Technical Report 1358

Automated Program Recognition by Graph Parsing

Linda Mary Wills

MIT Artificial Intelligence Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Technical Report No. 1358 July 1992

Automated Program Recognition by Graph Parsing

Linda Mary Wills

Abstract

The recognition of standard computational structures (clichés) in a program can help an experienced programmer understand the program. Based on the known relationships between the clichés, a hierarchical description of the program's design can be recovered. We develop and study a graph parsing approach to automating program recognition in which programs are represented as attributed dataflow graphs and a library of clichés is encoded as an attributed graph grammar. Graph parsing is used to recognize clichés in the code.

We demonstrate that this graph parsing approach is a feasible and useful way to automate program recognition. In studying this approach, we have experimented with two medium-sized, real-world simulator programs. There are three aspects of our study. First, we evaluate our representation's ability to suppress many common forms of program variation which hinder recognition. Second, we investigate the expressiveness of our graph grammar formalism for capturing programming clichés. Third, we empirically and analytically study the computational cost of our recognition approach with respect to the real-world simulator programs.

Copyright © Massachusetts Institute of Technology, 1992

The research described here was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the following organizations: National Science Foundation under grants IRI-8616644 and CCR-898273, Advanced Research Projects Agency of the Department of Defense under Naval Research contract N00014-88-K-0487, IBM Corporation, NYNEX Corporation, and Siemens Corporation.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the policies, expressed or implied, of these organizations.

Acknowledgments

I would like to thank my thesis advisor, Chuck Rich, for his continual support and encouragement over all my years at MIT. He has provided valuable guidance and advice at crucial times and he has shared many interesting ideas with me. I admire his energy, generosity, and integrity.

I am thankful to Richard Waters for his constant encouragement and cheerfulness, and for providing many fresh insights.

I am grateful to the members of my committee, David McAllester, Peter Szolovits, as well as Chuck Rich and Richard Waters, for their patient and careful reading of my thesis. They offered valuable insights and suggestions for presenting these ideas and they have broadened my perspective.

I appreciate Rudi Lutz's willingness to discuss the subtleties of his parsing algorithm. I have also benefited from helpful discussions with Yishai Feldman, John Hartman, Stan Letovsky, and Dilip Soni.

Several members of the AI Lab have provided encouragement and interesting discussions, especially Bonnie Dorr, Eric Grimson, Bob Hall, Ellen Hildreth, Tomas Lozano-Perez, Howard Reubenstein, Monica Strauss, Tanveer Fathima Syeda-Mahmood, and Yang Meng Tan. I greatly appreciate the friendliness and exceptional helpfulness of Andrew Chien, Bill Dally, and Mike Noakes.

The generosity, support, and encouragement of Christian Bauer, Avelino Gonzalez, and Soheil Khajenoori made it possible for me to finish this thesis. I am grateful to Ashok Goel, Janet Kolodner, Robert McCurley, and Spencer Rugaber for many interesting technical conversations.

I appreciate the moral support of my friends, Janet Allen, Anita Killian, Anuja Mariwala, Elizabeth Turrisi, and especially Jean Moroney.

I am thankful to my family – Mom and Dad, Len and Janet, Judy Ann and Jim, Diane, Tom, Stephen, Mark, Mom and Dad Wills, Kitty and Stevie – for providing many happy distractions.

I am fortunate to have a wonderful husband, Scott, who gives me unfailing love and support, and so much happiness.

Finally, I am grateful to my parents for their constant love and their confidence in me. This thesis is dedicated to them.

Contents

1 Introduction **5**

1.1 Motivations 5

1.2 Toward a Hybrid Program Understanding System 8

1.3 What is Involved in Automating Program Recognition? 10

1.4 Graph Parsing Approach 12

1.5 Goals and Contributions 15

1.6 Outline of Report 18

2 The Knowledge, Program Corpus, and Recognition Examples **19**

2.1 What are the Clichés? 19

2.1.1 Simulation Domain Context 20

2.1.2 Informal Cliché Acquisition Strategy 21

2.1.3 Sequential Simulation Clichés 23

2.1.4 The General-Purpose Clichés 33

2.2 Real-World Programs 33

2.3 Recognition Examples 38

2.3.1 Common Program Variations 38

2.3.2 Examples of Capabilities 39

2.4 Breadth of Coverage 53

3 The Flow Graph Formalism **59**

3.1 Flow Graphs 60

3.2 Flow Graph Grammars 62

3.2.1 Embedding Relation 62

3.2.2 Flow Graph Grammar Derivations 64

3.2.3 Attribute Conditions and Transfer Rules 65

3.3 Motivations for Formalism: Program Recognition Application 69

3.3.1 The Partial Program Recognition Problem 73

3.4 Extensions to the Flow Graph Formalism 74

3.4.1 Structure-Sharing 76

3.4.2 Aggregation 80

3.5 Chart Parsing Flow Graphs 100
3.5.1 Recognizing Share-Equivalent Flow Graphs 109
3.5.2 Recognizing Aggregation-Equivalent Flow Graphs 110
3.5.3 Matching St-Thrus 113
3.6 Related Graph Grammar Work 119
3.6.1 Classes of Graphs 119
3.6.2 Embedding Mechanism 120
3.6.3 Graph Parsers 120

4 Applying Parsing to Recognition **122**
4.1 Expressing Programs and Clichés in the Flow Graph Formalism 122
4.1.1 Attribute Language 123
4.1.2 The Plan Calculus 130
4.1.3 Codifying Clichés: Using the Plan Calculus as a Stepping Stone . . 134
4.1.4 Examples of Codifying Simulation Clichés 143
4.2 Architectural Details 154
4.2.1 Translating Programs to Flow Graphs 154
4.2.2 Additional Monitor to Handle Recursion Unfolding 156
4.2.3 Paraphraser 160

5 Capabilities and Limitations **163**
5.1 Variations Tolerated 163
5.1.1 Syntactic Variation 164
5.1.2 Organizational Variation 167
5.1.3 Delocalized Clichés 169
5.1.4 Unrecognizable Code 169
5.1.5 Function-Sharing 174
5.1.6 Redundancy 174
5.1.7 Implementation Variation 175
5.2 Limitations 175
5.2.1 Missing or Derived Dataflow 176
5.2.2 "Missing" Cliché Parts 178
5.2.3 Expressing Clichés with Loose Constraints 179
5.2.4 Enqueuing New Messages and Events 183
5.2.5 Modifications to Example Programs 184
5.2.6 Conclusion 186

6 Analysis **187**
6.1 Cost 188
6.1.1 Brief Algorithm Description 188
6.1.2 Complexity 192

6.2 Counting Items . . . 195
6.2.1 Item Trees . . . 196
6.2.2 Constraints Prune Item Trees . . . 197
6.2.3 Grammar Facilitates Reusing Sub-Search Space Exploration . . . 208
6.2.4 Empirical Observations of Item Trees . . . 208
6.2.5 Modeling Constraint Consistency . . . 213
6.2.6 Counting Zip-ups . . . 214
6.2.7 Partial Node Orderings . . . 216
6.2.8 Summary of Item Count . . . 222
6.3 Component Costs . . . 222
6.4 Other Performance Improvements . . . 225
6.4.1 Decomposition . . . 225
6.4.2 Indexing . . . 227
6.4.3 Interleaved Decomposition and Indexing . . . 227
6.4.4 Avoiding Unnecessary Copying . . . 228
6.5 Conclusion . . . 229

7 Conclusions **231**
7.1 Empirical Studies . . . 232
7.2 Future . . . 233
7.2.1 Multiple Recursion . . . 233
7.2.2 Interfacing with Other Recognition Techniques . . . 233
7.2.3 Disambiguating Data Structure Operation Instances . . . 234
7.2.4 Side Effects to Mutable Data Structures . . . 237
7.2.5 Advising GRASPR . . . 240
7.3 Related Work . . . 243
7.3.1 Representation . . . 244
7.3.2 Other Recognition Techniques . . . 248
7.4 Applications . . . 254

A Flow Graph Recognition is NP-Complete **255**

B The Example Programs **259**

C The Grammar Encoding the Cliché Library **289**

Chapter 1

Introduction

Experienced engineers are able to quickly determine the behavior and properties of a complex device by recognizing familiar, standard forms in its design. These standard forms, which we call *clichés* [110, 112, 115, 137, 117], are combinations of primitive mechanisms which engineers use frequently because the combinations have been found useful in practice. From experience, the engineers have come to expect the clichéd forms to exhibit certain known behaviors. By relying on this "pre-compiled" knowledge, engineers are able to efficiently understand and build complex devices containing clichéd components without always reasoning from first principles. Rich [110, 112, 117] has developed a model of engineering problem solving in which synthesis and analysis methods are based on the recognition and use of clichés. He calls these *inspection methods.*

This report deals with automating the recognition of clichés in computer programs. Clichés in the software engineering domain are stereotypical algorithmic computations and data structures. Examples of algorithmic clichés are list enumeration, binary search, and quick-sort. Examples of data-structure clichés are sorted list, priority queue, and hash table.

Several experiments [58, 83, 128, 142] give empirical data supporting the psychological reality of clichés and their role in understanding programs. In trying to understand a program, an experienced programmer may recognize parts of the program's design by identifying clichéd computational structures in the code. Knowing how these structures implement other more abstract structures, the programmer can build a hierarchical description of the program's design. We call this process *program recognition.* Program recognition is one technique, among several, used by programmers in the more general task of understanding programs.

1.1 Motivations

It is because human software engineers recognize clichés that we would like to automate program recognition. This gives us both theoretical and practical motivations.

From a theoretical standpoint, automated program recognition is an interesting artificial

intelligence problem. It is an ideal task for studying how programming knowledge and experience can be represented and used. (However, in automating program recognition, the goal is not to mimic the cognitive *process* used by programmers to recognize clichés, but to mimic only the use of experiential knowledge in the form of clichés to achieve a similar result of understanding the program.)

Our practical motivation stems from an interest in building automated systems that assist software engineers with tasks requiring program understanding, such as inspecting, maintaining, and reusing software. Such collaboration requires that the automated assistant be able to communicate with engineers in the same way as they communicate with each other when performing these tasks. They refer to instances of clichés and assume knowledge of their well-known properties and behaviors. For example, they might discuss changing a program from using an ordered associative linked list to using a hash table to gain efficiency. They discuss the change at a high level of abstraction and justify their design decisions using the established properties of the clichés. They are also able to explain the design of a program to each other on multiple levels of abstraction. They can convince each other of the properties or behavior of a program by pointing out the existence of clichés in its design and then leveraging off the accumulated body of experience surrounding the clichés. The known properties of the clichés are used directly, rather than constructing formal proofs or performing formal complexity analyses to establish that the properties hold.

If an automated assistant is to collaborate with human engineers in the same way, it must share the same knowledge of clichés and their properties. It must be able to recognize instances of clichés, without requiring the human engineer to explicitly identify and locate them in a program.

This recognition ability would be a valuable component of automated software tools and assistants that perform tasks requiring program understanding. They would be able to explain their understanding of the program in terms familiar to a human engineer. They can respond to requests from the engineer that are phrased in terms of abstract computational structures in the program, rather than low-level commands that spell out actions to be performed on language primitives. (For example, Waters' `KBEmacs` [116, 117, 139] shows how an automated assistant can aid a human engineer while communicating at a high-level of abstraction. In `KBEmacs`, this model is constructed as the program is being built. A tool like `KBEmacs` can be used to maintain existing code (not written with the help of `KBEmacs`), if the clichés from which the code is constructed are recognized.)

Incorporating an automated recognition system into software tools and assistants yields more than just communications benefits for human-computer interaction. By mimicking the human engineer's "short-cut" to understanding a program's design, an automated recognition system provides an efficient way to reconstruct design information. It bypasses complex reasoning about how behaviors and properties arise from a certain combination of language primitives. The behaviors and properties can be used directly by these tools.

Collaboration between a person and an automated recognition system is mutually ben-

eficial. An automated recognition system provides capabilities which complement the person's abilities. An automated system has significantly better memory capabilities than a person. These are valuable in maintaining multiple possible views of the program and in keeping track of details about what has been found so far. Also, some clichés may be easier for the computer to recognize because they are hidden or delocalized in the textual code representation, but are localized in the computer's internal representation.

On the other hand, people have some capabilities that can greatly aid the recognition system. They may have access to many different sources of knowledge about the program, beyond the source code, including its goals or specification, documentation, comments, execution traces, a model of the problem domain, and typical properties of the program's inputs and outputs. Even though some of this information can be incomplete and inaccurate, it provides an important independent source of expectations about a program's purpose and design. These expectations can be used to guide the recognition system by focusing its search on particular parts of a program for particular clichés.

The person can also provide information not easily recoverable from the code which can help the recognition system to recognize more of the program. For example, the person can undo an optimization that takes advantage of an opportune dataflow equality. This may uncover a dataflow dependency that must exist for a particular cliché to be recognized. (More concrete instances of the type of information that can help push the recognition of some clichés through are described in Section 5.2.)

Automated tools are also being developed to aid the human engineer in extracting design information and generating expectations from many different sources in addition to the code. An exemplary system is DESIRE, which is being developed by Biggerstaff [12, 13]. A central part of DESIRE is a rich domain model, which contains machine-processable forms of design expectations for a particular domain as well as informal semantic concepts. It includes typical module breakdowns and typical terminology associated with programs in a particular problem domain. Techniques for recognizing patterns of organization and linguistic idioms in the program are being developed to generate expectations of the typical concepts associated with these patterns. These expectations can be used to quickly draw attention to sections of the program where there may be clichés related to a particular concept in the domain.

Other, more conventional techniques for reverse engineering large programs have focused on extracting a given system's module structure. This is typically done by using clustering [62] and slicing [59, 140, 141] techniques, which bring together parts of a program based on identifier and procedure names, data dependencies, and call relationships, among other features [13, 19, 46, 51, 56, 123, 124, 143]. Programming and maintenance environments, such as MicroScope [7], Cleveland's system [20], and Marvel [66], provide tools for performing various types of dependency, dynamic, and impact analyses and for browsing the results in the form of call graphs, dataflow graphs, execution histories, and program slices.

These techniques and environments can contribute to a user's understanding of a pro-

gram. While they alone do not provide a deep understanding, they extract information that can help a person generate advice and expectations. Based on these, the person can guide an automated recognition system, so that a deeper understanding may be obtained. The results of recognition can in turn enhance the capabilities of these automated techniques by providing a more abstract view of a program. For example, dependencies between more abstract data objects can be computed and used to create more abstract clusters.

1.2 Toward a Hybrid Program Understanding System

Because program understanding requires many different techniques besides program recognition, and draws upon various sources of knowledge besides the code, program understanding systems of the future will be hybrid systems. They will integrate many different special-purpose components for extracting design information from a program and its associated documentation, domain model, etc. The components will communicate with human engineers, who can provide additional guidance and information.

The benefits of such co-operation between specialists in solving complex problems that require several, diverse types of knowledge are well known. For example, research in blackboard architectures [37, 63, 99] and hybrid knowledge representation systems [113] study ways of achieving co-operative problem solving.

Figure 1-1 shows a model of a hybrid program understanding system. It is roughly divided into two complementary processes: expectation-driven (top-down) and code-driven (bottom-up). The heuristic top-down process uses knowledge such as the program's goals, domain model, and documentation to generate expectations about the program's design. These can be used to guide the code-driven process, which can confirm, amend, or reject them by checking them against the code.

Since there are many different types of things an engineer or application tool might wish to understand about a program, the program understanding system can be directed by specific questions from the engineer or application.

The details of this hybrid system have not yet been fleshed out. We believe that a key part of the code-driven component is an automated recognition system. The labels on the communication links between the expectation-driven and code-driven components are useful inputs and outputs to a code-driven system based on recognition. However, these do not entirely specify the communication between, or the nature of, these components. Also, the diagram is not meant to imply that all the techniques integrated into the hybrid system are either solely code-driven or expectation-driven. Some may themselves be hybrids.

Some of the questions that must be answered in the design of such a hybrid system are what techniques should be incorporated and what is the appropriate division of labor between them? There are also managerial problems in the co-ordination of techniques and the integration of different types of knowledge and representations [93].

Determining which techniques to incorporate and what their individual responsibilities

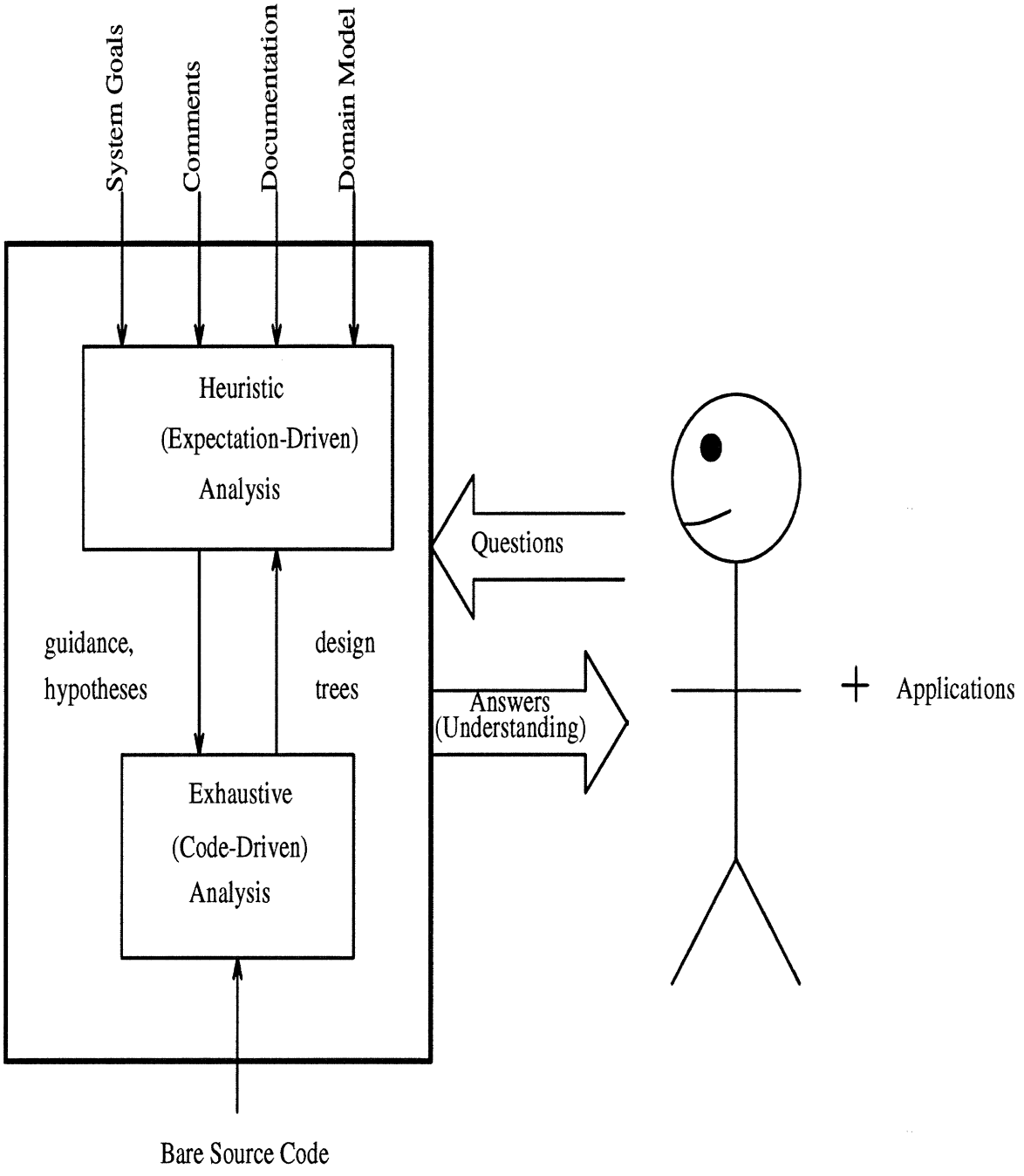


Figure 1-1: A hybrid program understanding system.

are requires analyzing the candidate techniques to determine their relative strengths, limitations, and computational expense. Our research takes a step toward the long-term goal of a hybrid program understanding system by exploring the strengths and weaknesses of a particular program recognition technique.

In particular, we develop and study a graph parsing approach to program recognition. This approach represents the program in a dataflow graph representation and the cliché library in a graph grammar and then uses graph parsing to recognize clichés in the code. The grammar rules capture implementation relationships between the clichés. The parsing technique yields a hierarchical description of a plausible design of the program in the form of derivation trees specifying the clichés found and their relationships to each other.

We demonstrate that the flow graph parsing approach is a feasible and useful way to automate program recognition. We also identify its shortcomings. This information will help us to make the appropriate division of labor between the integrated components of the hybrid program understanding system.

To do this, we developed an experimental system that performs recognition on realistic, medium-sized programs. Given a program and a library of clichés, it finds all occurrences of the clichés in the program and builds a hierarchical description of the program in terms of the clichés found. (In general, there may be several such descriptions.) We call our system GRASPR, which stands for "GRAph-based System for Program Recognition."

1.3 What is Involved in Automating Program Recognition?

To automatically recognize interesting clichés in real-world programs, a number of issues must be addressed. This section discusses the key issues.

What are the clichés? We must identify the clichés that programmers use. These include both general programming clichés that most programmers use (e.g., those found in textbooks on programming [3, 21, 76]) and domain-specific clichés that are used to solve particular problems. For the results of recognition to be useful, we also need to collect the information that is associated with each cliché, such as its behavior, pre- and post-conditions, complexity, and common design rationale for choosing it. In general, cliché library acquisition requires domain modeling, which is itself an entire area of active research [106].

How are clichés and programs encoded? Once clichés are identified, they must be expressed in a machine-manipulable form which makes relationships between the clichés explicit. To facilitate recognition, the representation of clichés and programs should suppress details that obscure the similarity between two instances of the same cliché. A negative example is a textual representation of clichés and programs. The program text contains details about how data and control flow is achieved in terms of programming language constructs. This introduces syntactic variation across programs that achieve the same data and control flow but use different constructs or different programming languages. Other

types of variation besides syntactic include variations in the implementations of some abstract cliché, the organization of components, the amount of redundant computation, and the contiguousness (or localization) of clichés. These are described further in Sections 2.3.1, 5.1, and 5.2. The representation should remove as much variation as possible between two instances of the same cliché.

How are clichés recognized efficiently? The recognition technique must deal with variation, allow partial recognition of a program, and have a flexible control strategy. To deal with the variation that the chosen representation cannot eliminate, the recognition technique might view the program in multiple ways and at several levels of abstraction, or introduce transformations to reveal the similarities between programs and clichés.

In addition to dealing with variation, the recognition technique should allow partial recognition of the program, since programs are rarely constructed entirely of clichés. Unfamiliar parts of the program must not deter recognition of the familiar parts.

Finally, the recognition technique should have a flexible control strategy, particularly if it is expected to interact with other components in a hybrid system. There may be a range of possible inputs to the recognition system as well as a variety of types of outputs desired from it. The types of inputs to the recognition system that tend to vary are the advice given to guide the search for clichés and the expectations and hypotheses generated from external knowledge sources. These vary depending on the amount of information that already exists about the program and its development (e.g., in its associated documentation). The input also changes as the recognition system and expectation-driven components interact. The task to which recognition is being applied also affects the type of information available as input. For example, in debugging, verification, or program tutoring applications, a specification of the program is often available from which strong guidance can be generated, while this information is often lacking in maintaining old code.

The application task can also place restrictions on the time and space allotted to the recognition system. For example, a real-time response may be required of the system if a person is using it interactively as an assistant in maintaining code. In this situation, it may be more desirable to quickly recognize clichés that are more "obvious" rather than spending more time to uncover clichés that are more hidden (e.g., by an optimization which must be undone for them to be revealed). It should be possible to prioritize the search for certain clichés, so that obvious ones are recognized early, while still reserving a "try harder" phase in which the more hidden clichés can be found. This allows us to gain efficiency without permanently sacrificing completeness.

Not only is it important that the recognition system be responsive to directions and additional information besides the code, it must have a control strategy that is flexible enough to perform a variety of recognition tasks. There are many reasons a human engineer or some application tool may want recognition to be performed, since they typically want to understand many different things about a program. The recognition task depends on what needs to be understood. For example, if the recognition system is going to be applied

to verification, it can use a strategy that finds *any* complete recognition of the program. On the other hand, if it were applied to documentation generation, it would be better for it to produce all possible full, as well as partial, analyses. For applications in which near-misses of clichés should be recognized, such as debugging, the best partial analysis might be desired. A flexible control strategy is needed that can be tailored to a variety of different recognition tasks.

To summarize, the main issues in automating recognition are: acquiring the cliché library, choosing a representation and efficient technique that tolerates variation, and providing a flexible control strategy. This report deals primarily with the problems of tolerating variation and providing a flexible, efficient recognition technique. It deals secondarily with the cliché acquisition problem by discussing experiences in manually acquiring our cliché library. It does not discuss aids for acquisition.

1.4 Graph Parsing Approach

There are two key aspects of our approach.

1. *Representation shift:* Instead of looking for clichés directly in the source code, GRASPR translates the program and clichés into a language-independent, graphical representation. The clichés and the relationships between them are encoded in graph grammar rules.

2. *Flexible recognition architecture:* Recognition is achieved by parsing the program's graphical representation in accordance with the graph grammar encoding of the clichés. A chart parsing algorithm is used which makes search and control strategies explicit, enabling them to accept advice and additional information from external agents.

Figure 1-2 shows GRASPR's architecture. In keeping with the bottom-up nature of the recognition process, the figure shows the program and cliché library inputs at the bottom and the more abstract results of recognition at the top. The recognition process is to be read upward. This also makes it easier to see how GRASPR fits into the hybrid system shown in Figure 1-1.

GRASPR translates the program into a *flow graph*, which is a restricted type of directed acyclic graph (as is described in Section 3). Basically, the graph represents operations in its nodes and dataflow dependencies between them in its edges. It is annotated with attributes which represent additional information about the program, for example, its control flow.

A program is translated into an attributed flow graph in two steps. The first step performs a data and control flow analysis of the program to yield a Plan Calculus representation of it. The Plan Calculus is a program representation developed by Rich, Shrobe, and Waters [110, 111, 112, 117, 127, 137] in which a program is captured in an annotated directed

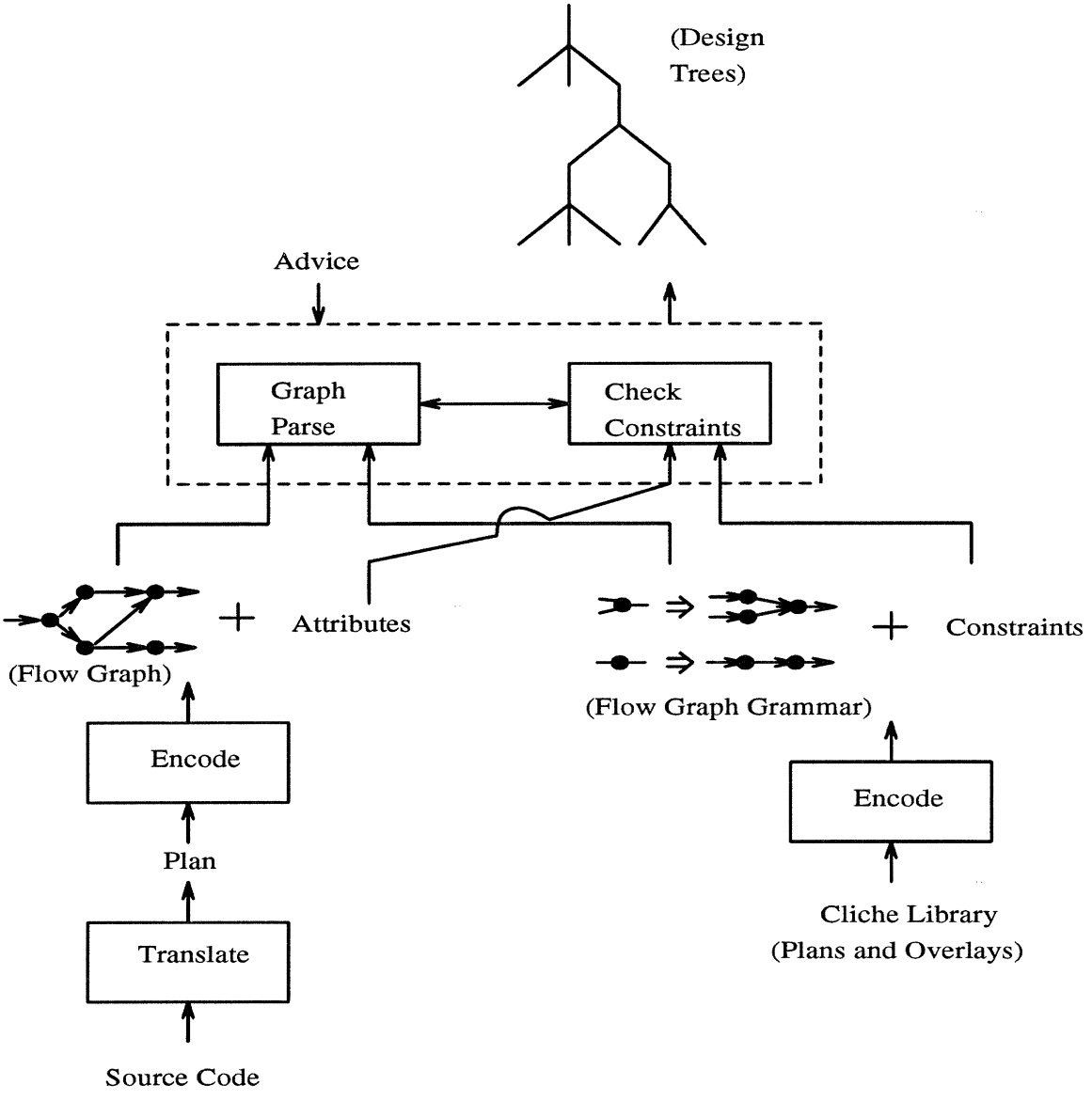


Figure 1-2: GRASPR's architecture.

graph, called a *plan*. The structure of this graph explicitly captures both data and control flow, as well as aggregate data structure accessors and constructors, and recursion. The second step of the translation encodes the plan in an attributed flow graph representation.

The Plan Calculus is used as a stepping stone in the translation of the program to an attributed flow graph. The main reason the program is not translated directly to the flow graph is that the attributes are easier to compute from the plan than to generate in one shot during the data and control flow analysis. A secondary reason is that GRASPR is intended as one component of an intelligent software engineering assistant, called the *Programmer's Apprentice* (PA) [117]. By being able to encode plans in its internal flow graph representation, GRASPR can more easily interface to other components of the PA, which all share the Plan Calculus representation.

The Plan Calculus is also a representation that has been found useful in representing the cliché library. It allows relationships between clichés to be captured in the form of *overlays*. These represent the knowledge that an instance of one cliché can be viewed as an instance of another (e.g., a specification cliché and an implementation cliché).

Clichés are translated from a Plan Calculus representation to an attributed flow graph grammar by a process similar to the encoding of plans in attributed flow graphs. The grammar rules encode the relationships specified in overlays. Each rule also places constraints on the attributes of any flow graph structurally matching the rule's right-hand side. These constraints explicitly encode the variations that are allowed in the values of attributes in cliché instances.

Once the program and cliché library are encoded in an attributed flow graph and flow graph grammar, recognition is achieved by parsing the flow graph in accordance with the grammar. Constraint checking is interleaved with parsing for efficiency (as described in Sections 3.2.3 and 6.2.2). Essentially, graph parsing matches the dataflow structure of clichés and constraint checking deals with the other details of clichés that cannot be represented in the graph structure or are sources of too much variation if graphically represented.

Parsing yields hierarchical descriptions of the program's design in the form of the possible derivations of the program's flow graph from the flow graph grammar that was extracted from the cliché library. These are called *design trees*.

By shifting the representation of programs and clichés from text to a flow graph, GRASPR is able to overcome many of the difficulties of syntactic variation and noncontiguousness. It abstracts away the syntactic features of the code, exposing the program's algorithmic structure. It concisely captures the data and control flow of programs, independent of the language in which they are written. Also, many clichés that are delocalized in the program text are much more localized in the flow graph representation.

The graph grammar captures relationships between clichés so that the results of recognition can be given on multiple levels of abstraction. Grammar rules relate abstract clichés to their implementations. This enables GRASPR to deal with implementation variation: two implementation clichés can be recognized as the same abstract cliché. The grammar also

captures commonalities between clichés so that large numbers of clichés can be encoded more compactly.

In using a graph parsing approach, we are not trying to mimic the recognition process of human programmers. No claim is being made that formal parsing is a psychologically valid model of how programmers understand existing programs. For the present work, a grammar is simply a useful way to encode the programmer's experiential knowledge about programming so that parsing can be used for program recognition.

1.5 Goals and Contributions

The goal of this research is to show that graph parsing is a good computational model for automating program recognition, and to identify its capabilities and limitations. We demonstrate the following:

- We can encode many interesting programming clichés and the relationships between them in a flow graph grammar.
- The flow graph formalism provides an effective representation for tolerating many classes of variation.
- Flow graph parsing can be used to recognize the clichés. The derivation trees that result provide a useful hierarchical description of the program, over multiple levels of abstraction.
- Limitations in the power of the recognition system to recognize certain clichés can be alleviated by accepting additional design information from an external agent (such as a person), who is interacting with it.
- Recognition by flow graph parsing can be performed efficiently in real-world situations.
- The complexity of the recognition process can be controlled if the parser's control strategy is sufficiently flexible and responsive to advice from an external agent.

We show these things by experimenting with real-world program examples, which are medium-sized (in the 500 to 1000 line range) simulation programs written in Common Lisp by members of a parallel-processing research group at MIT. (Section 2.2 describes them further.) We are able to express both general programming clichés and clichés from the simulation domain in a flow graph grammar. `GRASPR` recognizes these clichés in the example programs efficiently.

Our experimentation also reveals shortcomings in our graph parsing approach. Many of the limitations can be compensated for by other techniques and by using other sources of knowledge which may be available in the context of a hybrid program understanding system.

The specific contributions of this research are the following. (This list includes brief statements of how these contributions advance the state-of-the-art of recognition research. More details on related research are given in Section 7.3.)

- We develop and use a *flow graph grammar formalism* in which programs and clichés can be concisely represented so that much variation is eliminated and relationships between clichés are encoded.

  This graph-based representation has significant advantages over the text-based representations used by many other recognition systems, particularly in dealing with syntactic variation.

- We present a *recognition architecture* with a general, flexible control structure that can accept advice and guidance from external agents. The trade-off between recognition power and computational expense can be explicitly controlled so that some clichés are recognized quickly, while other more expensive recognitions are postponed to a "try-harder" phase. The algorithm exhaustively finds all possible recognitions of clichés and can generate multiple views of a program as well as partial "near-miss" recognitions. This architecture forms a seed for a hybrid program understanding system.

  Other recognition systems are committed to a rigid (often ad hoc) control strategy. Most search for a single best interpretation of the program, while permanently cutting off alternatives. They often build heuristics into the system for controlling cost that are chosen on a trial-and-error basis. They cannot try harder later to incrementally increase their power. They also cannot generate multiple views of the program when desired, nor provide partial information when only near-misses of clichés are present.

  Some recognition techniques can use information obtained from one or two other techniques (e.g., theorem proving or dynamic analysis of program executions) with which they are integrated. Many recognition techniques also take information about the goals and purpose of the program (in the form of a specification or model program). While these techniques show the utility of these additional sources of information, they *rely* on this information being given as input, rather than accepting it and responding to it if it becomes available.

- We *analyze* the graph parsing approach to program recognition to determine how it would fit into the context of a hybrid program understanding system.

  We address the questions:

  - What types of variations is the technique robust under? What types of variations are a problem. What other techniques must be used to remove the variation?
  - Are graph grammars expressiveness enough to encode programming clichés?
  - Is the technique feasible for large programs? How can the cost be controlled?

The observations we make in this analysis are based on our experiences in applying GRASPR to the recognition of two example programs. They do not represent complete lists of the capabilities and limitations of the graph parsing approach. Further experimentation is needed with more programs and in multiple problem domains.

Much of the early work in program recognition provides no analysis of the representations or techniques used. More recent research includes some empirical analysis, typically studying the accuracy of recognition and the recognition rates over sets of programs (usually student programs in program tutoring applications). With the exception of Hartman's work [55], discussions of limitations have focused mainly on practical implementational limitations, rather than on general limitations of the approach. They also do not describe how additional information or guidance can help.

- Our recognition system is able to recognize programs and clichés containing a wide range of types of program features. In particular, it is able to represent and recognize programs that contain conditionals, loops with any number of exits, recursion, aggregate data structures, and simple side effects due to assignments. (Suggestions for future work in dealing with side effects to mutable data structures are given in Section 7.2.4.) This allows GRASPR to recognize larger programs than existing recognition systems. It also enables encoding and recognition of domain-specific clichés as well as general-purpose ones, since many domain-specific clichés are aggregate data structure clichés. This allows empirical study of our recognition technique on programs that are not contrived nor biased toward our work.

  With the exception of CPU [84], existing recognition systems cannot handle aggregate data structure clichés and a majority do not handle recursion. Talus [95] heuristically handles some side effects to lists and arrays. The largest program recognized by any existing recognition system is a 300-line database program recognized by CPU. All other systems work with programs on the order of tens of lines. None deal with domain-specific clichés, except Laubsch's system [81, 82].

- A secondary contribution is a *graph parsing algorithm* which is an extension of the parsers of Lutz [90] and Brotsky [15] to handle a wider class of graph grammars. In particular, it is able to parse graph grammars that encode aggregation, which hierarchically groups graph edges, not just nodes. This algorithm has potential applications in areas other than program recognition, e.g., circuit verification and plan recognition. Section 7.2 discusses some applications.

- We do *not* contribute automated aids to the *acquisition* of the cliché library. However, we do discuss our experiences in manually acquiring the clichés.

  This type of discussion has not appeared in any other work on program recognition of which we are aware.

1.6 Outline of Report

Chapter 2 describes the cliché library and our experiences in acquiring it. It also demonstrates GRASPR's recognition of these clichés in the example simulation programs. Chapter 3 describes the flow graph formalism which forms the basis of our representation shift. It also presents a flow graph chart parsing algorithm, which provides a flexible recognition control strategy. It includes a summary of related work in the general area of graph grammar formalisms. Chapter 4 gives details of issues that arise in applying flow graph parsing to program recognition and how GRASPR solves them. Chapter 5 discusses the capabilities and limitations of the parsing approach in terms of the variations tolerated, and the expressiveness of flow graph grammars. Chapter 6 studies the computational cost of our approach, both empirically and analytically. Finally, Chapter 7 concludes with a summary of the strengths and weaknesses of the parsing approach, ideas for future work (particularly in the context of a hybrid system), and a brief comparative summary of related work in program recognition.

Chapter 2

The Knowledge, Program Corpus, and Recognition Examples

An important part of automating program recognition is codifying the knowledge that experienced programmers use to recognize programs. This knowledge is in the form of algorithmic and data structure clichés. It includes both general-purpose clichés that occur in programs over all problem domains, as well as those specific to a particular domain.

Our library must capture and express these clichés at a level of abstraction that allows them to be recognized in a broad range of programs. The ideal is that the clichés be concisely represented, but efficiently recognized in many forms. Recognition of a cliché should be immune to many common syntactic and implementational variations. For example, the same clichés should be recognized in programs that differ only in which syntactic binding and control constructs they use or in which programming languages they are written. Also, an abstract clichéd operation that exists in two programs should be recognized in both, even if the programs differ in which standard implementation of the operation is used.

This chapter discusses the clichés we have captured so far in our library. It also describes the corpus of programs we chose on which to base both our cliché acquisition and our empirical study of recognition. Finally, it gives examples of the capabilities of `GRASPR` in recognizing these clichés not only in our example corpus, but also in a range of variations of them. (Chapter 3 discusses the formalism we use to abstractly and concisely capture our clichés to make this possible.) Our examples provide both a demonstration of what is feasible as well as motivation for our formalism and recognition technique.

2.1 What are the Clichés?

Our cliché library contains a core set of general-purpose, "utility" clichés, along with a set of clichés from the domain of sequential simulation. The domain-specific clichés are built on top of the core utility clichés (i.e., they use utility clichés as components or implementations).

The general-purpose clichés are well-known, widely used algorithms and data structures,

such as those described in introductory computer science textbooks (e.g., [3, 21, 76]). They are found in programs across all problem domains. They include common operations on priority queues, hash tables, lists, and first-in-first-out (FIFO) queues, as well as basic iteration clichés, such as sequence enumeration, filtering, accumulation, and counting.

The domain-specific clichés in our library are found in programs that *sequentially simulate parallel systems.* More specifically, we have encoded the subset of common algorithms and data structures found in this domain that are used to sequentially simulate *message-passing* parallel systems.

A message-passing system contains a collection of processing nodes which communicate with each other via messages. Each processing node contains a processor, a network interface, and a block of distributed memory. The message-passing system takes a program in the form of a set of message handlers and a starting message. The program begins by sending the starting message to its destination node. The node executes the handler for that message's type. In addition to changing the state of the node, this can cause the node to send messages to other nodes (e.g., to request the value of some variable or to delegate some sub-tasks). When these messages are handled by their destination nodes, additional messages might be sent.

It is possible for a message to be received by a node while it is handling another message. Therefore, each node has a local buffer which accumulates the messages received while the node is busy. When the node finishes handling a message, if its buffer is non-empty, the node pulls a message from the buffer and handles it. The buffer is emptied in FIFO order. This is done to maintain the invariant that two messages received by the same node must be handled in the order in which they are received.

The behavior just described is simulated by the programs in which our library's domain-specific clichés are found. This is a subset of the actual behavior of a real message-passing system, which also includes routing messages through the network, for example. However, this simplified model is a typical one simulated in parallel architecture research. The simulation allows statistics to be gathered on such properties as the number of nodes busy over time (a measure of concurrency), average message execution times, and average message waiting times.

2.1.1 Simulation Domain Context

It is instructive to see how the domain we have chosen fits into the larger world of simulation programs. It is a subset of the problem domain of *sequential* simulation, as opposed to *parallel* simulation, of parallel systems. Our cliché library contains only sequential algorithmic clichés.

Within the domain of sequential simulation, there are two types of simulators: *discrete-event* and *continuous.* Discrete-event simulators model the behavior of a system over discrete points in time. Continuous simulators model behavior that is characterized by state that

changes continuously. (Continuous simulators typically solve a set of differential equations that express how the system's state changes over time. Continuous simulation is used, for example, to study heat dissipation in computer systems.) Our simulation clichés are found in discrete-event simulators. The discrete points in time at which a message-passing system can be modeled are when a message is sent, received, or handled.

Within the domain of discrete-event sequential simulation, our class of simulator programs are most similar to simulators that model *queueing systems* [91]. In a queueing system, there is a collection of one or more *servers* which service *tokens* (sometimes called "customers"). There is a notion of arrival time and processing time of tokens; tokens get buffered in a queue if they arrive while a server is busy. The queueing discipline is typically first-in, first-out, but it can be a different one if tokens need not be serviced in the order in which they arrive. A common real-world situation captured by the queueing system model is the servicing of bank customers by one or more tellers, where the customers wait in a single line.

The queueing system model (using a FIFO queueing discipline) is similar to the message-passing multi-processor model. Servers are analogous to processing nodes and servicing a token is analogous to handling a message. However, there are two key differences. One is that in the queueing system, servicing a token does not create new tokens which feed back to the servers. In the message-passing machine model, handling a message can cause new messages to be sent. The other key difference is that in the queueing system model, the waiting tokens are not targeted for a particular server to service. Whichever server is idle when a token is removed from the queue is the one that gets the job. In the message-passing model, on the other hand, each message is sent to a particular node for handling. The message's destination is determined when the message is sent. Our class of simulator programs can be seen as modeling a multi-queue multi-server system with feedback (in which tokens are targeted for particular servers and servers have local FIFO queues for buffering tokens when the server is busy).

2.1.2 Informal Cliché Acquisition Strategy

In acquiring our domain-specific clichés, we used an informal strategy. (Developing a domain modeling methodology for cliché acquisition is beyond the scope of this research.) We worked in two directions. One was bottom up by manually understanding two program examples in our domain. (These are described in Section 2.2.) This allowed us to identify concrete computational structures that were used in the simulators' designs. The differences between the two programs in implementing the same high level operation helped us to generalize our clichés. The similarities between the programs pointed out common components that some clichés shared. We were fortunate in that the authors of the programs were accessible for answering our questions about the design of the programs. Their explanations helped us not only to understand the programs, but also to identify the clichés, since the

authors often referred to algorithms and data structures that they considered to be typical.

Our second direction was top-down. We read textbooks in the area of simulation, such as [91, 151], to pick up the vocabulary and descriptions of typical high-level computational structures that are used. We then mapped these down to portions of the example programs that embody them.

In identifying the clichés to be captured, we tried to identify the most general form of each cliché and then express it in a way that canonicalized specializations of it. (This was done both by using an abstract representation and by providing mechanisms for viewing specializations as the more general form.) However, sometimes this canonicalization was not possible and we needed to include specializations of the cliché in the library along with the generalized forms. In these cases, we relied on empirical frequency of occurrence of the specialized forms, to avoid enumerating all possible variations (which can be expensive and incomplete).

This issue came up most frequently in trying to capture clichéd operations on aggregate data structures. We encountered three distinguished types of parts of aggregate data structures:

- *Primary* – a part that holds a piece of data directly. (For example, a Hash Table data structure contains a Buckets part which is usually an array).

- *Handle* – a part that is used to look up a primary part. (For example, a data structure might contain a primary part Node that represents a processing node or it might contain an integer (an identification number) that is used to index into another data structure to retrieve the structure representing a node.)

- *Secondary* – a piece of data that is an unnecessary part of a data structure in that it can be computed from a primary part or a handle part of the data structure. These are usually cached values. (For example, a Circular-Indexed Sequence includes a sequence part, and two indices which keep track of the bounds on the filled-in portion of the sequence. It can have an additional secondary part which keeps a running count of the number of elements in the Circular-Indexed sequence. This part is unnecessary because it can be computed from the size of the sequence and the boundary indices.)

If we were to capture all aggregate data clichés in their general form – as aggregates of only primary parts – we would have trouble recognizing them in cases where handles are used and in cases where secondary (cached) parts are used to circumvent computation performed on primary parts. So, we capture these specialized forms, but only if they are common. That is, we capture data clichés that are common optimizations and common uses of handles.

Sometimes an optimization of some generalized cliché is possible in the particular context in which it is used, but this optimization is not a common one. Perhaps it takes advantage of a rare alignment with other clichés or of opportune dataflow equalities. Since it is not

common, it is not in the cliché library. (Likewise for handles.) Unless we can undo the optimization or use of a handle, the recognition of the cliché will be hindered. Section 5.1.5 describes a class of common optimizations which can be undone. Sections 5.2.2 and 5.2.1 discuss some optimizations and uses of handles that should be able to be undone, but which require advice from an external agent.

2.1.3 Sequential Simulation Clichés

There are two common designs for sequential simulators of parallel systems. One is a *synchronous* simulation, which mimics the real system by maintaining a global clock and simulating the actions of the nodes in "lock-step." On each tick of the clock, the simulator "advances" each node by simulating what the node would do in the real system on that clock tick. In this type of simulation, all simulated nodes are synchronized to the global clock. At each clock tick, the state of the simulated nodes gives a snapshot of the state of the system at the time represented by the clock tick.

The other common sequential simulator design is *event-driven.* In this type of simulator, there is an agenda of *events*, which represent work to be done by the nodes. The simulator iteratively pulls an event from the agenda and performs the work associated with it. This may cause new events to be generated, which are added to the agenda. The simulation ends when the agenda is empty. Unlike in synchronous simulation, the actions of the nodes are simulated asynchronously rather than all being in step with a global clock. The nodes each keep track of their own local time, which is updated when they process an event.

Our cliché library contains algorithmic and data structure clichés that make up the designs of event-driven and synchronous simulators for message-passing systems. The next two sections discuss these designs and the clichés from which they are constructed.

A Common Synchronous Simulation Design

A common design used in synchronous simulators of message-passing systems has data structures representing processing nodes and messages. (In this discussion, we denote the data structure representing a node as `SYNCH-NODE` to distinguish it from the real processing node. Similarly, `MESSAGE` denotes the data structure representing a real message.) Each `SYNCH-NODE` contains a Local-Buffer part, whose value is a FIFO queue of messages, and a Memory part which represents the state of the node being represented. Each `MESSAGE` data structure contains a Destination-Address which specifies the node to which the message it represents was sent. It also typically contains a message Type, which is used to look up a handler for the message, Arguments which are used in executing the handler, and Storage-Requirements which specify how much local memory space is need to store arguments and locals during handler execution.

All `SYNCH-NODE`s are collected in a sequence, called an `ADDRESS-MAP`, which maps an integer address to a `SYNCH-NODE`. The `SYNCH-NODE` indexed by an integer i is the one representing the

real node whose address is i in the machine being simulated. A global buffer of MESSAGEs is also maintained to help model message delivery delay, as is explained below.

A common algorithm used for synchronous simulation proceeds as follows. The simulation is begun by adding a "start" MESSAGE, which is given as input, to the global MESSAGE buffer. On each iteration of the simulation, the following actions are taken.

- A termination condition is checked and if satisfied, the simulation stops. This condition is that the global MESSAGE buffer and all the Local-Buffers of the SYNCH-NODEs are empty.

- The MESSAGEs in the global buffer are "delivered," which means each is placed in the Local-Buffer of the SYNCH-NODE to which they were sent (i.e., the SYNCH-NODE in the ADDRESS-MAP indexed by the MESSAGE's Destination-Address part).

- Each SYNCH-NODE is polled to see if it has any work to do, i.e., if it has any MESSAGEs in its Local-Buffer. If so, a MESSAGE is pulled from the buffer (maintaining FIFO order) and handled. If any new MESSAGEs are sent as a result, they are buffered in the global MESSAGE buffer.

The global MESSAGE buffer is used to ensure that delivery delay is modeled. Buffering the MESSAGEs sent during a clock cycle prevents a message from being sent and handled during the same cycle.

The invariant that messages to the same node are handled in the order in which they are received is modeled by using a FIFO queue to locally buffer the MESSAGEs that a SYNCH-NODE must handle. A MESSAGE will not be handled by a SYNCH-NODE until all the MESSAGEs enqueued on the FIFO queue ahead of it have been handled.

What it means for a MESSAGE to be "handled" (or what action of a processing node is simulated) by the simulator varies across simulators. It depends on why a simulation is being performed and which aspects of a message-passing system are of interest. For example, some simulators might want to simulate the message handler execution on the node in order to gather statistics about operation frequencies or average message execution time on each node. Other simulators might only want to simulate message sends that result from handler execution, in order to gather information about average message waiting times, typical size of buffers needed, and the number of nodes busy. In addition, the set of message handling actions that are simulated varies over the machines that are being simulated. The machine architecture of a real node determines which actions it performs; only these can be simulated.

We have begun to identify and capture some clichés in the area of simulating node actions. These include algorithms for looking up and executing message handlers as well as clichés found in the domain of program execution. Below we discuss the clichés we have captured so far and Section 5.2 describes the difficulties we encountered in acquiring them.

Although we have identified some clichés in this area, it is unlikely that the code for simulating the actions of nodes will always be a cliché. There is a wide variety of reasons to simulate a message-passing system, resulting in a wide range of node behaviors to mimic. This variation is reflected in the diverse code responsible for simulating a node's actions. So, we also look at the issues involved when an integral part of an algorithmic cliché for synchronous or event-driven simulation may be filled with unfamiliar, non-clichéd code. It is difficult to encode such a cliché in a flow graph grammar so that it can be recognized by graph parsing. This is discussed in Sections 4.1.4 and 5.2.3.

There are many variations of the algorithm described in this section that still achieve synchronous simulation. For example, on each iteration, our algorithm performs three actions in the following order: test for termination, deliver messages, and poll and advance nodes by one step. The other variations of this algorithm in which a different ordering is used also perform synchronous simulation. However, the current cliché library contains only the one given above as an algorithmic cliché. Section 5.2 discusses the problems we face in trying to concisely encode and recognize the other variations.

The algorithm and data structures used in this synchronous simulation design are captured in our cliché library as clichés. However, the clichés are not flat structures, but are hierarchically built out of other clichés. The hierarchical organization allows sharing of common sub-computations among clichés, which helps us avoid redoing work during recognition. This also highlights the salient characteristics between two similar clichés which is useful in controlling recognition cost and choosing between near-miss recognitions of the clichés. (However, no static organization can do this perfectly, since saliency is relative.)

Figure 2-1 shows the names of the algorithmic clichés upon which the Synchronous-Simulation algorithmic cliché is built. Lines connecting the names indicate relationships between the named clichés. (This is only a portion of the cliché library. Figure 2-3 shows additional algorithmic clichés used in a common event-driven simulation design which is described in the next section. Also, the fringe of the trees in Figures 2-1 and 2-3 contain the names of general-purpose clichés and small triangles to indicate that the sub-tree of cliché names upon which they are built is not shown. Refer to Figure 2-5 for these cliché names and how they relate to the other general-purpose clichés in the library.) Figure 2-2 shows the aggregate data clichés in our library and how they relate to each other.

The trees of cliché names are shown only to give a flavor of the structure of the cliché library. More description of the clichés and details of how they are encoded are given in Section 4.1.

There are three types of relationships between the clichés in the library. One type of relationship is *composition*: Clichés may contain other clichés as parts. (This relation is shown in the trees of Figures 2-1 and 2-2 as a set of branching lines, grouped by a circular arc. The root name represents a cliché that is composed of the clichés named by the branches.)

For example, the aggregate data structure `SYNCH-NODE` consists of two parts, a Buffer and

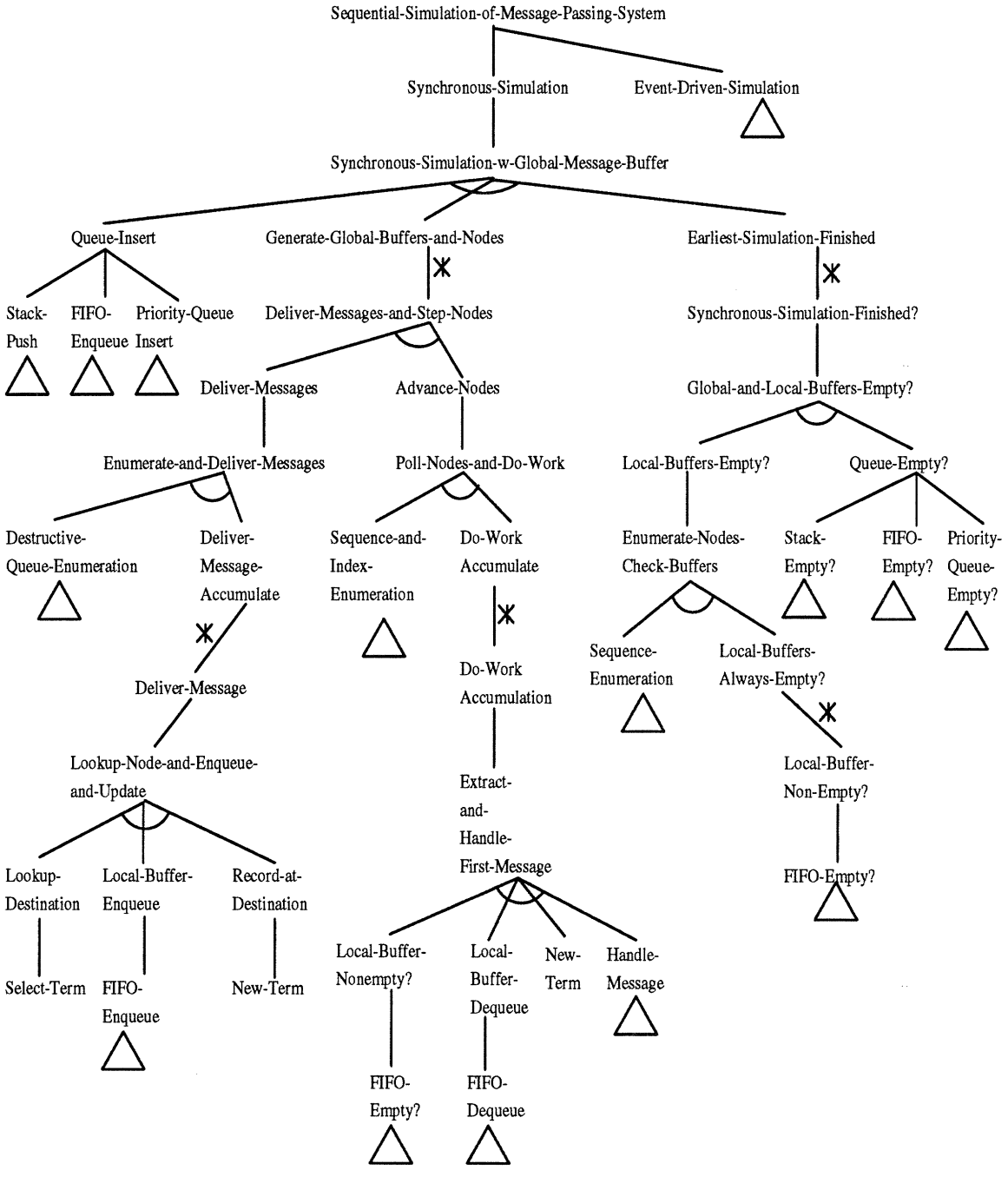


Figure 2-1: Synchronous simulation clichés.

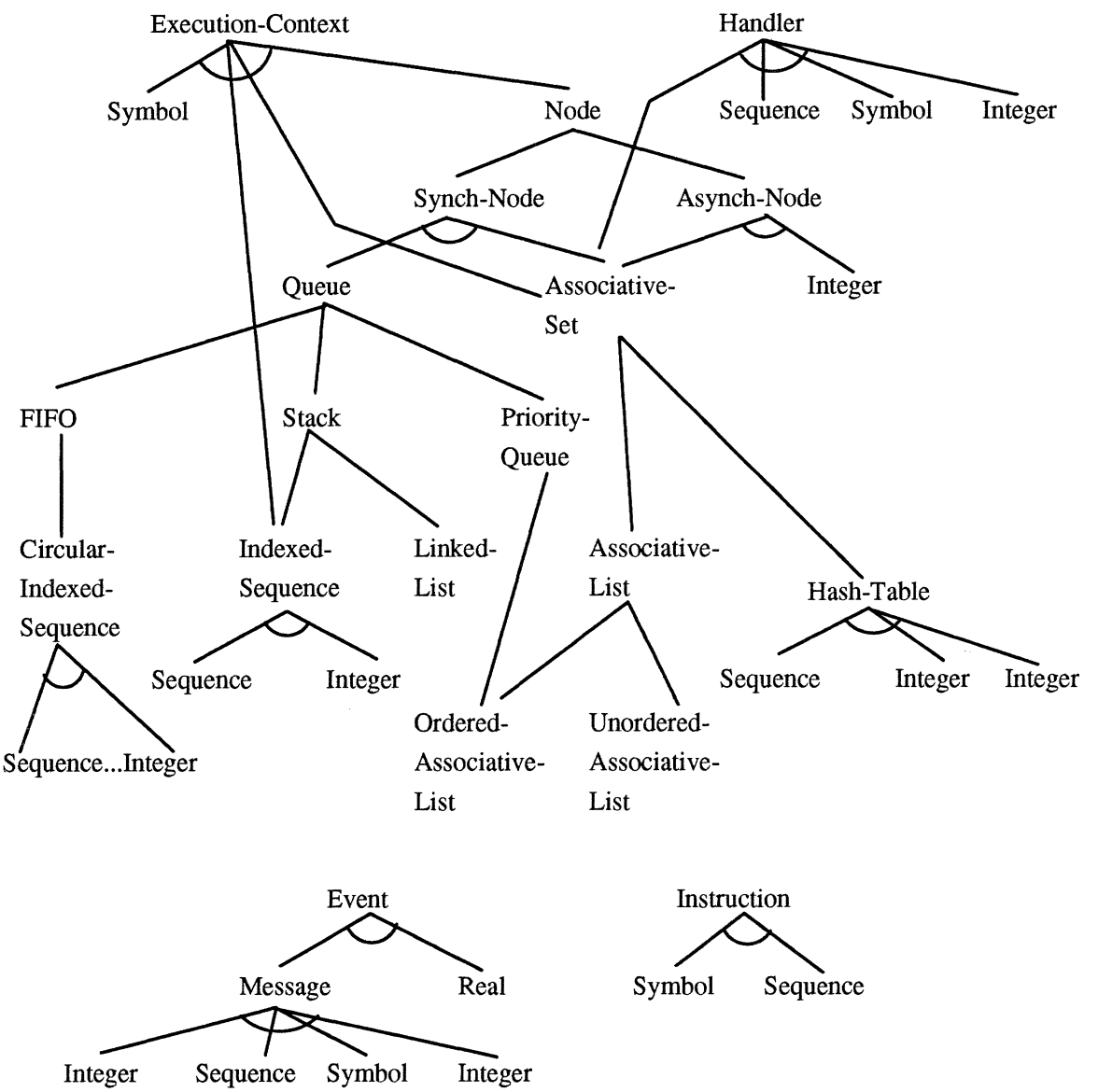


Figure 2-2: Aggregate data clichés.

a Memory, each of which is another cliché: a Queue and an Associative Set, respectively. A similar relationship can occur between algorithmic clichés. The algorithmic cliché of Synchronous Simulation using a Global Message Buffer is composed of three other clichés: Queue-Insert, Generate-Global-Buffers-and-Nodes, and Earliest-Simulation-Finished.

The second type of relationship that can occur between two clichés is an *implementation* relationship: A cliché may implement a more abstract cliché. For example, a FIFO, Stack, or Priority Queue can implement a Queue. Poll-Nodes-and-Do-Work is an implementation of Advance-Nodes. (Lines between cliché names in Figures 2-1 and 2-2 that are not grouped or starred represent this relationship. Of two clichés connected by a line, the upper one is implemented by the lower. Branching ungrouped lines represent alternative implementations of the root.)

The third type of relationship occurs when one cliché is a *temporal abstraction* of another. Temporal abstraction is a technique developed by Waters [117, 137, 138] and further extended by Rich and Shrobe [110, 127], in which a clichéd fragment of iterative computation is viewed more abstractly as an operation on a sequence of values – the sequence of values that are processed over time, one per iteration. For example, Sum is a temporally abstract operation that takes a sequence of numerical values and produces their total. This is a temporal abstraction of a loop fragment in which each iteration computes the sum of a new value and the result of the sum computed on the previous iteration. The temporal abstraction of this fragment views the sequence of new values accumulated in the sum as the input to Sum. (Lines marked with an asterisk in Figure 2-1 indicate that the upper cliché name is an operation that temporally abstracts the lower iterative cliché.) In Figure 2-1, Generate-Global-Buffers-and-Nodes is an example of a temporally abstract operation. It takes the initial global `MESSAGE` buffer and the initial collection of `SYNCH-NODE`s and creates a sequence of new global `MESSAGE` buffers and `SYNCH-NODE` collections. (This is a temporally abstract view of the iterative computation performed on each iteration of the simulation in which `MESSAGE`s are delivered and `SYNCH-NODE`s are stepped.)

A Common Event-Driven Simulation Design

This section describes a common event-driven simulator design for message-passing systems. It has data structures `ASYNCH-NODE` and `MESSAGE`, representing processing nodes and messages, respectively. It also has an `EVENT` data structure, which represents the arrival of a `MESSAGE` at an `ASYNCH-NODE`. Each `ASYNCH-NODE` data structure maintains its own local Clock. It also has a Memory part, holding its state. There is a sequence containing all `ASYNCH-NODE`s, called an `ADDRESS-MAP`, which maps each integer address to an `ASYNCH-NODE` (as in the synchronous simulation design). `MESSAGE`s typically have the same parts as those in the synchronous simulation design (Destination-Address, Type, Arguments, Storage-Requirements). An `EVENT` contains an Object, which is a `MESSAGE` to be handled, and a Time at which the work to be done on the object (i.e., handling a message) was scheduled (i.e., when the `MESSAGE` arrives

at an ASYNCH-NODE).

A global agenda, called the EVENT-QUEUE, keeps track of EVENTs that need to be processed. The agenda is implemented as a Priority Queue, in which the EVENT with the earliest Time has the highest priority.

The event-driven simulator is given an initial EVENT, whose Object is a starting MESSAGE and whose Time is the MESSAGE's arrival time. This is added to the EVENT-QUEUE. On each step of the simulation, the highest priority EVENT is pulled from the EVENT-QUEUE and processed. Processing an EVENT means simulating the handling of the MESSAGE in the EVENT's Object part. The simulated message handling is done in the context of the ASYNCH-NODE that represents the real node that is the destination of the message. This is looked up using the Destination-Address part of MESSAGE as an index into the sequence ADDRESS-MAP. (As we mentioned earlier, the portion of the simulator that simulates a processing node's message handling actions varies. Below, we describe an initial set of clichés that may be used. However, this portion of the simulator is not guaranteed to always be clichéd.)

When an EVENT is processed, the Clock of the destination ASYNCH-NODE for its MESSAGE Object is updated: the ASYNCH-NODE's Clock becomes the maximum of its current time and the arrival time of the MESSAGE (i.e., EVENT's Time). (The ASYNCH-NODE's current time can be later than the arrival time if the simulator is mimicking a real situation in which the real node was busy when the message arrived. The arrival time can be later than an ASYNCH-NODE's current time if in the real situation being simulated, the real node is idle when the message arrives.)

Handling a MESSAGE can cause other MESSAGEs to be sent. These are added to the EVENT-QUEUE. The event-driven simulation ends when the EVENT-QUEUE is empty.

An important characteristic of this algorithm is that the MESSAGEs are handled non-preemptively, which means that once an ASYNCH-NODE starts to handle a MESSAGE, it will not be interrupted, e.g., by receiving another MESSAGE.

Another property of the algorithm is that at each step, the globally earliest unprocessed MESSAGE received so far is chosen to be handled. Since the EVENT pulled from the EVENT-QUEUE is always the one with the earliest Time, and since Time is the arrival time of the MESSAGE in the EVENT's Object part, the MESSAGE chosen to be handled next is always the one with the earliest arrival time of the MESSAGEs that have not yet been handled.

These two properties ensure that once a MESSAGE is chosen for handling, no MESSAGEs will subsequently be generated that have an arrival time earlier than the MESSAGE chosen. In other words, MESSAGEs are handled in the order they arrive. So the simulator models the invariant obeyed by the real machine: messages to the same node are handled in the order in which they are received.

Figure 2-3 shows the structure of the portion of the cliché library that contains the event-driven simulation cliché and the clichés it is built upon. (For data clichés, refer to Figure 2-2.)

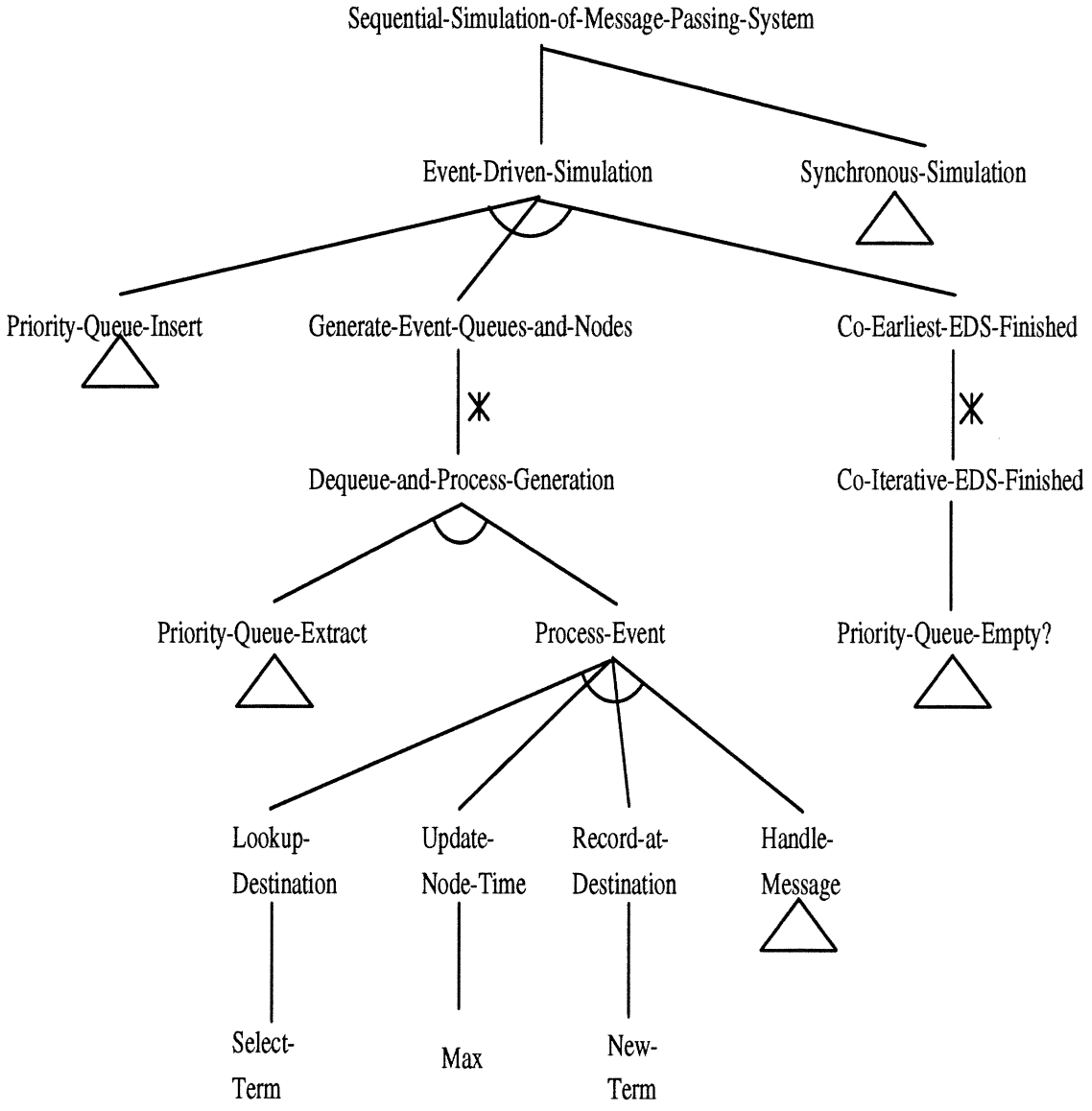


Figure 2-3: Event-driven simulation clichés.

Node Action Simulation Clichés

The two simulators for message-passing parallel systems contain a component that simulates some or all of the actions that a real processing node takes when handling a message. Which actions are simulated depends on the behavior of interest for the simulation. We have begun to collect some clichés that occur in simulators that model message handler lookup and execution on a node. These clichés are found in the broader domain of program execution in general, and the domain of program interpretation (or evaluation) in particular [1]. Figure 2-4 shows the structure of this portion of the library.

The clichés we have collected so far are those for the following.

- Looking up a handler based on a `MESSAGE`'s Type, which is typically an Associative-Set-Lookup or Property-List-Lookup, depending on how the handlers are stored.

- Loading the `MESSAGE`'s Arguments into the Memory part of an `ASYNCH-NODE` or `SYNCH-NODE` (depending on whether the simulator is event-driven or synchronous). This involves looking up the `ASYNCH-NODE` or `SYNCH-NODE` indexed by the `MESSAGE`'s Destination-Address, enumerating the Arguments, accumulating them in a sequence, and adding the sequence to the Memory part (typically an Associative Set).

- Executing the handler on the input data given in the Arguments. An `EXECUTION-CONTEXT` data structure is used to keep track of the Node executing the handler (which is an `ASYNCH-NODE` or `SYNCH-NODE`), the Status of the execution (a Symbol), Bindings of variable names to Memory locations (in an Associative Set), and the Instructions being executed (which is an Indexed Sequence: a data structure with two parts: a Base sequence of `INSTRUCTION`s and an integer Index which acts as an instruction pointer). An `INSTRUCTION` consists of an Operator (symbol), and a set of Arguments (typically in a list or an adjustable-length sequence), which may be other `INSTRUCTION`s.

  The handler execution involves iteratively fetching the next instruction to be executed using the current value of the instruction pointer. A standard Lisp `EVALUATE/APPLY` recursion is then used to interpret the `INSTRUCTION` with respect to the current values of the variable names stored in Memory. The Operator part of the `INSTRUCTION` is used to look up a Common Lisp function for simulating the actions of the processing node in applying that operator type to arguments. The `EVALUATE/APPLY` recursion "evaluates" an `INSTRUCTION` by iterating through its Arguments, recursively evaluating each one, and then applying the function associated with the `INSTRUCTION`'s Operator to the results.

We have made a first attempt at capturing the knowledge needed to recognize program execution clichés. Our experiences in encoding these clichés in the graph grammar helped us to understand both the strengths and weaknesses of the formalism for expressing certain types of programming ideas. This is discussed further in Chapter 5.

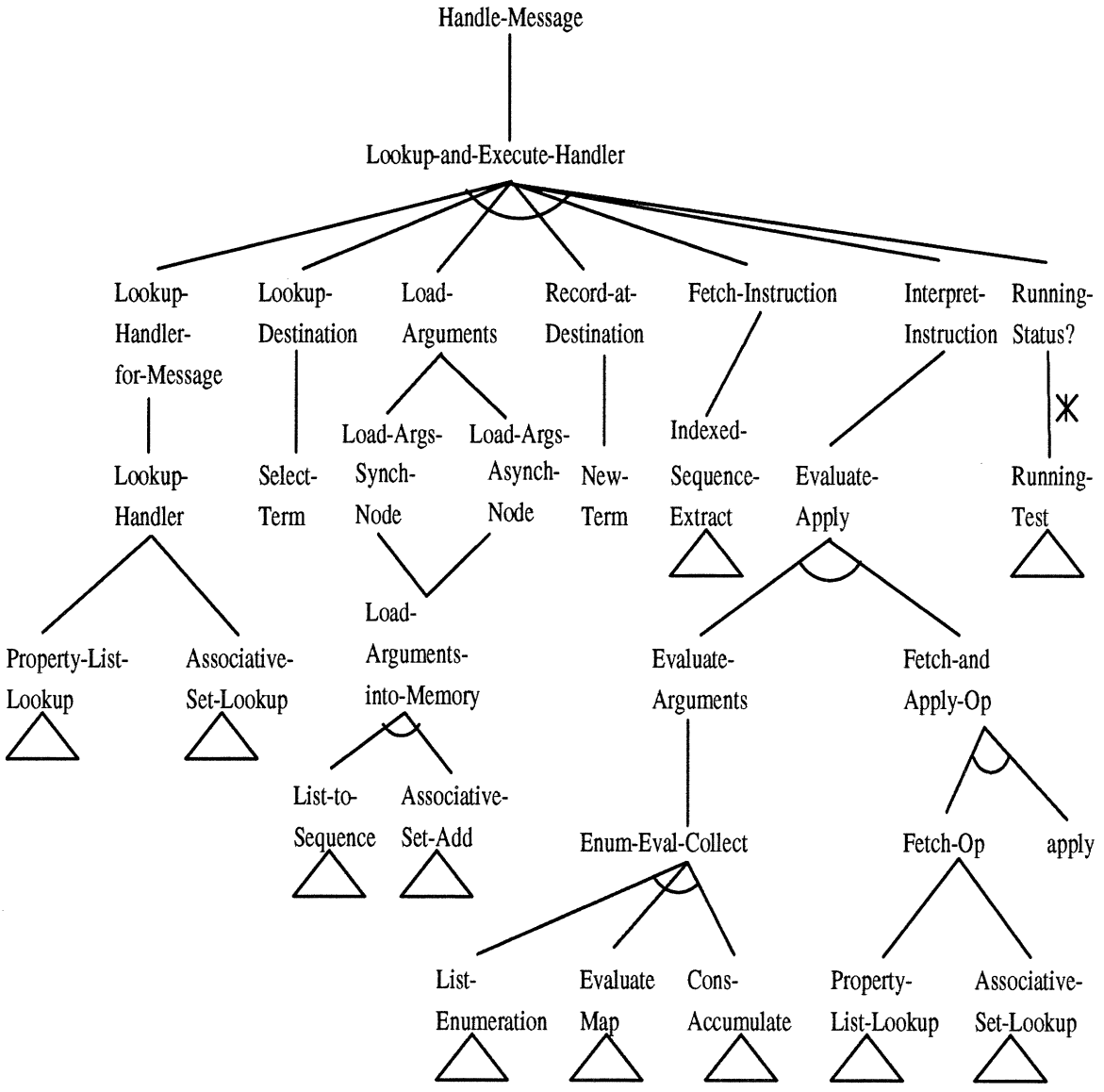


Figure 2-4: Node action simulation clichés.

2.1.4 The General-Purpose Clichés

Figure 2-5 gives an abstract picture of the relationships between the groups of general-purpose clichés that are contained in the library. Each box represents a set of algorithmic clichés that represent either operations on some aggregate data structure cliché (e.g., Priority-Queue) or basic iteration or computational clichés (e.g., Sum, Sequence-Enumeration, Absolute-Value). Each box contains the names of some of the clichés contained in the group it represents.

The arrows between the boxes indicate that the clichés in the source group use the clichés in the sink group as components, or the clichés in the source group are abstractions of those in the sink group. For example, the arrow from FIFO to Circular-Indexed-Sequence (CIS) indicates that clichéd operations on FIFOs can be implemented as clichéd operations on CISs. The arrow from CIS to Basic-Iteration-Clichés indicates that the operations of manipulating a CIS use basic iteration clichés as components (e.g., the operation of enumerating a CIS uses a Bounded-Count operation as a component, which generates a sequence of integers within some interval).

The cliché library does not contain all existing algorithmic clichés that operate on the data structures mentioned in Figure 2-5. We captured a fair number, but due to time limitations, we could not collect a complete set.

2.2 Real-World Programs

In studying program recognition, we focused on two programs which were written in Common Lisp by researchers in a parallel architecture group at MIT. The programs sequentially simulate the parallel execution of programs by a fine-grain message-passing parallel machine (which is described in [26]).

One program, called `PiSim`, simulates the parallel execution of programs in terms of the operations of a "parallel interface" (Pi) [146, 147]. (A parallel architecture interface separates parallel programming model issues from machine hardware issues, in a way analogous to the von Neumann interface for sequential computers. For more details, see [146].) It uses the event-driven algorithm and the program interpretation clichés that are in our library.

The other simulator simulates the parallel execution of programs written in a language called "Concurrent SmallTalk" [25]. We will refer to this simulator as `CST`. It uses the synchronous simulation design.

The `CST` simulator program is actually a module in a larger program which provides a programming environment for compiling, simulating, tracing, and gathering and displaying statistics on the execution of Concurrent SmallTalk code. Functions that call the simulator are not analyzed, neither are the metering, tracing, and plotting functions that it calls.

There are a few important points about the example simulators that are relevant to our study of recognition. One is that currently, `GRASPR` is unable to recognize clichés in programs

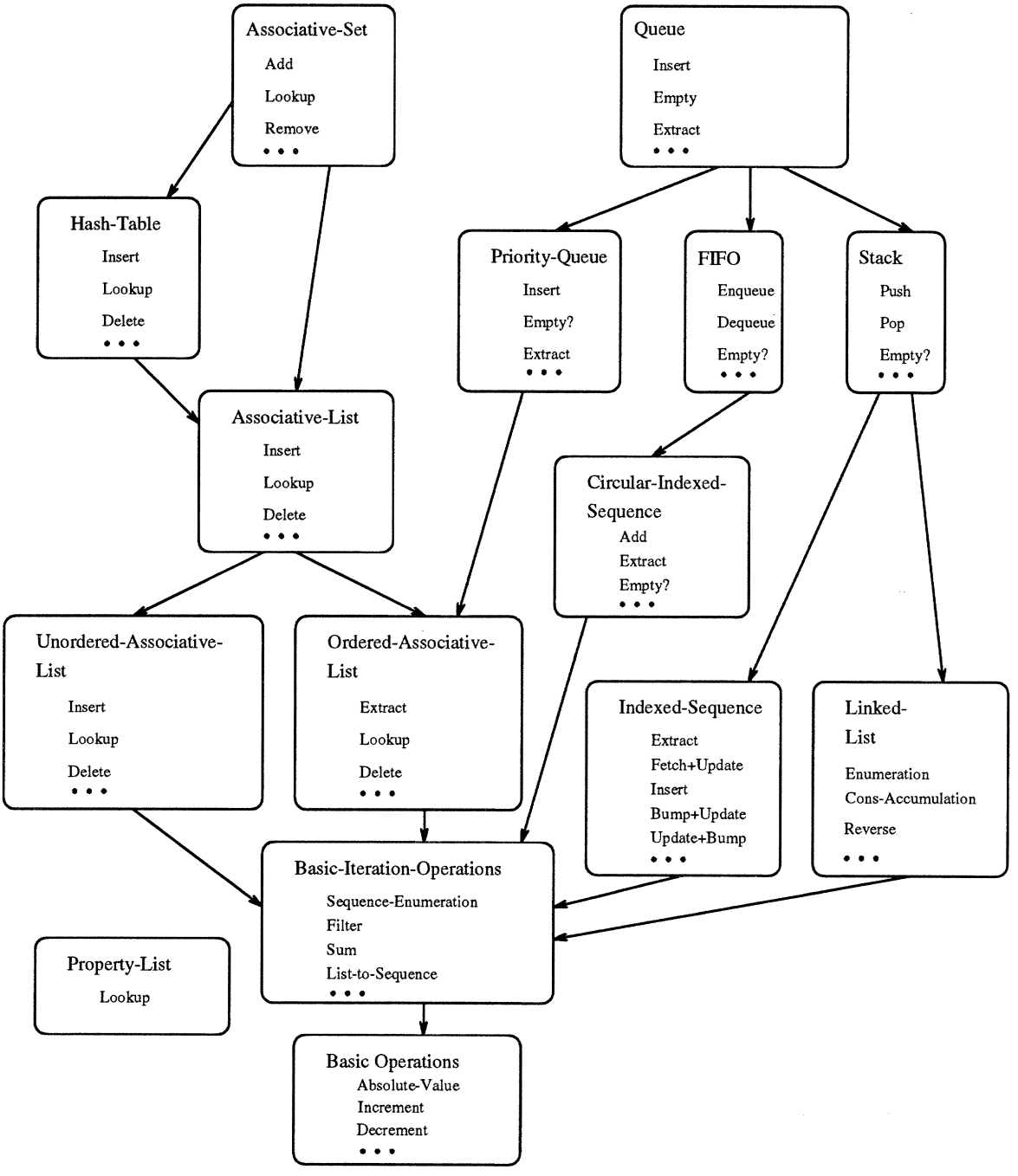


Figure 2-5: General-purpose clichés.

that contain operations that destructively modify mutable data structures. Our plan is to study the recognition of aggregate data structures, independent of issues concerning side effects to them, and then attempt to tackle the problems of mutable data structures later. So, we manually converted the example programs to programs that contain only non-destructive versions of the data structure operations. For example, we replaced destructive alterations to data structures with changes to *copies* of the data structures. We also propagated these changes to the data structures that pointed to the altered data structure, and so on. We essentially routed the dataflow by hand so that all aliasing was taken into account. (Section 7.2.4 gives more details. Appendix B contains the original versions of the two simulator programs, followed by their functional translations.)

In doing the translation, we found that many of the translation steps are automatable. For certain types of side effects, it may be possible to automatically uncover straightforward types of aliasing patterns and replace them with their non-destructive counterparts. The insights we gained should help us extend `GRASPR` in the future to deal with side effects to mutable objects, as discussed in Section 7.2.4.

All of the clichés in our current library are "pure" in that they include no destructive operations (such as `RPLACD`, `RPLACA`, or `SETF` in Common Lisp).

Another important point concerns how the programs simulate message handling. We mentioned earlier that we have only begun to encode the clichés found in code that is responsible for simulating a processing node's action of handling a message. We have experimented with recognizing these clichés in `PiSim`, which contains them. However, we would also like to explore the issues that arise when an integral part of an algorithmic cliché can be filled with unfamiliar, perhaps loosely constrained code. The `CST` program allows us to explore these difficulties because it contains code for simulating a node's action that is not clichéd (at least with respect to our current library of clichés). Details of these difficulties and suggestions for solving them are given in Sections 4.1.4 and 5.2.3.

Our final point is that even though `PiSim` contains clichéd node action simulation code, problems still arise in expressing and recognizing certain clichés. This is because part of the information about how to simulate a node's action is given as *input*, rather than being statically contained in the program. In particular, `PiSim` takes a set of message handlers as input. Each handler provides a set of instructions to be executed when handling a certain type of message. For example, Figure 2-6 gives a handler for a `Factorial` message, which iteratively computes the factorial of a single argument (`N`). (The `X` is a local variable.) The instructions in the handlers are written in a language of Machine Operations (e.g., `Times`, `Branch-Zero`). Each Machine Operation has a Common Lisp function associated with it that specifies how to simulate the actions of the processing node in executing that machine operation. They are defined in terms of simulator functions. For example, Figure 2-7 shows the functions that are associated with the operations `Times` and `Branch-Zero`.

Like the set of handlers, the definitions of Machine Operations are *inputs* to `PiSim`. This means they are not available for analysis or recognition. The problem that this poses is

```
(define-handler Factorial (N) (X)
  (print-user "~&running simple loop test~%")
  (write (self) X 1)
  Loop
  (branch-zero (read (self) N) Done)
  (write (self) X (times (read (self) X) (read (self) N)))
  (write (self) N (minus (read (self) N) 1))
  (branch-zero 0 Loop)
  Done
  (print-user "~&the answer is ~d~%" (read (self) X))
  (destroy-segment (self)))
```

Figure 2-6: A message handler for Factorial.

that the data and control flow of the entire PiSim program cannot be statically computed. It depends on the input for a particular simulation. The implication of this is that we do not have complete knowledge about who calls the simulator functions or how their inputs and outputs are connected. The problems we have encountered as a result are discussed in Section 5.2.

Choice of Programs: Breaking Out of the Toy Program Rut

In choosing programs to use in our study of recognition, our goal was to break out of the rut of automating the recognition of "toy" programs, in which most earlier recognition research has been caught. Both simulator programs (PiSim and CST) do this. Their sizes fall in the 500 to 1000 line range, rather than being on the order of tens of lines, which is the typical size of programs dealt with in previous recognition research.

Program length is only an approximate indicator of the potential difficulty of recognizing a program. In addition to choosing larger programs, we have chosen programs not written by us (the designers of the recognition system). The simulator programs are not contrived examples. They were written, without bias, to solve a particular real-world problem.

A key advantage of this is that it provides challenges to the recognition approach that might not be anticipated by us, as developers of it. Even though we may need to change or simplify the original program to allow recognition to occur, we are aware of the limitation of our approach that requires this. We also are aware of the type of transformation that should be made or the advice that should be given to help deal with the shortcoming. (Section 5.2 discusses the limitations observed and Section 5.2.5 summarizes changes made to the original programs to yield the programs that GRASPR recognizes.)

Additionally, the programs indicate which characteristics of programs are typical. This helps us in analyzing our recognition technique. For example, recognition by graph parsing can be expensive if there are excessive amounts of redundant computation, which causes

```
(Define-Operation Times (Active-Task X Y)
  (multiple-value-bind (New-Time Task-Node New-Task)
    (Increment-Time-Of Active-Task 1)
    (values (* X Y) New-Task)))

(Define-Operation Branch-Zero (Active-Task Test-Variable Label)
  (multiple-value-bind (New-Time Task-Node New-Task)
    (Increment-Time-Of Active-Task 1)
    (if (zerop Test-Variable)
        (values Label
                (Make-Task :Handler (Task-Handler New-Task)
                           :Node (Task-Node New-Task)
                           :Segment (Task-Segment New-Task)
                           :IP Label
                           :Status (Task-Status New-Task)))
        (values nil New-Task))))
```

Figure 2-7: The definition of two Machine Operations.

ambiguity. However, this characteristic is rare in the example simulator programs. Knowing which characteristics are typical or rare in real-world programs helps us determine which factors influence the practicality of our approach.

Another aspect of the simulator programs which distinguishes them from the "toy" programs studied previously is that they contain domain-specific clichés. These go beyond general-purpose clichés, such as operations on queues, stacks, and hash tables, which have been the focus of previous recognition research. The programs contain common simulation algorithms and data structures. By recognizing these clichés, GRASPR provides more useful program understanding capabilities than if it recognized the general-purpose clichés alone. This allows us to explore the expressiveness of the graph grammar formalism as a representation for domain-specific clichés. (On the other hand, the current cliché library has been acquired with the example programs in mind. More empirical studies are needed to evaluate the ability of the existing system to recognize new programs with the same library and to determine how much the library must change to recognize them.)

The simulator programs also contain a fair amount of unfamiliar code mixed in with clichéd computational structures. In experimenting with them, we test GRASPR's abilities to perform partial recognition, which is required in dealing with any realistic, non-trivial program.

2.3 Recognition Examples

Besides identifying the knowledge needed to understand and construct programs, it is important to capture this knowledge in such a way that it can be applied to a broad range of programs. In automating program recognition, our goal is to codify programming clichés at a level of abstraction that allows us to recognize them in programs that vary widely in such details as syntactic constructs used, programming language chosen, data structure and subroutine decomposition, and implementational choices. In addition, we provide recognition techniques that are robust under other types of variation, such as variation due to function-sharing optimizations and unfamiliar code.

This section gives examples of the recognition capabilities of `GRASPR`. This serves to demonstrate what `GRASPR` can do in terms of the classes of variation it can tolerate. It also provides motivating examples of the goals we have for our representational formalism and recognition technique.

2.3.1 Common Program Variations

Program recognition is difficult due to the wide range of possible variations among programs. An instance of a cliché may appear in a variety of forms. The following is a list of some of the common types of variation found in programs. (This does not provide a complete list of the variations we encountered in our empirical recognition studies with `PiSim` and `CST`. Chapter 5 discusses more variations, both those tolerated and not tolerated by our current system.)

- *Syntactic variation* in control and binding constructs. There are typically many ways to achieve the same net flow of data and control. Variable, function, data structure, and part names vary widely. Also, syntax varies over programming languages.

- *Implementation variation.* A given abstraction can often be implemented by a set of different concrete algorithms and data structures.

- *Delocalization.* Parts of a cliché are sometimes widely scattered throughout the text of a program, rather than being contiguous.

- *Unrecognizable code.* Not all programs are constructed completely of clichés. Recognition must be able to ignore an unpredictable amount of unrecognizable code.

- *Variation in the organization of components.* Programs can be decomposed into subroutines in a variety of ways. Also, data structures can aggregate pieces of data in a multitude of different nested organizations.

- *Redundancy.* Programs may vary in how much computation is repeated in the same instance of a cliché. For example, when the result of some inexpensive computation

is needed more than once, the program may simply recompute the value each time it is needed rather than caching the result in a temporary variable.

- *Optimizations.* A great deal of variation occurs between optimized and unoptimized programs even though they may contain the same abstract cliché. A common form of optimization introduces *function-sharing* in which the implementations of two or more distinct abstract structures are merged.

2.3.2 Examples of Capabilities

GRASPR is able to recognize both CST and PiSim as sequential simulators of message-passing parallel systems. It recognizes the synchronous simulation design in CST and the event-driven simulation design in PiSim. It also recognizes the message-passing program execution clichés in the portion of PiSim's code that simulates handling messages.

The primary output of GRASPR is a forest of *design trees.* A design tree indicates the clichés found in the program and how they are related to each other. Figure 2-8 shows a portion of the design tree produced in recognizing PiSim. Subtrees that are not shown are collapsed into small triangles below a cliché name. The dashed lines at the tree's fringe are links to primitive operations in the source code, which indicate the location of a particular cliché in the code. The drawing of the design tree is a simplified version of the actual description produced by GRASPR. The description is simplified (for presentation purposes) in that only operations are specified in the leaves of the tree, while the actual description includes information about the data involved in each cliché instance. In general, GRASPR may produce several design trees, representing recognition of multiple, perhaps overlapping, clichés in the code.

(The design trees are graph grammar derivation trees, which are described in Section 3.2.2. In general, they may be graphs in that a recognized cliché may be a component or implementation of two or more higher-level clichés.)

A secondary way to view the output of GRASPR is provided by a tool, called "Paraphraser," which takes the design trees produced during recognition and generates textual documentation based on them. Paraphraser knits together schematized textual fragments associated with the recognized clichés, filling in slots with identifiers taken from the source code (e.g., *EVENT-QUEUE*). It bases the structure of the text on the relationships between the clichés.

Figure 2-9 shows some of the documentation generated for the design tree shown in Figure 2-8. The documentation, although stilted, does describe the important design decisions in the program and can help a programmer locate relevant objects in the code (via the identifiers).

One potential benefit of automated program recognition is to use such automatically produced documentation to maintain poorly documented or undocumented programs. Automatically produced documentation can be updated whenever the source code changes,

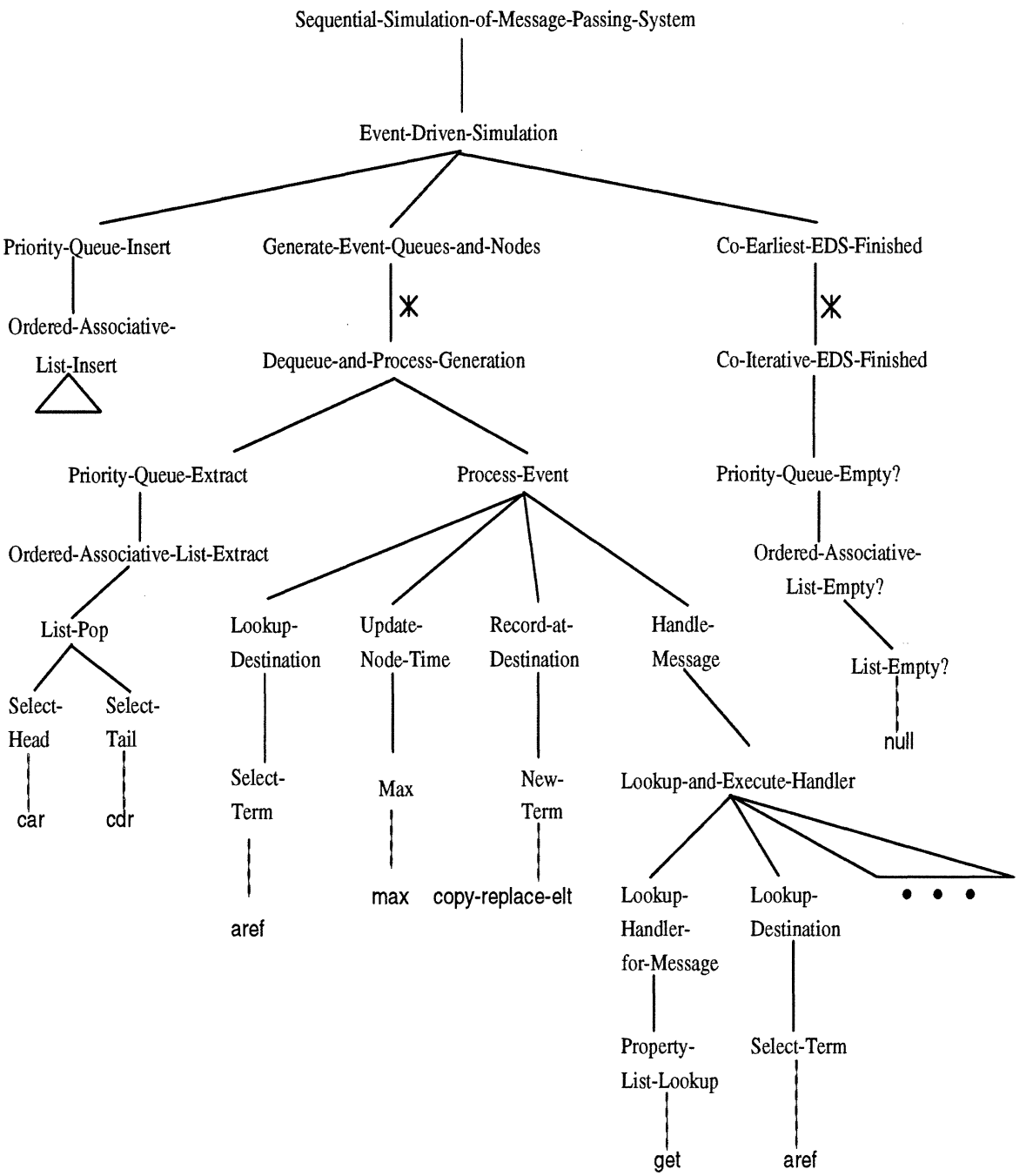


Figure 2-8: Design tree for `PiSim`.

```
PISIM sequentially simulates a parallel message-passing system.
It is implemented as an Event-Driven Simulation.
   1: Event-Driven Simulation asynchronously simulates a collection of
   processing nodes handling messages, using an event-driven algorithm.  An
   event-queue *EVENT-QUEUE* of events is maintained.  To start, an initial
   event EVENT is inserted in the event-queue.  On each step, an event is
   pulled off and processed, which may create new events to be added to the
   event-queue.  The asynchronous nodes (which represent processing nodes)
   are collected in an address-map, called *NODES*.
   Event-Driven Simulation is composed of a Priority-Queue Insert, a Co-Earliest
   Event-Driven Simulation Finished and a Generate Event Queues and Nodes.
      2: Priority-Queue Insert inserts EVENT in the priority queue
      *EVENT-QUEUE*.  An element's priority P is higher than another's Q,
      if P < Q.  If an element already exists in the priority queue with
      the same priority, then the new element is inserted into the queue
      after the existing element.
      Priority-Queue Insert is implemented as an Ordered Associative List Insert.
        3: Ordered Associative List Insert inserts EVENT in the
        ordered associative list *EVENT-QUEUE*...
      2: Co-Earliest Event-Driven Simulation Finished takes a sequence of
      event-queues and a sequence of address-maps and returns the address-map
      in the sequence of address-maps that corresponds to the first empty
      event-queue in the sequence of event-queues.
      Co-Earliest Event-Driven Simulation Finished temporally abstracts
      Co-Iterative Event-Driven Simulation Finished.
         3: Co-Iterative Event-Driven Simulation Finished terminates
         the simulation when the current event-queue (*EVENT-QUEUE*)
         is empty, returning the current value of the address-map (*NODES*).
         The event-queue is implemented as a Priority Queue.
         The Event-Driven Simulation Finished Test is implemented as a
         Priority Queue Empty.
            4: Priority Queue Empty tests whether the priority queue
            *EVENT-QUEUE* is empty....
      2: Generate Event Queues and Nodes generates event-queues and address-
      maps by repeatedly dequeuing the current event-queue and processing
      the event dequeued.  Processing an event causes new events to be added
      to the event-queue and a new address-map to be created.  The initial
      event-queue is *EVENT-QUEUE* and the initial address-map is *NODES*...
      Generate Event Queues and Nodes temporally abstracts Dequeue and
      Process Generation....
```

Figure 2-9: Some of the documentation generated for PiSim.

solving the pernicious problem of misleading, out-of-date documentation.

The current implementation of Paraphraser is heuristic and fragile. Documentation generation is not a primary focus of this research. The problem of applying recognition to program documentation needs further study, perhaps borrowing techniques from natural language generation.

Besides documentation, there are a variety of ways to present the results of recognition, depending on how the results will be used. Future work is needed to find the presentation appropriate for effective interaction with people and other automated tools.

Syntactic Variation

The design tree and documentation shown in Figures 2-8 and 2-9 were produced by `GRASPR` in recognizing `PiSim`. The top-level portion of `PiSim` is shown in Figure 2-10. (The source code for data structure definitions and some subroutines are not shown.) `Inject` is the top-level function which starts the `PiSim` simulator. It takes an initial start message type and the message's arguments. After some initialization, it creates a `Message` data structure, based on information about storage requirements computed from the `Handler` that is associated with the message type. It randomly generates a destination address for the message and computes the message's arrival time from the destination `Node`'s current time. Once the `Message` is created, an `Event` is constructed, whose Object part is the `Message` and whose Time is the arrival time. The `Event` is placed on the event-queue `*Event-Queue*` and `Execute-Events` is run to iteratively extract and execute the highest priority event on the event-queue.

Given a syntactic variation of this code, such as the code in Figure 2-11, `GRASPR` is able to recognize the *same* clichés to produce the same design tree and documentation (modulo identifiers). Recognition is robust under variations in variable names (`Length` versus `Memory-Needed`), binding and control constructs (`cond` versus `if`), and names of data structures and their parts (`Message` versus `Msg` and `Message-Destination` versus `Msg-Dest-Addr`). `Start-PiSim` also differs from `Inject` in the ordering of computations in the `let` binding clauses. It routes dataflow differently, using fewer local variables. It also passes the event queue around explicitly, rather than maintaining a global variable. Recognition robustness is achieved as a result of the representation shift performed by `GRASPR` which translates both programs into the same graphical representation. In this representation, syntactic details are suppressed.

Organization of Components

The representation used by `GRASPR` also suppresses details of how programs are decomposed into subroutines and how aggregate data structures are organized. For example, the code in Figure 2-12 differs from the original `PiSim` code shown in Figure 2-10 in structural organization. It bundles up the initialization and storage requirement computations into

```
(defvar *Event-Queue* nil "this is the global event-queue")
(defvar *Nodes* nil "this is the node array")
(defstruct Message
  (Destination nil)
  (Length 0)
  (Type nil)
  (Arguments nil))
(defstruct Event
  (Time 0)
  (Object nil))
(defun Inject (Type &rest Arguments)
  (Make-Nodes)
  (Clear-Nodes)
  (Clear-Event-Queue) ;; resets *Event-Queue* to NIL
  (let* ((Handler (Get-Handler Type))
         (Length (+ (Handler-Arity Handler)
                    (Handler-Number-Of-Locals Handler)
                    2))
         (Destination (random (Number-Of-Nodes)))
         (Arrival-Time (Node-Time (Translate-Node Destination)))
         (Message (Make-Message :Destination Destination
                                :Length Length
                                :Type Type
                                :Arguments Arguments))
         (Event (Make-Event :Time Arrival-Time
                            :Object Message)))
    (Enqueue-Event Event)
    (Execute-Events)))
(defun Enqueue-Event (New-Event)
  (if (or (null *Event-Queue*)
          (< (Event-Time New-Event)
             (Event-Time (first *Event-Queue*))))
      (setq *Event-Queue*
                (cons New-Event *Event-Queue*))
      (setq *Event-Queue*
            (Insert-Event New-Event *Event-Queue*))))
(defun Execute-Events ()
  (cond ((null *Event-Queue*)
         *Nodes*)
        (t (Execute-Next-Event)
           (Execute-Events))))
```

Figure 2-10: Top-level portion of PiSim code.

```
(defvar *P-Nodes* nil "collection of nodes")
(defstruct Msg
  (Dest-Addr nil)
  (Storage-Length 0)
  (Type nil)
  (Args nil))
(defstruct Event
  (Time 0)
  (Object nil))
(defun Start-PiSim (Start-Msg-Type Args)
  (Make-Nodes)
  (Clear-Nodes)
  (let* ((Address (random (Number-Of-Nodes)))
         (Msg-Handler (Get-Handler Start-Msg-Type))
         (Memory-Needed (+ (Handler-Arity Msg-Handler)
                           (Handler-Number-Of-Locals Msg-Handler)
                           2))
         (Pending-Events
           (Enqueue-Event
             (Make-Event :Time (Node-Time (Translate-Node Address))
                         :Object (Make-Msg :Dest-Addr Address
                                           :Storage-Length Memory-Needed
                                           :Type Start-Msg-Type
                                           :Args Args))
             nil)))
    (Execute-Events Pending-Events)))
(defun Enqueue-Event (New-Event Event-Queue)
  (if (or (null Event-Queue)
          (< (Event-Time New-Event)
             (Event-Time (first Event-Queue))))
      (setq Event-Queue
                (cons New-Event Event-Queue))
      (setq Event-Queue
            (Insert-Event New-Event Event-Queue)))
  Event-Queue)
(defun Execute-Events (Pending-Events)
  (if (null Pending-Events)
      *P-Nodes*
      (Execute-Events
        (Execute-Next-Event Pending-Events))))
```

Figure 2-11: A syntactic variation of the portion of PiSim shown in Figure 2-10.

```
(defvar *Message-Queue* nil "this is the global message queue")
(defvar *Nodes* nil "this is the node array")
(defstruct Msg
  (Destination nil)
  (Arrival-Time 0)
  (Data nil))
(defstruct Handler-Data
  (Type nil)
  (Length 0)
  (Arguments nil))
(defun Initialize-Simulator ()
  (Make-Nodes)
  (Clear-Nodes)
  (Clear-Message-Queue)) ;; resets *Message-Queue* to NIL
(defun Compute-Storage-Rqmts (Type)
  (let ((Handler (Get-Handler Type)))
    (+ (Handler-Arity Handler)
       (Handler-Number-Of-Locals Handler)
       2)))
(defun Inject (Type &rest Arguments)
  (Initialize-Simulator)
  (let* ((Length (Compute-Storage-Rqmts Type))
         (Destination (random (Number-Of-Nodes)))
         (Arrival-Time (Node-Time (Translate-Node Destination)))
         (Handler-Data (Make-Handler-Data :Type Type
                                          :Length Length
                                          :Arguments Arguments))
         (Message (Make-Msg :Destination Destination
                            :Arrival-Time Arrival-Time
                            :Data Handler-Data)))
    (Enqueue-Message Message)
    (Process-Messages)))
(defun Enqueue-Message (Message)
  (if (or (null *Message-Queue*)
          (< (Msg-Arrival-Time Message)
             (Msg-Arrival-Time (first *Message-Queue*))))
      (setq *Message-Queue*
            (cons Message *Message-Queue*))
      (setq *Message-Queue*
            (Insert-Message Message *Message-Queue*))))
(defun Process-Messages ()
  (cond ((null *Message-Queue*) *Nodes*)
        (t (Process-Next-Message)
           (Process-Messages))))
```

Figure 2-12: An organizational variation of the top-level portion of PiSim.

subroutines. It also aggregates data differently. The original code defines an **Event** data structure with two parts: an Object and a Time. The Object part is filled by a **Message** data structure, which has the parts Destination, Length, Type, and Arguments. Pending **Events** (containing **Messages** to be handled) are queued in an ***Event-Queue***.

In the variation of this code shown in Figure 2-12, there is no **Event** data structure. Instead **Msg** data structures are placed directly in an event-queue, called ***Message-Queue***. Each **Msg** contains all the data that is in a **Message** in the original code and additionally has an Arrival-Time part, which plays the role of the Time part of **Events** in the original code. Some of the data aggregated in **Msg** is aggregated further into a sub-structure, called **Handler-Data**. This structure contains the parts Length, Type, and Arguments found in **Message** originally and it is nested inside the **Msg** data structure, under the Data part.

Despite these differences, **GRASPR** recognizes the same clichés in this code as in the original code in Figure 2-10.

It is important that recognition be robust under organizational variations because the clichés in the current library are themselves organized hierarchically. It is crucial that the program need not mirror this same organization for the clichés to be recognized in it.

This is because the library organization is not necessarily based on the typical way these clichés are organized in programs. There are two reasons it is not. One is that there is not always exactly one "typical" or common decomposition of clichés into subroutines or nesting of aggregate data structures. The second is that it may be better to base the library's organization on other criteria besides what is typical. For example, the organization might be chosen to emphasize salient parts of clichés to facilitate recognition performance improvements or to help choose the best partial analysis during near-miss recognition.

On the other hand, information about typical decompositions may provide valuable expectations about the location of clichés in a program. This can considerably narrow down the search for clichés, as discussed in Section 6.4.1.

Our representation does not eliminate information about the boundaries of subroutines and user-defined data structures within the program. It merely suppresses it, so that the organizational variation does not hinder recognition. It places this information in annotations on the graphical representation of the program. So, although in general we do not require that a program's function and data structure organization match the organization of the clichés in our library, it is possible to impose constraints on the clichés being recognized, requiring that they occur within certain boundaries. These boundaries can be heuristically defined based on information, such as subroutine or data structure decomposition. (See Section 6.4.1 for more details.)

Delocalized Clichés and Unfamiliar Code

Programs are rarely constructed entirely of clichés. Non-trivial programs are usually a mix of clichéd computational structures and unfamiliar code. In addition, the clichés are

```
(defun cst-start (init-msg)
  (send-msg init-msg)
  (shell-go))
(defun send-msg (msg)
  (setq *step-queue*
        (enqueue *step-queue* msg)))
(defun shell-go ()
  (cond ((step-done) nil)
        (t (step-nodes)
           (shell-go))))
(defun step-nodes ()
  (when *profile* (profile-step))                          ;; ?
  (when *log* (log-step))                                  ;; ?
  (when *trace*                                            ;; ?
    (record-traced-selectors *trace-selectors*))           ;; ?
  (deliver-msgs)
  (when *meter-message-queues*                             ;; ?
    (record-message-queue-data))                           ;; ?
  (iteratively-step-nodes 0)
  (setq *step-nr* (1+ *step-nr*))))
(defun iteratively-step-nodes (x)
  (if (>= x (array-total-size *nodes*))
      nil
      (step-node x)
      (iteratively-step-nodes (1+ x))))
(defun step-node (node-nr)
  (let* ((node (get-node node-nr))
         (q (node-queue node)))
    (if (queue-empty? q)
        nil
        (multiple-value-bind (msg new-queue)
            (dequeue q)
          (setq node
                (make-node :queue new-queue
                           :objects (node-objects node)              ;; ?
                           :contexts (node-contexts node)
                           :busy-count (1+ (node-busy-count node))   ;; ?
                           :method-cache (node-method-cache node)))) ;; ?
          (setq *nodes* (copy-replace-elt node node-nr *nodes*))
          (multiple-value-bind (new-nodes new-step-queue)
              (process-msg msg *nodes* *step-queue*)
            (setq *nodes* new-nodes
                  *step-queue* new-step-queue))))))
```

Figure 2-13: Top-level portion of CST. Question marks indicate unfamiliar code.

often interleaved with unfamiliar computation as well as with each other. This means that parts of a cliché may be scattered throughout the text of a program. Both of these factors make recognition difficult not only to automate, but also for people to do correctly.

GRASPR is able to ignore unfamiliar code to *partially recognize* the program. It also addresses the difficulty of recognizing delocalized clichés by employing a program representation shift from source text to flow graph. Cliché parts that are separated by unrelated expressions in the text become neighboring nodes in a flow graph.

For example, Figure 2-13 shows the top-level portion of the CST program, which uses the synchronous simulation design. (The source code for data structure definitions and some subroutines are not shown.) In addition to the simulation algorithm and data structures, this code contains calls to functions that perform various metering, logging, and statistics-gathering operations. These operations are not clichéd, at least with respect to our current library. The figure indicates unfamiliar portions of the code with question marks. The clichés in the program are not found in one contiguous section of program text, but are interrupted with unrelated computations.

Not only are there unfamiliar computations interleaved with the *algorithmic* clichés, but there are also parts of *data structures* that are not recognizable as part of any data cliché. For example, the data structure node consists of a Queue part (which acts as the local FIFO buffer in the SYNCH-NODE data cliché) and a Contexts part (which contains a data structure that has a part corresponding to the Memory part of the SYNCH-NODE). The rest of the parts of node (Objects, Busy-Count, and Method-Cache) are novel, specific to this program. They are used for gathering statistics and simulating the action of handling a message.

Despite the delocalization of the clichés and the unfamiliar code, GRASPR is able to recognize clichéd parts of this program. The design tree and documentation produced are shown in Figures 2-14 and 2-15 (in abbreviated form).

Implementation Variation

Often, there is more than one clichéd implementation of an abstract operation or data type. This can introduce variability between programs that on a high level of abstraction perform the same abstract operation or use the same abstract data types. It is important that GRASPR be able to recognize the same abstract clichés in these variations.

For example, the CST program uses a FIFO queue to implement the queue of messages collected on each cycle of the synchronous simulation and then delivered on the next. The FIFO queue is implemented as a Circular Indexed Sequence, as shown in Figure 2-16. However, another possible implementation of the queue is a LIFO queue (or stack), as shown in Figure 2-17.

GRASPR produces the design-tree shown in Figure 2-18 for the code that uses this implementation. It differs from the tree in Figure 2-14 only in the subtrees that are highlighted by dotted boxes in the figure. The rest of the tree, including the high-level description of

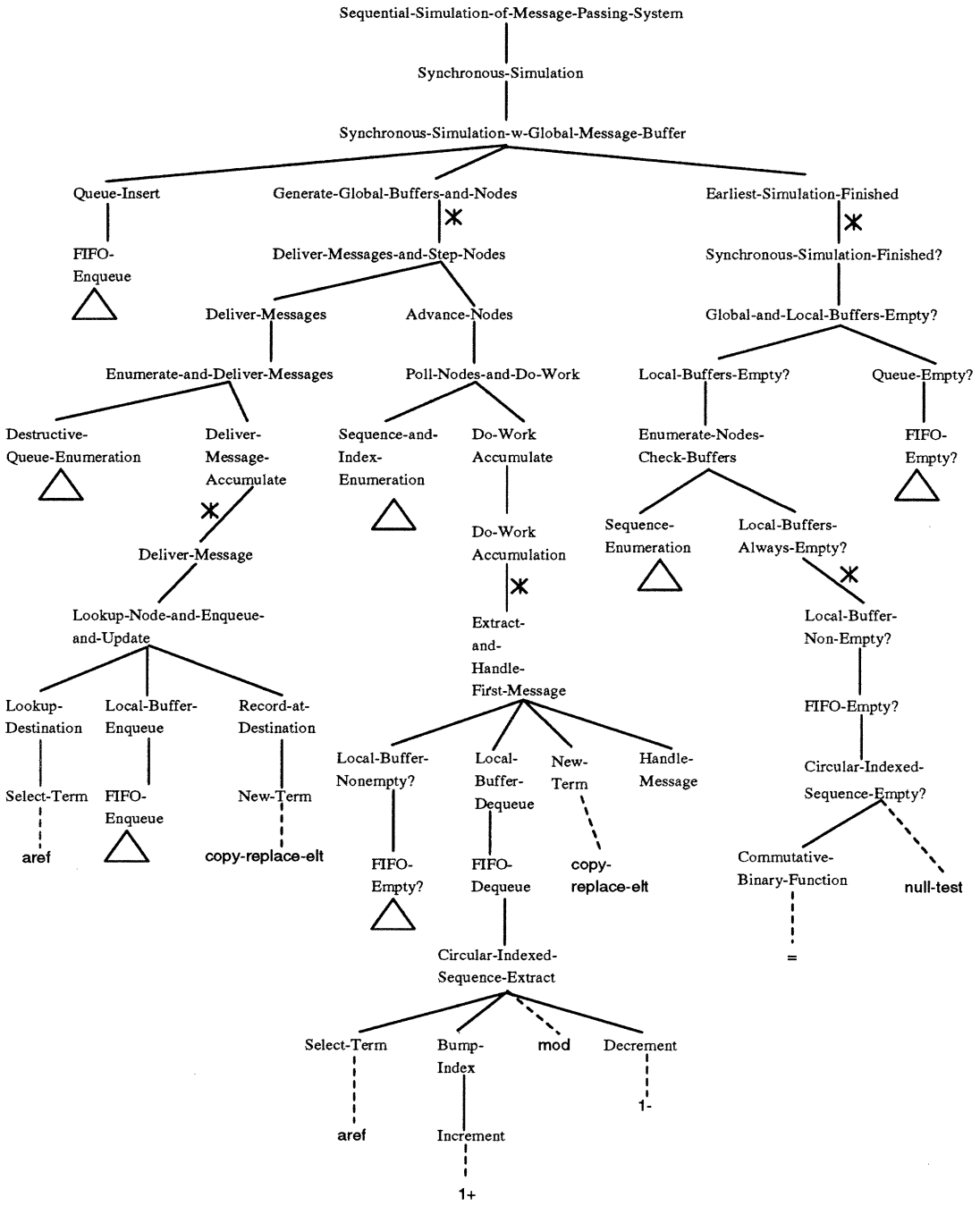


Figure 2-14: A portion of design tree produced in recognizing CST.

```
CST sequentially simulates a parallel message-passing system.
It is implemented as a Synchronous Simulation.
  1: Synchronous Simulation synchronously simulates a collection of processing
  nodes handling messages.  The synchronous nodes (which represent the
  processing nodes) are collected in an address-map, called *NODES*.  Each
  node maintains a local buffer of pending messages to handle.  Synchronous
  Simulation is implemented as a Synchronous Simulation using Global
  Message Buffer.
    2: Synchronous Simulation using Global Message Buffer iteratively advances
    each synchronous node in *NODES* by handling one message a piece.  It uses
    a global message buffer to ensure that nodes advance in lock-step.  The
    global buffer's initial value is *STEP-QUEUE*.  The simulation starts by
    adding an initial message INIT-MSG to *STEP-QUEUE*.  The simulation ends
    when no node has work to do (i.e., no more messages to handle) and the
    global message buffer *STEP-QUEUE* is empty.  As messages are handled, new
    messages are created which are buffered on the global message buffer.
    Synchronous Simulation using Global Message Buffer is composed
    of a Queue Insert, an Earliest Simulation Finished and a Generate
    Global Message Buffers and Nodes.
      3: Queue Insert enqueues INIT-MSG on the Queue *STEP-QUEUE*, which is
      implemented as a FIFO.  Queue Insert is implemented as a FIFO Enqueue.
        4: FIFO Enqueue enqueues INIT-MSG on the FIFO queue *STEP-QUEUE*,
        which is implemented as a Circular Indexed Sequence....
      3: Earliest Simulation Finished takes two input sequences: a sequence
      of address-maps, starting with *NODES*, and a sequence of global
      message buffers, starting with *STEP-QUEUE*.  It outputs the first
      address-map in the input sequence of address-maps that satisfies the
      predicate that all nodes in the address-map have empty local buffers
      and the corresponding global message buffer is empty.
      Earliest Simulation Finished temporally abstracts Synchronous
      Simulation Finished?.
        4: Iterative Synchronous Simulation Finished tests whether a
        synchronous simulation is finished by testing whether the
        global buffer and all of the nodes' local buffers are empty....
      3: Generate Global Message Buffers and Nodes generates address-maps
      and global message buffers by repeatedly delivering all
      messages in the global message buffer *STEP-QUEUE* and
      advancing the synchronous nodes in *NODES* by one step each....
```

Figure 2-15: A portion of the documentation generated for CST.

the program as a sequential simulation, remains the same.

It is impractical to enumerate all possible implementational variations of an abstract cliché in the cliché library. The hierarchical organization of the cliché library allows implementation variation to be represented compactly.

Function-Sharing

Programs can vary widely, depending on which optimizations they make. A type of optimization that occurs frequently in programs is one in which two abstract clichés share some functional part. In this case, the implementations of the clichés overlap. `GRASPR` is able to recognize the two clichés in a program whether or not their implementations overlap.

For example, one of the things the `CST` program does in gathering statistics is that it iterates through the `nodes` and computes the average length of their FIFO queues before it delivers messages on each clock cycle. Suppose we added the cliché to our library that performs this operation: it polls the `SYNCH-NODE`s, keeps a running total of their local buffer sizes, and divides the sum by the number of `SYNCH-NODE`s.

This cliché is found in the current `CST` code in the function `avg-queue-length`, which is called by `profile-step` in `step-nodes`, as shown in Figure 2-19. The recognition of this cliché results in the design tree shown in Figure 2-20. (This tree is generated by `GRASPR`, in addition to the design tree shown in Figure 2-14.)

Figure 2-21 shows a variation of the `CST` code in which the function-sharing optimization has been introduced. In this code, the average queue length computation has been moved into the iteration in `iteratively-step-nodes` that polls `nodes` and advances each one in lock step. This function is already iterating through the `nodes`. So, in addition to stepping each one, it has been made to keep a running total of their local queue lengths. Its caller, `step-nodes`, finishes off the averaging computation. This optimization increases the program's efficiency by enumerating the `nodes` only once.

`GRASPR` is able to recognize both the queue averaging cliché and the advance nodes cliché in this optimized program, even though the implementations of the clichés overlap. The resulting design trees share a sub-tree, as shown in Figure 2-22.

Redundancy

Sometimes a part of a cliché might appear more than once in the same instance of a cliché. The repeated part is most often some inexpensive computation whose result is needed more than once. The program may simply repeat this computation, rather than caching the result in a temporary variable. An example of this occurs in the function `Splice-in-Bucket` shown in Figure 2-23, which is used by a hash table insertion function contained in `PiSim`. `Splice-in-Bucket` creates and inserts an entry into a hash table bucket, called `Bucket-List`, which is an ordered associative list. It does this by "`cdr`'ing" down the `Bucket-List`, looking for a place to insert the new entry so that the entries remain ordered with respect to their

```
(defun cst-start (init-msg)
  (send-msg init-msg)
  (shell-go))
(defun deliver-msgs ()
  (cond ((queue-empty? *step-queue*) nil)
        (t (multiple-value-bind (msg new-step-queue)
               (dequeue *step-queue*)
             (setq *step-queue* new-step-queue)
             ...)
           (deliver-msgs))))
(defstruct queue
  (head 0)
  (tail 0)
  (length 0)
  (data-size *default-queue-size*)
  (data (make-array *default-queue-size* :adjustable t)))
(defun queue-empty? (queue)
  (= (queue-length queue) 0))
(defun enqueue (queue obj)
  (let* ((length (queue-length queue))
         (old-size (queue-data-size queue))
         (big-enough-queue (if (< length (1- old-size))
                               queue
                               (grow-queue queue))))
    (enqueue-base big-enough-queue obj)))
(defun enqueue-base (queue obj)
  (let ((old-size (queue-data-size queue)))
    (make-queue :head (queue-head queue)
                :tail (mod (1+ (queue-tail queue)) old-size)
                :length (1+ (queue-length queue))
                :data-size (queue-data-size queue)
                :data (copy-replace-elt obj
                                        (queue-tail queue)
                                        (queue-data queue)))))
(defun dequeue (queue)
  (let ((elt (aref (queue-data queue) (queue-head queue))))
    (setq queue (make-queue :head (mod (1+ (queue-head queue))
                                       (queue-data-size queue))
                            :tail (queue-tail queue)
                            :length (1- (queue-length queue))
                            :data-size (queue-data-size queue)
                            :data (queue-data queue)))
    (values elt queue)))
```

Figure 2-16: Buffer queue implemented as a FIFO, which in turn is implemented as a CIS.

```
(defun queue-empty? (queue)
  (null queue))
(defun enqueue (queue obj)
  (cons obj queue))
(defun dequeue (queue)
  (values (car queue)
          (cdr queue)))
```

Figure 2-17: Buffer queue implemented as a stack (LIFO).

Key parts. If an entry exists with the same Key as the new entry (`Key`), then the existing entry's Value part is changed to the new `Value`. `Number-Entries` keeps track of the number of entries in the hash table. It is incremented only if the new entry is inserted, not if an existing entry is changed.

This function repeats the computation of accessing the first element of `Bucket-List`, using `car`, as indicated in the figure by asterisks. However, the cliché for Ordered-Associative-List-Insert contains only one part corresponding to these expressions. It matches more closely the program shown in Figure 2-24. `GRASPR` is able to recognize Ordered-Associative-List-Insert in both variations.

2.4 Breadth of Coverage

The clichés captured in our library cover a broad range of programs. The domain-specific clichés occur in programs in the domain of sequential simulation of message-passing parallel systems, while our general-purpose utility clichés are found in programs across all domains.

However, the library's coverage is not absolute. Our "example-driven" cliché acquisition was based on an extremely small sample set of programs in a particular domain. We make no claims of fully modeling the simulation domain or even the subset of it that deals with message-passing systems. Also, our library does not contain all utility clichés used by experienced software engineers.

Despite these limitations, our library demonstrates the kinds of algorithms and data structures that can be expressed within a graph grammar formalism. This formalism captures these clichés at a level of abstraction that enables recognition by graph parsing to be robust under many common types of program variations.

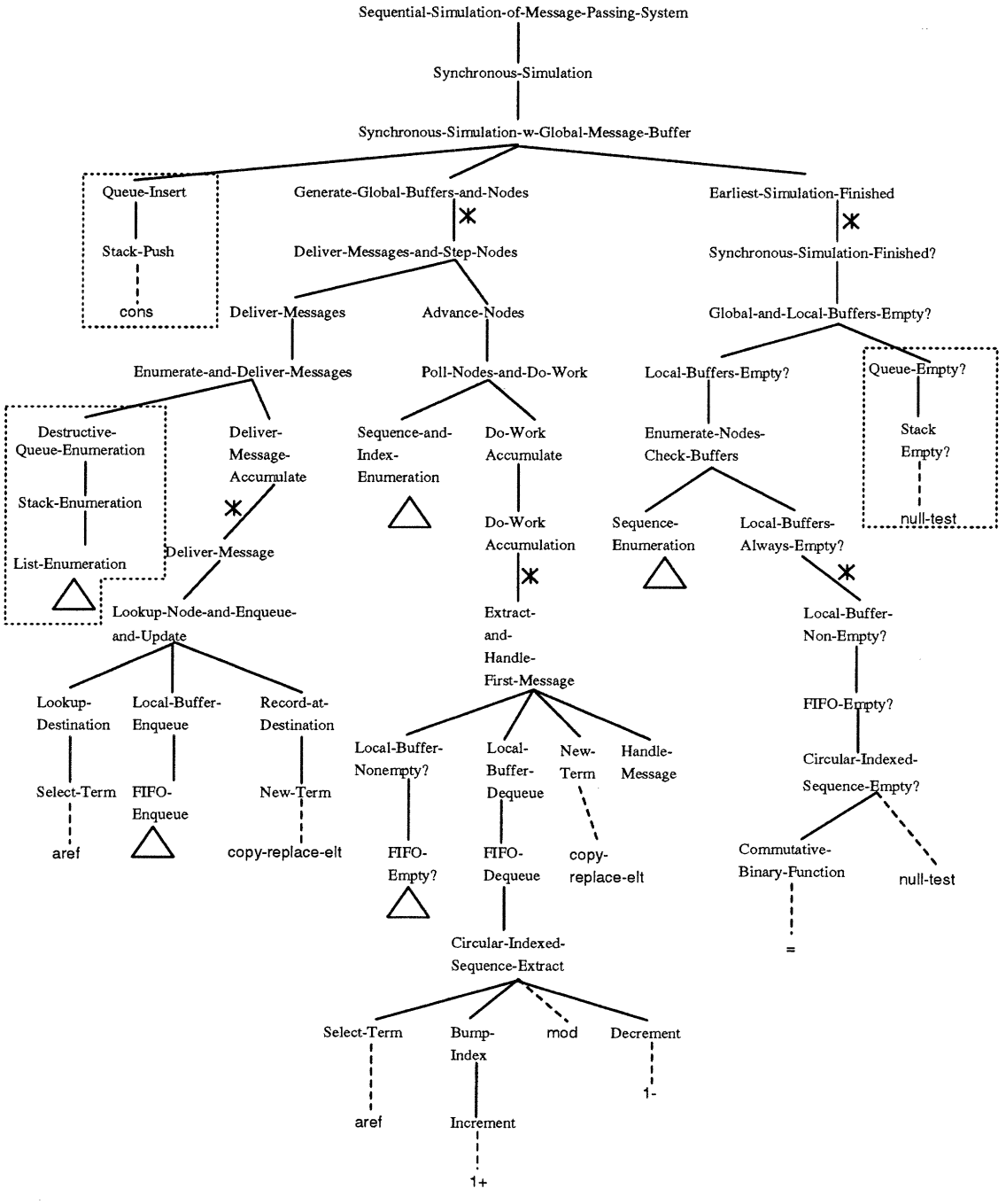


Figure 2-18: Design tree for implementational variation in which the buffer is a stack.

```
(defun step-nodes ()
  (when *profile* (profile-step))
  ...
  (iteratively-step-nodes 0)
  ...)
(defun profile-step ()
  ...
    (avg-queue-length)
  ...)
(defun avg-queue-length ()
  (let ((tql 0))
    (setq tql (sum-queue-lengths 0 tql))
    (/ tql (array-total-size *nodes*))))
(defun sum-queue-lengths (x tql)
  (if (>= x (array-total-size *nodes*))
      tql
      (sum-queue-lengths
        (1+ x)
        (+ tql (queue-length (node-queue (get-node x)))))))
(defun iteratively-step-nodes (x)
  (if (>= x (array-total-size *nodes*))
      nil
      (step-node x)
      (iteratively-step-nodes (1+ x))))
```

Figure 2-19: Portion of CST that averages node queue lengths.

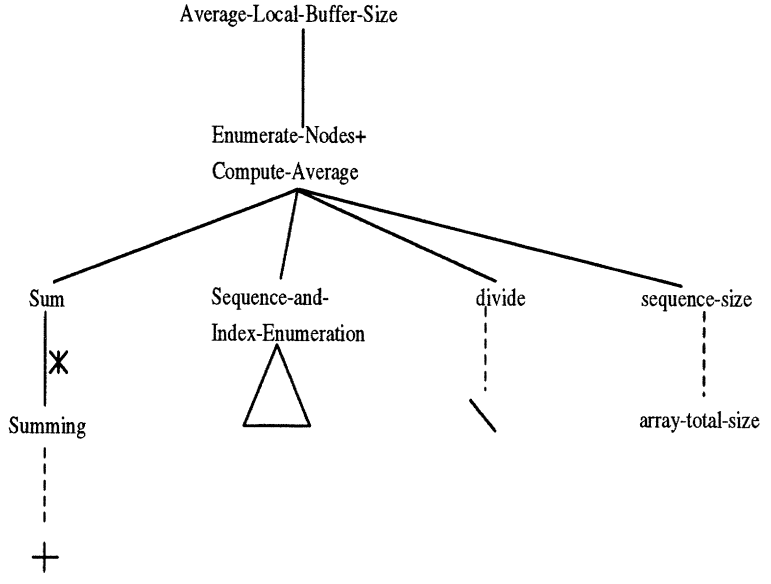


Figure 2-20: Design tree for queue length averaging computation.

```
(defun step-nodes ()
  (when *profile* (profile-step))
  ...
  (iteratively-step-nodes 0 0)
  ... (/ *total-queue-length*
         (array-total-size *nodes*)) ...
  ...)
(defun iteratively-step-nodes (x tql)
  (cond ((>= x (array-total-size *nodes*))
         (setq *total-queue-length* tql)
         nil)
        (t (step-node x)
           (iteratively-step-nodes
             (1+ x)
             (+ tql (queue-length (node-queue (get-node x))))))))
```

Figure 2-21: Optimization in which averaging is performed while advancing nodes.

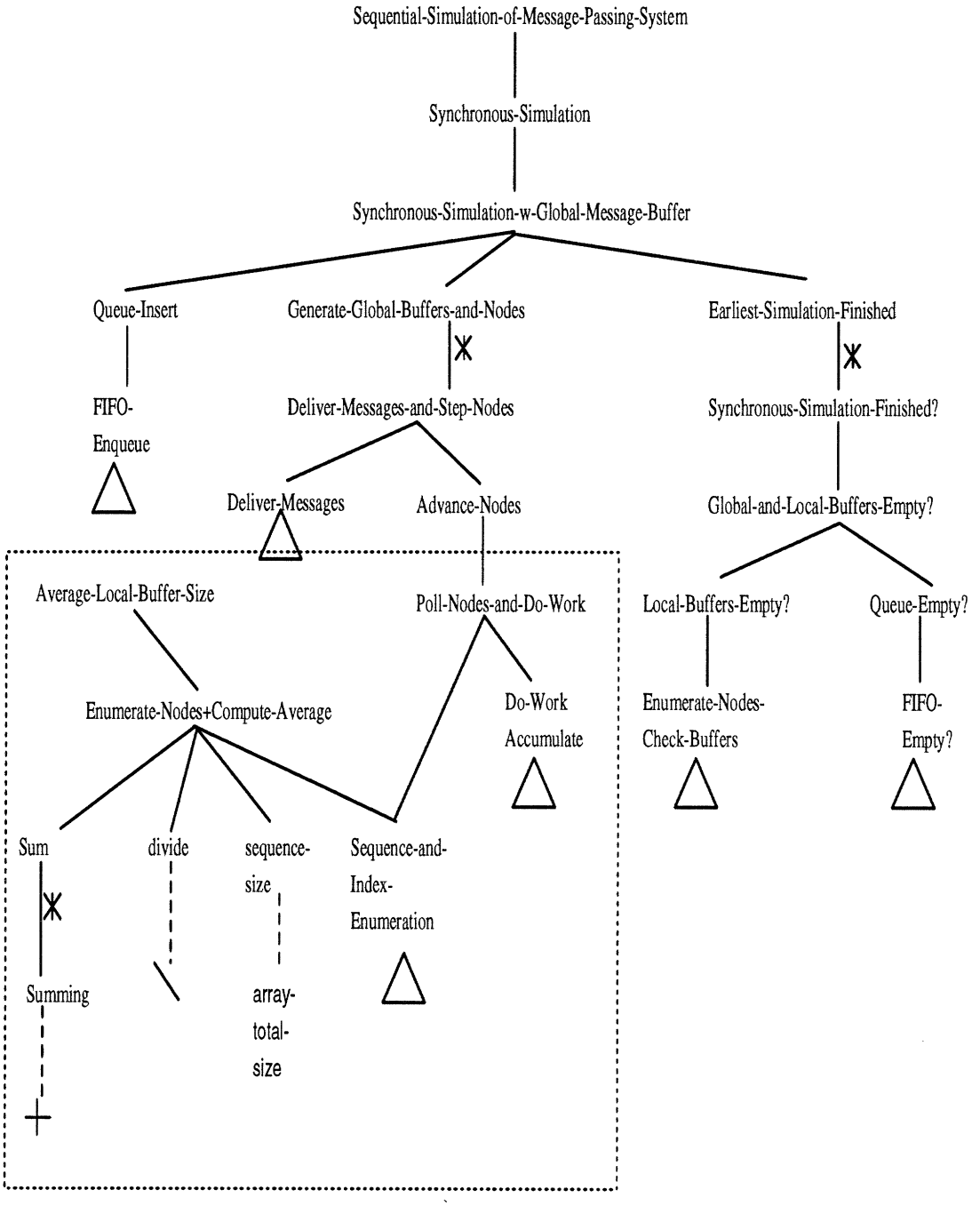


Figure 2-22: Design tree for optimized code, with shared sub-tree.

```
(defun Splice-In-Bucket (Value Key Bucket-List Number-Entries)
  (cond ((Empty-or-Low-Priority-Head? Key Bucket-List)
         (values (cons (Make-Entry :Key Key :Value Value)
                       Bucket-List)
                 (1+ Number-Entries)))
        ((string= Key
                  (Entry-Key (car Bucket-List)))          ;; *
         (values (cons (Make-Entry :Key Key :Value Value)
                       (cdr Bucket-List))
           Number-Entries))
        (t (multiple-value-bind (New-Bucket-List Num-Entries)
               (Splice-In-Bucket Value
                                 Key
                                 (cdr Bucket-List)
                                 Number-Entries)
             (values (cons (car Bucket-List)              ;; *
                           New-Bucket-List)
                     Num-Entries)))))
```

Figure 2-23: Code containing a redundant CAR computation.

```
(defun Splice-In-Bucket (Value Key Bucket-List Number-Entries)
  (cond ((Empty-or-Low-Priority-Head? Key Bucket-List)
         (values (cons (Make-Entry :Key Key :Value Value)
                       Bucket-List)
                 (1+ Number-Entries)))
        (t (let ((This-Entry (car Bucket-List)))          ;; *
             (cond ((string= Key
                             (Entry-Key This-Entry))      ;; *
                    (values
                      (cons (Make-Entry :Key Key :Value Value)
                            (cdr Bucket-List))
                      Number-Entries))
                   (t (multiple-value-bind (New-Bucket-List Num-Entries)
                          (Splice-In-Bucket Value
                                            Key
                                            (cdr Bucket-List)
                                            Number-Entries)
                        (values
                          (cons This-Entry New-Bucket-List)  ;; *
                          Num-Entries)))))))))
```

Figure 2-24: Code in which the result of CAR is cached and reused.

Chapter 3

The Flow Graph Formalism

GRASPR is able to tolerate many of the common types of program variations mentioned in Section 2.3.1 by using a dataflow graph representation for programs and by using a flow graph grammar to encode programming clichés. Program recognition is achieved by parsing the dataflow graph in accordance with the flow graph grammar. There are several advantages to using a graph grammar formalism to represent programs and clichés:

- Quasi-canonical form. Dataflow graphs abstract away irrelevant syntactic details and give the representation programming-language independence.
- Localization. Dataflow graphs make dataflow dependencies explicit, imposing a partial ordering on the program's operations (rather than the linear, total ordering imposed by text). The effect is that patterns that are textually delocalized (noncontiguous) can often become localized in a flow graph where only essential dataflow relationships are captured.
- Compact representation. Only primitive operations and dataflow between them are represented by the graph.
- Fragmentary patterns can be represented without including unnecessary details.
- Hierarchical relationships can be drawn between graphs, with the graph grammar formalism providing a firm mathematical basis.

In this chapter, we define the flow graph grammar formalism used to represent programs and clichés. We present the basic formalism first and then describe extensions to it that allow us to deal with variations due to redundancy versus structure-sharing, and variations in aggregation organization. We then present a chart parser for flow graphs in this formalism. Interleaved with the description of the formalism are sections that ground the description in the concrete application of program recognition. These may help clarify and motivate the restrictions on flow graphs and graph grammar rules. These sections are unnecessary for understanding the general description of the formalism, which has a broad range of

applicability to other problem domains besides program recognition (as discussed in Section 7.4). In the final section, we summarize related graph grammar research.

3.1 Flow Graphs

A *flow graph* is an attributed, directed, acyclic graph, whose nodes have *ports* – entry and exit points for edges. Flow graphs have the following properties and restrictions:

1. Each node has a *type* which is taken from a vocabulary of node types.

2. Each node has two disjoint tuples of ports, called its *inputs* and *outputs*. Each port has a *type*, taken from a vocabulary of port types. All nodes of the same type have the same number and type of ports in their input and output port tuples. The size of the input port tuple of a node is called the *input arity* of the node, while its *output arity* is the size of the node's output port tuple.

3. A node's inputs (or outputs) may be empty, in which case the node is called a *source* (or *sink*, respectively).

4. Edges do not merely adjoin nodes, but rather edges adjoin ports on nodes. All edges run from an output port on one node to an input port on another node. The ports connected by an edge must have the same port type.[1] (An exception to this is that a port of the special designated type `Any` can connect to ports of any type.)

5. More than one edge may adjoin the same port. Edges entering the same input port are called *fan-in edges*, while edges leaving a common output port are called *fan-out edges*.

6. Ports need not have edges adjoining them. Any input (or output) port in a flow graph that does not have an edge running into (or out of) it is called an *input* (or *output*) of that graph.

7. Each flow graph has a vocabulary of attributes, which is partitioned into two disjoint sets of node attributes and edge attributes. Each attribute has a (possibly infinite) set of possible values. Associated with each node type is a finite subset of the node attributes. These are the only attributes for which nodes of that type can hold values. All edges hold a value for each of the edge attributes.

Flow graphs were first defined by Brotsky [15], drawing upon the earlier work on *web grammars* [27, 94, 102, 105, 119]. Wills [144, 145] extended Brotsky's definition so that flow graphs can include sinks and sources (item 3 above), fan-in and fan-out edges (item 5), and attributes (item 7).

[1] In the future, a type hierarchy system may be used to allow ports to be connected if one port's type is a subtype of the other's.

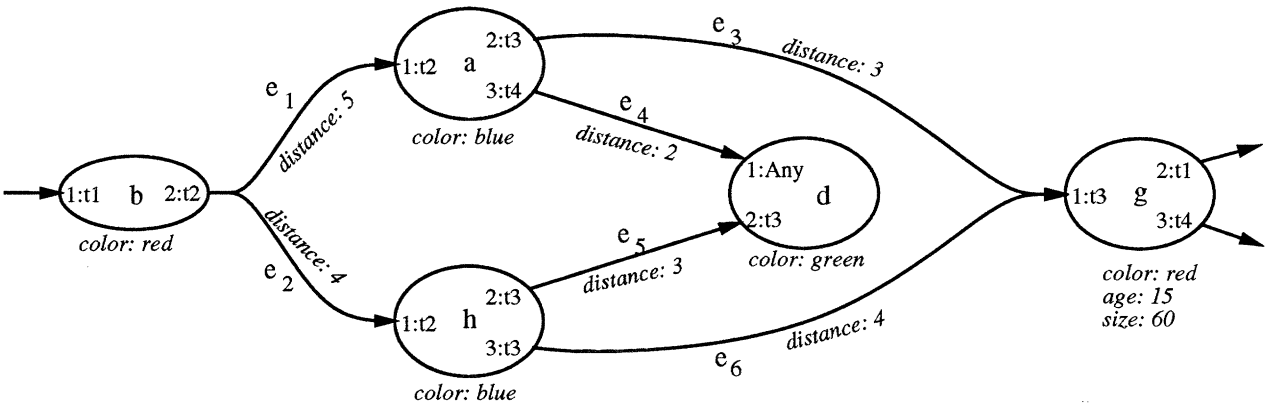


Figure 3-1: An example attributed flow graph.

Figure 3-1 shows an example flow graph. We refer to nodes by their node type. If there are two nodes with the same type, we precede the node type with a unique label. Ports are identified using numeric annotations on the nodes. Each numeric port identifier is followed by a colon and the port's type. The edges of the flow graph have been labeled with subscripted "e"s.

Edge e_5 connects two ports of type t_3, while edge e_4 connects a port of type t_4 with one of type `Any`. Edges e_1 and e_2 fan out of port 2 on node b, while edges e_3 and e_6 fan into port 1 of node g. Node d is a sink. Port 1 of node b is an input of the graph and ports 2 and 3 of node g are outputs of the graph. (Pictorially, we emphasize inputs and outputs of the graph by drawing edge stubs adjoining them.)

In the figure, attribute-value pairs (in the form *attribute:value*) are shown in italics near the node or edge which holds a value for the attribute. In this example, all node types have the node attribute *color*. The node type g additionally has the attributes *age* and *size* and the node of type g in this particular graph has values 15 and 60, respectively, for these attributes. All edges have the attribute *distance*.

Useful Definitions

A flow graph H is a *sub-flow graph* of a flow graph G if and only if H's nodes are a subset of G's nodes, and H's edges are the subset of G's edges that connect only those ports found on nodes of H.

Isomorphism can be defined between flow graphs using a variation of its standard definition, which accounts for edges adjoining ports, rather than nodes. Two flow graphs F_1 and F_2 are *isomorphic* if and only if there is a one-to-one mapping ϕ of the nodes of F_1 onto the nodes of F_2, such that adjacency is preserved – i.e., the i^{th} output of a node n_1 is connected to the j^{th} input of a node n_2 in F_1 if and only if the i^{th} output of the node $\phi(n_1)$ is connected to the j^{th} input of the node $\phi(n_2)$ in F_2.

3.2 Flow Graph Grammars

A flow graph grammar is a set of rewriting rules (or productions), each specifying how a node in a flow graph can be replaced by a particular sub-flow graph. All rules in a flow graph grammar rewrite a single left-hand side node to a right-hand side flow graph. The grammar specifies which flow graphs are in a particular set of flow graphs, called the *language* of the grammar.

In addition, the flow graph grammar may be attributed: Each rule can specify how to compute attribute values of the rule's nodes from the attributes of other nodes in the rule. Each rule can also impose constraints on the attributes of the rule's nodes. Every flow graph in the language of an attributed grammar has attribute values that satisfy the constraints of the rules generating the flow graph.

More precisely, a flow graph grammar G has four parts: two disjoint sets N and T of node types, called non-terminals and terminals, respectively, a set P of *productions*, and a set S of distinguished non-terminal types, called the *start types* of G. (By convention, non-terminal types are denoted by capital letters, while terminal types are in lower case.)

Each production in P consists of the following five parts:

- A flow graph L, called the *left-hand side*, containing a single node having a non-terminal type.
- A flow graph R, called the *right-hand side*, containing nodes of non-terminal or terminal types.
- An *embedding relation* C which specifies the correspondence between the ports of L and R.
- A set of *attribute conditions*, which impose constraints (in the form of relations) on the attribute values of nodes and edges in R.
- A set of *attribute transfer rules*, each of which specifies the value of an attribute of L's node in terms of the attributes of the nodes and edges in R.

Sections 3.2.1 and 3.2.3 discuss the embedding relation and the attribute conditions and transfer rules in more detail.

3.2.1 Embedding Relation

The embedding relation is necessary in flow graph grammar rules (unlike string grammar rules) to provide connectivity information when an occurrence of a left-hand side is rewritten during a derivation. It specifies how the ports connected to the left-hand side should be connected to the right-hand side flow graph, and possibly to each other, when the left-hand side is replaced by the right-hand side. (It is used in an analogous way in the reverse process

of reducing an occurrence of a rule's right-hand side to its left-hand side during recognition or parsing.)

The embedding relation C is a binary relation on $\mathcal{L} \times \mathcal{R} \cup \mathcal{L}$, where $\mathcal{L}$ denotes the set of left-hand side ports and $\mathcal{R}$ denotes the set of right-hand side ports of a rule. A left-hand side port l_i and a right-hand side port or another left-hand side port p_j are said to "correspond" if $(l_i, p_j) \in C$. The embedding relation is restricted in the following ways.

1. If a left-hand side port corresponds to a right-hand side port, then both ports must be of the same direction (input or output). If two left-hand side ports correspond to each other, they must be of opposite directions.

2. More than one right-hand side port and/or left-hand side port may correspond to the same left-hand side port. However, more than one left-hand side port may not correspond to the same right-hand side port.

3. Each left-hand side port corresponds to at least one right-hand side or left-hand side port. (A right-hand side port need not correspond to some left-hand side port.)

The right-hand side ports corresponding to ports on the left-hand side node need not be inputs or outputs of the right-hand side graph (i.e., they may be connected to other ports in the graph).

The definition of the embedding relation is extended (as described in Section 3.4.2) to encode aggregation information. However, the extended relation still obeys these restrictions.

When a left-hand side port l_1 corresponds with another left-hand side port l_2, the rule is said to contain a *straight-through* (abbreviated "st-thru"). We discuss the significance of st-thrus in the next section, where we describe how the embedding relation is used in the derivation of flow graphs.

Figure 3-2 shows an example flow graph grammar. In this example, ports are referred to as subscripted node types (e.g., a_1 refers to the port labeled 1 on the node with type a). Port types are not shown. The port correspondences of each rule are indicated pictorially by matching Greek letters. For example, left-hand side port A_1 corresponds to right-hand side port a_1. (This grammar does not have attribute conditions or attribute transfer rules, so they are not shown. See Section 3.2.3 for the details of attribute handling and Figure 3-5 for a complete picture.)

By convention, when a port correspondence involves an internal right-hand side port (not an input or output of the right-hand side graph), we draw an edge stub coming into or out of that port. We annotate the edge stub with the port correspondence label. For example, this is done in drawing the rule for non-terminal A in Figure 3-2. Also, when two or more right-hand side ports correspond to the same left-hand side port, the edge stubs from the right-hand side ports are drawn as if they are merged with each other. This abbreviated notation is used, for example, in depicting the rule for B. (This makes it easier

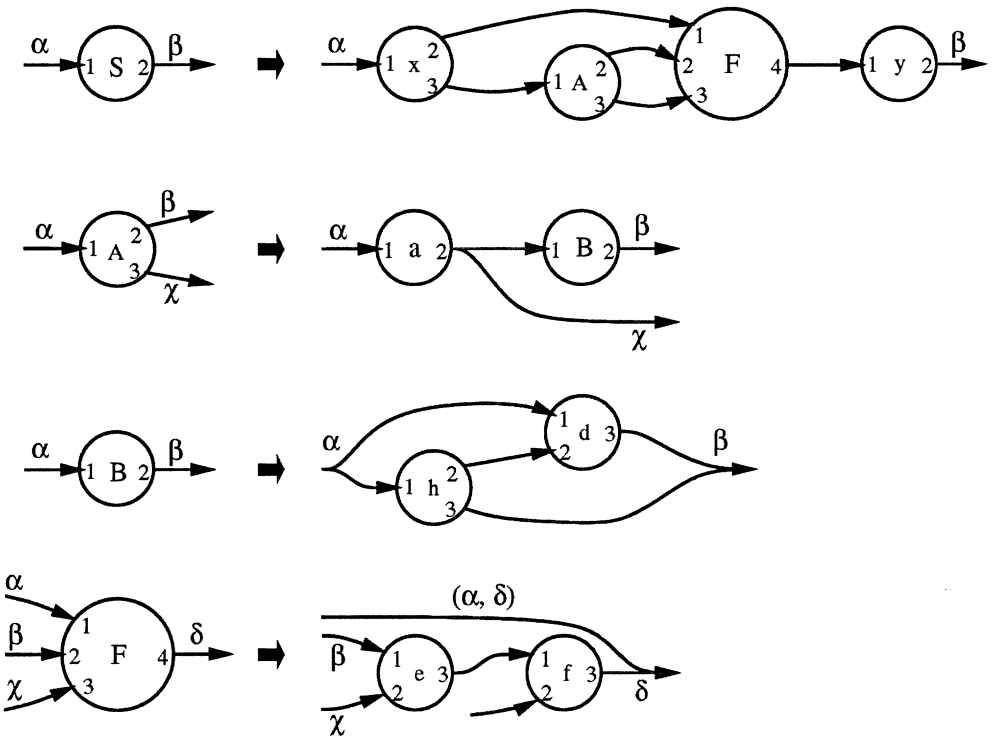


Figure 3-2: An example flow graph grammar.

to visualize how the right-hand side of a rule is embedded into a graph when the left-hand side is expanded during derivation.)

Similarly, st-thrus are depicted as lines which do not adjoin any port, but which may be merged with an edge stub and/or another st-thru. In drawings, they are annotated with the pair of correspondence labels associated with the left-hand side ports that correspond. The rule for F contains a st-thru, since ports F_1 and F_4 correspond.

3.2.2 Flow Graph Grammar Derivations

A flow graph is derived from a start type S_o of a flow graph grammar by starting with a flow graph containing a single node of type S_o and repeatedly applying the grammar's rewrite rules (productions) to the non-terminals in this graph until no non-terminals are left.

Each rewrite rule specifies how an isomorphic occurrence of the rule's left-hand side L can be replaced by the rule's right-hand side graph R. The embedding relation C of the rule is used to embed R in the graph once L has been removed. In particular, for each right-hand side port r_i and left-hand side port l_i related by C, r_i is connected to all of the ports that were connected to l_i before L was removed.

In addition, if a left-hand side input port l_i corresponds to a left-hand side output port l_j, then edges are drawn connecting each of the ports connected to l_i to each of the ports connected to l_j. In other words, when a rule contains a st-thru, the embedding relation

between the ports involved, l_i and l_j, imposes the constraint that the ports adjacent to l_i and l_j become connected directly to each other when the left-hand side is rewritten.

For example, a sample derivation of a graph from the grammar of Figure 3-2 is shown in Figure 3-3. When the non-terminal node A is expanded in the second step of the derivation, A is removed from the graph, along with the edges adjoining its ports. Then the right-hand side of the rule for A is added to the graph. Finally, edges are drawn between the right-hand side ports a_1, B_2, and a_2 and the ports to which A_1, A_2, and A_3 (respectively) had been connected (i.e., x_3, F_2, and F_3).

In string grammars, the derivation tree is used as a canonical representation of equivalent derivations, which abstracts away from the order in which productions are applied in the derivations. It is useful to make use of a similar representation for flow graph derivations.

As in the string case, a derivation tree has vertices labeled with the node type of a non-terminal that was expanded during the derivation. However, unlike the string case, the children of each vertex are related in a partial ordering. The right-hand side graph in the production for the vertex's label defines this partial ordering. (Derivation trees are normally shown without the edges between the nodes of the tree to reduce clutter.) For example, the derivation sequence of Figure 3-3 is represented by the derivation tree of Figure 3-4.

3.2.3 Attribute Conditions and Transfer Rules

So far, we have discussed the aspects of flow graph grammars that impose *structural* constraints on the flow graphs in their languages, for example, by constraining their node types and edge connections. This section describes how the *non-structural* aspects of a flow graph are constrained. Attributes are used to represent information that cannot be adequately expressed in the structure of a flow graph. Attribute conditions in grammar rules impose constraints on these attributes.

The concept of an attributed string grammar was formalized by Knuth [77] as a way to assign semantics to strings in a context free language. Attribute values are computed from other attribute values within a rule. This is called *attribute evaluation.* The attributes that are computed represent some aspect of the "meaning" of the string being parsed (e.g., the decimal value of a binary number).

Since then, attribute grammars have been used extensively in such areas as pattern recognition [16, 17, 39, 48, 86, 135], compiler technology [40, 41, 47, 68, 74, 78, 79], programming environments [6, 28], software specification and development [38, 97, 98, 101, 131], and test case generation [30]. Raiha [107] gives a bibliography of the early papers. These systems use attribute grammars to deal with nonstructural, semantic properties of a pattern and to reduce the complexity of the grammar. Much of the theoretical work in this area has focussed on developing efficient attribute evaluation strategies [28, 68, 73, 109], the complexity of checking that attribute grammars are well-formed [64], and assisting the writing of attribute grammars which contain complex dependencies among the attributes

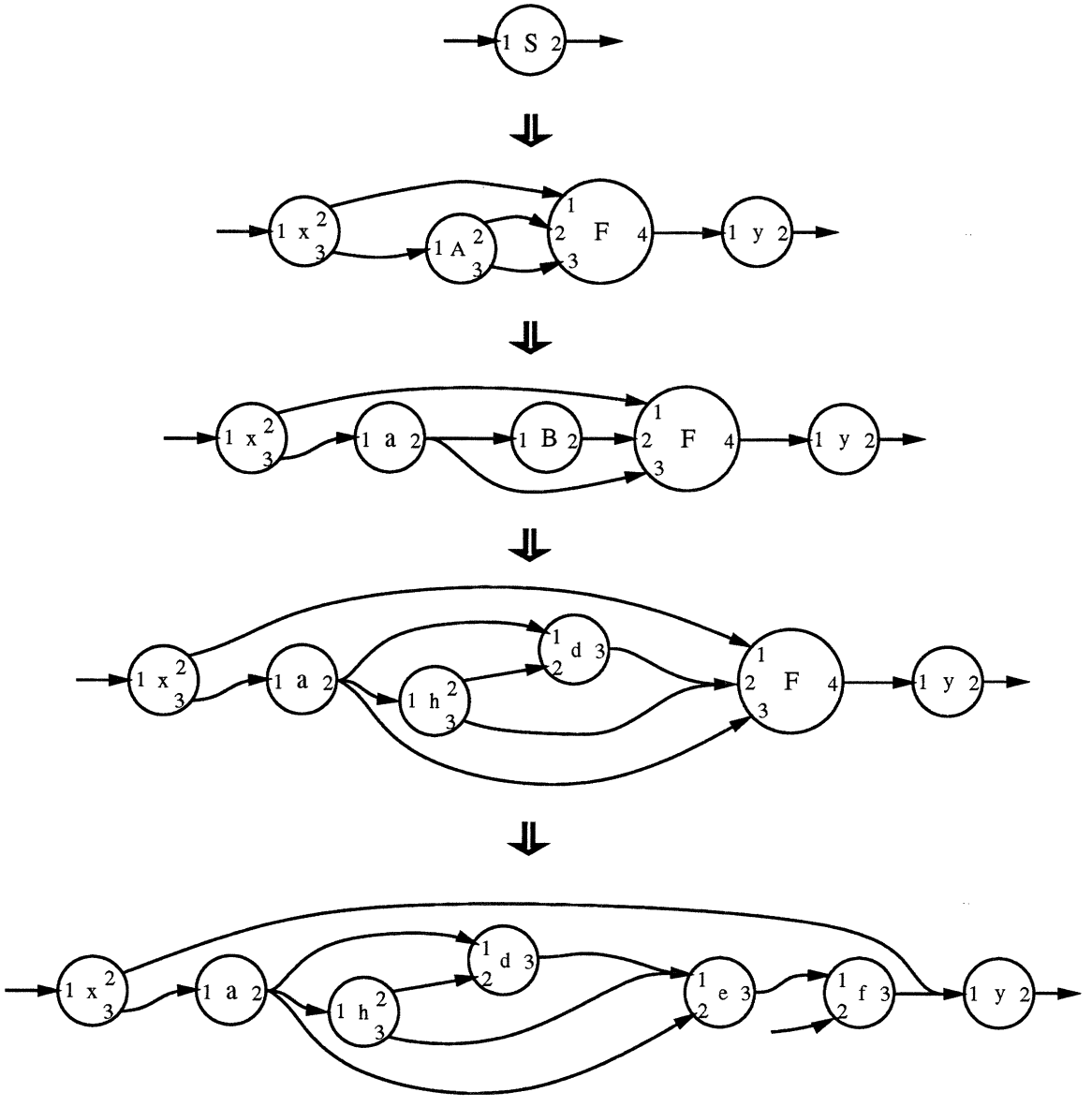


Figure 3-3: An example derivation sequence.

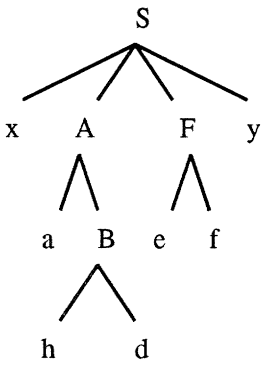


Figure 3-4: An example derivation tree.

[29].

Our flow graph grammars are attributed grammars in the sense that their productions contain *attribute transfer rules* for computing attribute values from the attribute values of other nodes and edges within the rule. (These are also called "semantic rules" [77], "attribute transfer functions" [16], or "attribute transfer specifications"[145].)

In general, attribute transfer rules can associate the attribute of some node or edge on *either* side of a rule with a function for computing its value from the attributes of the other nodes and edges (on either side) of the rule. Attributes that are computed for the left-hand side node from the attributes of the right-hand side are called *synthesized* attributes. Those that are computed for a right-hand side node or edge from the attributes of the left-hand side node and/or other nodes and edges in the right-hand side are called *inherited* attributes.

Currently, the flow graph grammar used by the recognition system uses only synthesized attributes. This is because our attributed flow graph grammars are not used so much for *computing* attribute values, as for imposing *constraints* on the attributes of the flow graph being parsed. Inherited attributes are useful if the value of an attribute involves complex dependencies across the derivation tree. However, the attribute values computed in the current system are based on simple relationships among attributes. Synthesized attributes are adequate.

Constraints are imposed on attributes in the form of *attribute conditions* on grammar rules. Attribute conditions are relations on the attribute values of the nodes and edges of a flow graph grammar rule's right-hand side. They specify constraints that must be satisfied by the attributes of a flow graph if it is in the language of the grammar. (These are also called "context conditions" [68], "constraints" [145], and "applicability predicates"[16].)

The attribute conditions and attribute transfer rules of a production are used primarily during parsing. (They *can* be used during generation to produce a set of conditions that must be satisfied by the attribute values of the flow graph generated. However, this is not how they are typically used.)

A parser for an attributed grammar engages in the following three activities when given

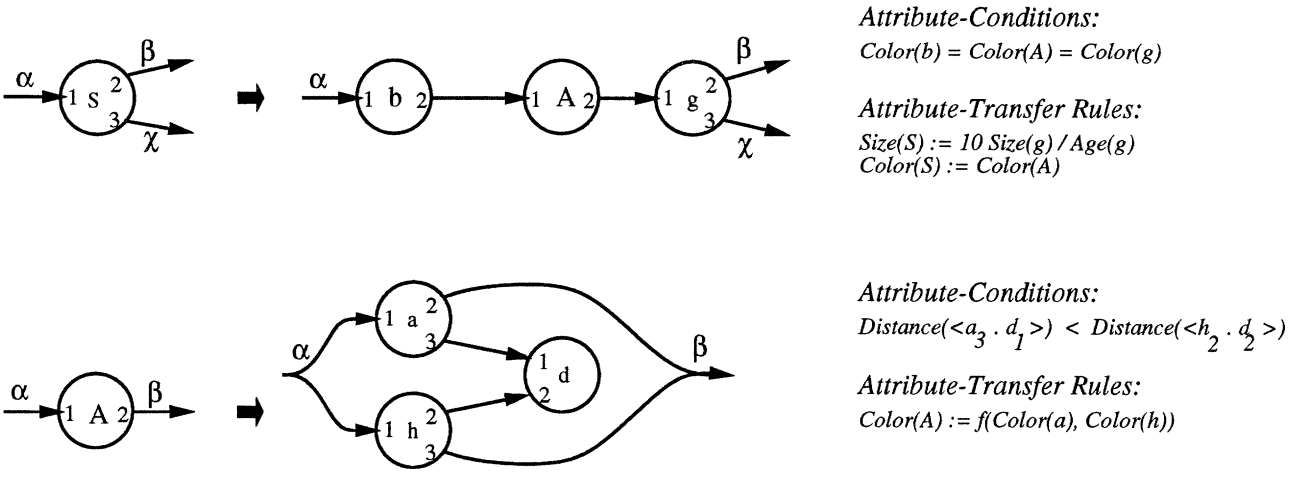


Figure 3-5: An example attributed flow graph grammar.

a string (or graph, in the case of attributed graph grammars) x:

1. Structural analysis – recover a derivation of x from a start type of the grammar and create a derivation tree to represent the derivation. If no derivation tree is found, reject x for membership in the language of the grammar. (This is the usual activity performed by recognizers for non-attributed grammars.)

2. Attribute evaluation – propagate attribute values throughout the derivation tree in accordance with the attribute transfer rules. Values for synthesized attributes move upward as a function of the attribute values of the descendants of a node, while inherited attribute values move downward from the ancestors.

3. Attribute condition checking – maintain the invariant that if all attribute values are known for the attributes related by an attribute condition, then the condition must hold. If a condition fails to hold, reject x.

If the recognizer finishes with an attributed derivation tree for x and all attribute conditions of all productions involved are satisfied, then x is recognized as a member of the language.

For example, Figure 3-6 shows the derivation tree that would result from parsing the attributed flow graph in Figure 3-1 in accordance with the grammar of Figure 3-5. The edges are drawn between the leaves of the derivation tree to show the edge attributes that are involved in the parse. Dashed arrows show the propagation of attribute values.

The three parsing activities can be interleaved. The interleaving is particularly simple in our parser, since only synthesized attributes are used. All attribute values of a derivation node depend only on the attributes of the node's descendants. Attribute conditions can be checked as soon as the right-hand side of a rule is recognized. Attribute values can

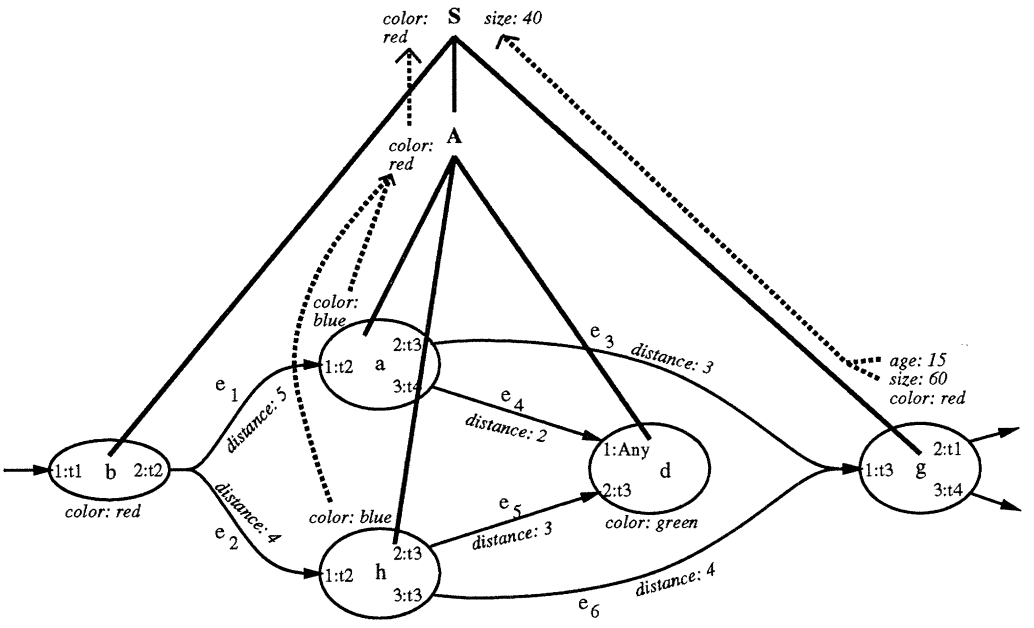


Figure 3-6: An attributed derivation tree.

be computed and transferred to the left-hand side node during the reduction of the right-hand side to the left-hand side. Because the attribute condition checking is folded into the structural parsing process (i.e., conditions are checked each time a reduction is attempted), invalid parses can be cut off early.

In the future, if inherited attributes are needed, a more sophisticated attribute evaluation and condition checking strategy will need to be employed (for example [28, 68, 73, 109]).

3.3 Motivations for Formalism: Program Recognition Application

So far, the basics of the flow graph formalism have been described. There are two major extensions to this formalism that increase the class of flow graphs and grammars that can be succinctly expressed in it. However, before they are described, this section briefly shows how the basic formalism is used in a particular application domain. This provides some rationale for the restrictions on the grammar formalism that have been described so far. (This section is not needed to understand the extensions. It may be read after the extensions have been discussed.)

We apply the flow graph formalism to the representation of programs and programming clichés. In particular, flow graphs serve as graphical abstractions of programs, flow graph grammars encode allowable implementation steps between abstract operations and lower-level operations, and the derivation trees resulting from parsing give the program's top-down design.

```
(DEFUN RIGHTP (HYPOTENUSE SIDE1 SIDE2)
  (LET* ((HYP-SQ (SQ HYPOTENUSE))
         (DIFF (- HYP-SQ
                  (+ (SQ SIDE1)
                     (SQ SIDE2))))
         (DELTA (IF (< DIFF 0)
                    (NEGATE DIFF)
                    DIFF)))
    (IF (<= DELTA (* HYP-SQ 0.02))
        T
        NIL)))
```

Figure 3-7: Testing whether the three input sides form a right triangle.

The flow graph is used to represent the operations of a program and the dataflow between them. Each non-sink node in a flow graph represents a function, with ports on the node representing distinct inputs and outputs of the function. The ports' types are determined by the signature of the function. Sink nodes represent conditional tests. The edges of a flow graph represent dataflow constraints between the functions and tests. When the result of a function is consumed by more than one function, the edges representing the dataflow fan out. Edges that fan in represent the conditional merging of more than one dataflow.

For example, Figure 3-8 shows the flow graph representing the code shown in Figure 3-7.[2] `RIGHTP` determines whether the inputs could be the lengths of the sides of a right triangle. It checks whether the square of `HYPOTENUSE` is approximately equal to the sum of the squares of `SIDE1` and `SIDE2`.

Two special nodes of type $\$B\$$ and $\$E\$$, which are not in $N \cup T$ cap the ends of the flow graph. These hold ports that represent the input and output values of data consumed or produced by the code. These nodes make it easy to represent the fan-out of input data to more than one function and the conditional fan-in of output data. For example, port 1 on $\$E\$$ receives fan-in representing the conditional output of either constant `T` or `NIL`.

Attributes on nodes and edges are used to capture characteristics of a program that cannot be adequately expressed in the structure of a flow graph. Control flow information is stored in the attributes of the flow graph representing a program. Each node has a *control environment* attribute whose value indicates under which conditions the operation represented by the node is executed. Nodes in the same control environment represent functions that are all executed under the same conditions. (Section 4.1.1 describes the vocabulary of attributes and attribute conditions used by the recognition system in more detail.)

Sink nodes, representing conditional tests, carry two additional attributes, *success-ce*

[2]The function RIGHTP is taken from Problem 3-9 (p.42) in [148].

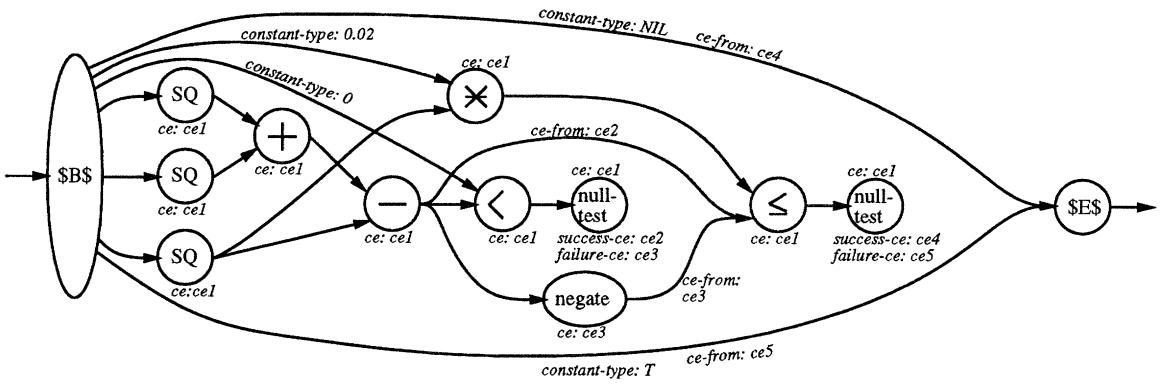


Figure 3-8: Attributed flow graph for `RIGHTP`.

and *failure-ce*. These specify the control environments whose operations are executed when the conditional test succeeds or fails, respectively.

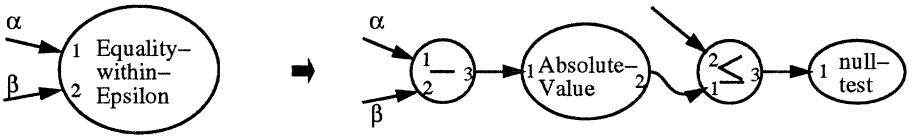
Each edge holds a *ce-from* attribute which indicates the control environment in which the edge carries dataflow. (In Figure 3-8, only ce-from attributes of edges that fan-in are shown, to reduce clutter. The edges that do not fan-in all have ce_1 as their ce-from attribute value.)

Each edge also carries a *constant-type* attribute whose value is either a constant (such as `T`, `NIL`, `0`) or `undefined`, depending on whether the edge represents dataflow from a constant. For edges whose source is not a port on node $\$B\$$, the constant type is always `undefined`. This attribute is not shown in Figure 3-8 for edges for which its value is `undefined`.

Program clichés are encoded in flow graph grammar rules. Informally, a rule can be seen as specifying how an abstract operation, represented by the rule's left-hand side node, is implemented in terms of lower-level operations, represented by the right-hand side flow graph. (Section 4.1 gives more details of how this is done, as well as other relationships between clichés, besides implementation relationships, which are captured in grammar rules.)

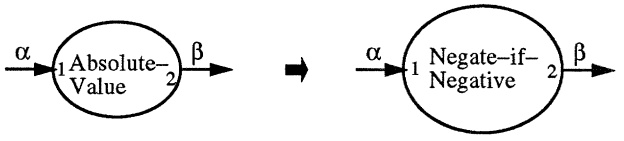
Figure 3-9 shows a grammar containing a rule that represents the common cliché of testing whether two numbers are within some "epsilon" of each other. The rules representing two common implementations of the Absolute Value cliché demonstrate that the grammar allows us to modularly specify implementation variations. The rules have typical embedding relations. In the rule for Negate-if-Negative, two right-hand side ports ($<_1$ and $negate_1$) correspond to the same left-hand side port. This represents the constraint that the input to an isomorphic instance of the right-hand side must come from a source that fans out to both $<_1$ and $negate_1$.

The rule for Negate-if-Negative also has a right-hand side port ($<_2$) that does not correspond to any left-hand side port. This right-hand side port represents the input coming from the constant 0. It is important that in our formalism a right-hand side port is not required to correspond to a left-hand side port, since otherwise we would have to add an input to Negate-if-Negative to correspond to $<_2$. This would destroy the modularity of the



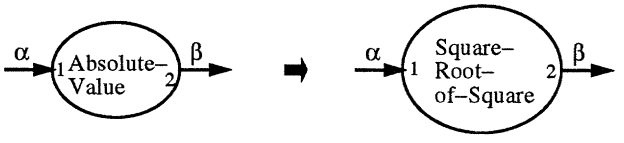
Attribute–Transfer Rules:

ce := ce(null–test).
success–ce := failure–ce(null–test).
failure–ce := success–ce(null–test).



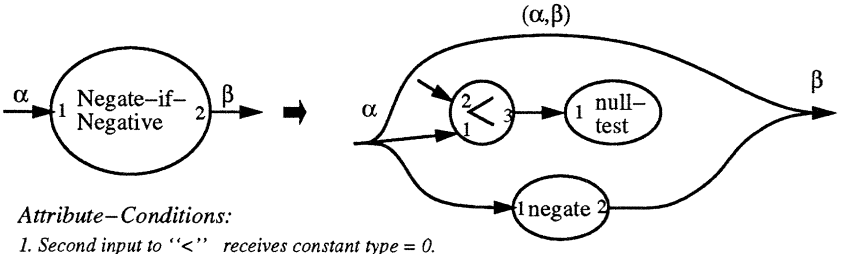
Attribute–Transfer Rules:

ce := ce(Negate–if–Negative).



Attribute–Transfer Rules:

ce := ce(Square–Root–of–Square).

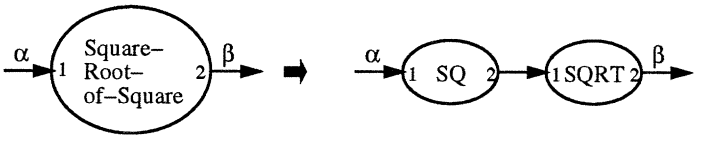


Attribute–Conditions:

1. Second input to "<" receives constant type = 0.
2. Data flows out from "negate" in failure–ce(null–test).
3. Data flows straight–through from input to output in success–ce(null–test).

Attribute–Transfer Rules:

ce := ce(null–test).



Attribute–Transfer Rules:

ce := ce(SQRT).

Figure 3-9: Flow graph grammar encoding clichés found in `RIGHTP`.

grammar, since the extra input must be propagated up through the rules that use Negate-if-Negative. We would need to add an input to the Absolute-Value node, but this extra input would be meaningless for Absolute-Value's other implementation as Square-Root-of-Square.

The rule for Negate-if-Negative also shows how st-thrus are used to represent clichéd operations in which some of the input data is not acted upon, but passes directly to the output.

This grammar also shows typical attribute conditions and attribute transfer rules. (These are stated informally in English in Figure 3-9. Section 4.1.1 gives a more formal description of the actual attribute language used in encoding clichés.) A typical attribute condition placed on an edge's attribute in a grammar rule is that it must carry dataflow in a particular control environment (e.g., the failure-ce of some test).

Attribute conditions and transfer rules may refer to attributes of nodes and edges of the rule's right-hand side. In addition, they may refer to edges in the input graph whose sources or sinks match the inputs or outputs of the rule's right-hand side, or to edges matching st-thrus. For example, the rule for Negate-if-Negative constrains the input to $<_2$ to come from a constant source of type 0. It also constrains the ce-from attribute of edges whose sources match $negate_2$ and of edges matching the st-thru.

3.3.1 The Partial Program Recognition Problem

We formulate the problem of recognizing clichés in programs in terms of solving a parsing problem for flow graphs. This section defines these problems.

The *parsing problem* for flow graphs is: Given a flow graph F and a flow graph grammar G, if F is in the language of G, then produce all possible parses for F (i.e., all possible derivation trees that yield F).

The *subgraph parsing problem* for flow graphs is: Given a flow graph F and a flow graph grammar G, find all possible parses of all *sub-flow graphs* of F that are in the language of G.

There are two types of program recognition: *total*, in which the entire program is recognized as a single cliché, and *partial*, in which the program may contain unrecognizable parts but as much of the program as possible is recognized as one or more clichés.

The *total recognition* problem for programs is: Given a program and library of clichés, determine which clichés in the library are instantiated by the program as a whole. (Usually a single program is recognizable as an instance of only one cliché, but this general definition includes cases in which a program can be viewed in more than one way.)

The *partial recognition* problem is: Given a program and a library of clichés, find all instances of the clichés in the program (i.e., determine which clichés are in the program and their locations).

In this work, we are more interested in the partial recognition problem for programs. (The total recognition problem is subsumed by it.) When we say "program recognition" we

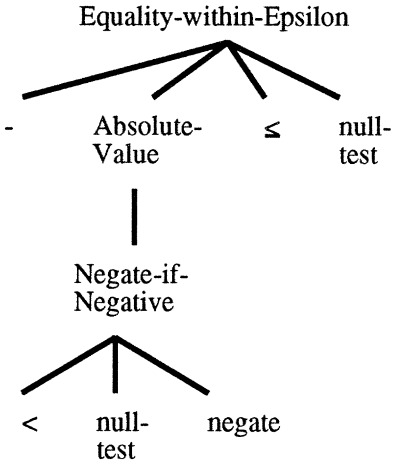


Figure 3-10: Clichés recognized in `RIGHTP`.

mean partial program recognition.

The partial program recognition problem is solved by formulating it as a subgraph parsing problem: Given a flow graph F representing the program's dataflow and a cliché library encoded as a flow graph grammar G (with all non-terminals that represent clichés as start types), solve the subgraph parsing problem on F and G.

The derivation trees that are produced are called *design trees*. The root of the tree identifies a particular cliché that was recognized and the yield of the tree indicates where the cliché was found. Intermediate non-terminals in the tree indicate the subclichés that implement the cliché that was found. Thus, casting partial program recognition as a parsing problem yields as output not only the set of clichés and their locations, but also relationships between the cliché instances.

For example, Figure 3-10 shows the design tree produced by partially recognizing the program `RIGHTP`, represented as the flow graph in Figure 3-8 and using the graph grammar of Figure 3-9.

When a program is partially recognized, one or more sub-flow graphs of the program's flow graph encoding are recognized as members of the language of the graph grammar which encodes the cliché library. From the definition of a sub-flow graph, we can see that it is possible to ignore portions of a flow graph before and after a recognizable sub-flow graph, as well as portions that fan out from or into an internal port in the sub-flow graph.

3.4 Extensions to the Flow Graph Formalism

The next two sections discuss two major extensions to the flow graph grammar formalism described so far. The first extension follows closely an extension made by Lutz [90] to a graph formalism similar to ours, while the second is novel to our research. The extensions are the following.

1. We expand the language of a flow graph grammar to include all flow graphs derivable not only from a start type of the flow graph grammar, but also from flow graphs that are "share-equivalent" to a sentential form[3] of the grammar. The notion of share-equivalence captures the types of variation due to *structure-sharing* that the extended formalism abstracts away. In a structure-sharing flow graph, a node plays the role of more than one node of the same type by generating output that fans out or by receiving input that fans in.

2. We extend the expressiveness of the flow graph grammar to allow it to capture the rewriting of a single input (or output) of a non-terminal node into an *aggregation* of inputs (or outputs) of a sub-flow graph. We then further expand the language of a flow graph grammar to include all flow graphs that are "aggregation-equivalent" to the flow graphs derivable from the grammar. The notion of aggregation-equivalence defines the variation tolerated in how aggregates are organized.

In the program recognition application, the first extension is needed to deal with variation due to the common engineering optimization of function-sharing. The second extension is important in being able to represent and recognize clichéd operations on aggregate data structures.

These extensions to the formalism are described in this section. However, the mechanisms by which the parsing problem is solved for flow graphs in the extended formalism are described in Section 3.5, after the parsing process for the basic unextended formalism is presented.

We make these extensions to remove some forms of variation between semantically equivalent programs that are not abstracted away by the graph representation alone. We essentially do this by imposing an equivalence relation on the graphs representing the programs. Alternatively, we could impose the equivalence relation at the source text level by transforming program expressions directly. For example, a great deal of work has been done in the term rewriting area [60, 61, 75]. These techniques are good for canonicalizing localized parts of a program (e.g., by algebraic simplification and normalization). However, if the expression that we want to rewrite is delocalized and interleaved with unrelated expressions, we need to first apply subexpression shuffling and copying transformations to localize it. This is avoided in the graph representation which tends to localize related operations. Expression-based techniques also fall prey to syntactic variation. It would be useful to combine the expression-based rewriting techniques with graph-based parsing. One way is to canonicalize the text as much as possible first and then convert to the graph-based representation and parse. Another is to interleave the two (maintaining multiple representations) so that expression-based simplifications and normalizations can be done to aid recognition and the graph-based representation can localize expressions to rewrite and abstract away

[3] A *sentential form* of a graph grammar is any flow graph that is derivable from a start type of the grammar by the application of zero or more productions of the grammar.

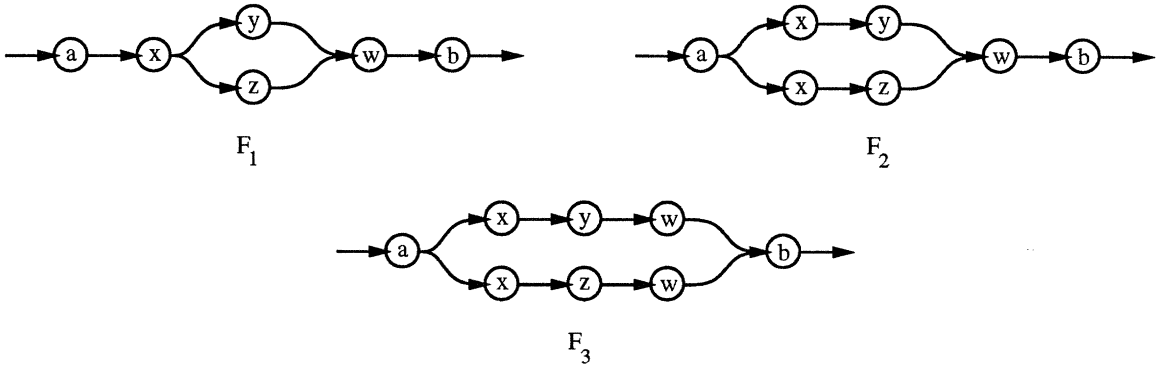


Figure 3-11: These flow graphs should all be seen as equivalent.

syntactic differences.

3.4.1 Structure-Sharing

Flow graphs can be used to represent collections of components having inputs and outputs that are produced or consumed by each other. In using this representation, we would like to be able to view a flow graph in which two or more components of the same type are collapsed into a single shared component as being equivalent to a flow graph in which the two components are not collapsed. See Figure 3-11.

This is important in dealing with variation due to function-sharing, in engineering applications of the formalism. Function-sharing is a common engineering optimization made during design, in which one component fulfills more than one purpose. For example, in an optimized program, two or more functions may be applied to the result of a single (shared) function application.

We employ a notion of *share-equivalence* to capture the relationship between flow graphs, such as those in Figure 3-11. This notion was introduced by Lutz [90] for graphs similar to ours. Share-equivalence is defined in terms of a binary relation *collapses* (denoted $\lhd$) on flow graphs. Flow graph F_1 collapses flow graph F_2 if and only if there are two nodes n_1 and n_2 of the same node type t in F_2, having input arity I and output arity O, such that all of these conditions hold:

1. Either one or both of the following are true:

   (a) $\forall i = 1...I$, the i^{th} input port of n_1 is connected to the same set of output ports as the i^{th} input port of n_2.

   (b) $\forall j = 1...O$, the j^{th} output port of n_1 is connected to the same set of input ports as the j^{th} output port of n_2.

2. F_1 can be created from F_2 by replacing n_1 and n_2 with a new node n_3 of type t with the i^{th} input (resp., output) of n_3 connected to the union of the ports connected to

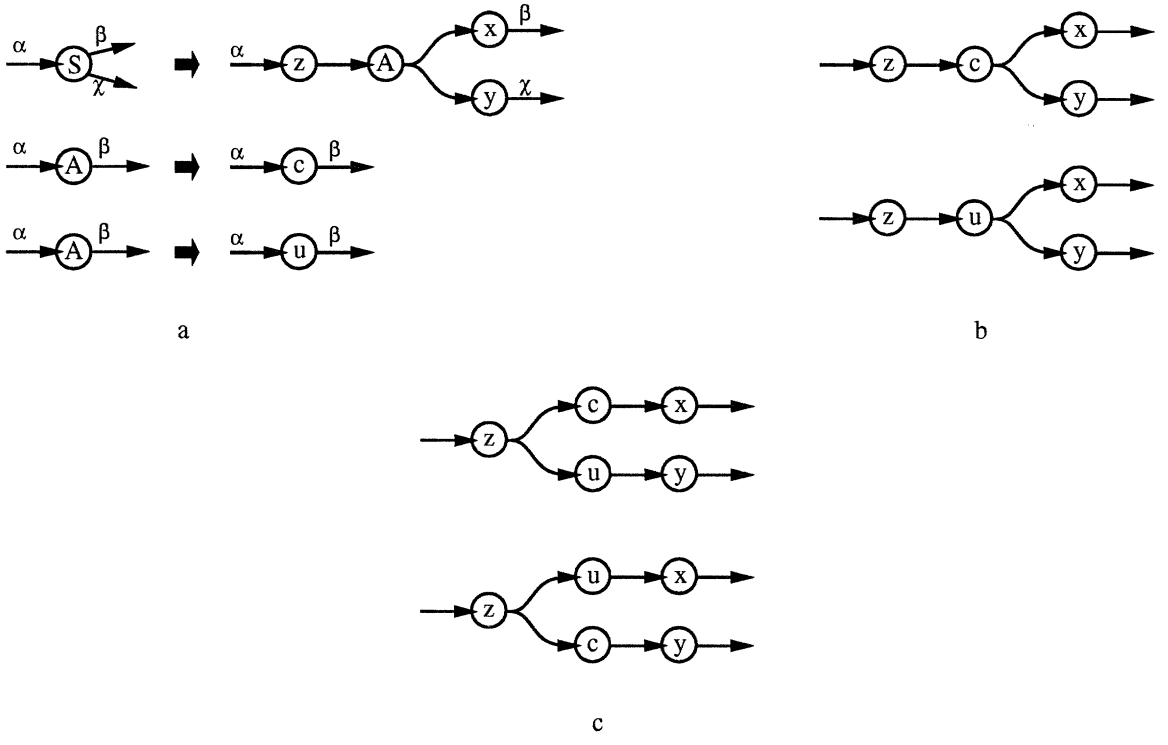


Figure 3-12: a) A grammar. b) Its core language. c) Some flow graphs in its expanded language.

the i^{th} inputs (resp., outputs) of n_1 and n_2.

3. The attribute values of n_1 and n_2 can be "combined." This is done by applying an *attribute combination function*, which is defined for each attribute, to the attribute values of n_1 and n_2. The attribute combination functions may be partial functions. If the function is not defined for n_1 and n_2's attributes, then the attribute values cannot be combined (and F_1 does not collapse F_2).

For example, in Figure 3-11, F_1 collapses F_2 which collapses F_3. Performing the transformation in condition 2 from F_2 to F_1 is called "zipping up" F_2. Its inverse is referred to as "unzipping".

The reflexive, symmetric, transitive closure of collapses, $\lhd^*$, defines the equivalence relation *share-equivalent*. (In Figure 3-11, F_1, F_2, and F_3 are all share-equivalent.)

The *directly derives* relation ($\Rightarrow$) between flow graphs is redefined as follows. A flow graph F_1 directly derives another flow graph F_2 if and only if either F_2 can be produced by applying a grammar rule to F_1, $F_1 \lhd F_2$, or $F_2 \lhd F_1$.

As in string grammars, the reflexive, transitive closure of $\Rightarrow$, is the *derives* relation ($\Rightarrow^*$). The language of a flow graph grammar G (denoted $L(G)$) is the set of all flow graphs, whose nodes are of terminal type and which can be derived from a start type of G.

Thus, the notion of a language of a flow graph grammar G has been extended to include

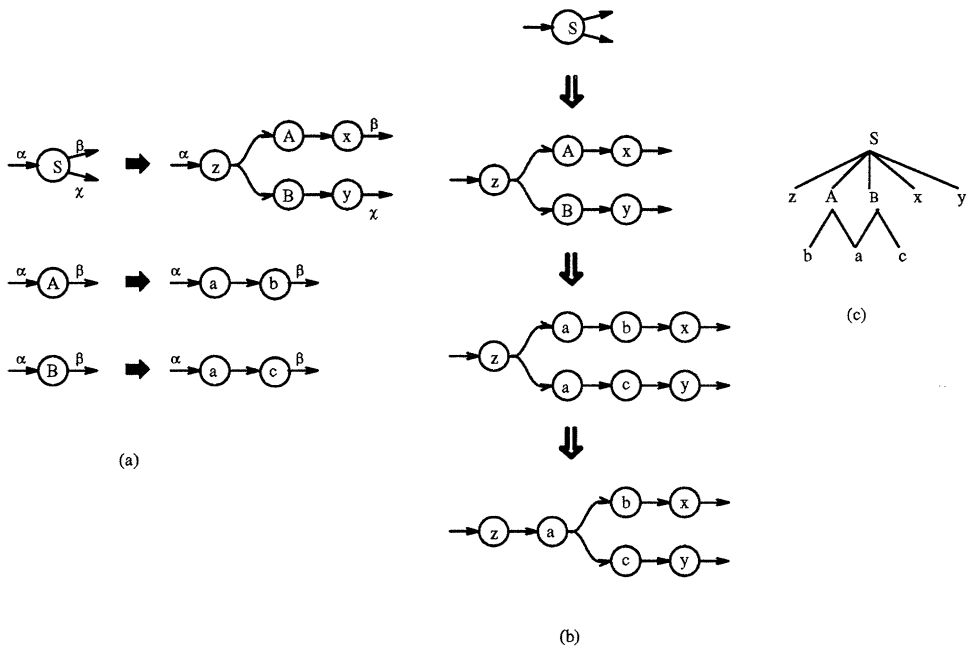


Figure 3-13: a) A grammar. b) A derivation sequence. c) A derivation graph representing the derivation.

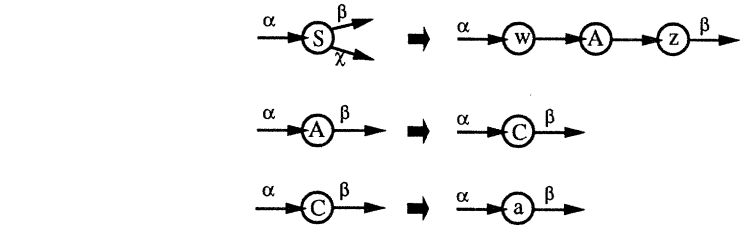
flow graphs that are generated by a series of not only production rule applications but also zip-up and unzipping transformations. Since a zip-up or unzipping step can happen anywhere in the derivation sequence, the language of a graph grammar G in this extended formalism is a superset of the set of flow graphs share-equivalent to flow graphs in the "core" language of G in the unextended formalism. For example, the flow graphs in Figure 3-12c are included in the language of the grammar in Figure 3-12a, even though they are not share-equivalent to either of the flow graphs in the grammar's core language, shown in Figure 3-12b.

Both generators and parsers for the language of a flow graph grammar can interleave zipping and unzipping transformation steps with their usual expansion and reduction steps. The parser used by the program recognition system reported here simulates the introduction of these transformations into its reduction sequence, as is described in Section 3.5.1.

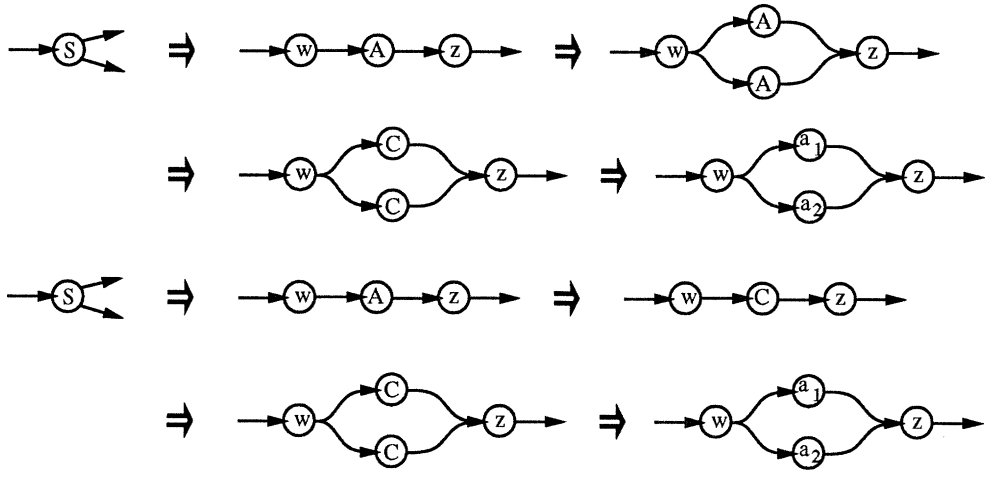
Structure-Sharing Derivation "Trees"

The extensions to the language of a flow graph grammar affect how equivalent derivation sequences are captured in a single canonical tree representation. Because flow graph zip-up can occur as part of a derivation sequence and this results in a shared subderivation, the representation of a derivation as a *tree* is no longer possible. Derivations must be represented as *graphs*. For example, see Figure 3-13.

In addition, there may be different derivation graphs, depending on when unzipping is done in the derivation sequence. For example, Figure 3-14a shows a simple flow graph



(a)



(b)

(c)

Figure 3-14: (a) A grammar. (b) Two derivations of same flow graph. (c) Two derivation graphs representing the derivations.

grammar and Figure 3-14b gives two possible derivation sequences. In the first sequence, the unzipping transformation happens in the second step. In the second derivation sequence, this transformation happens in the third step. An unzipping step is represented in a derivation graph by a vertex that is a group of instances of that vertex, each with its own sub-derivation. The two derivation sequences are represented by the two derivation graphs in Figure 3-14c.

We arbitrarily choose those derivation graphs as canonical that represent derivation sequences in which unzipping occurs at the earliest possible moment in the derivation sequence (i.e., unzip a non-terminal before it is expanded). In our example, the derivation graph on the left is taken as canonical.

3.4.2 Aggregation

Grammar rules in our flow graph formalism specify how a non-terminal node can be rewritten as a particular grouping of terminal and non-terminal nodes (in the form of a flow graph). We now extend it to also specify how a single input or output of a non-terminal node can correspond to an *aggregation* of the inputs or outputs of a flow graph to which the non-terminal node is rewritten.

In engineering application domains, this is useful in representing not only how aggregations of components make up a higher-level component, but also how the inputs and outputs of the components are aggregated into fewer, more abstract types of inputs and outputs of the higher-level component. In the programming domain, for example, the Circular Indexed Sequence Insert cliché has two inputs: an element to insert and a clichéd aggregate data structure (the Circular Indexed Sequence). The insert is implemented by a group of primitive operations with several of their inputs representing the various parts aggregated by the single Circular Indexed Sequence data type.

This section first considers a way to capture the aggregation of port types without extending the formalism. This is found to be too intolerant of the variation that may exist in the way port types are aggregated. However, it provides useful insights into what is required to handle the variation. In particular, a notion of *aggregation-equivalence* is defined to relate flow graphs that differ only in how they aggregate port types. The language of a flow graph grammar is expanded to consist of all flow graphs aggregation-equivalent to flow graphs derivable from a start type of the grammar.

Using Make and Spread Nodes

This section sets up a straw man which is a simple way to capture the aggregation of port types into a single, more abstract port type without extending the graph grammar formalism. This technique will work in restricted cases. However, as the next section shows, it is too intolerant of variations in the organization of aggregates.

A simple way to capture the aggregation of port types into fewer, more abstract port

types is to use special nodes, called *Make* and *Spread* nodes. A Make node represents the aggregation of input port types into the output port type, while a Spread node represents the decomposition of the input port type into the output port types.

Each Make node has a tuple of input ports whose types compose the type of the Make's single output port. The node type of a Make node is defined by the ordered tuple of its output ports' types and its aggregate input port's type. Two Make nodes match if they collect the same tuple of input port types into the same aggregate output port type. Spread nodes are analogous to Make nodes, but have a single input port of aggregate port type and a tuple of output ports which have types composing the input port's type.

Make and Spread node types come in pairs, called *corresponding pairs.* For each Make node type, there is a corresponding Spread node type (and vice versa) for the same aggregate type, such that the i^{th} *input* of the Make corresponds to the i^{th} *output* of the Spread in that they have the same port type and represent the same part of the aggregate port type.

Using Make and Spread nodes, we can now write production rules such as the ones shown in the grammar of Figure 3-15. For example, in the right-hand side of the rule for A, Spread and Make nodes explicitly show how the inputs and outputs of nodes a and b are aggregated into the abstract port type P. This port type is the type of both the input and the output of the left-hand side node A. These types of rule require no extension to the graph grammar formalism describe in Section 3.2. F_1 in Figure 3-16 is the (only) flow graph in the language of the grammar in Figure 3-15.

To simplify the discussion, we assume right-hand sides only have Spreads and Makes on fringes and that no nesting of Spreads or Makes occurs on any right-hand side. A flow graph grammar can always be transformed so that this is true.

We also assume that abstraction monotonically increases as we move up through the grammar rules. Left-hand side port types are always either aggregates of (i.e., more abstract than) their corresponding right-hand side port types or are of the same type as their corresponding right-hand side port types. Right-hand side port types are never aggregates of left-hand side port types. This means no flow graph in the language of a flow graph grammar has inputs going to a Make node or outputs coming from a Spread node.

Problems Due to the Inflexibility of Makes and Spreads

The flow graph F_1 in Figure 3-16 is the only one derivable from the start type S. However, we would like to expand the language of the grammar to include flow graphs that differ from this one solely in the way port types are aggregated within the graph. In particular, the organization of aggregated port types may vary in any of the following ways:

1. Port types may be aggregated in any order, since aggregation is *commutative.* For example, flow graph F_2 in Figure 3-16 aggregates types x and y into P in the opposite order in which F_1 does.

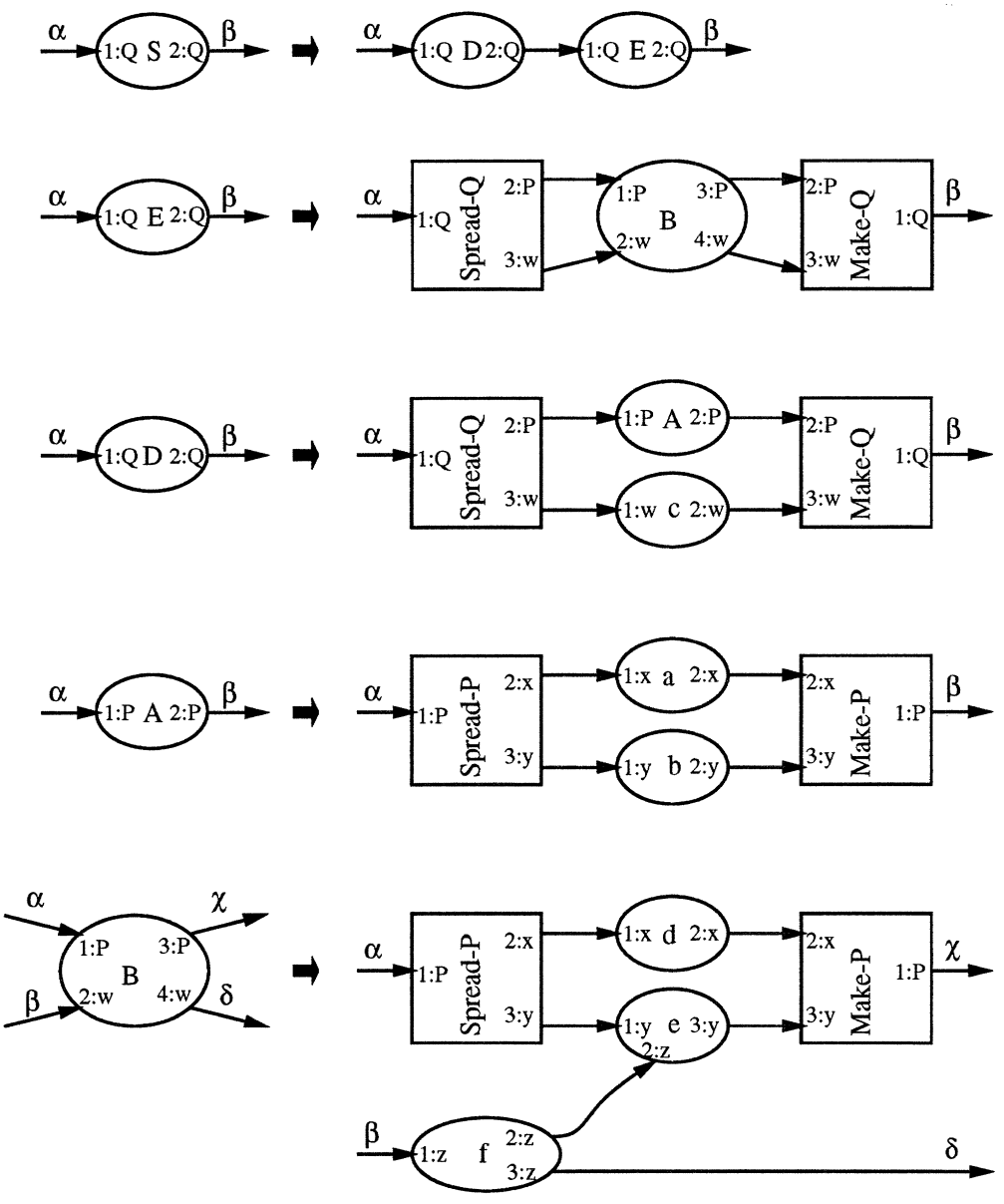


Figure 3-15: A grammar representing aggregation, using Spread and Make nodes.

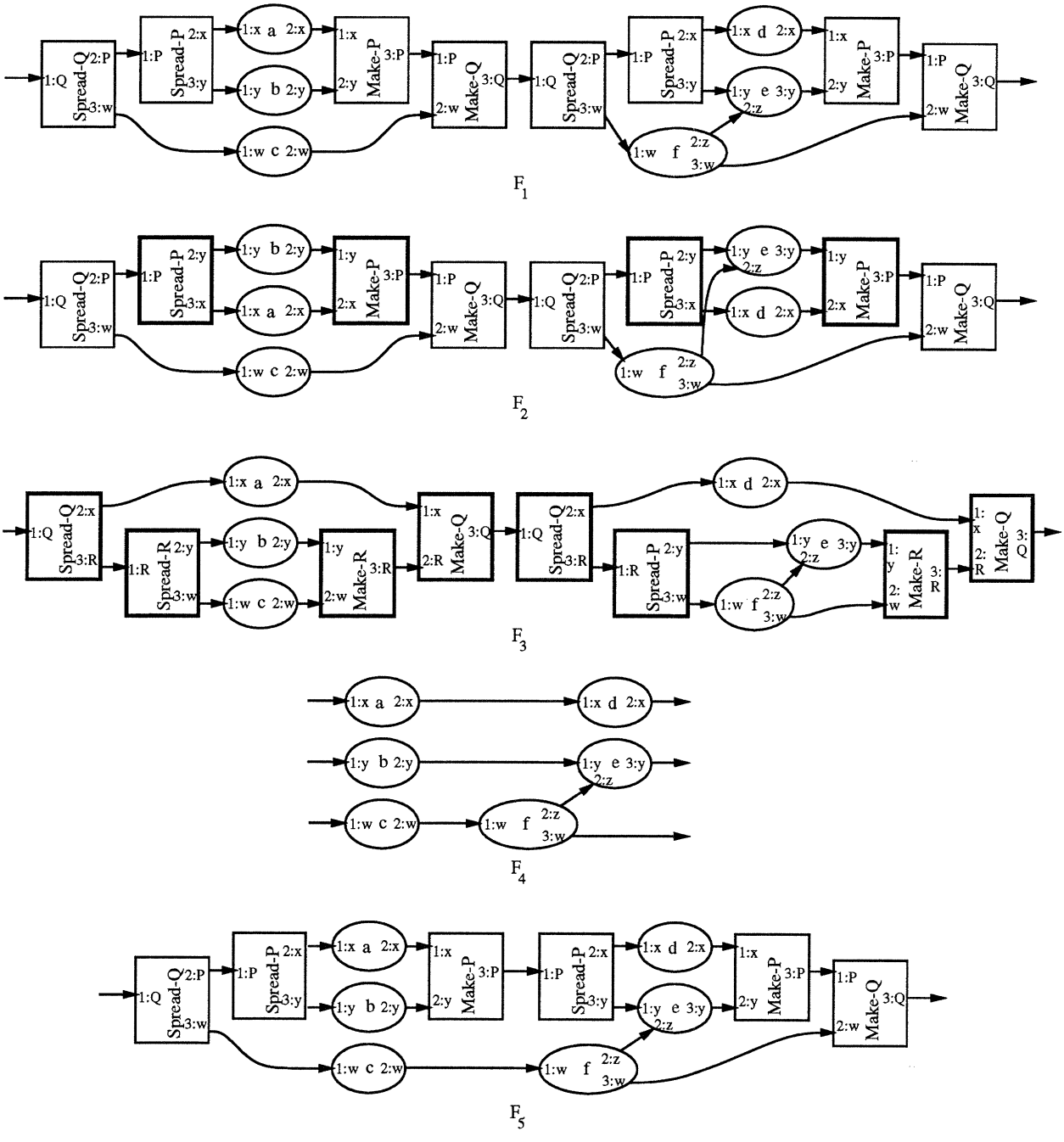


Figure 3-16: F_1 is the flow graph in the language of the grammar in Figure 3-15. The rest are flow graphs aggregation-equivalent to it.

2. Aggregations of port types may be nested within other aggregations and the organization of this nesting does not matter, since aggregation is *associative*. For example, flow graph F_3 aggregates y and w into type R and then aggregates x and R, while F_1 groups together x and y into P which is then aggregated with w.

3. Port types might not be aggregated at all. For example, flow graph F_4 is a variation of flow graph F_1 in which no aggregation is done. A special case of this type of variation is the variation due to the choice of which compositions of Spreads with Makes (and vice versa) to simplify. For example, flow graph F_5 results from the simplification of F_1's composition of a Spread with a Make.

Aggregation-Equivalence

We would like the flow graphs $F_2, ..., F_5$ to be in the language of the grammar of Figure 3-15, not just F_1. To describe the relationship between these flow graphs, we define the equivalence relation *aggregation-equivalent* on flow graphs.

First, we need to define the following terms.

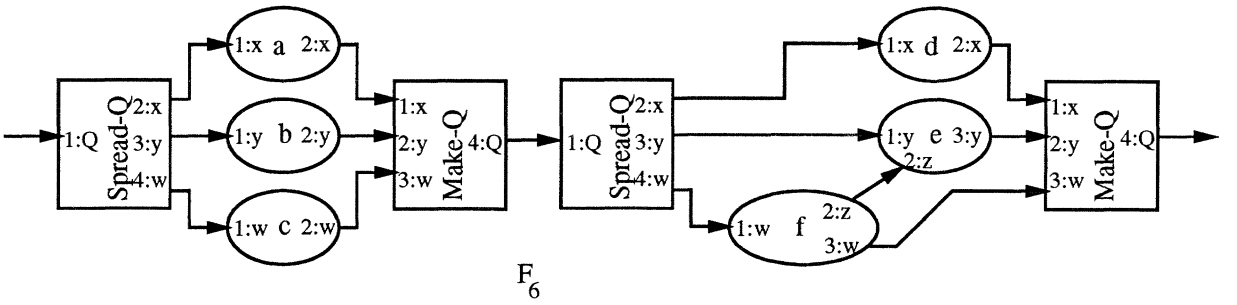
- A *Make-of-Spread composition* is a Spread node connected to a Make node of corresponding type via edges between their corresponding part type ports. More precisely, a Make-of-Spread is a corresponding pair of Make and Spread nodes, such that $\forall i = 1, ..., m$, the i^{th} output of the Spread node connects directly to the i^{th} input of the Make node and there are no other edges adjoining these ports (where m is the number of part port types aggregated).

- A *Spread-of-Make composition* is analogous. It is a Make node connected to a Spread node of corresponding type via an edge between the Make's output port and the Spread's input port.

Now we can define the reflexive, symmetric, transitive relation *aggregation-equivalent*. A flow graph F_1 is aggregation-equivalent to another F_2 (denoted $F_1 \equiv_A F_2$) if and only if there exists a flow graph F_3, such that F_1 and F_2 can each be transformed to a flow graph isomorphic to F_3, using a (possibly empty) sequence of the following transformations:

1. For some corresponding pair of Spread and Make node types, T_S and T_M, permute the outputs of all (Spread) nodes of type T_S and the inputs of all (Make) nodes of type T_M, keeping connections intact and using the same permutation for all the Spreads and Makes. (The flow graphs F_1 and F_2 in Figure 3-16 can be transformed into each other using this transformation.)

2. For all compositions of Spread nodes, replace the composition sub-flow graph with a single Spread whose output arity, m, is the number of outputs of the sub-flow graph and $\forall i = 1, ..., m$, the i^{th} output of the new Spread has the same port type and



F_6

Figure 3-17: F_3 and F_1 can be transformed to this flow graph by flattening nested Makes and Spreads.

connections as the i^{th} output of the sub-flow graph. Flatten all compositions of Make nodes analogously. (This can be used to transform F_1 to F_6 (shown in Figure 3-17) and F_3 to F_6, so $F_1 \equiv_A F_3$ in Figure 3-16.)

3. For any Make-of-Spread composition, replace the Make-of-Spread composition with edges from the ports adjacent to the input of the Spread to the ports adjacent to the output of the Make.

4. For any Spread-of-Make composition, replace the Spread-of-Make composition with new edges drawn in the following way: $\forall i = 1, ..., m$ connect the ports adjacent to the i^{th} input of the Make to the ports adjacent to the i^{th} output of the Spread (where m = the Make's input arity = the Spread's output arity). (F_5 results from applying this transformation to F_1 in Figure 3-16.)

5. Remove any Spread node whose input is an input of the flow graph and remove any Make node whose output is an output of the flow graph. (F_5 can be transformed to F_4 by using this transformation and by removing the Spread-of-Make composition.)

Transformations 1 and 2 allow variation due to commutativity and associativity of aggregation, respectively, while conditions 3 and 4 allow variability in the simplification of Spread-Make compositions. Transformation 5 is needed to allow flow graphs, like F_4, that use no aggregation to be in the language of a grammar that aggregates port types.

We will call the first transformation the *permutation* transformation, since it permutes the part port tuples of Makes and Spreads. The rest of the transformations are *aggregation-removal* transformations. We will call the inverse of aggregation-removal transformations *aggregation-introduction* transformations, since they insert Spreads and Makes into a flow graph.

We can use the aggregation-equivalence relation to expand what we mean by the language of a flow graph grammar. If we call the set of flow graphs derivable from the graph grammar (using the "derives" relation defined in Section 3.4.1) the "core" language of the

grammar, then we can define the *language* of the grammar to consist of all flow graphs aggregation-equivalent to flow graphs in the core language.

Useful Definitions and Facts

A flow graph F_1 is said to be *less-aggregated* than another F_2 if and only if F_1 can be generated from F_2 by applying any of the aggregation-removal transformations above. This relation is transitive. If there is no flow graph less-aggregated than a flow graph F, then F is said to be *minimally-aggregated.*

There is only one minimally-aggregated flow graph less-aggregated than or isomorphic to a particular flow graph that can be obtained by the aggregation-removal transformations. (However, there may be more than one minimally-aggregated flow graph less-aggregated or isomorphic to a particular flow graph F that is aggregation-equivalent to F. These can be transformed into one another by applying the permutation transformation.)

Whether the minimally-aggregated flow graph has any Spreads or Makes depends on whether the formalism allows ports on terminal nodes to have aggregate port types. If terminal nodes have no ports of aggregate type, then minimally-aggregated flow graphs will have no Spreads or Makes.

To see this, suppose we have a minimally-aggregated flow graph F, with a Spread or Make node n. The node n cannot be on F's fringe since otherwise it could be removed by Transformation 5 to create a flow graph less-aggregated than F. So, n must be an internal node. It must also be flat (i.e., it is not nested with another Spread or Make node), since otherwise Transformation 2 could be applied to create a less-aggregate flow graph. Since n is internal, its aggregate port p_1 is connected to another port p_2, which must be of aggregate port type. However, p_2 must be the aggregate port of a node of corresponding Make or Spread type, since only Spreads and Makes can have ports of aggregate type. This would mean F contains a Spread-of-Make composition, which means F is not minimally-aggregated. Therefore, a minimally-aggregated flow graph cannot contain a Spread or Make node if there are no aggregate port types allowed on terminal nodes.

On the other hand, if terminal nodes have ports of aggregate type, then minimally-aggregated flow graphs might have one or more Spread or Make nodes. Using reasoning similar to that above, we can see that all Spread or Make nodes would be internal and flat, with their aggregate port connected to ports on terminal nodes that are not Spread or Make nodes.

These facts are useful in developing a recognizer for languages of flow graph grammars that aggregate port types.

Recognizing Aggregation-Equivalent Flow Graphs

A generator or parser for the language of a flow graph grammar may perform the permutation, aggregation-introduction and aggregation-removal transformations as steps in their

derivation or reduction sequence. Because there are many possible orderings in which to apply the transformations and because doing this efficiently involves an extension to the embedding relation of the graph grammar formalism, it is important to discuss how such a recognizer is constructed. (A generator for the language is not described here, since we are more interested in building recognizers for languages than we are in constructing language generators, for the purposes of program recognition. A generator can easily be imagined by reversing the recognition process.)

One way a recognizer for the language can work, given an input flow graph F, is in two stages. The first would apply some sequence of the permutation, aggregation-removal and aggregation-introduction transformations to F to produce a flow graph F', while the second would apply a recognizer for the core language to F'. A flow graph F would be recognized if a sequence of transformations is found which yields a new flow graph F' that is accepted by a recognizer for the core language. Unfortunately, the first stage could involve a great deal of search to find the appropriate transformation sequence.

A more promising approach is to divide up the stages differently so that no choices need to be made. In the first stage only aggregation-removal transformations that work "downward" by creating less-aggregated flow graphs are applied until a minimally-aggregated flow graph is obtained. Then in the second stage, the aggregation-introduction and permutation transformations are interleaved with the reduction actions of the recognizer for the core language. The idea is that the grammar rules can provide guidance as to what to aggregate and how to organize the aggregation so that the flow graph will be recognizable as a member of the core language. The aggregation guidance is found in the Spreads and Makes of the rule's right-hand side. This section gives the details of how the interleaving of recognition with aggregation-introduction transformations works.

This is explained first for a restricted formalism in which no terminal nodes have ports of aggregate port type and the union port type `Any` is a union of only primitive (non-aggregate) port types. This simplifies the discussion since each minimally-aggregated flow graph in the language of the graph grammar contains no Spreads or Makes.

Then a second formalism is considered in which the restriction is relaxed to allow the type `Any` to be a union of all port types (including aggregate port types). This formalism is still restricted in that the only (possibly) aggregate port type a (non-Spread, non-Make) terminal node's port may have is `Any`. In this case, the minimally-aggregated flow graphs in the graph grammar's language might contain Spreads and Makes. However, as discussed above, these Spreads and Makes will each be flat and internal. Each Spread node must have its input aggregate port connected to a port of type `Any`. The same must be true for each Make node's output aggregate port.

```
(DEFUN POP-TWICE2 (STK)
  (LET* ((FIRST (AREF (STACK-ELTS STK)
                      (STACK-PTR STK)))
         (NEW-STK (MAKE-STACK :ELTS (STACK-ELTS STK)
                              :PTR (1+ (STACK-PTR STK))))
         (SECOND (AREF (STACK-ELTS NEW-STK)
                       (STACK-PTR NEW-STK)))
         (NEWER-STK (MAKE-STACK :ELTS (STACK-ELTS NEW-STK)
                                :PTR (1+ (STACK-PTR NEW-STK)))))
    (VALUES FIRST SECOND NEWER-STK)))

(DEFUN POP-TWICE (A I)
  (LET* ((FIRST (AREF A I))
         (NEW-I (1+ I))
         (SECOND (AREF A NEW-I))
         (NEWER-I (1+ NEW-I)))
    (VALUES FIRST SECOND A NEWER-I)))
```

Figure 3-18: Two programs each performing two consecutive Stack Pops.

What the Restrictions Mean in the Program Recognition Application

These two restricted formalisms are sufficient for capturing the types of aggregation that arise in dataflow graphs representing programs that operate on aggregate data structures.

Allowing only non-aggregate port types on terminals, although restrictive, is still very useful in representing a wide class of programs and clichés in the program recognition domain. For example, the minimally aggregated flow graph for both of the programs shown in Figure 3-18 is given in Figure 3-19. (Attributes are not shown.) Each program can be recognized as a Stack Pop, followed immediately by another Stack Pop, where the Stack is implemented as an Indexed Sequence aggregate data cliché whose parts are an Index (an integer) and a Base (a sequence).

(When we create the minimally-aggregated flow graph representing a program that uses user-defined aggregate data structures, we remove Spread and Make nodes, which contain naming information that is useful for presenting the results of recognition. We convert this information to another form (attributes). See Section 4.2.3 for a discussion of how this information is used.)

The second less-restrictive formalism is useful in representing programs in which aggregate data structures are collected into primitive data types such as arrays and lists (in Common Lisp). The accessors and constructors of these primitive data types (e.g., `CAR`, `CONS`, `AREF`) are primitives. They cannot be treated like Spreads or Makes of aggregate data structures that have fixed, named parts, because their "parts" are accessed and inserted

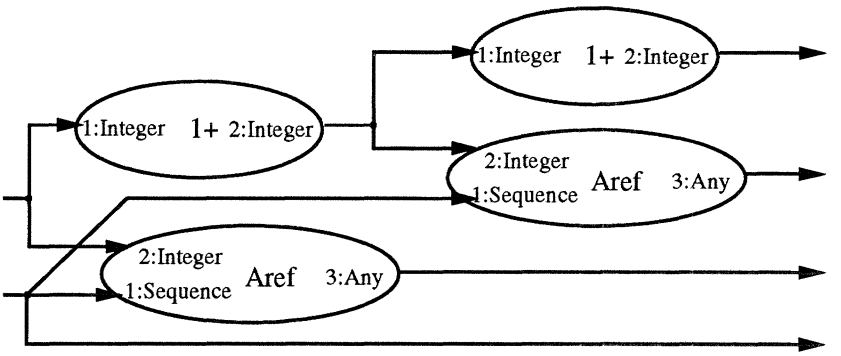


Figure 3-19: The flow graph for the programs `POP-TWICE` and `POP-TWICE2`.

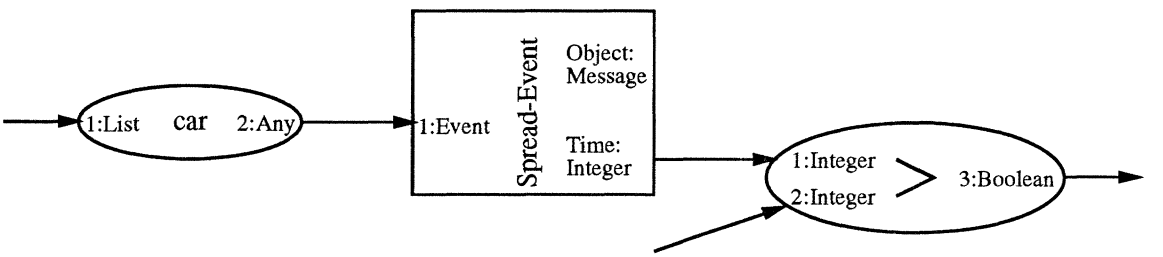


Figure 3-20: Flow graph with a node whose output port is of type Any.

at variable, computed positions. These primitive accessors and constructors have ports of type `Any`.

For example, the code fragment `(> New-Time (Event-Time (car Event-Queue)))` is part of a program for inserting a user-defined data structure, called an Event, into a Priority Queue which is implemented as an Ordered Associative List. The Event has parts Time (an integer) and Object (a Message, which is a user-defined type). The Event is treated as a priority queue element, whose priority is the Time part. This code fragment is testing whether the first element of the input list, Event-Queue, has a Time part less than the value of `New-Time` (which is the Time of the event being inserted).

The attributed flow graph representing this code fragment is shown in Figure 3-20. Its `CAR` has an output of type `Any`. (Rather than numeric port labels, the Spread in this example uses mnemonic names, such as Time, for clarity.)

No Aggregate Port Types on Terminals

This section shows how the actions of a recognizer for the core language are interleaved with aggregation-introduction transformations in a formalism that does not allow ports of aggregate type on terminal nodes.

Since minimally-aggregated graphs have no Spreads or Makes, the Spreads and Makes in the right-hand sides of rules cannot be matched. Only a sub-flow graph of the right-hand side can be matched to nodes in the input graph. This sub-flow graph, called the

non-aggregated rhs, consists of the subset of nodes that are not Spreads or Makes and the subset of edges connecting their ports.

Since right-hand sides of rules are assumed to contain no internal Spreads and Makes, the non-aggregated rhs is the right-hand side graph minus its boundary Spreads and Makes. These boundary Spreads and Makes contain valuable information about how the inputs and outputs of the non-aggregated rhs should be aggregated to recognize a left-hand side that has aggregate port types. We move this information into the embedding relation. We remove the boundary Spreads and Makes so the right-hand side of each graph grammar rule becomes the non-aggregated rhs.

Recall that the embedding relation, as described so far, relates left-hand side ports to right-hand side ports and other left-hand side ports. (That is, C is a binary relation on $\mathcal{L} \times \mathcal{R} \cup \mathcal{L}$, where $\mathcal{L}$ and $\mathcal{R}$ are the sets of left- and right-hand side ports, respectively.) A single left-hand side port can correspond to a non-empty set of right-hand side and left-hand side ports, while a single right-hand side port can correspond to at most one left-hand side port.

We extend this embedding relation to relate each left-hand side port to a *tuple* of right-hand side and left-hand side port sets, where the position in the tuple is significant. More precisely, the embedding relation C is now on $\mathcal{L} \times (2^{\mathcal{R} \cup \mathcal{L}})^n$ where n varies. (A left-hand side port and each right-hand side port in the tuple related to it are still said to "correspond" with each other.)

The right-hand side ports are tupled and related to the left-hand side ports based on the fringe Spread and Make nodes that are removed from each rule's right-hand side. When a Spread node of output arity m is removed, the left-hand side input port corresponding to its input port becomes related to a tuple in which $\forall i = 1, ..., m$ the i^{th} element of the tuple is the set of right-hand side ports (if any) connected to the i^{th} output of the Spread. Similarly, when a Make node of input arity m is removed, the left-hand side output port corresponding to its output becomes related to a tuple, in which $\forall i = 1, .., m$, the i^{th} element of the tuple is the set of right-hand side ports (if any) connected to the i^{th} input of the Make.

The rule for A in Figure 3-21a becomes the rule shown in Figure 3-21b when Spreads and Makes are removed. Left-hand side port A_1 is related to the tuple of right-hand side ports $< \{a_1, d_1\}, b_1 >$. This is shown by tupling the Greek annotations associated with each left-hand side port to reflect the aggregation of right-hand side ports corresponding to the left-hand side port. (For simplicity, elements of tuples that are singleton sets degenerate to the single element of the set in drawings. Tuples containing one element degenerate to that one element.)

If any Spread node has an output j that connects directly to an input k of a Make node, then a st-thru results between the left-hand side ports (l_1 and l_2) that originally corresponded with the input of the Spread and the output of the Make, respectively. Specifically, the j^{th} element of the tuple corresponding with l_1 contains l_2 and the k^{th} element of the

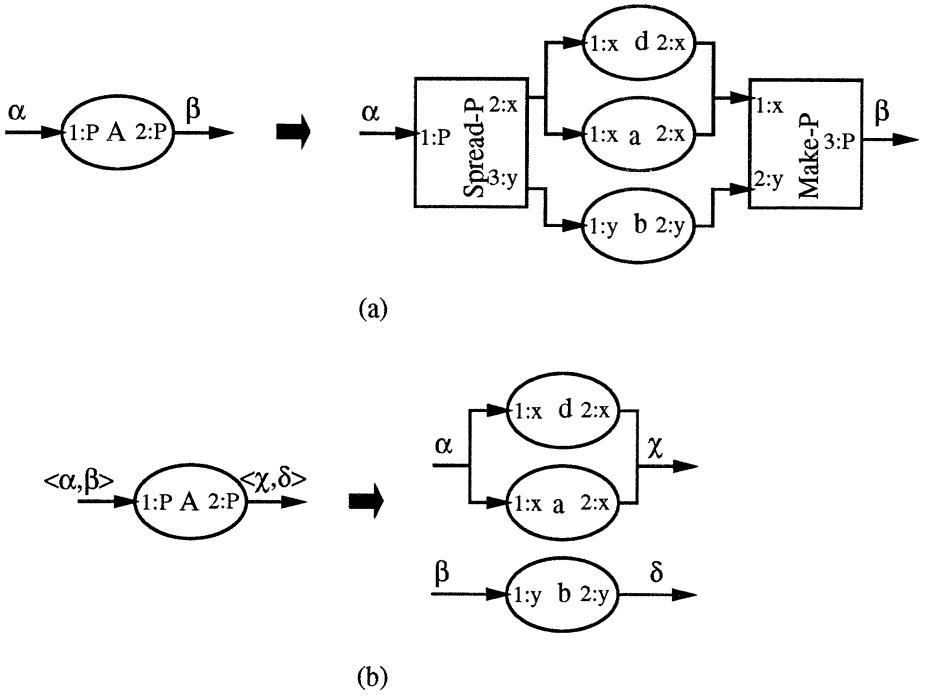


Figure 3-21: (a) A rule which aggregates port types. (b) The same rule with aggregation information moved to the embedding relation.

tuple corresponding with l_2 contains l_1.

This is illustrated in Figure 3-22 where the rule in part (a) is converted to the rule of part (b) which contains a st-thru. A_1 corresponds with A_2 in part y of aggregate port type P.

Relation To Concrete Application Domain: St-Thrus in Data Aggregation

This case arises quite frequently in the program recognition domain. Operations on aggregate data structures in which *all* parts of the data structure are used and/or changed are rare in the simulator programs. Most operations work on only a subset of the parts. For example, the operation for removing the first element from the clichéd aggregate data structure Circular Indexed Sequence (abbrev. CIS) accesses only four of its five parts and changes only two parts. As shown in Figure 3-23, the CIS data structure has a Base, which is a sequence, a Size, which is an integer, a Fill-Count, which is an integer count of the number of elements in the CIS, and two index pointers (First and Last), which are positive integers that specify the indices of the first and last elements in the CIS. The removal operation uses the CIS's First part as an index into its Base part to retrieve the first element. Then the First part is updated by being incremented or decremented (depending on the direction of growth), modulo the Size part. The Fill-Count is also decremented. The Last part is not used or changed. Also, the Base and Size parts are used but not changed. So,

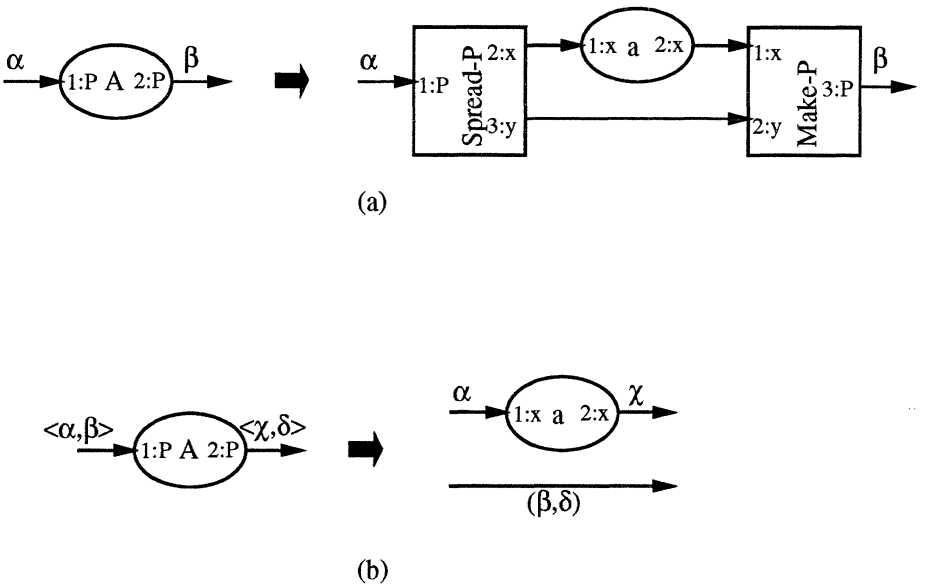


Figure 3-22: (a) An edge connects a Spread and Make. (b) This edge becomes a st-thru when aggregation information is moved to the embedding relation.

there are three st-thrus in the rule for CIS Extract, representing the Last, Base, and Size parts. The rule for CIS Extract is shown in Figure 3-24. (The CIS part names corresponding to the elements of the tuples of correspondence labels are shown in the lower left-hand corner.)

Using the Embedding Relation in Reduction

The embedding relation plays a key role in reduction which is at the heart of the recognition process. A flow graph is recognized if it can be reduced to a single node having a start type. Reduction steps are analogous to rewriting (or generation) steps. Rather than rewriting an occurrence of the left-hand side of a rule to a sub-flow graph isomorphic to the rule's right-hand side, we reduce an isomorphic occurrence of the right-hand side to an instance of the left-hand side. In both cases, the embedding relation is used to determine how to connect the replacement sub-flow graph to the rest of the graph, called the *host graph.*

The following is only a conceptual description of the reduction mechanism. While a recognizer can be implemented to perform exactly these actions, it is not necessary that it do so. In most generators, recognizers, and parsers, the flow graph is not destructively transformed at each derivation or reduction step. The rewriting or reduction is simulated in the state of the generator, recognizer, or parser. This allows backtracking and multiple results to be formed (e.g., for ambiguous grammars).

Recall that the unextended embedding relation is used as follows. When a sub-flow graph R is reduced to an instance of a rule's left-hand side L, an edge is created between a port p_i in the host graph and a port L_j of L, if and only if p_i was connected to a port in R

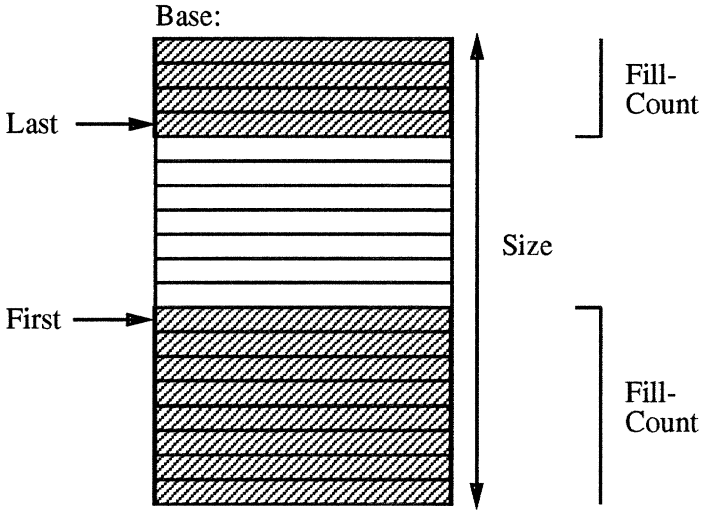


Figure 3-23: Circular Indexed Sequence data structure.

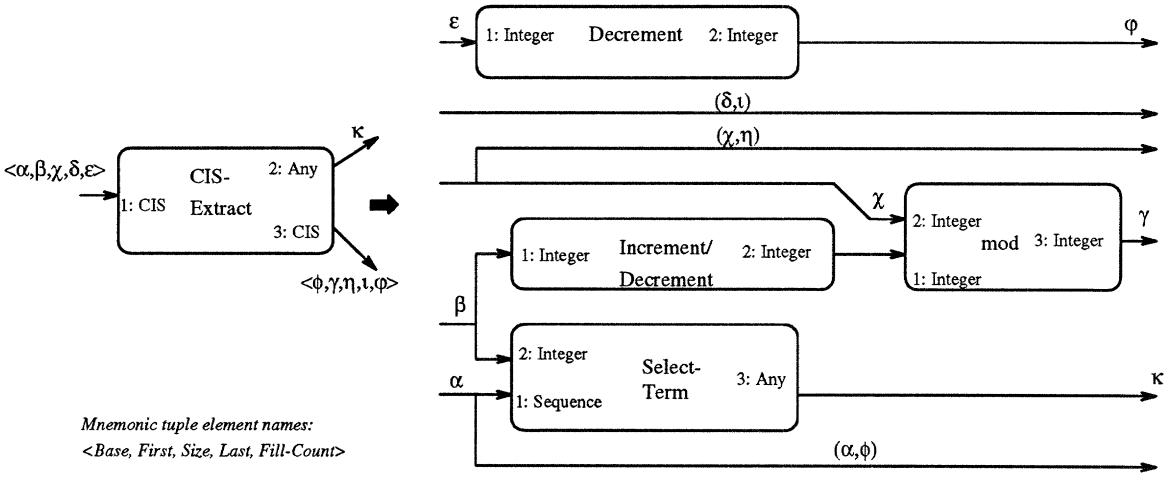


Figure 3-24: The rule for Circular Indexed Sequence Extract.

that corresponds to L_j, according to the embedding relation.

Reduction using the extended embedding relation is more complicated. Several right-hand side ports may correspond to the same left-hand side port, but we do not want all ports in the host graph that are connected to these right-hand side ports to become connected to the left-hand side port when the right-hand side is replaced with the left-hand side. Instead, before we connect the left-hand side instance up to the ports of the host graph, we insert Make and Spread nodes into the graph surrounding the left-hand side to bundle up the inputs and outputs coming from or going to the ports of the host graph.

More specifically, for each left-hand side input port L_j having an aggregate port type, a Make node is inserted. Its output is connected to L_j and its i^{th} input is connected to the host graph ports that are connected to the right-hand side ports in the i^{th} element of the tuple corresponding to L_j. Likewise, for each left-hand side output port L_k having an aggregate port type, a Spread node is inserted. L_k is connected to the Spread's input and the i^{th} output of the Spread is connected to the host graph ports that are connected to the right-hand side ports in the i^{th} element of the tuple corresponding to L_k.

The Make and Spread nodes specify how the minimally-aggregated flow graph should be aggregated to recognize it as the left-hand side of the rule. When the reduction results in a Make-of-Spread composition, the composition is simplified. (Note that Spread-of-Makes are never created by this action.)

For example, the flow graph grammar of Figure 3-15, which expresses aggregation using Spreads and Makes, is converted to the flow graph grammar of Figure 3-25, which expresses aggregation in the embedding relation. A sample reduction sequence using the rules of this grammar is shown in Figure 3-26.

A flow graph is recognized if it is reduced to a flow graph consisting of node of a start type of the grammar, with (possibly empty) trees of nested Makes and Spreads, whose roots are connected to the start type node's inputs and outputs, respectively.

The reduction transformation described here is *simulated* by our parser. Spreads and Makes are not actually added to the graph being parsed (just as the graph being parsed is not destructively reduced). Section 3.5.2 gives details of how the parser does this simulation.

No Aggregate Port Types on Terminals Except "Any"

We now slightly relax the restriction on our formalism that no terminal nodes have ports of an aggregate type. We allow ports of type `Any` on terminal nodes to take on any port type, including an aggregate port type. In this formalism, the minimally-aggregated flow graphs in a graph grammar's language might contain Spreads and Makes which are flat and internal. We call these *residual* Spreads or Makes. Each residual Spread node must have its input aggregate port connected to a port of type `Any`. Likewise, the output aggregate port on each residual Make node must connect to a port of type `Any`.

The main difference this makes to the reduction mechanism is that the simplification

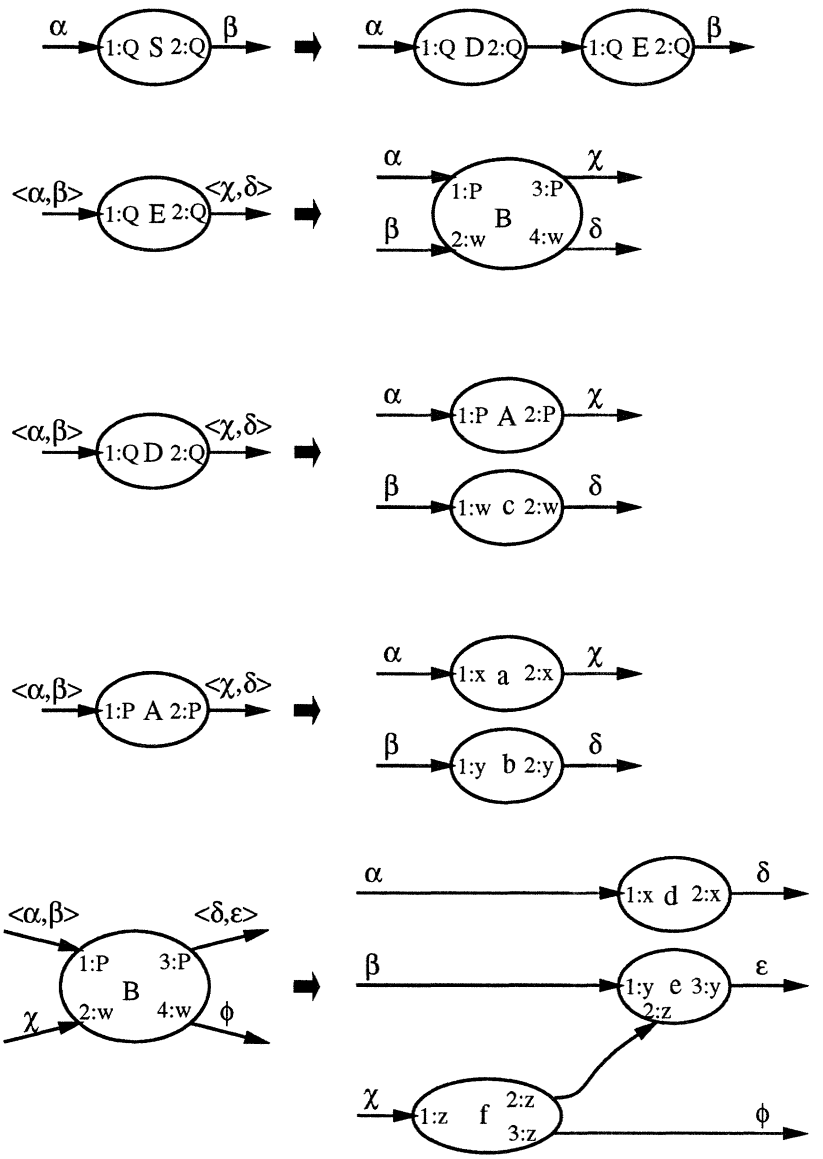


Figure 3-25: The grammar of Figure 3-15 with aggregation encoded in the embedding relation.

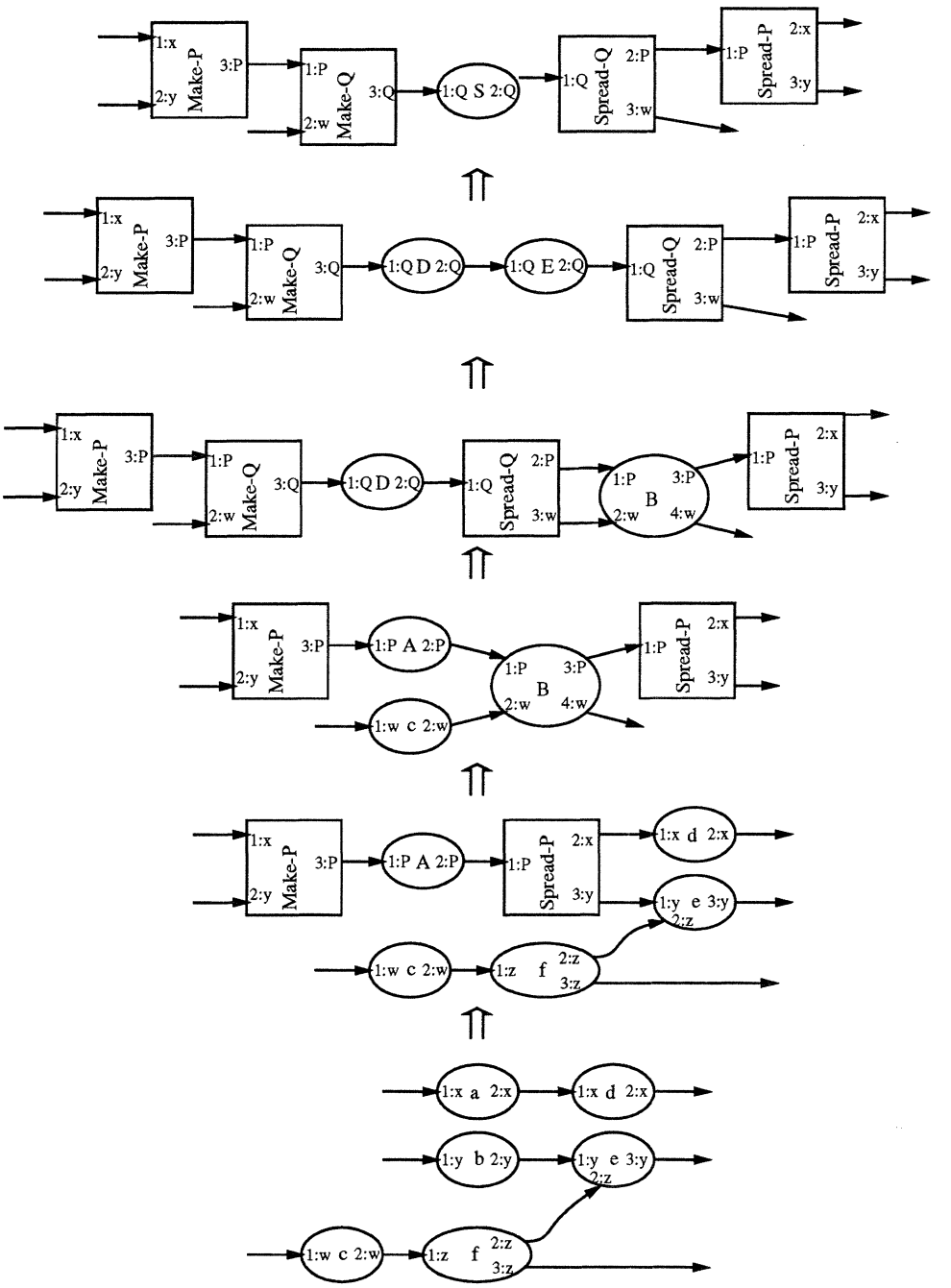


Figure 3-26: A reduction sequence using the grammar of Figure 3-25.

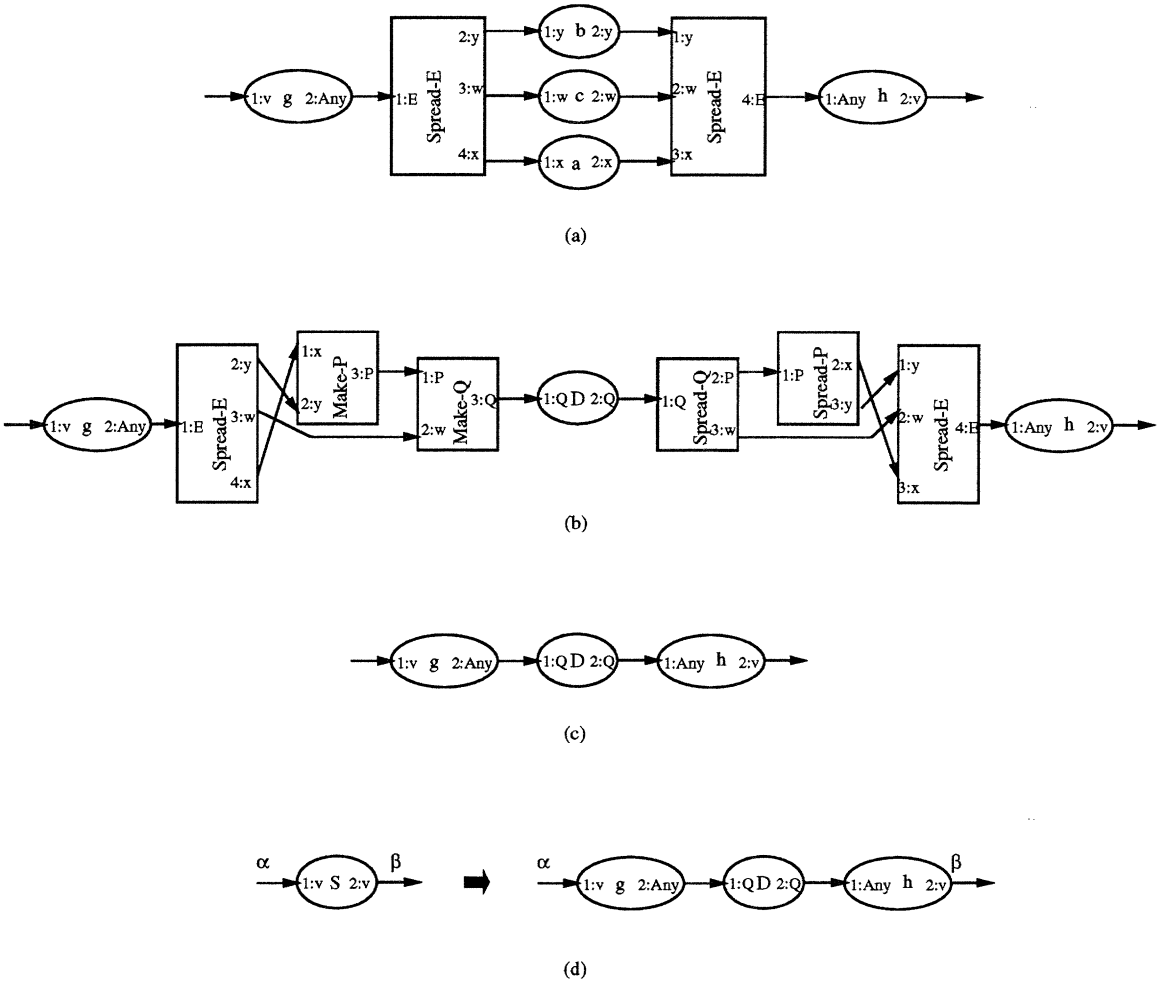


Figure 3-27: The reduction of a sub-flow graph using the rule for D from Figure 3-25.

of Spreads and Makes is not as straightforward. When a sub-flow graph isomorphic to the right-hand side is reduced to a left-hand side with surrounding Makes and Spreads, the Makes and Spreads may become connected to residual Spreads and Makes.

A composition of a Make with a Spread node may arise. However, the Make and Spread will not usually be of corresponding type. The residual Make or Spread may even become connected to a tree of nested Spreads or Makes, respectively. The usual, straightforward Make-of-Spread simplification cannot be applied to this composition.

For example, the sub-flow graph containing nodes a, b, and c in Figure 3-27a is reduced to a non-terminal node of type D, surrounded by Makes and Spreads, using the rule for D from Figure 3-25. The result of the reduction is shown in Figure 3-27b.

There are two solutions to this. One is built on the other and is more powerful in that it allows a useful form of partial recognition to be done. The basic solution is to perform a special-case simplification to the composition. In particular, if all of the outputs of a residual Spread are connected to inputs of a Make or tree of nested Makes (as they are

in Figure 3-27), then we can simplify this composition by drawing an edge from each port connected to the residual Spread's input to each port connected to the output port of the Make or of the root of the Make tree. We can simplify compositions involving residual Makes in an analogous way.

For example, the flow graph in Figure 3-27b would simplify to the one in Figure 3-27c, which can be recognized as an S, whose rule is in Figure 3-27d.

The main limitation of this basic solution is that it does not enable us to handle a form of partial recognition that we find crucial in performing partial program recognition. In particular, we would like to be able to recognize aggregate port types that aggregate only a *subset* of the parts that are aggregated by a port type used in the input flow graph.

For example, suppose we have the flow graph shown in Figure 3-28a and we want to recognize an S in it, whose rule is shown in Figure 3-28b. (Perhaps the flow graph in Figure 3-28a represents a program in which some clichéd operation is being done to some parts (of type x and y) of a user-defined data structure F, where these parts compose a clichéd data structure P. At the same time, the user-defined data structure might contain additional parts (of type m and n) that are keeping track of some statistics, such as how many times the parts of type x and y are accessed. The operations (p and q) to the statistics-keeping parts are unfamiliar and need to be ignored when partially recognizing the program.)

The key to partial recognition of flow graphs is the ability to separate recognizable portions of a flow graph from unrecognizable portions. For partial recognition of a flow graph F to succeed, the recognizable section must be a sub-flow graph of F. (Recall the discussion of Section 3.3.1.) The problem here is that residual Spreads and Makes keep the unrecognizable portion of the input flow graph connected to the recognizable portion, preventing simplification and recognition of a sub-flow graph of the input flow graph.

The reduction of the flow graph using the rule for A yields the flow graph in Figure 3-28c. We cannot simplify the composition of the residual Spread (`Spread-F`) with the Make (`Make-P`) as we do in the first solution because not all of the residual Spread's outputs are connected to the Make's inputs. The same is true for compositions involving residual Makes.

(Note that if there are no aggregate port types on terminal nodes, there are no residual Spreads or Makes. So this form of partial recognition is handled easily in the more restricted formalism.)

To solve this, we make use of the fact that fan-in and fan-out facilitate partial recognition in that unrecognizable portions of a flow graph that fanout from or into ports internal to recognizable portions can easily be ignored simply by not being included in the sub-flow graph matched.

The idea is to break up residual Spreads into two Spreads, one of whose outputs connect to the recognizable portion while the other's outputs connect to the unrecognizable portion. (The input port types of the two Spreads become some brand new type.) The inputs to the Spreads are connected to edges which fanout from the port(s) of type `Any` that connected

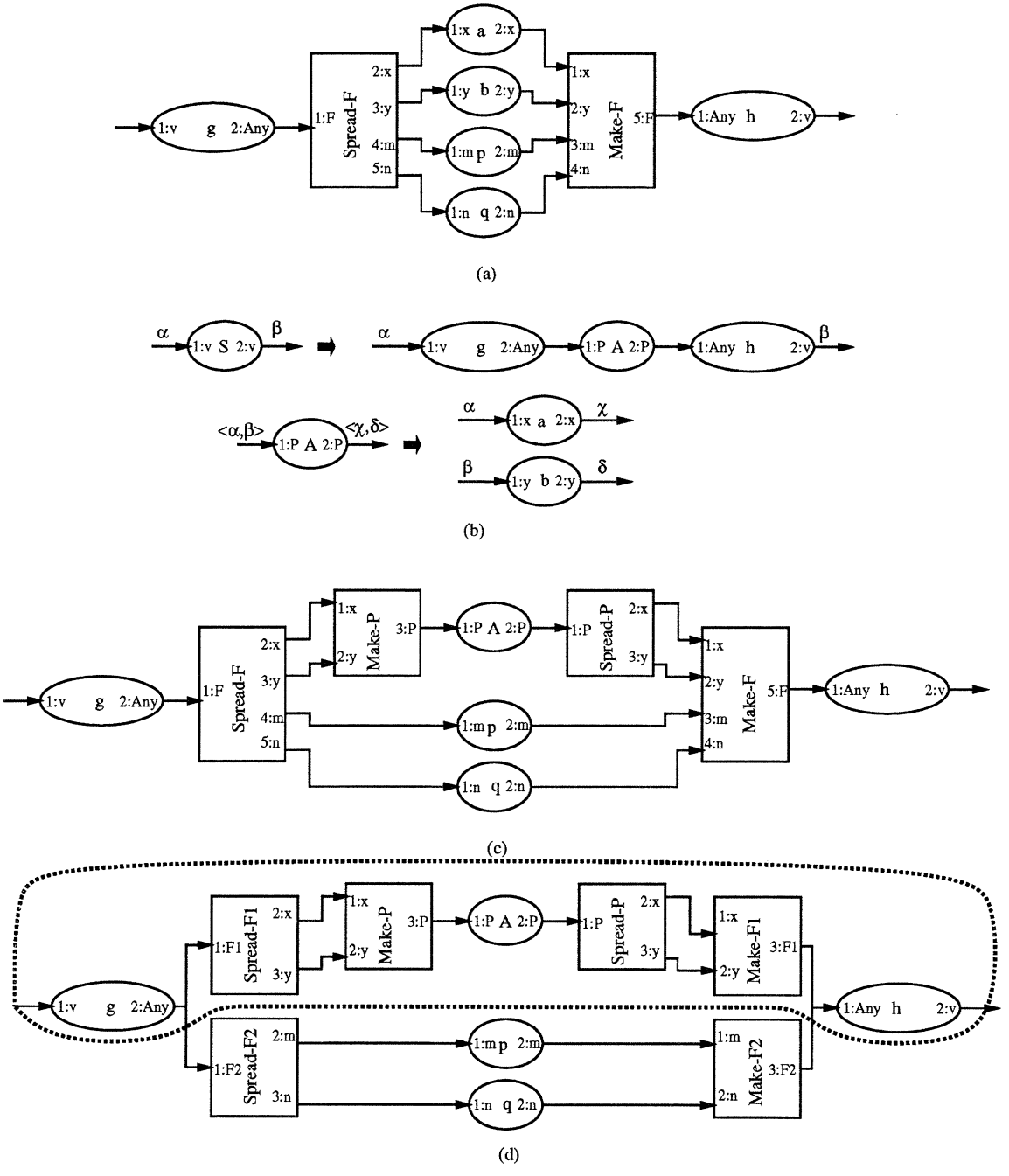


Figure 3-28: (a) A flow graph only partially recognizable as the non-terminal S, whose rule is in (b). (c) Result of reduction. (d) Breaking up residual Spreads and Makes to facilitate partial recognition.

to the input of the original residual Spread. Residual Makes are broken up into two Makes analogously. Thus, we isolate the recognizable portion from the unrecognizable portion by inserting a fan-in or fan-out. For example, the sub-flow graph enclosed in a dashed line in Figure 3-28d can be recognized as an S once the residual Spreads and Makes are broken up.

How a residual Spread or Make is to be broken up is determined by which connections we are trying to make with ports of type `Any`. In other words, the decomposition is not guessed. It is determined by what we are trying to connect together. It may be broken up in more than one way, depending on how many subsets of parts of an aggregate port type can be partially recognized as distinct aggregate port types.

As is the case with the rest of the reduction mechanism discussed so far, this is all simulated in the state of the parser. No graph operations are actually done. See Section 3.5.2 for more details.

3.5 Chart Parsing Flow Graphs

`GRASPR` uses a new graph parser which has evolved from Brotsky's flow graph parser [15]. It also has been influenced by a chart-based flow graph parsing algorithm developed by Lutz [90]. See Figure 3-29. Brotsky's parsing algorithm generalized Earley's string parsing algorithm [32] to flow graphs. Kay [71, 72] and Thompson [132, 133] also generalized Earley's parser to create string *chart parsing*. This was a generalization of the control of Earley's algorithm to allow flexibility in the rule-invocation and search strategies employed. Lutz then generalized string chart parsing to a type of flow graph that is a slightly restricted form of the flow graphs defined in this report. (Section 3.6 explains the difference.) The flexibility of control in Lutz's flow graph chart parsing algorithm has been adopted by the flow graph parser presented here.

An earlier version of our parser (described in [144, 145]) was an extension of Brotsky's parser that allowed it to handle flow graphs that contain edges that fan-in or fan-out. It also dealt with some variations due to structure-sharing (in particular, for parsing flow graphs in which the derivations of two non-terminals overlap). Lutz independently developed more techniques for dealing with structure-sharing variations. These techniques have been incorporated into our parser.

Our formalism further extends that of Lutz and our earlier formalism to include graph grammars that encode aggregation information. Our parser also extends the class of flow graph variations that are tolerated to include variations due to aggregation organization.

The main characteristics of the parser are:

- It deterministically simulates a non-deterministic parser.
- It finds all possible parses and keeps track of all partial analyses.
- It can handle ambiguous grammars.

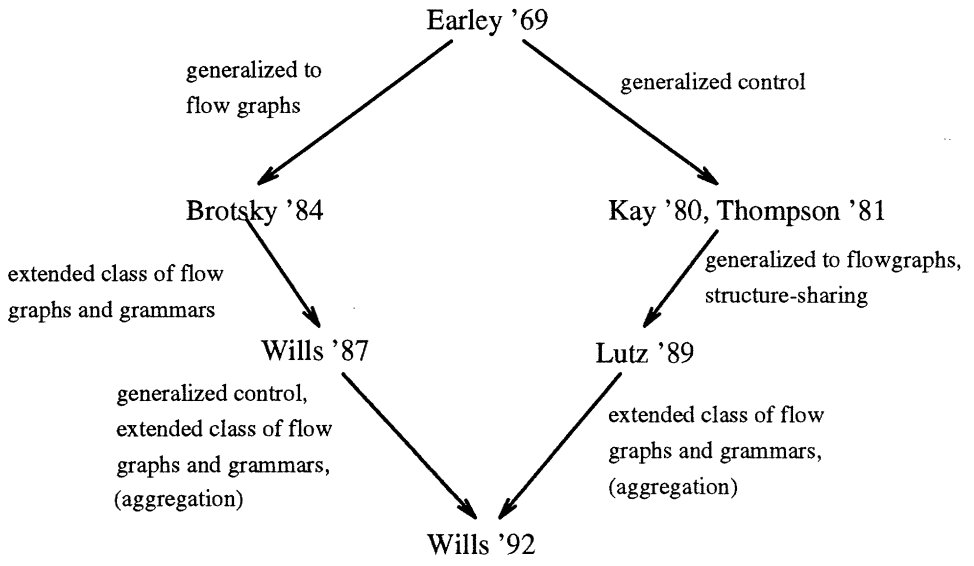


Figure 3-29: Flow graph parser evolution.

- It reuses previously found parses so that it can avoid re-doing work (i.e., it shares subderivations).
- It has a flexible control structure. Its rule invocation strategy (top-down vs. bottom-up) and its search strategy can be specified as part of its inputs.
- The order in which parses are constructed does not matter. (This is useful in being able to incrementally construct parses and to advise the parser to focus on certain parts of its search while postponing others.)
- It is able to make use of analyses it has obtained while parsing to create alternative views of the input graph. This can in turn allow more analyses to be constructed.
- During reduction, it can aggregate not only a set of right-hand side nodes into a single left-hand side non-terminal, but also an aggregation of inputs (or outputs) of a right-hand side flow graph into a single input (or output) of a left-hand side non-terminal.

The Basics of Chart Parsing

Chart parsers maintain a database, called a *chart*, of partial and complete analyses of the input. This is shown in Figure 3-30. The elements in the chart are called *items*. (In string chart parsing, they are called "edges." Lutz [90] calls them "patches.") An item might be either complete or partial. Complete items represent the recognition of some terminal or non-terminal in the grammar. Partial items represent a partial recognition of a non-terminal.

A complete item for a terminal node is created for each node in the input graph during initialization. A complete item for a non-terminal node is created when there are complete

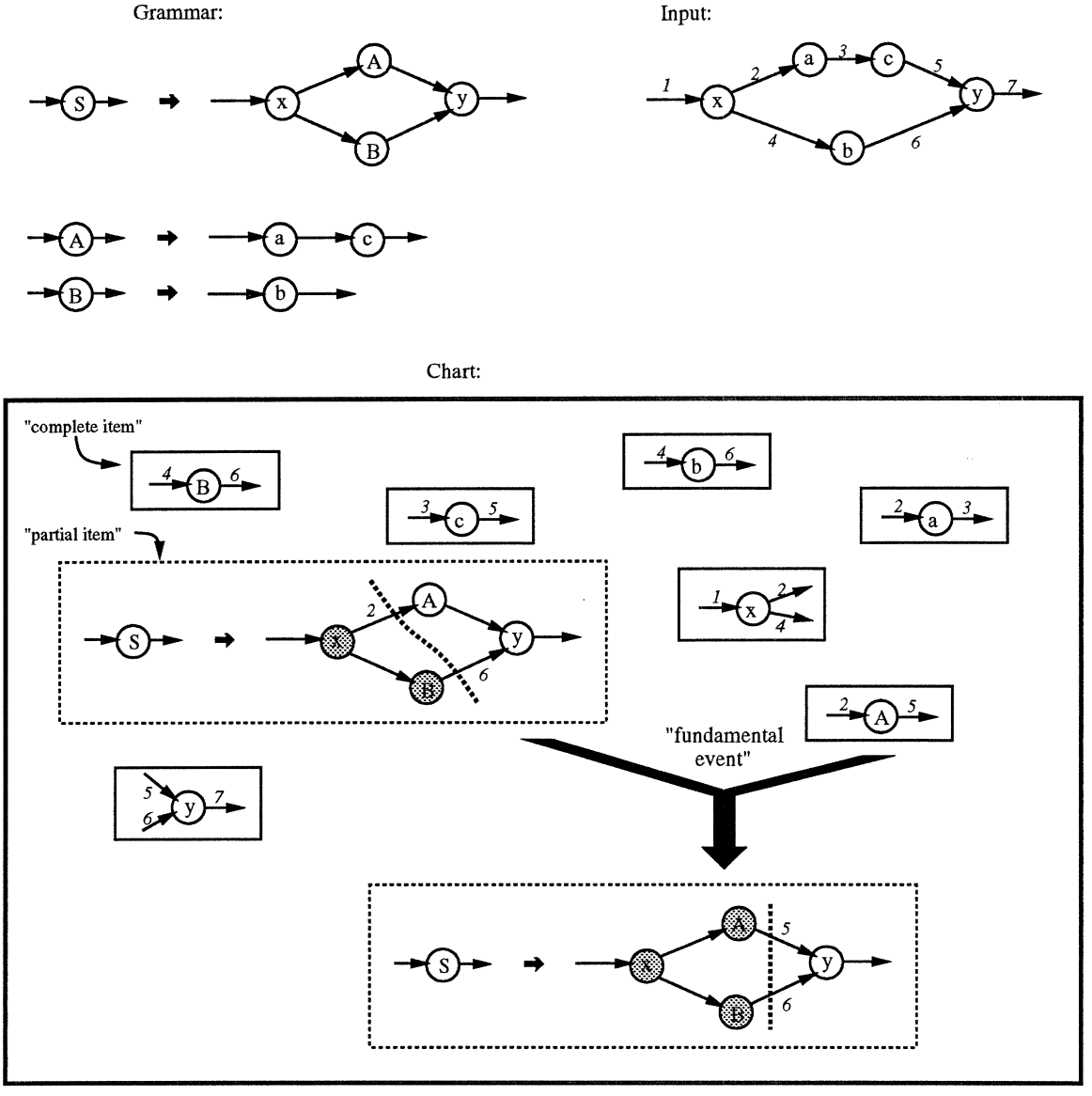


Figure 3-30: Graph chart parsing.

items for each of the constituents of the right-hand side of some rule for the node's type, and the locations of the constituents satisfy the right-hand side's edge connection constraints. Each complete item keeps track of the location in the input graph at which the instance of the node type has been found. It also contains pointers to the subitems on which it depends, as well as other information.

Partial items, on the other hand, contain information about how much of a rule's right-hand side has been recognized so far. It contains a dotted rule, which specifies the non-terminal being recognized, the rule used to recognized it, which constituents have been found, and which constituents are still needed.

Fundamental Event

The most basic operation of a chart parser is to create new items by combining a partial item with a complete one. This is called the *fundamental event.* If there is a partial item that needs a non-terminal A at a particular location and if there is a complete item for non-terminal A at that location, then the partial item can be *extended* with the complete item. During extension, a copy of the partial item is created and augmented. This results in a new item which is added to the chart. (When a partial item is extended with a complete one, they are said to be "combined.") Duplicate items are never added to the chart. This avoids redoing work. (Also, items are never removed from the chart.)

In the string chart parsing literature, the chart is described as a graph. The nodes represent locations in the string being parsed and the edges represent the partial or complete recognition of some terminal or non-terminal between two locations. In string chart parsing, the retrieval of pairs of edges to participate in the fundamental event is based primarily on location. Whenever a partial and complete edge meet (i.e., satisfy the adjacency criterion), the pair becomes a candidate. The set of pairs are then further refined by an *extendibility criterion* (which typically checks terminal or non-terminal types).

In string chart parsers, it makes sense to use the adjacency criterion as the first filter in retrieving pairs of edges to be combined. It only requires looking up the edges that start at a particular node in the chart (graph). Then the extendibility criterion can be applied to these edges.

However, in graph parsing, the "edges" (items) are between *sets* of ports. The adjacency criterion now requires that the inputs and outputs of the completed item be a *subset* of the outputs and inputs (respectively) of the partial one. Since there can be many possible pairs of items that satisfy this criterion, we use part of the extendibility criterion to help retrieve pairs of items to combine. Additional constraints have been added to the extendibility criterion as a way of narrowing down the search for analyses. For example, some of the non-structural constraints on attributes have been incorporated into the criterion. The choice of which constraints to include depends on the cost of checking the constraints at this point in the parsing. (See Section 6.2.2.)

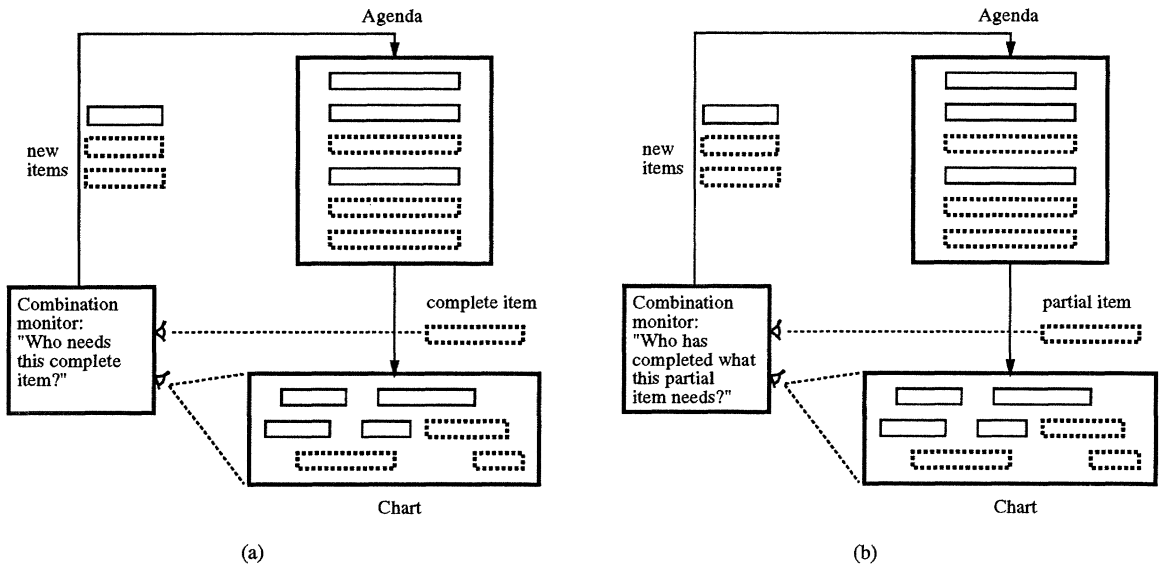


Figure 3-31: (a) Adding a complete item to the chart. (b) Adding a partial item to the chart.

Agenda-Based

In chart parsers, an agenda is used to queue up the items to be added to the chart. Items are continually pulled off the agenda and placed in the chart. As an item is added, it is paired with other items with which it can be combined. If the item being added is a complete item, then it is paired with partial items that need it. On the other hand, if the item added is a partial item, then it is paired with any complete items for the non-terminals it needs. These two cases are illustrated in Figure 3-31.

The agenda makes it easy to control which things are added to the chart and when they are added. This explicit control can be used to enforce a particular rule invocation strategy or search strategy.

For example, we can make the parser adopt a bottom-up parsing strategy, as shown in Figure 3-32. Whenever a complete item is added to the chart, new empty items can be added to the agenda for each rule that needs the complete item to get started (i.e., the rule has a minimal right-hand side node that is of the same type as the type derived by the complete item). The new item is instantiated at a location that depends on the location of the complete item.

Likewise, we can achieve a top-down parsing algorithm. First, during initialization, empty items must be added for each rule that derives a start type of the grammar. (An "empty" item is a partial item that needs complete items for *all* of its rule's right-hand side constituents.) For each such rule, an empty item must be instantiated at each of the possible matchings of the inputs of the input graph to the inputs of the rule's left-hand side. Second, whenever a partial item is added to the chart, a new empty item must be added to

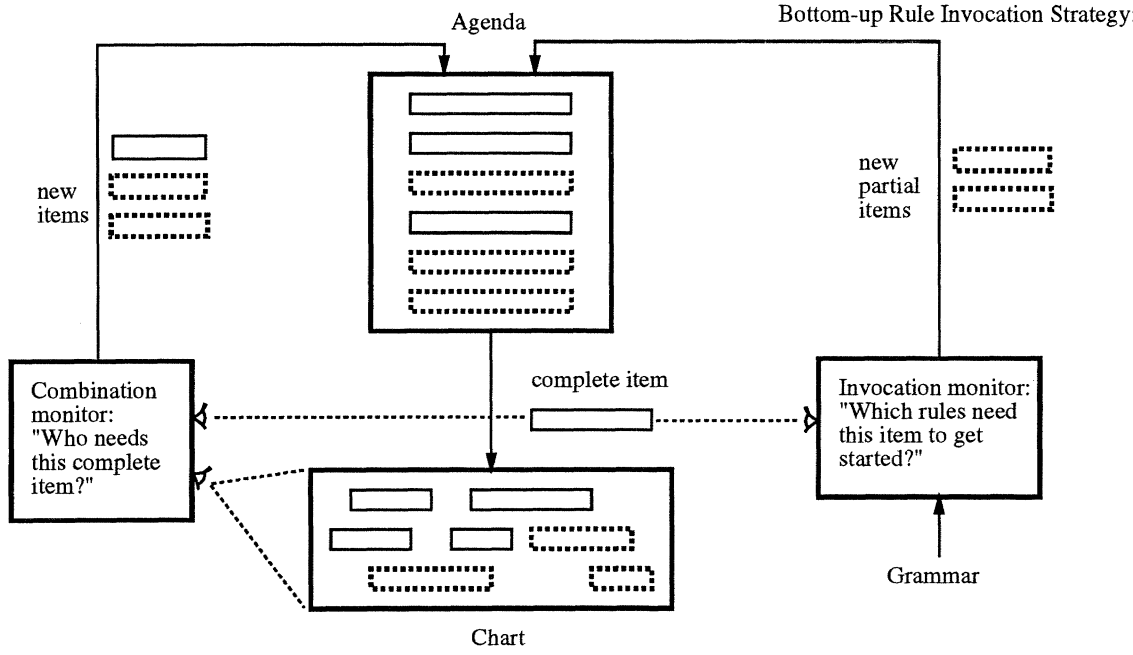


Figure 3-32: A bottom-up rule invocation strategy affects adding a complete item to chart.

the agenda for each rule that derives a non-terminal needed by the partial item. The new item must be instantiated at a location that depends on where the partial item needs the non-terminal constituent.

(In the current program recognition system, we use only a bottom-up strategy, since this facilitates partial recognition. This also makes it easier to recognize non-terminals for which there are rules with mismatching arity between the left-hand and right-hand sides. This is necessary in handling rules whose right-hand sides have inputs (representing constants) that do not correspond to left-hand side input ports. Allowing a right-hand side to have more inputs and outputs than the left-hand side is also crucial in allowing the type of embedding relation that encodes aggregation relationships. A top-down strategy would require that we predict the organization of aggregation when each empty item is first instantiated (before the item's rule's right-hand side is matched). In other words, it requires searching for the appropriate sequence of aggregation-introduction transformations needed to recognize the flow graph, as discussed in Section 3.4.2.)

The way in which the agenda is maintained determines not only the rule invocation strategy, but also the parser's search strategy. While we can control whether the parsing algorithm proceeds top-down or bottom-up by controlling what gets added to the agenda, we can choose a particular search strategy (e.g., depth-first or breadth-first), simply by controlling the order in which items are pulled off of the agenda. The agenda might be maintained as a first in, first out (FIFO) queue to achieve breadth-first search, for example.

The strategy for maintaining the agenda can be given by the user. It is one of the ways

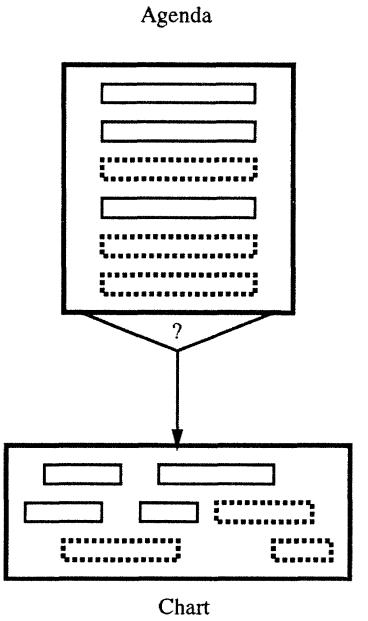


Figure 3-33: Search strategy as input to parser.

advice from an expectation-driven component or a human user can be incorporated into the code-driven component. See Figure 3-33.

The parser is guaranteed to find every parse exactly once, no matter which rule invocation or search strategy is used.

Additional Monitors

One final aspect of the architecture of the parser is that it contains additional monitors that watch the chart. See Figure 3-34. These detect the existence of certain kinds of items or collections of items in the chart which can be used to generate other items. In particular, they look for opportunities to view part of the input graph in an alternative way in order to yield more parses. The graph is not explicitly changed to the alternative view. Instead, new items are created which represent the alternative views and these are added to the agenda.

An example of this is employed in simulating the zipping up of an input graph as explained in Section 3.5.1, which describes how share-equivalent flow graphs are recognized.

Selectively Trying Harder

We do not necessarily want the parser to generate all of the alternative views of the input graph. So, the opportunities for generating new items representing these views are queued on an agenda. These opportunities can be selectively pulled from the agenda and performed. The parser can be given advice from an external agent about how and when to make the selection. The parser can be made to incrementally try harder. It can report

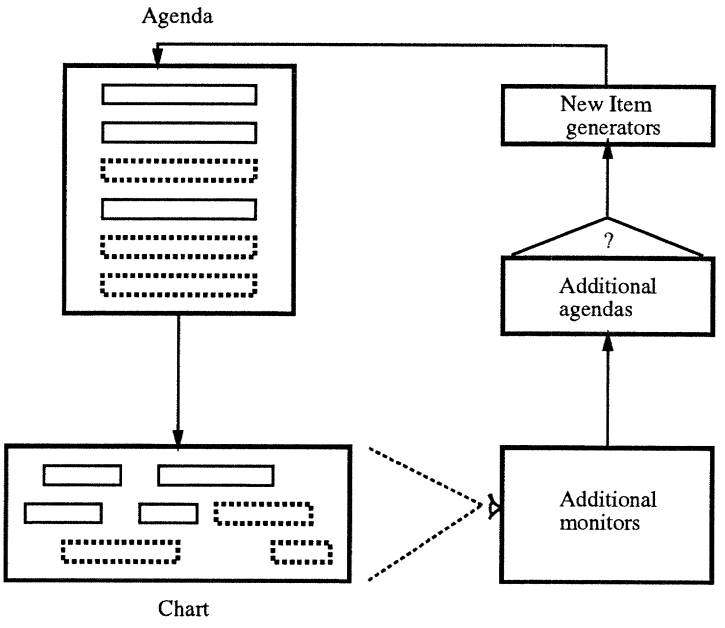


Figure 3-34: Additional monitors.

easy recognitions early, and then be given more time later to generate alternative views that uncover the obscured clichés. So, quick results can be obtained, without sacrificing completeness in the long run.

The parser can also be directed to generate alternative views only within a certain area of the input graph. For example, if no clichés were found in a particular area of the input graph, the parser could try generating alternative views in that area in case this would allow more clichés to surface.

Asking for Advice

The monitors might also detect question-triggering patterns in the chart. These are patterns that indicate that a particular constraint is likely to hold. This is useful if the constraint is costly for the parser to check. When such a pattern is found, the recognition system can ask whether the constraint is satisfied. The question might be more easily answered by some other source (such as an expectation-driven component in a hybrid recognition system).

Now that the basic operation of the chart parser for flow graphs has been described, the next three sections give details of how the extensions to the formalism and st-thrus are handled.

Motivations for Copying Before Extension

Each time a partial item is extendable by a complete one, a copy of the partial item is created and the copy is extended. There are three reasons that the parser extends a *copy* of partial item, rather than the original. One is that the parser is leaving itself open to

the possibility of *ambiguity.* It might be possible in the future for the partial item to be extended with another complete item for the same right-hand side node. By not changing the original partial item, the parser continually has a partial item that can accept alternative derivations for its immediately needed nodes.

The alternative complete item need not be a duplicate of the first. If both satisfy the constraints of the partial item, with respect to its matching so far, then both can extend the partial item. For example, the two complete items might have overlapping locations, but if the partial item only constrains the location that is shared by the two items, both can extend the partial item. So the parser is using copying to deal with partial ambiguity.

The second reason is that copying facilitates *partial recognition.* When a complete item is recognizing a partial item's immediately needed node that is on the left fringe, then extending a copy of the partial item allows the partial item to be extended with a different complete item, representing an instance of the left-fringe node at a different location in the input graph. (This is a special case of ambiguity.)

A third reason to copy before extending is that this facilitates *incremental analysis* [149]. There are two forms of incremental analysis. One is incrementally analyzing a static input graph. This is achieved in chart parsing by iteratively adding complete items for each of the input graph's nodes to the chart. A depth-first retrieval of items from the agenda can ensure that all partial analyses of the input graph considered so far are created before another node of the input graph is considered (i.e., the complete item for the node is added to the chart).

The other type of incremental analysis is useful to do when the input graph is changing. (This might happen when the recognition system is being used to aid maintenance, for example.) It involves updating the results of a previously parsed input graph to account for a modification to the input graph. This type of incremental analysis requires 1) creating analyses of the new sub-flow graph and incorporating them into the existing analyses, and 2) retracting analyses that depend on the old sub-flow graph that has changed. Augmenting existing analyses based on the new information is another case of the first type of incremental analysis. Retracting analyses that are no longer valid involves first finding the items to retract and then doing the retraction.

Copying before extension makes doing the retraction of an item easy. All partial items whose copies were extended with the item are still around, unmodified. They represent intermediate states in the search for an analysis, before the complete item advanced the search. Retraction of an item can be done by "killing" the item in the chart and each partial item it extended, as well as their item tree descendants. The original partial item will remain.

Finding the items to retract requires keeping track of dependencies between the input graph's structure (and attributes) and the items that represent recognitions of it. Most of this dependency information is contained in the item's structure in the form of links to sub-items that represent its components. The leaves of these links are the items for terminal

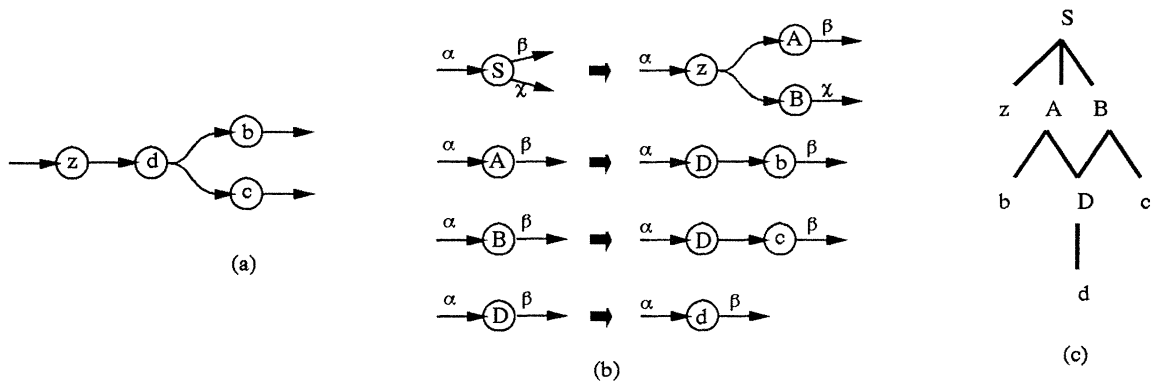


Figure 3-35: Sharing a sub-derivation.

nodes in the input graph. However, more dependency information must be maintained than is in the current implementation. If any edges are added or attributes are changed, constraints might no longer be satisfied. The information of how items depend on the nodes, edges, and attributes of the input graph is important not only in deciding which items to retract, but also which previously failing items or item combination attempts might now be valid. So this dependency information is also relevant in the incremental addition of analyses and the augmentation of existing analyses.

3.5.1 Recognizing Share-Equivalent Flow Graphs

Recall from Section 3.4.1 that a recognizer or parser for a structure-sharing flow graph grammar may work by interleaving zipping and unzipping transformation steps with the usual reductions steps. Our chart parser *simulates* this introduction in two ways. First, unzipping the input graph is simulated by allowing sub-derivations, in the form of sub-items, to be shared. For example, suppose we give the parser the input flow graph shown in Figure 3-35a with the grammar of Figure 3-35b. Once the parser creates a complete item for D, it is shared between the items for A and B. Parsing yields the derivation graph shown in Figure 3-35c.

Second, zipping up the input graph is simulated using a "zip-up" monitor. For example, an input flow graph might redundantly contain two instances of the same non-terminal A, where the inputs and/or the outputs of the two instances fan out from or into the same port(s). (See Figure 3-36b.) The right-hand side flow graph that we are looking for might maximally share a single instance of the non-terminal (as does the rule for S in Figure 3-36a). We would like to view the input program as maximally sharing the two instances of A, so that the right-hand side flow graph will match. This is done by generating an item for A that "zips up" the two items for A that were created. (See Figure 3-36c.) The location and sub-items of the new zipped up item is the union of the locations and sub-items (respectively) of its zip-up components.

Also, the attribute values of the zipped up item's left-hand side are computed based on those of the zip-up components. The attribute combination function associated with each attribute held by the zip-up components' left-hand sides is used to compute a new value of the attribute. In particular, for each attribute a_i associated with the left-hand side's non-terminal type, a_i's combination function is applied to the attribute values held for a_i by the left-hand sides of the zip-up components. (The attribute combination functions may be partial functions. If the function is not defined for the attributes of some left-hand sides whose items are being zipped up, then the zip-up attempt fails.)

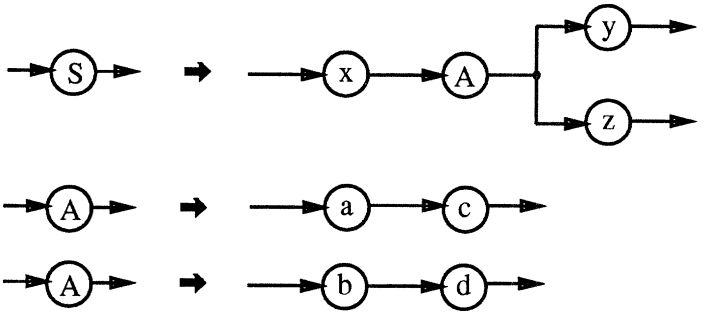
3.5.2 Recognizing Aggregation-Equivalent Flow Graphs

Following the discussion of Section 3.4.2, this section describes the recognition of aggregation-equivalent flow graphs first for the restricted formalism in which no terminal has an aggregate port type and then for the less restrictive formalism. Recall that the recognition process for the restricted formalism included "inserting" Spread and Make nodes whenever an isomorphic occurrence of a right-hand side is reduced to a left-hand side non-terminal node with aggregate ports. The Spread and Make nodes serve to bundle up the edges surrounding the non-terminal node. The recognition process also "simplified" any Make-of-Spread composition that results from the insertion of Spreads and Makes. These actions are simulated by the flow graph chart parser.

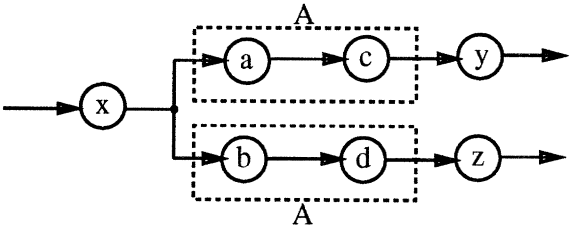
In particular, items keep track of where the right-hand side is found, using a set of *location pointers*, which indicate which edges correspond to the inputs and outputs of the right-hand side of the item's rule. To represent the addition of a Make or Spread, the location pointers are placed in tuples, which are nested in tree structures. The nested tuples reflect the organization of the aggregation of the edges to which they refer. An element of the tuple can be either another tuple or a set of location pointers. (A set of more than one location pointer represents fan-in or fan-out.) When items are combined, their location pointers are compared to see if they represent a Make-of-Spread that simplifies correctly. The corresponding parts of the tuples are compared. If both parts are tuples, they are compared recursively. If both are sets, the sets must have a non-empty intersection for the comparison to succeed. If one is a set and the other a tuple, the comparison fails.

For example, Figure 3-37a shows the flow graph in the language of the grammar in Figure 3-25, whose reduction is shown in Figure 3-26. Location pointers are shown as integers annotating the edges and edge stubs. Figure 3-37b shows the items created by the parser in parsing this graph. The nested tuple on the input in the item for D, for instance, represents the nested Make nodes "inserted" during the reduction sequence of Figure 3-26. The creation of the complete item for S shows the comparison between the nested tuples on the output of D and the input of E.

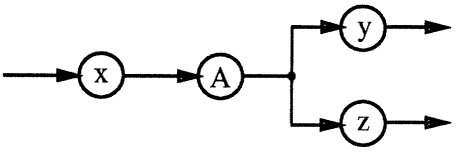
Note that the simulation method used by the parser relies on using a bottom-up rule invocation strategy. It compares the tuples of location pointers that are organized based



(a)



(b)



(c)

Figure 3-36: (a) A graph grammar that maximally shares the non-terminal A. (b) An input flow graph containing two redundant instances of A. (c) An alternative view created by "zipping up" the input graph.

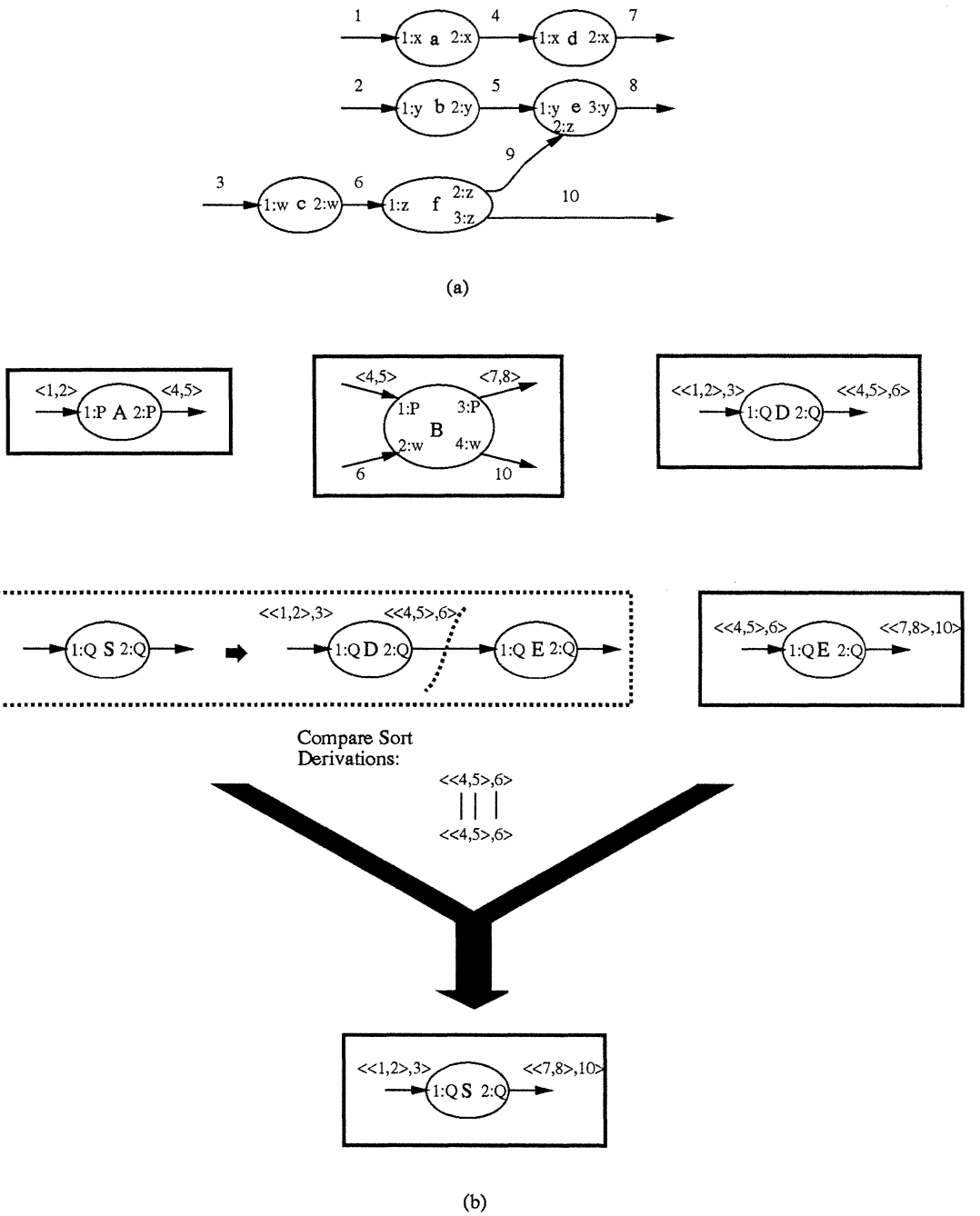


Figure 3-37: (a) A flow graph with location pointers. (b) Items created during parsing.

on the recognition of a rule's right-hand side, rather than predicting the organization and then verifying it by trying to match the right-hand side at the predicted location.

We now consider recognizing flow graphs in the less restrictive formalism in which there still are no aggregate port types on terminal nodes, but the type `Any` is a union type of aggregate and non-aggregate types. Recognition involves a special-case simplification of compositions of residual Makes (or Spreads) with the nested Spreads (or Makes) that are "inserted" during reduction. Recall that to perform partial recognition, in which parts of an aggregate port type used in the input graph are ignored, we need to "break up" the residual Spreads (or Makes) so that recognizable portions of the flow graph are separated from unrecognizable portions.

This is simulated in the state of the parser, using operations on the location pointers of items. Residual Spreads and Makes are removed from the input flow graph. They are replaced with fan-out and fan-in, respectively.

(As is discussed in Section 4.2.3, some of the information found in residual Spreads and Makes is useful for generating documentation about which data structure clichés were found in a program and how their parts relate to user-defined structures' parts. This information is placed in attributes on the fan-out or fan-in edges that replace a Spread or Make.)

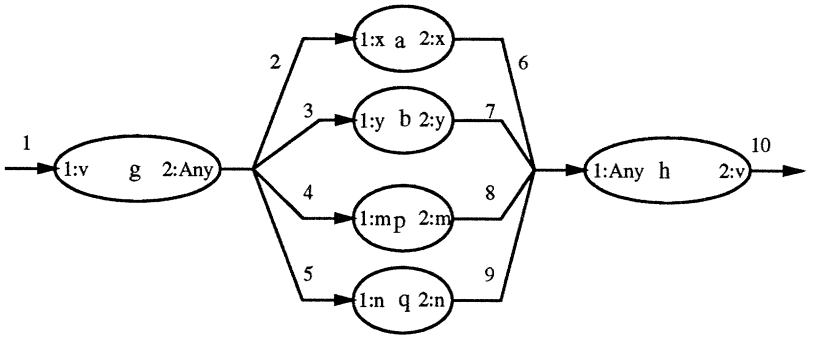
In the combination operation, a nested tuple of location pointers "inserted" during reduction of a rule's right-hand side may be compared with a flat, unordered set of location pointers, representing the fan-out or fan-in edges that replaced a residual Make or Spread. The combination is valid if for each list L_p of location pointers in the fringe of the tree formed by the nested tuple, at least one location pointer in L_p is a member of the flat set of location pointers. Not all of the pointers in the flat set of location pointers need to be members of some list of location pointers within the nested tuple.

For example, the input flow graph generated from the example of Figure 3-28 is shown in Figure 3-38. In creating a complete item representing the recognition of S, the flat set of location pointers representing the residual Spread, $\{2,3,4,5\}$, is compared with the tuple of location pointers, $< 2,3 >$, representing the aggregation of types x and y into A's input port type P. (See Figure 3-38b.) Likewise, the tuple $< 6,7 >$ is compared with the flat set of pointers $\{6,7,8,9\}$. Both comparisons succeed.

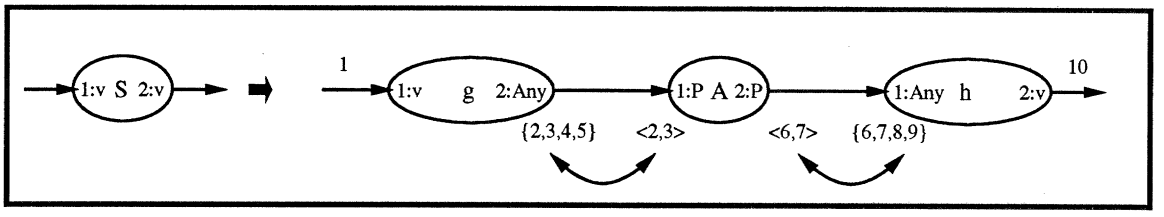
3.5.3 Matching St-Thrus

When two left-hand side ports of a rule correspond with each other in the embedding relation, the rule contains a st-thru. Because st-thrus are part of the embedding relation rather than the right-hand side flow graph, they are not matched in the same way as nodes and edges of the right-hand side. They can possibly match any edge in the input flow graph.

St-thrus impose a global constraint. Suppose a rule for a non-terminal A contains a st-thru involving ports labeled 1 and 3 on A, as in Figure 3-39. If an item completes for A and is combined with a partial item, the complete item places a constraint on the locations



(a)



(b)

Figure 3-38: Simulating the break up of residual Spreads and Makes.

of non-terminals that are connected to A at ports 1 and 3 in the partial item's rule. The constraint requires that these adjacent non-terminals be located at endpoints of the same edge. The st-thru essentially imposes a constraint that the non-terminals connected to A at ports 1 and 3 be connected to each other. (See Figure 3-40.)

St-thrus differ based on whether or not they are structurally constrained and whether or not they are optional. A st-thru is *structurally constrained* if the embedding relation restricts it to matching edges that fan out (or in) with edges coming into (or out of) an isomorphic occurrence of a right-hand side. In other words, a st-thru is constrained if one or both of the two corresponding left-hand side ports also correspond to some right-hand side port.

Structurally unconstrained st-thrus are not restricted in this way. They exist when two left-hand side ports correspond to each other and no other right-hand side port. These types of st-thrus often arise when a right-hand side with Spreads and Makes is translated to a non-aggregated right-hand side. If the output of a boundary Spread connects directly to an input of a boundary Make and neither port connects any other ports, a structurally unconstrained st-thru arises.

We refer to structurally constrained st-thrus as simply "constrained" st-thrus (and structurally unconstrained ones as "unconstrained"), with the understanding that this is referring only to structural constraints. Most st-thrus, including unconstrained ones, have non-structural constraints (in the form of attribute conditions) imposed upon them by their

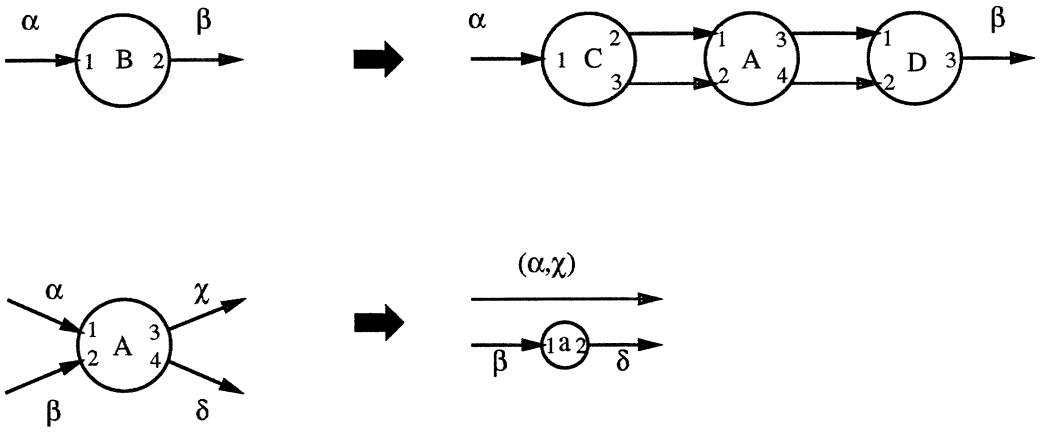


Figure 3-39: Grammar containing a rule with a st-thru.

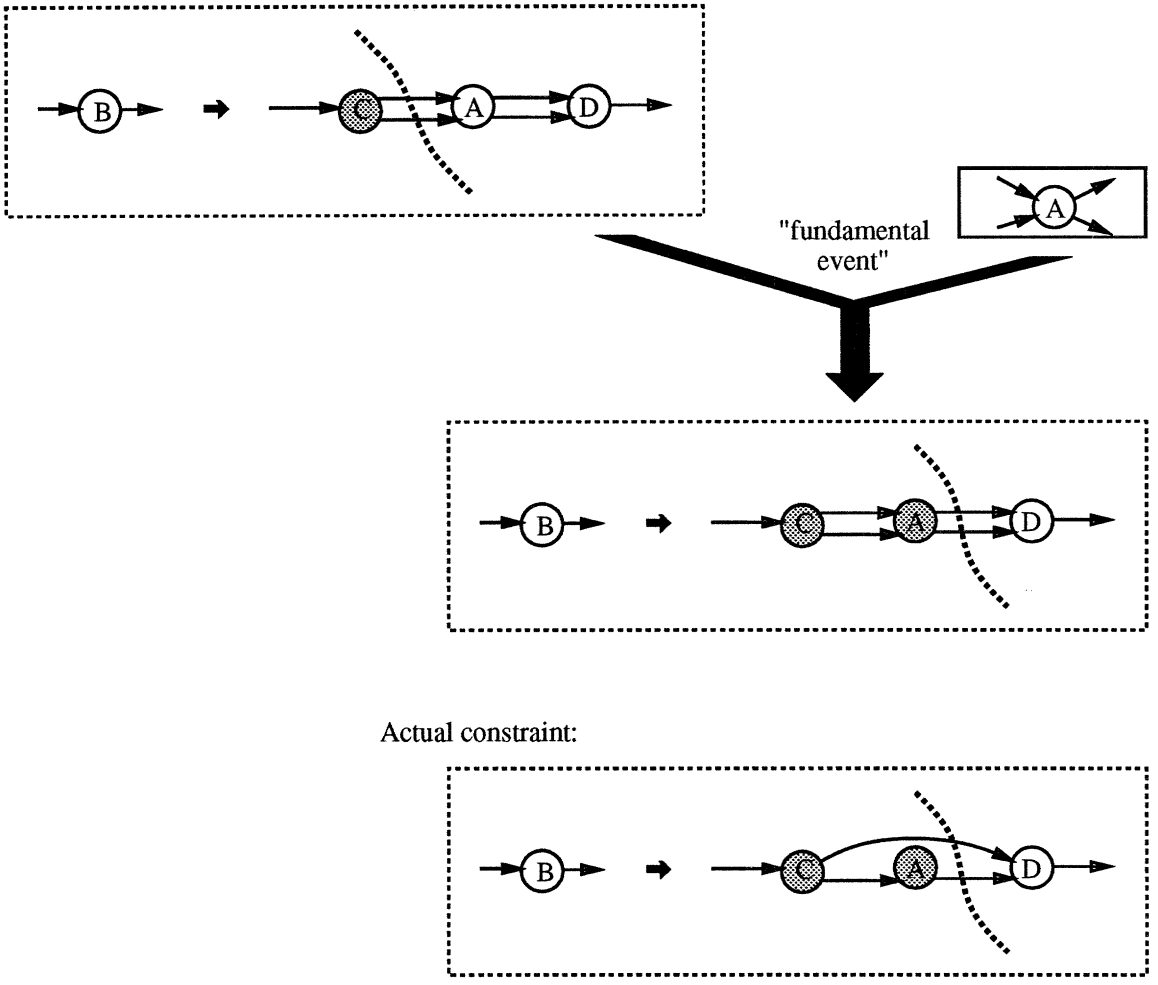


Figure 3-40: Constraint on combination imposed by st-thrus.

rule.

Constrained and unconstrained st-thrus are both matched to a set of edges, which is then narrowed down, based on the context in which its rule's right-hand side is reduced to its left-hand side. An unconstrained st-thru initially matches the set of all edges, while the constrained st-thru matches the subset of edges that satisfy the restrictions imposed by the embedding relation. These sets of matching edges are shrunk as non-structural constraints are checked and the reduction of higher-level non-terminals in the parse tree occurs.

For example, suppose a Circular Indexed Sequence Insert and a Circular Indexed Sequence Extract non-terminal were recognized in the input graph, as shown in Figure 3-41. When the locations of the Insert and Extract non-terminals are compared during combination, the location pointer tuples are compared element-by-element. The First part of the output of CIS Insert represents an unconstrained st-thru and is initially matched to all edges (shown pictorially by a wild-card $*$). During combination, this First part is matched with the First part of the input to the CIS Extract instance. This narrows down its matching set of edges to those indicated by location pointers 10 and 13. The Size part of the CIS Insert output also comes straight through CIS Insert's right-hand side, but because it fans out with the input to MOD, it is constrained to be matched to a small number of edges (those indicated by location pointers 5 and 6).

Global constraints represented by the st-thru are imposed by propagating reductions in sets of matching edges across non-terminals and across edges. For example, once the item for CIS Extract extends the partial item of Figure 3-41, the wild-card matches can be reduced to a small set of matches. Figure 3-42 shows the result of propagation of st-thru match reduction. Now CIS Extract's output constrains the location of its Last part (to location 9), restricting the location at which the second CIS Insert should be found.

Constrained and unconstrained st-thrus can additionally be described as either *optional* or *required.* Required st-thrus must be assigned a match, while optional st-thrus need not.

Optional st-thrus are useful in the program recognition domain, where it is often the case that there is no edge matching a st-thru. This occurs if no operation makes use of the data represented by the st-thru. For example, the edge indicated by the location pointer 18 in Figure 3-41 might not exist if no operation following the CIS Extract uses the Base part of the output CIS. St-thrus representing data structure parts are optional. An example of a required st-thru is that of the rule representing the Negate-if-Negative implementation of the Absolute Value cliché. (See Figure 3-9.)

The only difference this designation makes is in what it means if the reduction of sets of matching edges results in an empty set of possible matches. If the st-thru is required, this empty set means the recognition of the rule's left-hand side failed. Otherwise, the set of possible matches of an optional st-thru can become empty without causing the recognition to fail.

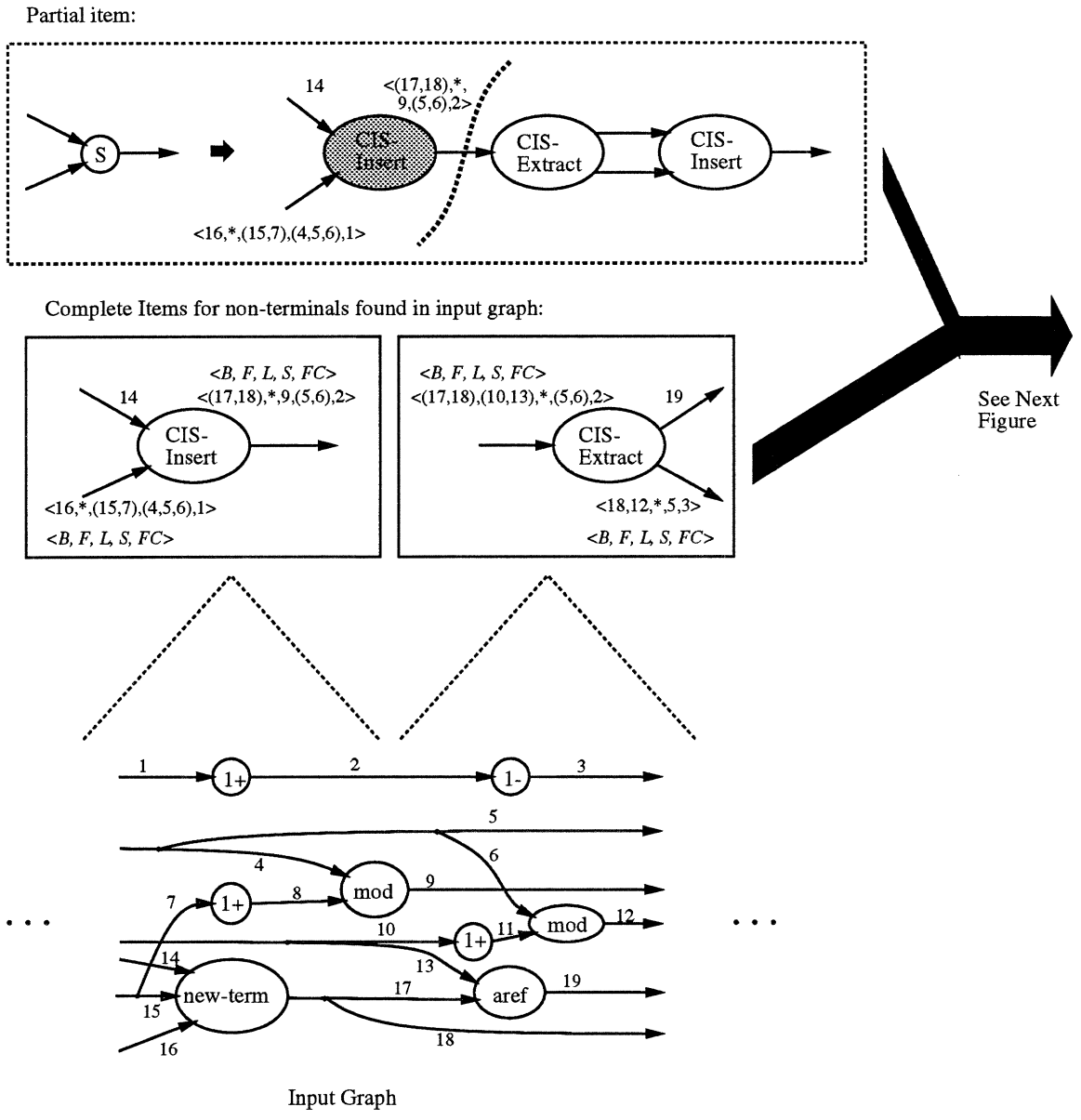
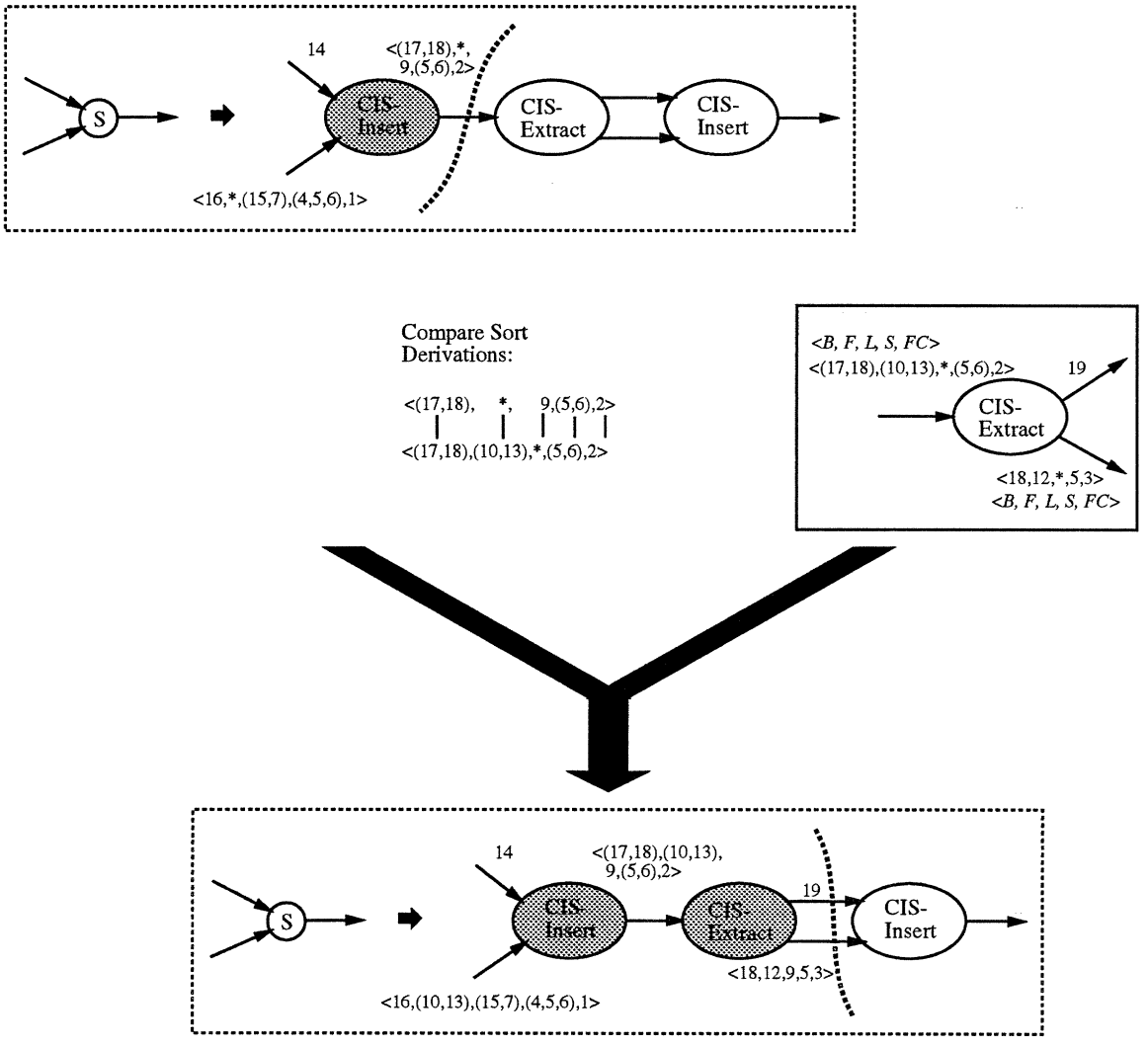


Figure 3-41: Constrained and unconstrained st-thrus.



Resulting partial item. Notice that the location pointers have been propagated to replace the wild-cards.

Figure 3-42: Propagating matches of st-thrus.

3.6 Related Graph Grammar Work

Graph grammars have been used widely in automatic circuit understanding and verification, pattern analysis, compiler technology, and in software development environments. (See [34, 35, 134] for several examples in these areas.)

There are many varieties of graph grammar formalisms. They vary both in the classes of graphs that are generated and by the embedding mechanisms used. In this section, we briefly discuss the classes of graphs commonly studied and relate our flow graphs to them. Then we discuss typical embedding mechanisms. Finally, we describe interesting graph parsers related to ours.

3.6.1 Classes of Graphs

Early graph grammar work focused on traditional graphs, in which nodes do not have distinct entry and exit points ("ports"). This includes work on *webs* and *web grammars* [27, 94, 102, 105, 119]. These traditional types of graphs are also generated by *node-label controlled* (NLC) graph grammars [120] and by the *algebraic rewriting* approaches [23, 33]. (NLC grammars are controlled by node labels (i.e., our node types) in that labels are important in choosing a node to rewrite and in that the embedding relation is defined in terms of labels, rather than specific nodes in a rule's right-hand side or in the host graph. *Edge-label controlled* graph grammars [52, 92] are closely related in that they can simulate NLC grammars.) NLC grammars and algebraic rewriting is discussed further in Section 3.6.2. Their relation to each other is studied by Kreowski and Rozenberg in [80].

Traditional graphs are a special case of graph classes in which nodes have ports. These more general graph classes include Lutz's *flowgraphs* [90] and *hypergraphs* [53], as well as our flow graphs.

Lutz's [90] "flowgraphs" are a special type of our flow graph. They contain, in addition to nodes, ports, and edges, *tie-points* which are intermediate points through which ports are connected to each other. Since each port is connected to exactly one tie-point, fan-in and fan-out are not captured to the same level of granularity as is captured by flow graphs. For example, they cannot express the following situation: an output port p_1 fans out to input ports p_3 and p_4, while output port p_2 is only connected to p_4.

Hypergraphs can be seen as flowgraphs (in Lutz's sense), where nodes in a hypergraph correspond to tie-points and hyperedges correspond to flowgraph nodes. Engelfriet and Rozenberg [36] and Vogler [136] study the relationships between hypergraph grammars and boundary NLC graph grammars. (In boundary NLC grammars, no two non-terminal nodes are neighbors in any right-hand side [121].)

3.6.2 Embedding Mechanism

Our basic flow graph formalism makes use of a simple embedding relation to specify the connectivity of the right-hand side with the host graph when a left-hand side is expanded during derivation. This type of embedding mechanism is quite common. However, in some formalisms, embedding is more complicated.

In NLC rewriting, the connectivity of the right-hand side nodes with the nodes in the "embedding area" (i.e., those nodes adjacent to the left-hand side node being expanded) is determined by a connection relation on node labels (types). In particular, a right-hand side node is connected to a node in the embedding area if their node labels are related by the connection relation. (For example, if label l_1 is related to label l_2 all right-hand side nodes having label l_1 become connected to all nodes of label l_2 in the embedding area.)

In set-theoretic approaches [96], the embedding can involve nodes that are not in the immediate neighborhood of the left-hand side being replaced. The nodes to which the right-hand side nodes are connected are specified by *path expressions*, such as "all nodes that can be reached from the left-hand side node by following an outgoing edge of label k and then an incoming edge of label i." These complicated embedding transformations are used mainly in graph generation (e.g., for specification purposes in software development environments [98, 97]).

Part of each production in the algebraic approach [38] is a set of *gluing points*, which can be edges as well as nodes. Both the left- and right-hand sides of the productions can be graphs containing more than one node. The gluing points are two sets of nodes and/or edges, one for each side of the production. These sets are in bijective correspondence with each other. They remain when the left-hand side is removed and form an anchor for the right-hand side that replaces it. In other words, the embedding relation is captured in the sets of corresponding gluing points.

3.6.3 Graph Parsers

Work on applications of graph grammars has focused mostly on graph generation, rather than analysis. However, recently there has been more interest in developing graph parsers.

Bamji [8, 9] developed a special case of a chart parser for graphs equivalent to Lutz's flow graphs. The interesting aspect of Bamji's graph grammar formalism is that his grammar rules have an embedding relation in which each left-hand side port can be related to a *set* of right-hand side ports. Unlike tuples in our embedding, these sets are not ordered and the right-hand side ports aggregated in them are homogeneous in that they have the same type and are not distinguished by position in the set. The chart parser imposes simple set-intersection conditions between the port sets of adjacent non-terminals in right-hand sides of rules.

Bamji developed this formalism for the purposes of representing and verifying circuit designs. His parser's efficiency is gained by using only deterministic grammars and using

a straightforward rewriting: whenever a right-hand side matches a subgraph, replace it (destructively) with the left-hand side. Bamji's parser does not try to obtain all possible parses, just one is sufficient for verification.

Franck [44] and Kaul [69, 70] study precedence graph grammars. They both present a precedence graph parser which is a straightforward extension of string precedence parsing using the well-known Wirth-Weber precedence relations. Graphs can be parsed in linear time with these parsers. However, precedence graph grammars are restricted to be unambiguous, and uniquely invertible. Precedence techniques may be useful to use on subsets of our graph grammar that have these properties.

Bunke and Haller [18] and Peng, et al. [103] have both developed a parser for plex grammars which are generalizations of Earley's algorithm similar to Brotsky's.

Wittenburg, et al. [150] give a unification-based, bottom-up chart parser which is similar to Lutz's and our chart parser. Grammar rules place a strict (total) ordering on the nodes in their right-hand sides. This ordering determines the order in which items are extended. This creates fewer partial analyses, which is advantageous in terms of efficiency, but is a drawback in terms of generating partial results when the graph contains unrecognizable sections.

Chapter 4

Applying Parsing to Recognition

Chapter 2 described the clichés that we have collected in our library and Chapter 3 described the basics of the parsing technique that we apply to recognize them in a wide range of programs. This chapter fills in the details of encoding programs and clichés in the flow graph formalism and of applying the flow graph parser to the partial program recognition problem. Sections 3.3 and 3.4.2 gave glimpses of how programs and clichés are encoded in the flow graph formalism. In Section 4.1, we review and fill in more details of this encoding. Then in Section 4.2, we complete the picture by providing details of GRASPR's architecture.

4.1 Expressing Programs and Clichés in the Flow Graph Formalism

We use the flow graph formalism to represent programs and programming clichés. In particular, flow graphs serve as graphical abstractions of programs, flow graph grammars encode allowable implementation steps between abstract operations and lower-level operations, and the derivation trees resulting from parsing give the program's top-down design.

The flow graph is used to represent the operations of a program and the dataflow between them. Each non-sink node in a flow graph represents a function, with ports on the node representing distinct inputs and outputs of the function. The ports' types are determined by the signature of the function. Sink nodes represent conditional tests. The edges of a flow graph represent dataflow constraints between the functions and tests. When the result of a function is consumed by more than one function, the edges representing the dataflow fan out. Edges that fan in represent the conditional merging of more than one dataflow. For example, Figure 3-8 shows the attributed flow graph representation of the program RIGHTP, given in Figure 3-7.

Information about a program's control flow, recursion, and data aggregation is captured in the attributes of the flow graph representation of the program. Section 4.1.1 describes the key attributes and conditions used in representing programs and programming clichés.

Attributed flow graphs and grammar rules can become difficult for *people* to read. For

presentation purposes, we make use of a macro-notation, called the Plan Calculus (developed by Rich, Shrobe, and Waters [110, 114, 117, 127, 137]), which graphically summarizes some classes of attributes and conditions, making them more readable. Section 4.1.2 introduces this notation. The Plan Calculus is used here only as a visual aid; the primary representation used by `GRASPR` is the flow graph.

The Plan Calculus aided us in building the cliché library. It formed a representational stepping stone between English descriptions of clichés and their encoding as attributed flow graph grammar rules. It facilitates the capture of relationships between clichés, such as implementation relationships and temporal abstractions. Section 4.1.3 discusses this further.

Section 4.1.4 demonstrates how the event-driven simulation cliché and the clichés it is built upon are expressed in the flow graph formalism. It goes from the English description of the clichés to their Plan Calculus rendering and then to the flow graph grammar rules that `GRASPR` actually uses to recognize `PiSim`.

4.1.1 Attribute Language

Attributes on flow graphs store control flow, recursion, and data aggregation information about a program. In particular, each node has a *control environment* attribute which specifies when the operation represented by the node is executed, relative to when other operations in the program are executed. Nodes in the same control environment represent operations that are performed under the same conditions (so they are each performed the same number of times). These nodes are said to *co-occur.*

Nodes that represent conditional tests have two additional attributes, *success-ce* and *failure-ce.* Operations in the success-ce (resp. failure-ce) control environment are executed when the conditional test succeeds (resp. fails).

Control environments form a partial order. A control environment ce_i is less than or equal to another control environment ce_j (denoted $ce_i \sqsubseteq ce_j$) iff nodes in ce_j are performed at least as many times as those in ce_i. For example, the success-ce of a node representing a conditional test is less than or equal to the control environment of the same node, because operations on a conditional branch are performed less often than the conditional test.

A flow graph representing a recursive function F contains a node whose type is F. This is called the *recursive node.* We assume our recursive functions always have at least one exit test and are singly recursive. (Section 7.2.1 discusses extensions for modeling multiple recursion in the future.) Figure 4-2 shows the flow graph representing the program `HT-Insert` given in Figure 4-1. (This is a simple hash table program in which `Structure` is an array of buckets. Each bucket is a list of strings, ordered lexicographically.) The recursive node is the one labeled "Splice-In-Bucket."

We distinguish three control environments in flow graphs representing recursive functions:

```
(defun HT-Insert (Element Structure)
  (let* ((Key (Hash Element Structure))
         (Bucket (aref Structure Key)))
    (copy-replace-elt (Splice-In-Bucket Element Bucket)
                      Key
                      Structure))))
(defun Splice-In-Bucket (Element Bucket)
  (if (null Bucket)
      (cons Element Bucket)
      (let ((Entry (car Bucket)))
        (if (string> Entry Element)
            (cons Element Bucket)
            (let ((Rest (cdr Bucket)))
              (if (string= Entry Element)
                  (cons Element Rest)
                  (cons Entry (Splice-In-Bucket Element Rest))))))))
```

Figure 4-1: A recursive function with multiple exits.

- *recur-ce* – the top-most control environment of the flow graph representing the recursive function. It is the control environment of the node representing the first operation performed by the recursive function. In Figure 4-2, this is $ce2$.
- *feedback-ce* – the control environment of the node representing the recursive call within the body of the recursive function. In Figure 4-2, this is $ce8$.
- *outside-ce* – the control environment in which the recursive function is called and into which it exits. In Figure 4-2, it is $ce1$. (If the recursive function is analyzed independent of any callers, a new control environment is created to be the outside-ce.)

The feedback-ce and the outside-ce are always $\sqsubseteq$ the recur-ce. Operations performed before the exit test (i.e., in the recur-ce) are always performed more times than the recursive call or the operations done upon exit, since they are performed when the recursion exits as well as when it repeats. If there is only one exit, then the node representing the exit test has the recur-ce as its control environment, the feedback-ce as its failure-ce, and the outside-ce as its success-ce. (If a new control environment had been created to represent the outside-ce, then it becomes equal to the success-ce of the test.)

Summing Incomparable Control Environments

Some subsets of control environments are said to be *incomparable*. In particular, if ce_a and ce_b are the success-ce and failure-ce of the same node, then the set $\{ce_a, ce_b\}$ is incomparable.

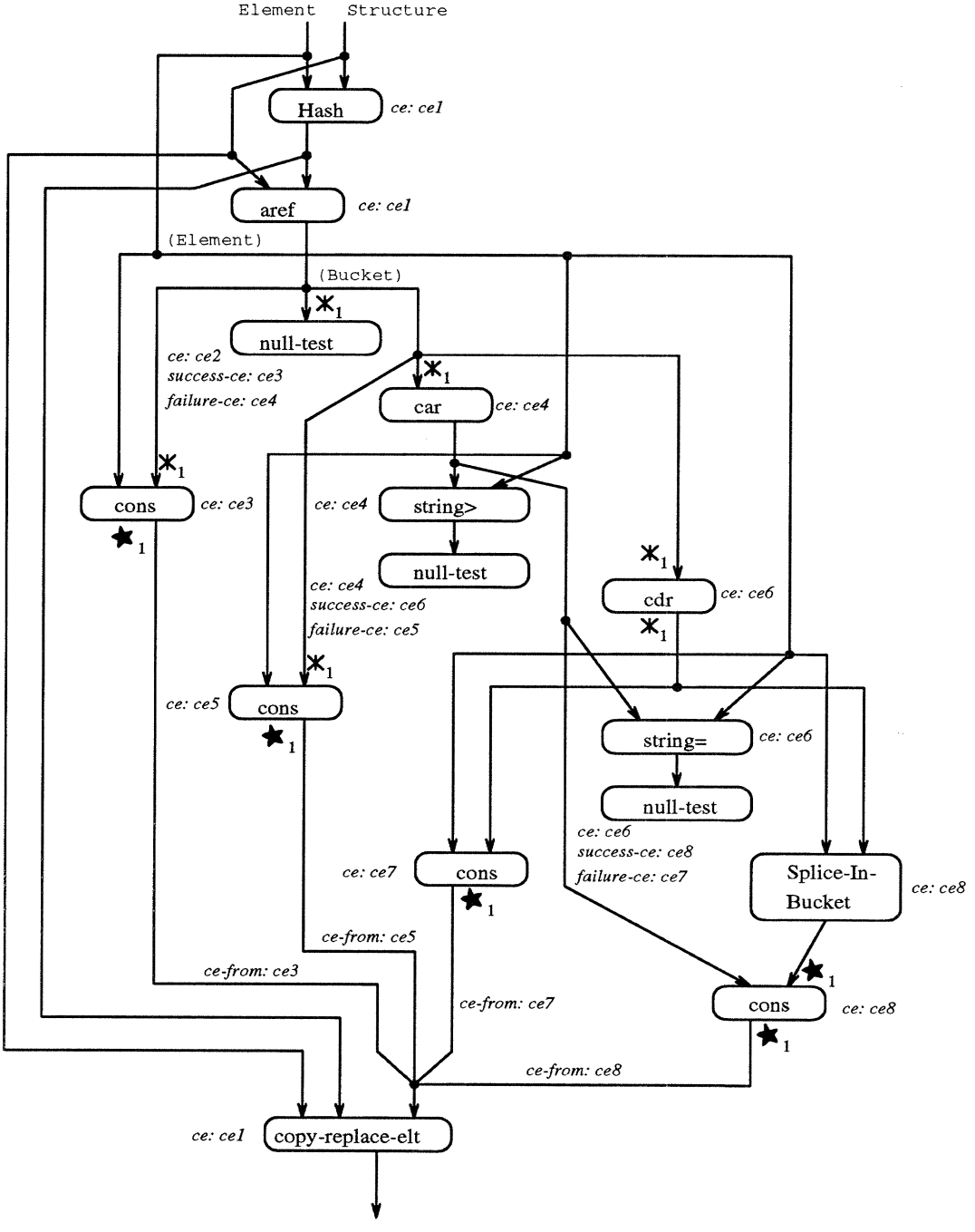


Figure 4-2: Flow graph representing `HT-Insert`.

In addition, the set of control environments in which a recursion is exited are incomparable. (There will be more than one such control environment if the recursion has multiple exits.) These are the set of control environments of the nodes that are executed in the base cases of the recursion. For example, in Figure 4-2, the set $\{ce3, ce5, ce7\}$ is incomparable.

We define a partial function $+_{ce}$ as the following. If a set S of control environments is not incomparable, then $+_{ce}(S)$ is undefined. Otherwise, if S is a success-ce/failure-ce pair for the same node, then $+_{ce}(S)$ is the control environment of that node. If S is a set of control environments in which a recursion is exited, then $+_{ce}(S)$ is the outside-ce of that recursion. In Figure 4-2, $+_{ce}\{ce3, ce5, ce7\} = ce1$, while $+_{ce}\{ce3, ce5\}$ is undefined. (Intuitively, the result of $+_{ce}$ can be viewed as the control environment in which operations are performed as many times as the combined number of times operations in the control environments of the incomparable set are performed.)

Another function $\sum_{ce}$ on sets of control environments is defined recursively in terms of $+_ce$ as:

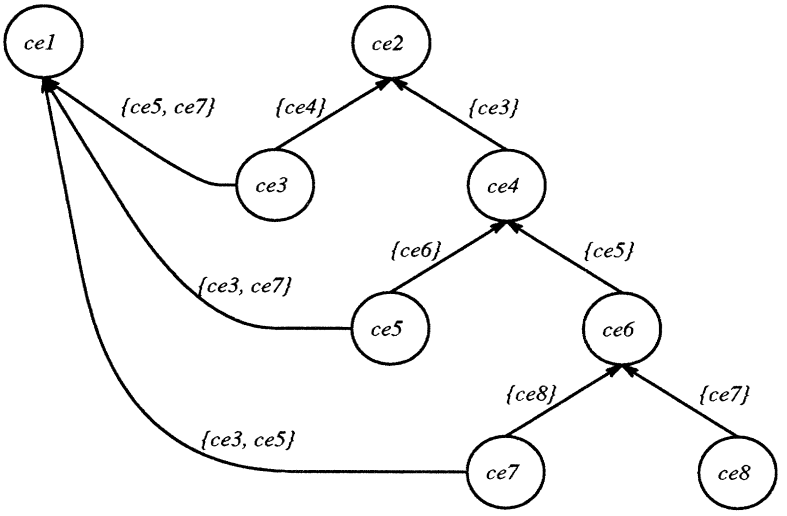
- If $|S| = 2$, then $\sum_{ce} S = +_{ce}(S)$.
- If there is a set $S' \subseteq S$ which is incomparable, then $\sum_{ce} S = \sum_{ce}(+_{ce} S' \cup (S - S'))$.
- Otherwise, $\sum_{ce} S$ is undefined.

In other words, if a single control environment can be obtained by recursively reducing (using $+_{ce}$) all incomparable subsets of the input set S, then that control environment is the result. Otherwise, $\sum_{ce} S$ is undefined. For example, in Figure 4-2, $\sum_{ce}\{ce3, ce5, ce7, ce8\} = \sum_{ce}\{ce3, ce5, ce6\} = \sum_{ce}\{ce3, ce4\} = ce2$. Also, $\sum_{ce}\{ce3, ce5, ce8\}$ = undefined, while $\sum_{ce}\{ce3, ce5, ce7\} = ce1$.

This summing function is used as the attribute combination function for control environment attributes. Recall from Section 3.5.1 that when two items are zipped up, the attribute values of the resulting item's left-hand side are computed based on those of the zip-up components. Each attribute has an attribute combination function associated with it. This is used to compute a new value of an attribute, based on the values of that attribute held by the zip-up components' left-hand sides. For all control environment attributes, the attribute combination function is $\sum_{ce}$. This is a partial function. If the sum is not defined for the set of control environments being combined, the zip-up of the items involved fails.

Partial Order Graph of Control Environments

We represent the partial ordering of control environments in an annotated partial order graph which facilitates the operations of checking $\sqsubseteq$ and computing $+_{ce}$ and $\sum_{ce}$. The annotated partial order graph has nodes representing control environments. An edge is drawn from one node representing ce_i to another representing ce_j iff $ce_i \sqsubseteq ce_j$. This edge is annotated with the set of control environments that together with the source ce_i form an incomparable set.



Recursion information: [recur-ce: ce2, feedback-ce: ce8, outside-ce: ce1]

Figure 4-3: Annotated partial order graph representing the relationships between the control environments of `HT-Insert`.

Associated with this graph is a set of triples, one for each recursive function call represented by the flow graph. (There may be more than one if the flow graph represents a program that calls more than one recursive function, including nested recursions.) Each triple contains the recur-ce, feedback-ce, and outside-ce of the flow graph representing the recursive function.

For example, Figure 4-3 shows the annotated partial order graph for the control environments of the flow graph in Figure 4-2. One triple of recursion information is associated with the graph.

Edge Attributes

Besides attaching control environment attributes to nodes, control flow information is contained in attributes on edges. Each edge holds a *ce-from* attribute, which indicates the control environment in which the edge carries dataflow. For example, in Figure 4-2, the ce-from attribute on the edge from the top-most `cons` (in the figure) to the `copy-replace-elt` indicates that the operation `copy-replace-elt` receives dataflow only in the control environment *ce*3 which is the success-ce of the first `null-test` node. (Edges that fan in represent conditional merging of dataflow.)

Each edge also carries a *constant-type* attribute whose value is either a constant (such as `T`, `NIL`, `0`) or `undefined`, depending on whether the edge represents dataflow from a constant.

Flow graphs for programs containing user-defined aggregate data structures hold attributes that represent the aggregation information. Each edge holds an *accessor* attribute that describes how the data it carries results from the destructuring of some data struc-

ture. Each edge also holds a *constructor* attribute that describes how the data it carries becomes part of some data structure. (The value of these attributes is undefined if the edge is not carrying data involved in some aggregation.) The attributed flow graph can be seen as the flow graph that results from 1) making a flow graph that includes Spreads and Makes to represent aggregation and then 2) transforming it into a minimally aggregated flow graph using aggregation-removal transformations, and 3) replacing any residual Spreads and Makes with fan-out and fan-in edges, respectively.

As these nodes are removed, the naming information they contain is placed into attributes. This information is useful in presenting the results of recognition and can be a source of guidance for the recognition system, as discussed in Section 4.2.3, 6.4.1, and 7.2.3. Because these attributes are primarily used by the Paraphraser, we defer describing them until Section 4.2.3.

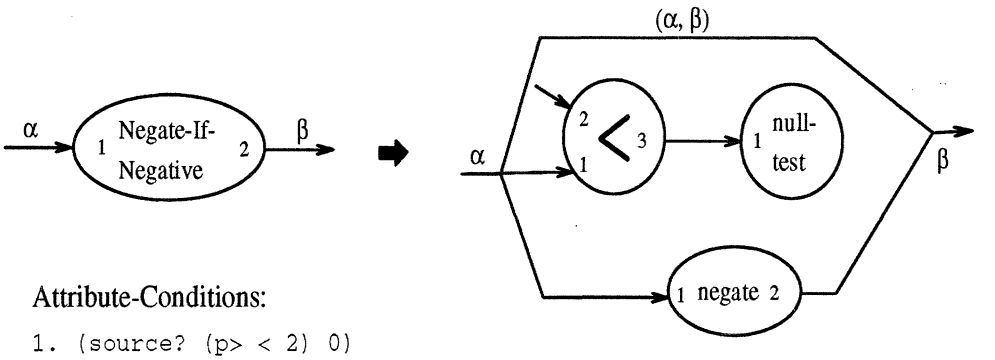
Input and Output Correspondences

In addition to control environment attributes, flow graphs for recursive functions have attributes which represent the relationship between the inputs (resp. outputs) of the flow graph and the inputs (resp. outputs) of the node representing the recursive call. In particular, an output port p_o *input-corresponds* to an input port p_i iff p_o is connected to the jth input of the recursive node and p_i represents an input to an operation that receives dataflow from the jth input of the recursive function.[1] Similarly, an input port p_i *output-corresponds* to an output port p_o iff p_i is connected to the kth output of the recursive node and p_o represents an output that sends dataflow to the kth output of the recursive function.) The input-corresponds and output-corresponds relations are not symmetric, transitive, or reflexive.

For example, in the flow graph representing `HT-Insert`, shown in Figure 4-2, the output port on the `cdr` node input-corresponds with each of the input ports of `null-test`, `car`, `cdr`, and the second input of each of the `cons`'s in control environments $ce3$ and $ce5$. (Input and output correspondences are illustrated by subscripted asterisks and stars, respectively.) The second input of the `cons` in the feedback-ce output-corresponds with the output port of each of the `cons` nodes.

Because recursions can be nested within each other, it is necessary to be more specific about the conditions under which a pair of ports input- or output-correspond (i.e., in which recursion does the correspondence occur). This is done by associating with each correspondence relation the feedback-ce of the recursion in which the ports correspond. All correspondences in this flow graph have the feedback-ce $ce8$ associated with them.

[1]The input-corresponds relation was previously called *feeds-back* [145] in flow graphs representing tail-recursive functions, but it was renamed in the current representation which is generalized to represent regular recursion, as well as tail recursion.



Attribute-Conditions:

```
1. (source? (p> < 2) 0)
2. (ce= (ce-from (e> negate 2 Negate-If-Negative 2))
        (failure-ce (n> null-test)))
3. (ce= (ce-from (st-thru> 1 2))
        (success-ce (n> null-test)))
```

Attribute-Transfer Rules:

```
1. ce := (ce (n> null-test))
```

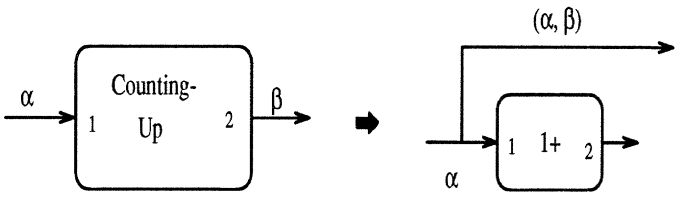
Figure 4-4: Flow graph grammar rule for Negate-if-Negative, with actual attribute conditions.

Attribute Conditions and Transfer Rules

Graph grammar rules impose constraints on the attributes of the flow graphs to which their right-hand sides match. The attribute conditions and attribute-transfer rules are expressed in terms of:

- Functions that map a port, node, or edge in a rule's right-hand side or a rule's st-thru to the port, node, or edge in the input graph to which it is matched when the right-hand-side (and st-thru) are recognized. These are `p>`, `n>`, `e>`, and `st-thru>`.
- Attribute accessor functions which when given a node or edge return the value of that attribute of the node or edge. For example, `ce-from` computes the ce-from attribute value of an edge. These accessor functions are both primitive accessor retrieval functions and functions built on top of them, such as control environment computations involving $+_{ce}$.
- Relations on the attribute values, such as $\sqsubseteq$, and predicates on nodes and edges that are defined in terms of these primitive relations and the attribute accessor functions. For example, `co-occur` is a predicate that takes two nodes and checks whether their control environments are equal.

For example, Figure 4-4 gives the rule for Negate-if-Negative, a common implementation of the Absolute-Value cliché. (This rule is repeated from Figure 3-9, where the attribute conditions were given informally.) In the first condition, `(p> < 2)` refers to the input graph port matching the port labeled 2 on `<`. `Source?` tests whether this port receives dataflow from a constant equal to 0.



Attribute Conditions:

```
1. (input-corresponds? (p> 1+ 2) (p> 1+ 1)
                       (feedback-ce (innermost-recur (n> 1+))))
2. (ce= (ce-from (st-thru 1 2))
        (recur-ce (innermost-recur (n> 1+))))
```

Attribute-Transfer Rules:

```
1. ce := (ce (n> 1+))
```

Figure 4-5: Grammar rule for counting-up cliché.

In the second condition, `e>` is used to refer to an edge in the input graph whose source matches an output of the rule's right-hand side. It constrains this edge to have a ce-from attribute that is equal to the failure-ce of the node that matches `null-test`.

The third condition uses `st-thru>` to refer to an edge that matches the st-thru. It constrains this edge to have a ce-from attribute that is equal to the success-ce of the node that matches `null-test`.

The attribute-transfer rule computes the control environment of the left-hand side node to be the control environment of the node matching `null-test`.

Attribute accessor functions are provided to compute the recursion information for the innermost recursion containing a particular node. These are used in many constraints for iterative clichés. A typical constraint is that two ports input-correspond or output-correspond in the feedback-ce of the innermost recursion containing some node.

For example, Figure 4-5 shows the grammar rule representing the iteration cliché, *counting-up*, which repeatedly increments the value of its input, which starts with some initial value and is subsequently the result of the increment performed on the previous iteration. The rule constrains the input graph ports matching the output and input ports of `1+` to input-correspond in the feedback-ce of the innermost recursion in which the input graph node matching `1+` occurs.

4.1.2 The Plan Calculus

Flow graphs annotated with the attributes and conditions described in the previous section can become difficult for *people* to read. For presentation purposes, we make use of a graphical notation, called the Plan Calculus [110, 117], which aids people in viewing flow graphs with

certain classes of constraints pertaining to programming. However, although the Plan Calculus is used as a visual aid, the underlying attributed flow graph representation is conceptually primary to our recognition approach.

The Plan Calculus is a graphical formalism for representing programs, clichés, and relationships between clichés. In the Plan Calculus, both clichés and individual programs are represented as *plans*. The relationships between clichés are captured in *overlays*. This section briefly describes plans and overlays as they relate to our attributed flow graph formalism. (For more details, see Rich [110, 117].)

A plan graphically represents the operations of a program and the data and control flow constraints between them in what is called a *plan diagram*. (Plans also specify preconditions and postconditions in a separate logical language.) A plan diagram is a hierarchical graph structure composed of boxes and arrows. Boxes denote operations and tests, while arrows denote control flow and dataflow.

Plan diagrams can be seen as graphical depictions of flow graphs with certain classes of attributes and conditions – those that pertain to control flow and data aggregation. Plan diagrams and flow graphs share the same dataflow structure in that boxes represent operations and arcs denote dataflow between them. However, plan diagrams also have arcs that denote control flow and *join* boxes that represent the merging of control flow. A control flow arc from a box A to a box B denotes that B eventually (not necessarily immediately) follows A. A branch in control flow is represented by a *test* box. The rejoining of control flow is represented by a *join* box. It has two sets of incoming dataflow arcs, one for each case of the corresponding test that caused the control flow to branch out. The set of dataflow arcs leaving the join carry the data of the set of inputs on either the `T` or the `F` side of the join, depending on whether the `T` or the `F` branch (respectively) of the conditional is taken.

Like flow graph edges, dataflow arcs may fan out (which means the result of an operation is used by more than one operation). However, they cannot fan into the same input, as edges can in flow graphs. Instead, they are merged by join boxes. Control flow arcs may fan in or out.

Figure 4-6 shows an example of a plan diagram, representing the following code fragment.

```
(let ((tax 0.0))
  (when (> gross min)
    (setq tax (* percent gross)))
  (- gross tax))
```

Solid arcs denote dataflow; cross-hatched arcs denote control flow. Each box in the plan has a label, composed of a part name and a type. For instance, the label "multiply:*" specifies that the plan in Figure 4-6 has a part named "multiply" of type "*." The part names serve to distinguish between boxes in the plan that have the same type. The part names in a given plan diagram must be distinct. The part "test" is a test box. Although in this example, "test" has no data outputs, in general, data may flow out of a test box from either the side labeled `T` or the side labeled `F`, depending on whether the output is produced

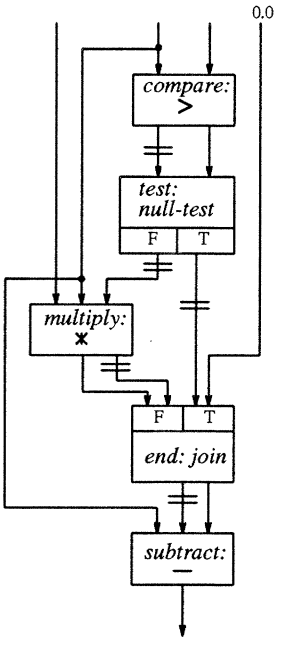


Figure 4-6: The plan diagram for a code fragment.

when the test succeeds or fails, respectively. The box named "end" is a join. Its outgoing dataflow arc carries the data coming from "multiply" when `GROSS>MIN` (and the `F` branch of "test" is executed), and 0.0, otherwise.

The control flow arcs, test, and join boxes represent the control flow information that is in the control environment attributes. Boxes that represent operations that are tied together by control flow arcs correspond to nodes that are all in the same control environment in our flow graphs. The relationships between control environments are reflected in the structure of the control flow arcs. The ce-from attributes and conditions on dataflow edges are represented by dataflow routed through joins, which explicitly specify in which case of a conditional branch data flows from a particular operation to another.

Control flow arcs are sometimes omitted when there is no conditional structure (i.e., all operations are in the same control environment). For example, in Figure 4-6, the control flow arcs between "compare" and "test" and between "end" and "subtract" can be omitted.

Plans may contain other plans as parts. If the type of a plan and a subplan within it are the same, then the plan is *recursively defined.* An example is given in Figure 4-7. This is the plan diagram representing the following code fragment which iterates over a list `L`, counting the number of elements in it. A dashed box delimits the recursive subplan, with enough details filled in to show the input- and output-corresponds relations.

```
(LET ((COUNT 0))
  (LOOP (WHEN (NULL L) (RETURN COUNT))
        (SETQ L (CDR L))
        (SETQ COUNT (1+ COUNT))))
```

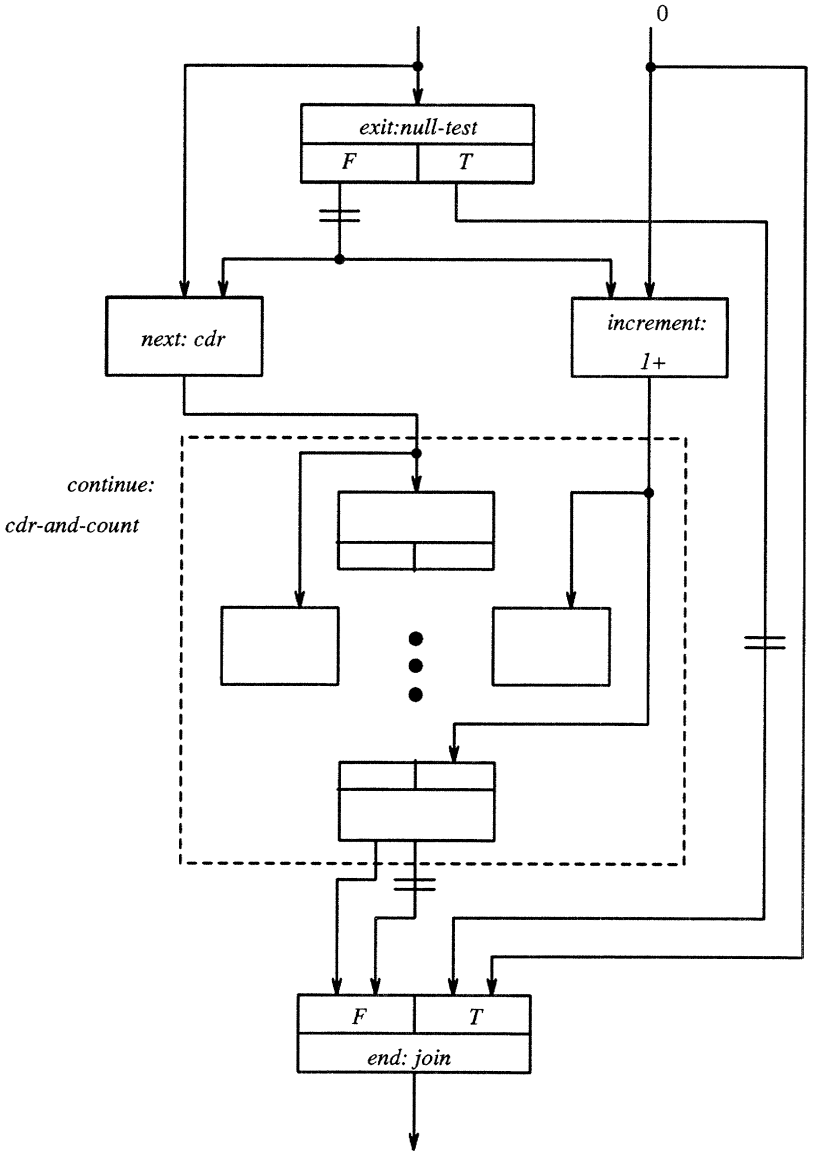


Figure 4-7: A recursively defined plan.

Figure 4-8: Data plan for Circular Indexed Sequence.

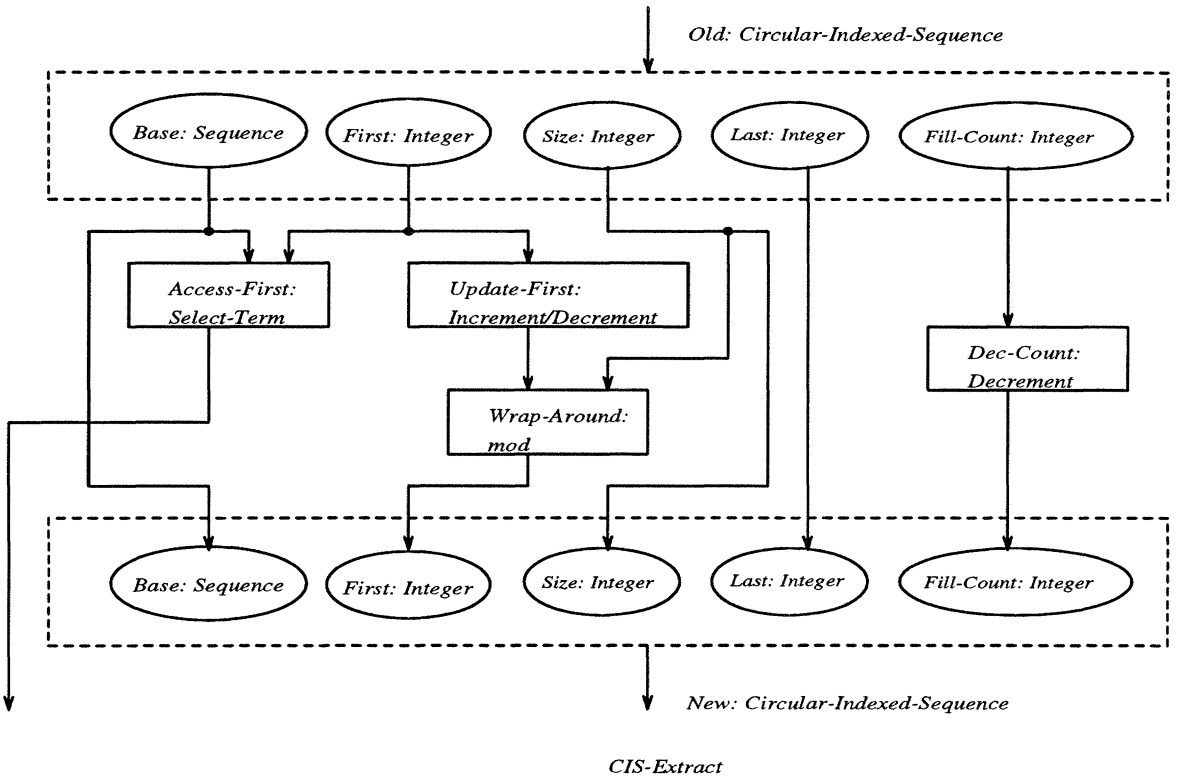


Figure 4-9: Plan for extracting an element from a Circular Indexed Sequence.

Plan diagrams can contain data as parts. A *data plan* is a plan whose parts are all either data or (hierarchically) data plans. For example, Figure 4-8 shows a data plan diagram representing the Circular Indexed Sequence (CIS) data structure. Figure 4-9 shows a hierarchical plan that contains both data and computational parts. It is the plan diagram for the familiar computation of extracting an element from a CIS. The two data subplans, which represent the aggregation of data, depict the accessor and constructor information that we encode in accessor and constructor edge attributes on flow graphs.

4.1.3 Codifying Clichés: Using the Plan Calculus as a Stepping Stone

Plans are used in the Plan Calculus both to represent programs and to define clichés. Relationships between clichés are represented by *overlays*. An overlay is a pair of plans and a set of correspondences between their parts. They show how an instance of one cliché can be viewed as an instance of another. Overlays provide a general facility for representing common shifts of viewpoint, such as implementing specifications and data abstractions, and temporally abstracting iterations.

As grammar writers, we found it easier to express clichés in the Plan Calculus first and then to translate the plan definitions and overlays into graph grammar rules.

This section describes overlays and shows examples of how relationships between clichés are captured in them. It then describes how overlays and plan definitions of clichés are

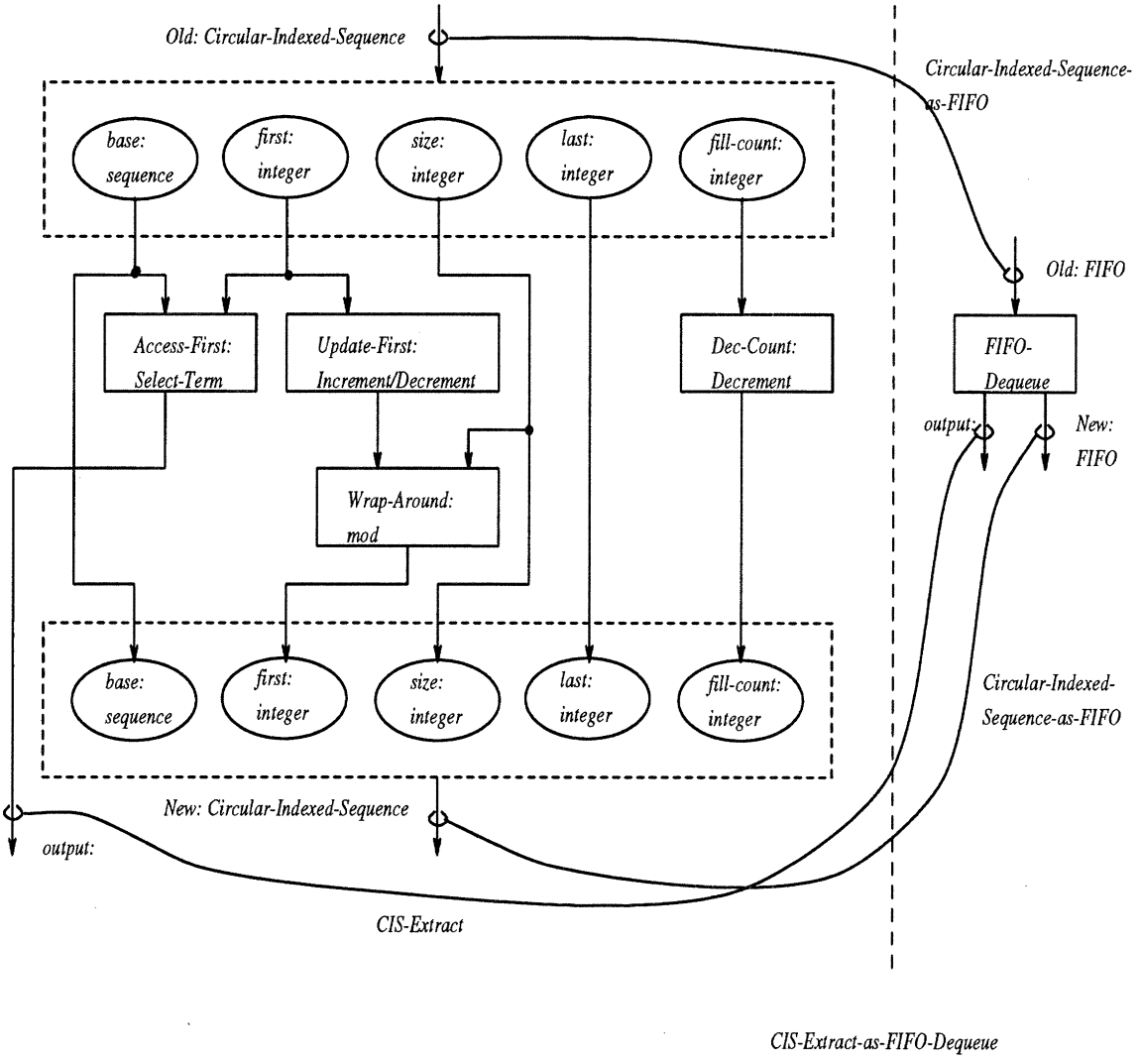


Figure 4-10: Implementation overlay showing how FIFO-Dequeue can be implemented by CIS-Extract.

encoded in attributed flow graph grammar rules.

Implementation Relationships

Recognizing clichés on multiple levels of abstraction requires being able to view some clichés as implementations of more abstract clichés. In the Plan Calculus, *implementation overlays* capture these relationships.

The plan on the right of an implementation overlay is the plan definition for an abstract operation or data structure. The plan on the left of the overlay is the plan definition of a correct implementation of the abstract operation or data structure represented on the right.

For example, Figure 4-10 shows an implementation overlay that expresses the relationship between the abstract clichéd operation FIFO-Dequeue and one possible implementation

of it, which is as a CIS-Extract cliché. The correspondences between the two sides of the overlay show how the inputs and outputs of the abstract operation are related to those of the implementation. They may be labeled with names of data overlays, as is the correspondence between the input FIFO on the right and the input CIS on the left. The CIS-Extract-as-FIFO-Dequeue overlay represents an implementation of the FIFO-Dequeue operation, in which the FIFO is implemented as a Circular Indexed Sequence. The old and new FIFOs of the FIFO-Dequeue operation correspond to the old and new Circular Indexed Sequences of the implementation plan. These correspondences are labeled with the name of the Circular-Indexed-Sequence-as-FIFO data overlay, which means that the old (resp. new) CIS of CIS-Extract, when viewed as a FIFO correspond to the old (resp. new) FIFO of FIFO-Dequeue.

Encoding Implementation Overlays in Grammar Rules

Our grammar formalism was developed to make it easy to represent shifts of viewpoint from both abstract operations and abstract data structures to their implementations. It is specifically able to encode the relationships expressed in implementation overlays, including those in which the left-side plan definition contains data plans for aggregate data structures as subplans.

Each plan definition of the algorithmic clichés is encoded in a flow graph grammar rule. The type of the left-hand side node of the rule is the plan's name. The right-hand side is the flow graph encoding of the plan, in which the control flow constraints summarized in the structure of the plan are listed in attribute conditions. If the inputs or outputs of the plan definition are data plans, the aggregation they represent is encoded in the embedding relation of the rule.

In particular, suppose an input (or output) of a plan definition is an aggregate data structure of type D, represented by a data subplan. The rule encoding of the plan definition will have a left-hand side port whose type is D which corresponds to a tuple of right-hand side and left-hand side ports. For each part p_i of the data plan, the ith element of the tuple is the set of right-hand side ports (if any) that encode the inputs or outputs of boxes to which the part is connected. If the part is connected directly to a part in another data plan in the plan definition, then the tuple will include the left-hand side port that encodes that data plan.

(One way to see this encoding is: the ports in the tuple are determined as if the input (or output) data plan were replaced by a fringe Spread (or Make) node. The embedding relation that results from removing these fringe nodes (as described in Section 3.4.2) is the same as the embedding resulting from this encoding.)

For example, Figure 4-11 shows the flow graph grammar rule encoding of the CIS-Extract plan definition of Figure 4-9. (This figure is a repeat of Figure 3-24.) Attribute conditions and transfer rules are not shown.

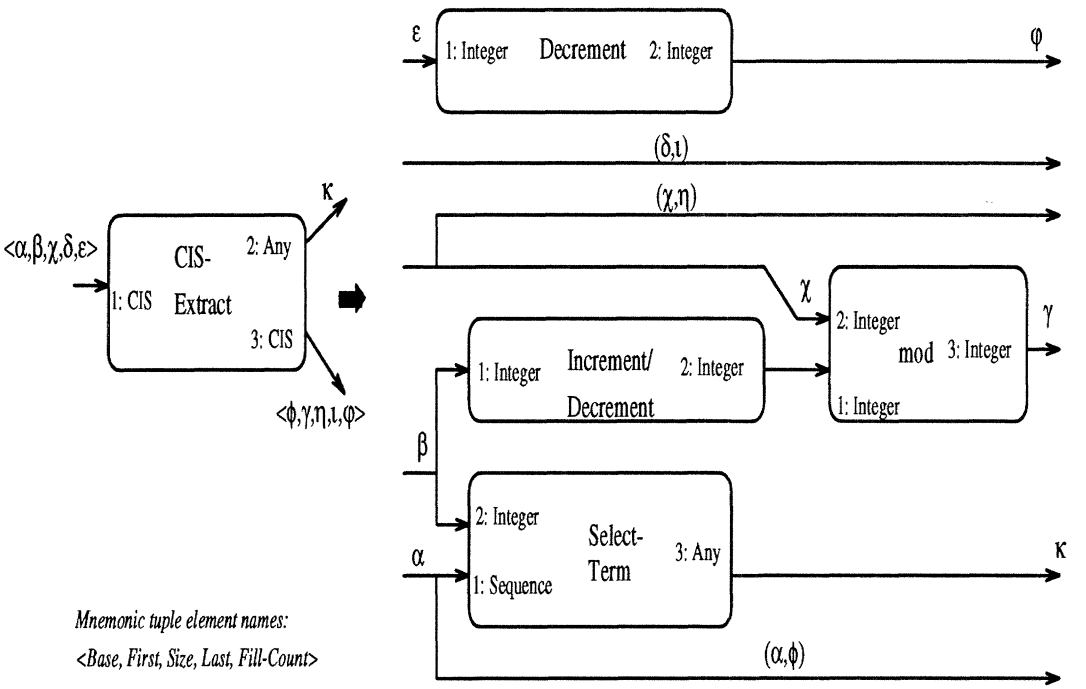


Figure 4-11: Rule encoding plan for CIS-Extract.

Currently, we are limited to encoding only those plans that contain data subplans only at its *inputs* or *outputs*. However, internal data subplans can be represented by collapsing a sub-flow graph of the flow graph that represents the left side of the overlay into a non-terminal. This sub-flow graph can have the data plan as its input/output.

In addition to plan definitions of clichés, each implementation overlay is encoded as a flow graph grammar rule. These rules contain single nodes on both sides. The left-hand side node's type is the type of the abstract operation on the *right* side of the overlay. The right-hand side node's type is the name of the implementation plan on the overlay's *left* side.

The embedding relation encodes the correspondences between the two sides of the overlay. If there is a correspondence between an input (or output) of the abstract operation on the right side of the overlay and an input (or output) of the implementation plan, then the left- and right-hand side ports that encode them in the grammar rule correspond to each other in the rule's embedding relation. For example, Figure 4-12 shows the grammar rule encoding of the overlay of Figure 4-10.

Sometimes a correspondence is labeled with the name of a data overlay that maps an abstract data type to a concrete one. This mapping information is associated with the corresponding ports in the rule. Different ports may have different data mappings associated with them, even if they are of the same type.

When a rule that encodes an overlay is used in a parse, it uncovers a design decision to implement a certain abstract operation or data structure as another operation or data

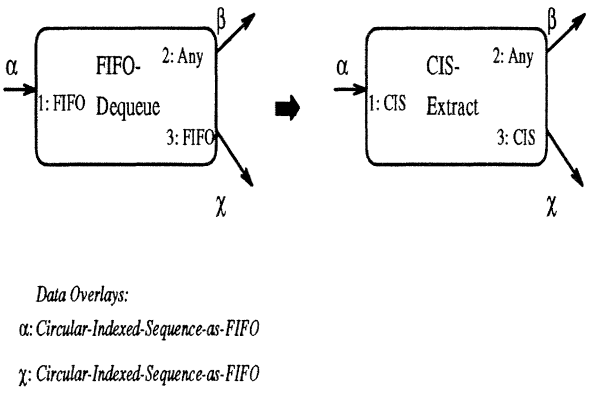


Figure 4-12: Rule encoding the CIS-Extract-as-FIFO-Dequeue overlay.

structure. The overlay mapping information is used to generate documentation of this design decision.

Temporal Abstraction

In recognizing an iterative program, it is often useful to view clichéd fragments of iterative computation as operations on a sequence of values. This technique is called *temporal abstraction.* (See [110, 117, 127, 138].)

For example, a common computation that occurs in iterative programs is: on each iteration a function is applied to the result of the previous application of the function (or to an initial value on the first iteration). This is called the *generation* cliché. The plan diagram for this iteration cliché is shown on the left in the overlay of Figure 4-13. A common instance of generation is *counting-up*, in which the generating function is `1+`.

The temporally abstracted view of generation is as an operation Generate that takes an initial value and a generating function and creates a sequence of values – the values processed over time, one per iteration. For example, the temporal abstraction of the counting-up cliché is the operation Count, which takes an initial value (i) and produces the sequence of values $[i, i+1, (i+1)+1, ...]$.

The temporal abstraction of iteration clichés is formalized in the Plan Calculus using *temporal overlays.* These relate a temporally abstract operation (on the right side of the overlay) to the plan for an iteration cliché (on the left side). Figure 4-13 shows a temporal overlay formalizing the temporal abstraction of generation as a Generate operation.

The correspondence labeled with an asterisk is called a *temporal correspondence.* This denotes the relationship between the left side data part (the input to `apply`) and the right side *temporal sequence* (the output of Generate). It specifies that the first term of the temporal output sequence of Generate is equal to the initial input to `apply`; the second term is equal to the same part of the recursively defined plan; and so on recursively. Temporal overlays always contain at least one temporal correspondence.

Temporal abstraction allows an iterative program that is composed of iteration clichés

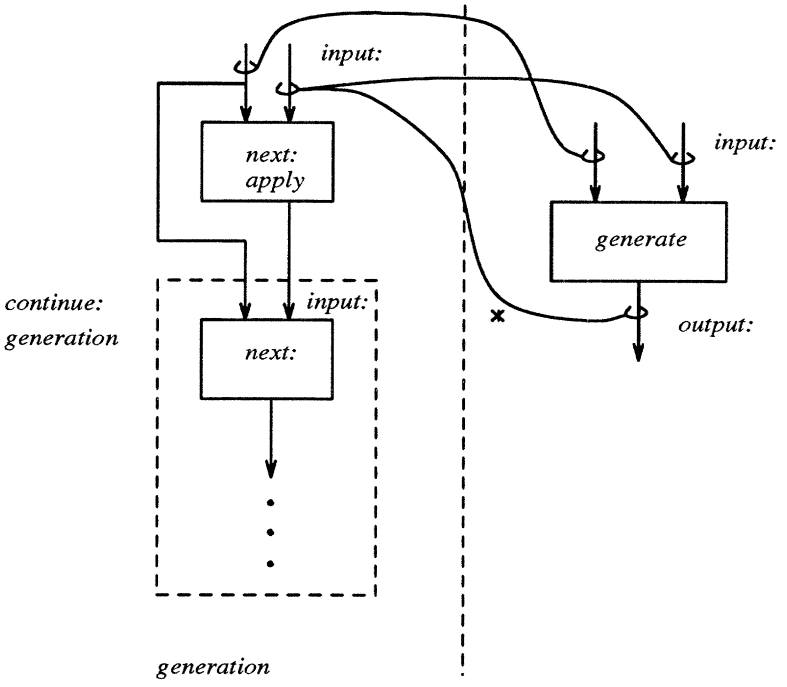


Figure 4-13: Temporal overlay showing the view of Generation as a Generate operation.

to be seen as a composition of functions on sequences. This makes the program as easy to understand and reason about as a non-iterative (straight-line) program.

Temporal abstraction also enables GRASPR to undo common function-sharing optimizations within iterative programs, such as loop-jamming, using the same techniques it uses to deal with function-sharing due to common subexpression elimination. (These are the techniques for parsing structure-sharing flow graphs, as is discussed further in Section 5.1.5.)

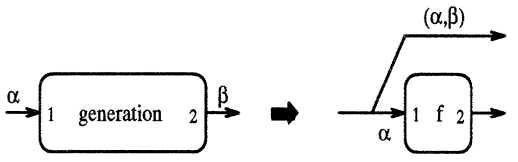
Also, it is easy to encode clichés by building them out of temporally abstract operations, rather than expressing them as large, flat iteration patterns. Additionally, a composition of abstract operations is easier to describe than a combination of overlapping, interleaved iteration clichés.

Encoding Temporal Abstractions in Grammar Rules

As with implementation relationships, flow graph grammar rules are able to capture temporal abstractions by a straightforward encoding of temporal overlays.

Like any other algorithmic cliché, the plan diagram for an iteration cliché is encoded in a grammar rule whose left-hand side is a node whose type is the name of the cliché. The right-hand side is the dataflow structure of the plan diagram.

The relationships between the inputs (resp. outputs) of the recursively defined plan and the inputs (resp. outputs) of the recursive subplan are captured in "input-corresponds?" and "output-corresponds?" conditions. For example, the rule for generation is shown in Figure 4-14. It has attribute conditions that constrain the output of `f` to input-correspond



Node-Type Constraints:

```
f: (lambda (node-type) T)
```

Attribute Conditions:

```
1. (input-corresponds? (p> f 2) (p> f 1)
                       (feedback-ce (innermost-recur (n> f))))

2. (ce= (ce-from (st-thru 1 2))
        (recur-ce (innermost-recur (n> f))))
```

Attribute-Transfer Rules:

```
1. ce := (ce (n> f))
2. generating-function := (node-type (n> f))
```

Figure 4-14: Grammar rule encoding the plan for Generation.

to the input of `f`.

This rule's right-hand side is not exactly the dataflow structure of generation's plan definition. The plan definition takes a function as input, which is iteratively applied, but the right-hand side flow graph does not explicitly represent this functional input and application. Instead, the right-hand side node has a *generalized node type*, which means the rule imposes a constraint on the types of input graph nodes or non-terminal instances that can match this node. In the rule for generation, the node type constraint is loose: any node type matches. So any instances of a clichéd unary operation or a unary primitive operation that satisfies the input-corresponds relationships will be recognized as an instance of generation. (Generalized node types are used as a shorthand for several rules that have the same left- and right-hand sides, except for variation in the node types of the right-hand side nodes.)

The reason the `apply` operation is not encoded directly in the grammar rule as a node of type "apply" is that there would not be an input graph node to match it. Also, this grammar rule cannot be used to recognize generation in programs in which the generating function is an arbitrary composition of functions. This limitation is discussed in more detail in Section 5.2.3.

The type of the input graph node matching the right-hand side is transferred to the left-hand side's *generating-function* attribute. This can be constrained in attribute conditions of rules that use generation.

Control flow constraints captured in the iteration cliché's plan are encoded in attribute conditions referring to the control environments of the recursion (recur-ce, feedback-ce, and outside-ce). For example, the plan diagram for the cliché *iterative-search* is shown on the left in the overlay of Figure 4-15. This iteration cliché is the familiar pattern of repeatedly

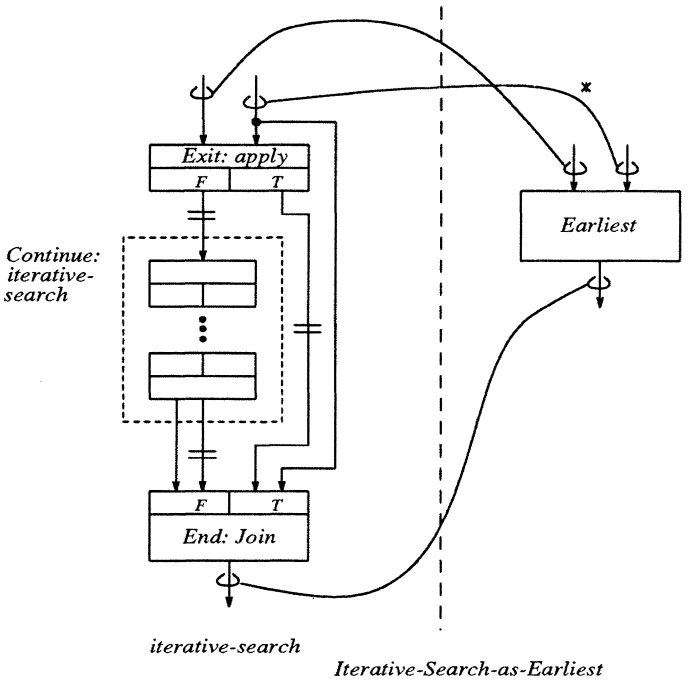


Figure 4-15: Temporal overlay relating the plan for Iterative Search and the operation Earliest.

applying some test until it is satisfied by some value. When the test succeeds, the iteration is terminated and the value is made available outside the iteration. This iteration cliché is encoded in the flow graph grammar rule shown in Figure 4-16. (In the figure, `ce<=` stands for $\sqsubseteq$ and `ce=` is the equality relation between control environments.)

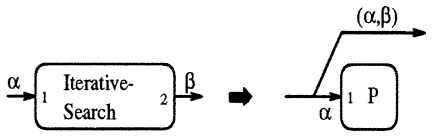
The first condition in this rule encodes the constraint summarized by the control flow arcs, test, and join: the test must be an exit test of the iteration. This constraint translates to a condition on how the control environments of the test and the recursion relate. In particular, the recursive call should occur in the failure-ce of the test and the recursion should be exited in the success-ce of the test.

The attribute condition actually loosens this constraint slightly to allow for other exit tests of the recursion. The two parts of the condition are:

1. It must be *possible* for the recursive call to occur in the failure-ce of the test (but another exit test may occur in the failure-ce which can prevent this from happening). This is expressed as: the feedback-ce of the innermost recursion containing the test must be $\sqsubseteq$ the failure-ce of the test.

2. The success-ce of the test is one *possible* way to exit the recursion (but there may be another exit test in whose success-ce the recursion is also exited). This is expressed as the success-ce must be $\sqsubseteq$ the outside-ce of the recursion.

This constraint occurs in the encoding of many iteration constraints, so we defined a



Node-Type Constraints:

```
P: (lambda (node-type) (predicate? node-type))
```

Attribute Conditions:

```
1. (and (ce<= (feedback-ce (innermost-recur (n> P)))
              (failure-ce (n> P)))
        (ce<= (success-ce (n> P))
              (outside-ce (innermost-recur (n> P)))))
2. (ce= (ce-from (st-thru> 1 2))
        (success-ce (n> P)))
3. (ce= (ce-from
          (output-edge (recursive-node (innermost-recur (n> P)))
                       (edge-sink (st-thru> 1 2))))
        (feedback-ce (innermost-recur (n> P))))
```

Attribute-Transfer Rules:

```
1. ce := (ce (n> P))
2. search-predicate := (node-type (n> P))
3: success-ce := (success-ce (n> P))
4. failure-ce := (failure-ce (n> P))
```

Figure 4-16: Grammar rule for Iterative Search cliché.

predicate, **exit-predicate**, that takes a terminal or non-terminal test node and checks these conditions. So the abbreviate form of the first condition in Figure 4-16 is **(exit-predicate (n> P))**. For example, the top-most **null-test** terminal node in Figure 4-2 is an exit-predicate.

The second attribute condition in the rule for iterative-search constrains the output to carry dataflow in the success-ce of the test. This expresses the constraint that the output of the iterative-search cliché is the first element to pass the test.

The third condition encodes the constraint that is depicted by the data and control flow edges from the recursive sub-plan to the exit join in the plan diagram of Figure 4-15. This constraint is that the output dataflow of the recursion that merges with the st-thru must carry dataflow in the feedback-ce of the innermost recursion containing the test. This ensures that there is no additional computation being performed on the way up out of the recursion.

The function **recursive-node** finds the input graph node that represents the recursive call of the recursion containing the exit test. The function **output-edge** finds the edge from some output port of a recursive node to an input port. This function is only used when the recursive node is expected to have only one output port that connects to the input port. (The constraint fails if this is not true.) In this case, **output-edge** finds the edge that shares its sink with the edge matching the st-thru.

This rather awkward type of condition is imposing a structural constraint (as well as the ce-from constraint) which cannot be expressed in the structure of the rule's right-hand

Attribute-Transfer Rules:

```
1. ce := (outside-ce (innermost-recur (n> Iterative-Search)))
2. search-predicate := (search-predicate (n> Iterative-Search))
```

Figure 4-17: Grammar rule encoding the temporal overlay Iterative-Search-as-Earliest.

side flow graph. It requires that there be an edge from a recursive node directly to the output that merges with the st-thru. This constraint is expressed in attribute conditions, rather than in the structure of the right-hand side of the rule because there is no way to represent the edge from the recursive node to the output without including the recursive node in the right-hand side. The edge cannot be expressed as a st-thru, since its source is not an input to the non-terminal. If we did include the recursive node, we would have to specify its arity. This would severely restrict the programs in which it can be matched to only those with recursive nodes of the specified arity.

The attribute-transfer rules shown in Figure 4-16 specify that all of the control environment attributes of the exit predicate are transferred to the non-terminal representing iterative-search.

A temporal abstraction of iterative-search is the *Earliest* operation. This operation takes a sequence of values and a predicate and finds the first term in the sequence satisfying the predicate. This relationship is shown in the overlay of Figure 4-15.

A temporal overlay is encoded in a grammar rule in the same way as implementation overlays. Figure 4-17 shows the rule for Earliest.

When an iteration cliché is viewed as a temporally abstract operation, the operation is seen as being in the control environment from which the iteration is called (i.e., its outside-ce). This is expressed in the attribute-transfer rules of the rule encoding a temporal abstraction: the control environment of the temporally abstract operation is the outside-ce of the innermost recursion containing the iteration cliché.

4.1.4 Examples of Codifying Simulation Clichés

We used the Plan Calculus as a stepping stone in capturing our clichés and then encoding them in a flow graph grammar. This section gives a flavor for how we did this. It shows the plan definitions and overlays that capture some of the clichés that were described in English in Chapter 2. It then gives the grammar rules GRASPR uses in recognizing these clichés.

Encoding Event-Driven Simulation Clichés

Recall from Section 2.1.3, that the event-driven simulation algorithm consists of the following key steps:

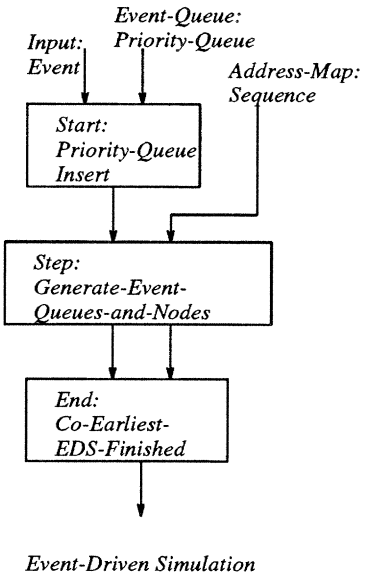


Figure 4-18: Plan definition for Event-Driven Simulation cliché.

- The event-driven simulator is given an initial `EVENT`, whose Object is a starting `MESSAGE` and whose Time is the `MESSAGE`'s arrival time. This is added to the `EVENT-QUEUE`.
- On each step of the simulation, the highest priority `EVENT` is pulled from the `EVENT-QUEUE` and processed.
- Processing an `EVENT` means simulating the handling of the `MESSAGE` in the `EVENT`'s Object part. This involves:
  - looking up the `ASYNCH-NODE` in the `ADDRESS-MAP` that is indexed by the Destination-Address part of the `MESSAGE`.
  - updating the `ASYNCH-NODE`'s Clock to be the maximum of its current time and the Time part of the `EVENT`. This creates a new `ASYNCH-NODE`.
  - creating a new `ADDRESS-MAP` in which `MESSAGE`'s Destination-Address part is mapped to the new `ASYNCH-NODE`.
  - handling `MESSAGE` in the context of the `ASYNCH-NODE`.
- The event-driven simulation ends when the `EVENT-QUEUE` is empty.

The event-driven simulation algorithm is encoded as a composition of two temporally abstract operations, called Generate-Event-Queues-and-Nodes and Co-Earliest-EDS-Finished, and a Priority-Queue Insert. The Priority-Queue Insert is the operation performed on the first step of the simulation, which is to add a starting `EVENT` to the `EVENT-QUEUE`.

The temporally abstract operations embody the following temporally abstract view of the iterative actions of the simulator. The simulator generates two sequences: one is a

sequence of EVENT-QUEUEs and the other is a sequence of ADDRESS-MAPs, using an operation called Generate-Event-Queues-and-Nodes. It does this by repeatedly applying a function that extracts the highest priority element (an EVENT) from the EVENT-QUEUE and processes it. These two sequences feed into a temporally abstract operation called Co-Earliest-EDS-Finished. This operation returns the ADDRESS-MAP in the input sequence of ADDRESS-MAPs that corresponds to the first empty EVENT-QUEUE in the other input sequence of EVENT-QUEUEs. (These two operations are described further below.)

Temporal abstraction allows us to express this cliché as a simple composition of temporally abstract operations. The complexity of how data feeds back during iteration and how the output relates to the exit predicate is pushed down into the encoding of the individual operations.

Generate-Event-Queues-and-Nodes

Generate-Event-Queues-and-Nodes is a temporal abstraction of the iteration cliché Dequeue-and-Process-Generation, as shown in the overlay in Figure 4-19. This iteration cliché is a special case of the generation cliché. The generating function is a composition of Priority-Queue Extract and Process-Event.

This is slightly more complicated than the generation cliché described in Section 4.1.3 in that it generates two sequences, rather than one. On each iteration, the generating function is applied to the two results of the function's application on the previous iteration.

Co-Earliest-EDS-Finished

Co-Earliest-EDS-Finished is a special case of a more general temporally abstract operation, called Co-Earliest, which is related to the Earliest operation described in Section 4.1.3. Co-Earliest takes two input sequences, S_1 and S_2, and a predicate and it returns the term of S_2 that corresponds to the first term of S_1 satisfying the predicate. Co-Earliest-EDS-Finished is an instance of Co-Earliest in which the predicate is a test for whether the simulation is finished.

It is a temporal abstraction of the Co-Iterative-EDS-Finished iteration cliché, as shown in the overlay of Figure 4-20. This iteration cliché is the iterative fragment that terminates the simulation when the current EVENT-QUEUE is empty, returning the current value of the ADDRESS-MAP.

The temporally abstract operation Co-Earliest-EDS-Finished views the sequences of EVENT-QUEUEs and ADDRESS-MAPs processed over the iterations as its two inputs. It returns the ADDRESS-MAP in the sequence of ADDRESS-MAPs that corresponds to the first empty EVENT-QUEUE in the sequence of EVENT-QUEUEs.

The grammar rules in Figures 4-21 and 4-22 encode the information in the plan definitions and overlays discussed so far. A legend specifies port type abbreviations used in the figure. (The plan definitions, overlays, and the corresponding grammar rules for the

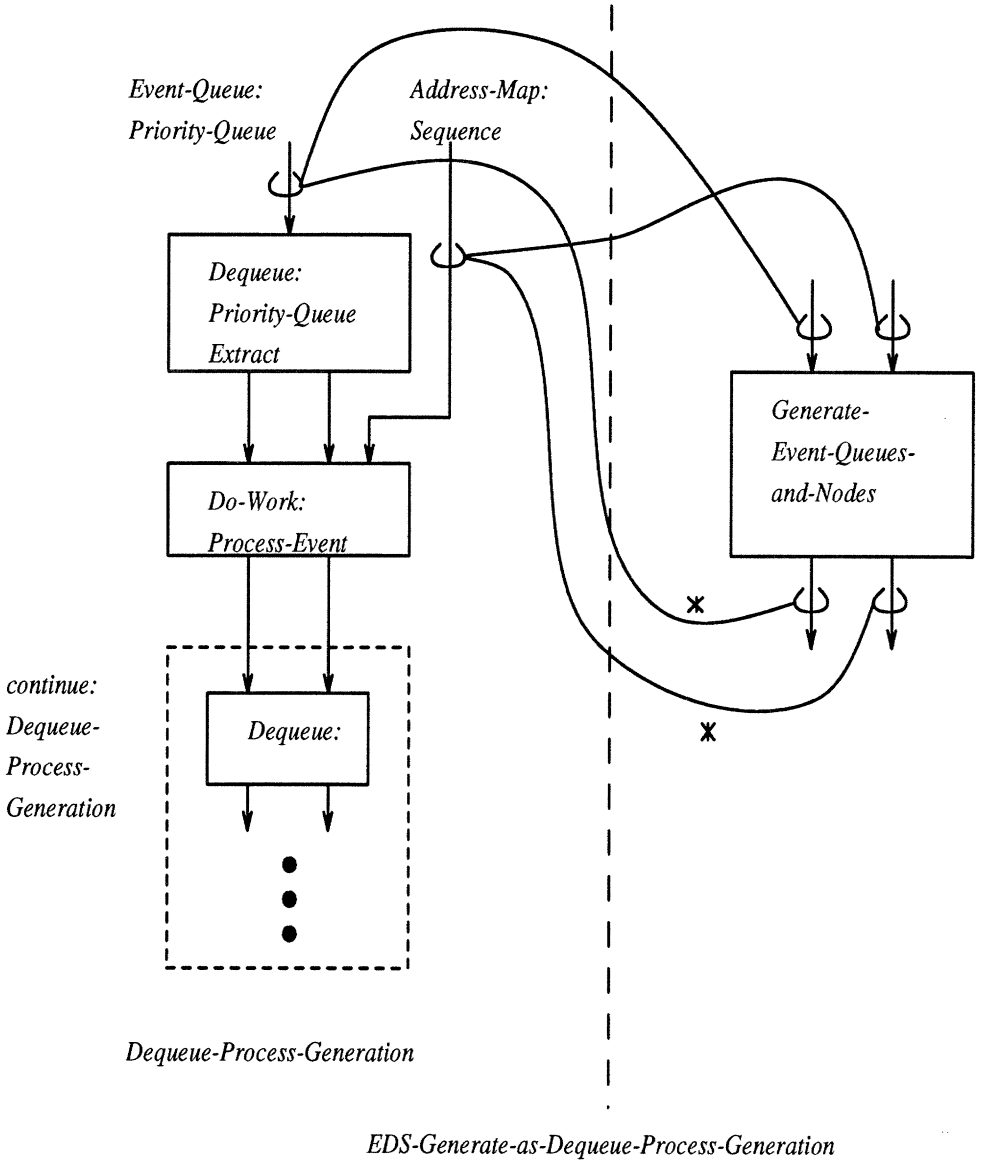


Figure 4-19: Overlay showing the temporal abstraction of the iteration cliché Dequeue-and-Process-Generation.

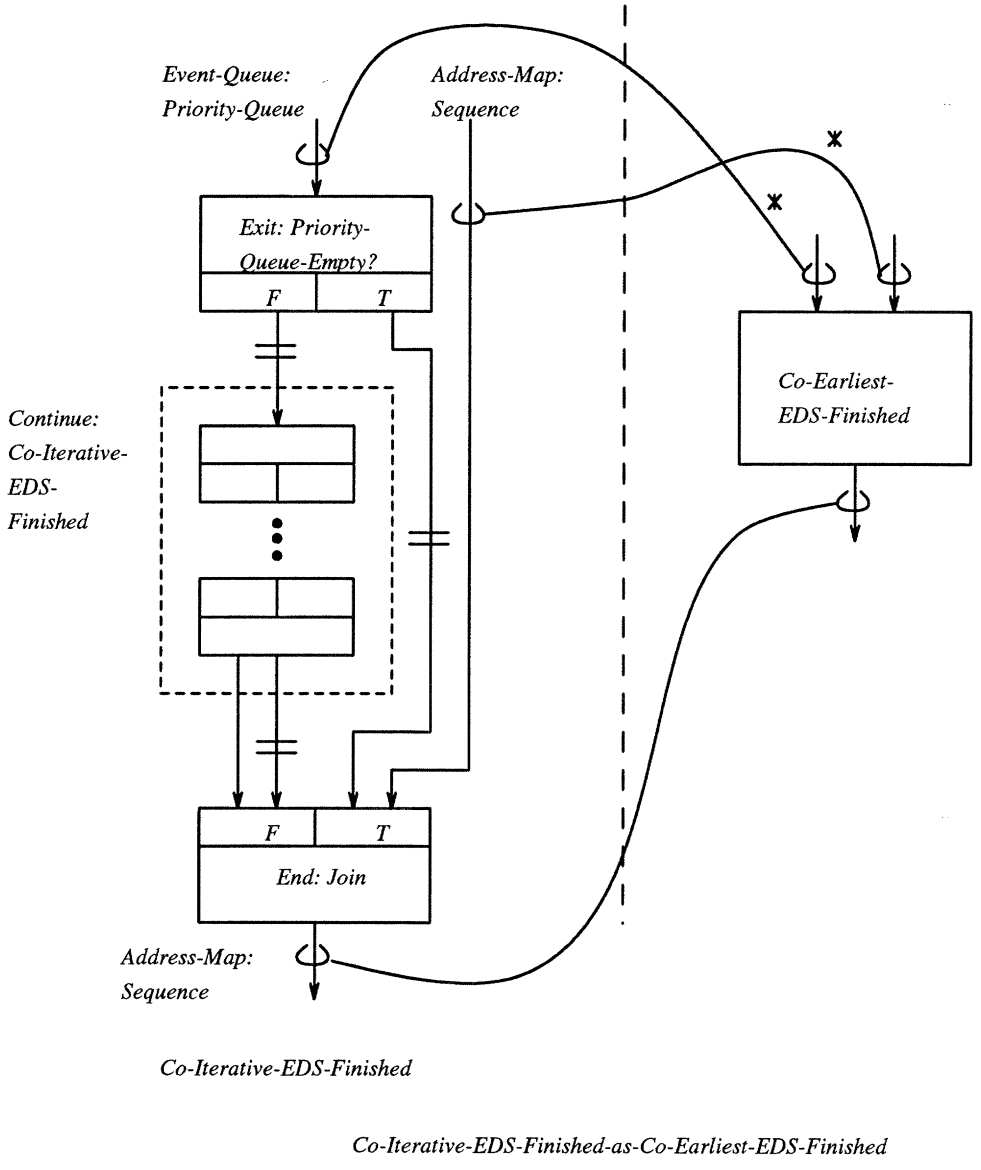
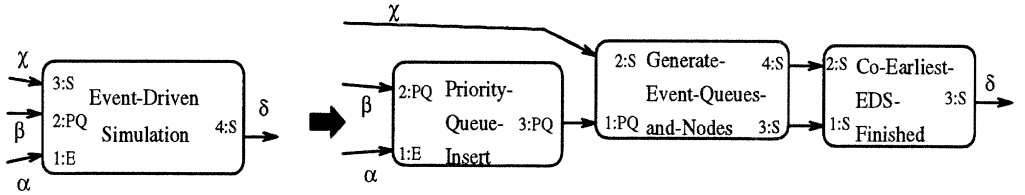


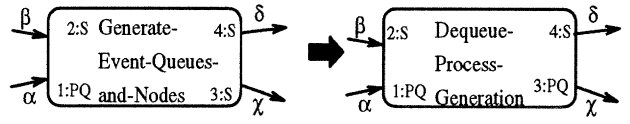
Figure 4-20: Overlay showing the temporal abstraction of the iteration cliché Co-Iterative-EDS-Finished.



Attribute Conditions: [All nodes co-occur]

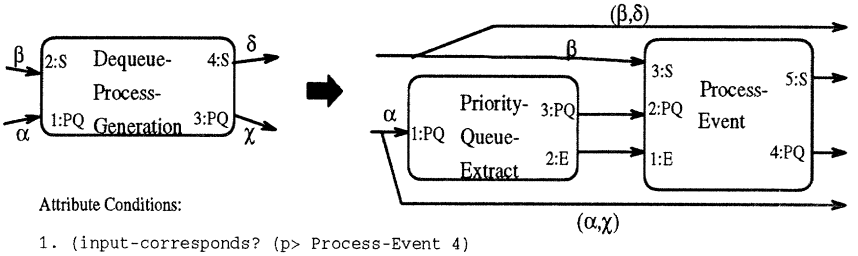
Attribute-Transfer Rules:

```
1. ce := (ce (n> Priority-Queue-Insert))
```



Attribute-Transfer Rules:

```
1. ce := (outside-ce (innermost-recur (n> Dequeue-Process-Generation)))
```



Attribute Conditions:

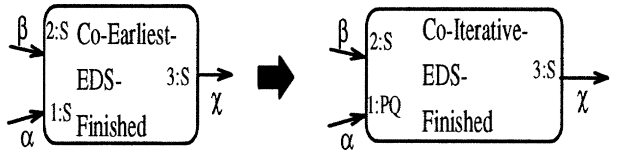
```
1. (input-corresponds? (p> Process-Event 4)
                       (p> Priority-Queue-Extract 1)
                       (feedback-ce (innermost-recur (n> Priority-Queue-Extract))))
2. (input-corresponds? (p> Process-Event 5)
                       (p> Process-Event 3)
                       (feedback-ce (innermost-recur (n> Priority-Queue-Extract))))
3. (co-occur (n> Priority-Queue-Extract) (n> Process-Event)))
```

Attribute-Transfer Rules:

```
1. ce := (ce (n> Process-Event))
```

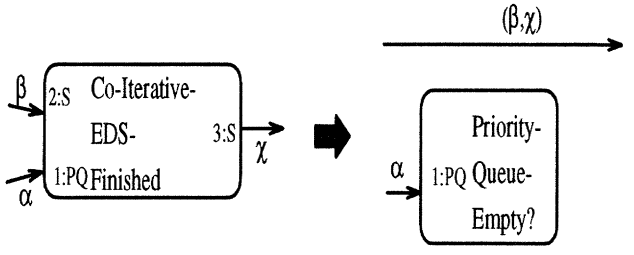
Legend:
E=Event
PQ=Priority-Queue
S=Sequence
A=Any
AN=Asych-Node
M=Message
I=Integer

Figure 4-21: Grammar rules for some Event-Driven Simulation clichés.



Attribute-Transfer Rules:

```
1. ce := (outside-ce (innermost-recur (n> Co-Iterative-EDS-Finished)))
```



Attribute Conditions:

```
1. (exit-predicate (n> Priority-Queue-Empty?))

2. (ce= (ce-from (st-thru> 2 3))
        (success-ce (n> Priority-Queue-Empty?)))
3. (ce= (ce-from (output-edge (recursive-node (innermost-recur (n> Priority-Queue-Empty?)))
                                (edge-sink (st-thru> 2 3))))
        (feedback-ce (innermost-recur (n> Priority-Queue-Empty?))))
```

Attribute-Transfer Rules:

```
1. ce := (ce (n> Priority-Queue-Empty?))

2. success-ce := (success-ce (n> Priority-Queue-Empty?))

2. failure-ce := (failure-ce (n> Priority-Queue-Empty?))
```

Figure 4-22: Grammar rules for clichés used by Event-Driven Simulation cliché.

Priority-Queue operations of Empty?, Insert, and Extract are not shown here, since they do not illustrate any new points.)

Process-Event

The plan definition for the Process-Event cliché is shown in Figure 4-23. This cliché consists of the four operations that are performed when an event is processed (as described at the beginning of this section): looking up a destination ASYNCH-NODE, updating its Clock, updating the ADDRESS-MAP, and handling the MESSAGE.

This plan contains a hierarchical data plan within it, which represents the EVENT data cliché. It has two parts: an Object (a MESSAGE) and a Time (an integer). The Object part is a MESSAGE data plan, which has four parts. The Destination-Address part (an integer) is used to index into the ADDRESS-MAP sequence to look up the destination ASYNCH-NODE. This ASYNCH-NODE is then given as input to the Update-Node-Time cliché, along with the Time part of the EVENT. A new ASYNCH-NODE is returned and NEW-TERM is used to insert it into a copy of the input ADDRESS-MAP, using the Destination-Address part of the MESSAGE as an index. Finally, a Handle-Message operation is used to simulate the handling of the MESSAGE in the Object part of EVENT. This operation takes the new ADDRESS-MAP and the EVENT-QUEUE as inputs, as well as the MESSAGE, and returns an ADDRESS-MAP and EVENT-QUEUE.

Figure 4-24 shows the rule that encodes the Process-Event cliché, plus two rules that derive the non-terminals Lookup-Destination and Record-at-Destination. These two additional rules are needed because we cannot directly encode the hierarchical data plan for EVENT in the embedding relation of one grammar rule. Grammar rules can only represent one level of aggregation at a time. (This is a limitation of the current implementation of GRASPR. It does not appear to reflect an inherent difficulty with the graph parsing approach.) To get around this limitation, we decompose the dataflow graph structure of the plan so that we separate those parts that access parts of the MESSAGE from those that access the EVENT. We then create rules taking the non-terminals Lookup-Destination and Record-at-Destination to the sub-flow graphs representing those parts that access the parts of MESSAGE.

The rules for Lookup-Destination and Record-at-Destination contain embedding relations in which a left-hand side port is mapped to a tuple containing some empty elements (denoted by asterisks). This represents the fact that not all of the parts of the MESSAGE data structure are used by the operations represented by nodes on the rule's right-hand side.

Part of the Process-Event cliché is the Handle-Message operation. We have grammar rules that encode one possible clichéd implementation of this operation. (These are not shown here, since they are more of the same type we have seen already.)

However, we would also like to allow Process-Event (and the rest of the Event-Driven Simulation cliché) to be recognized in simulators in which the Handle-Message operation is non-clichéd. That is, we would like to think of this as applying a non-clichéd function to the MESSAGE which simulates the handling of a real message by a real processing node.

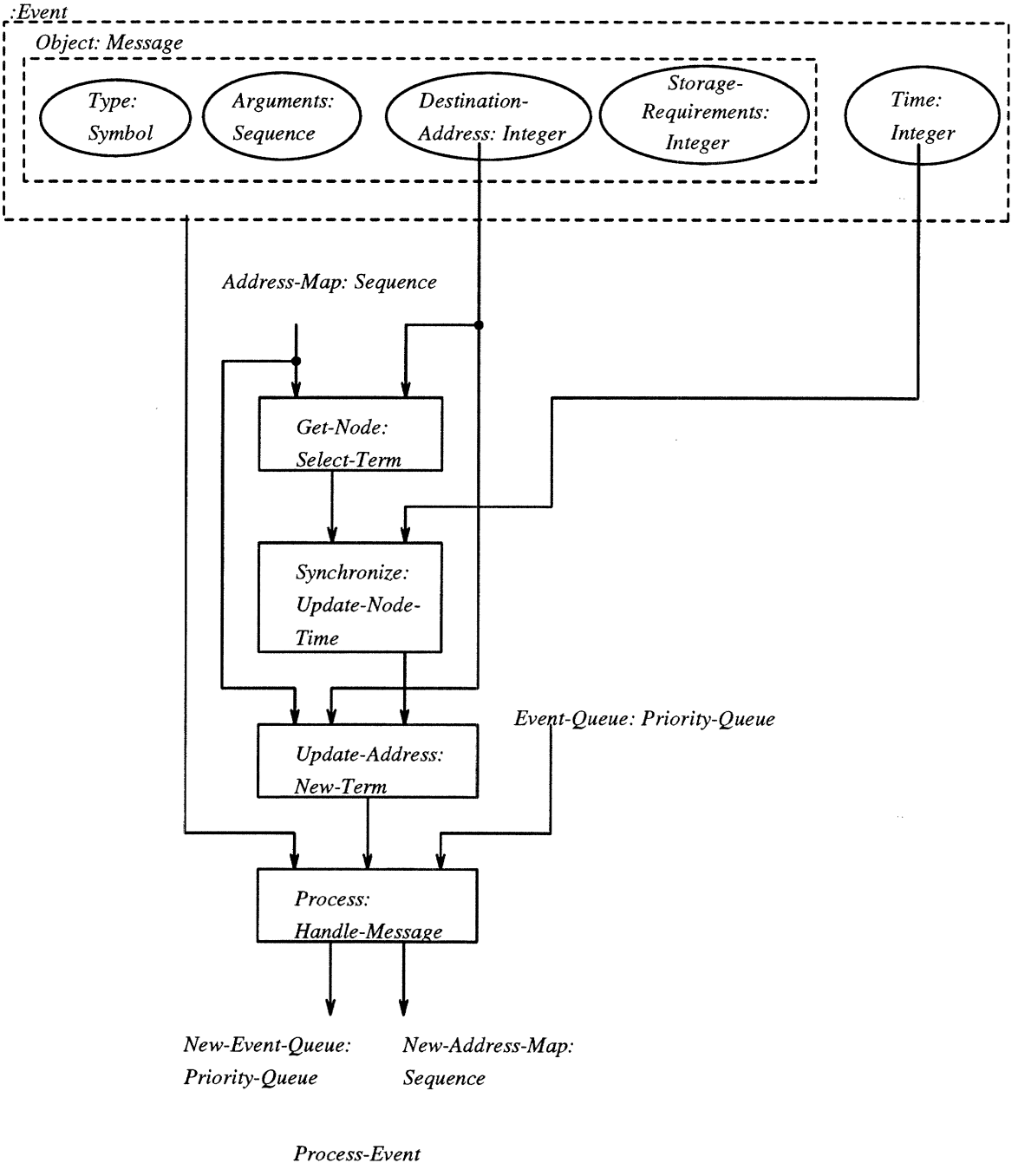
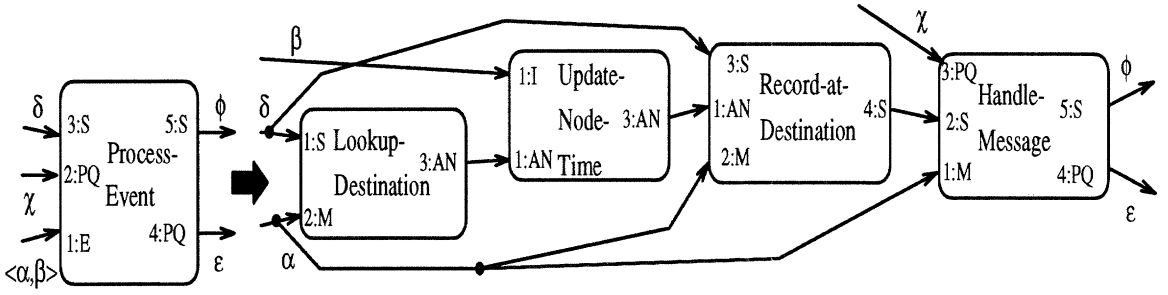


Figure 4-23: Plan definition for the Process-Event cliché.

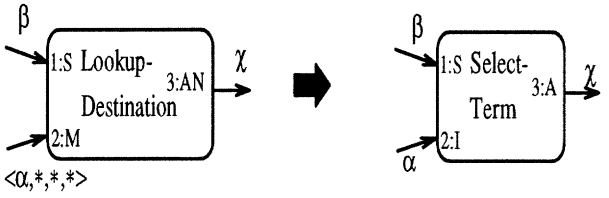


Attribute Conditions: [All nodes co-occur]

Attribute-Transfer Rules: `1. ce := (ce (n> Lookup-Destination))`

Mnemonic tuple element names:

<Object, Time>

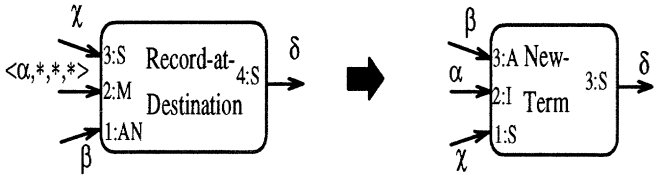


Attribute-Transfer Rules:

`1. ce := (ce (n> Select-Term))`

Mnemonic tuple element names:

<Destination-Address, Type, Arguments, Storage-Requirements>



Attribute-Transfer Rules:

`1. ce := (ce (n> New-Term))`

Mnemonic tuple element names:

<Destination-Address, Type, Arguments, Storage-Requirements>

Figure 4-24: Rules for Process-Event cliché.

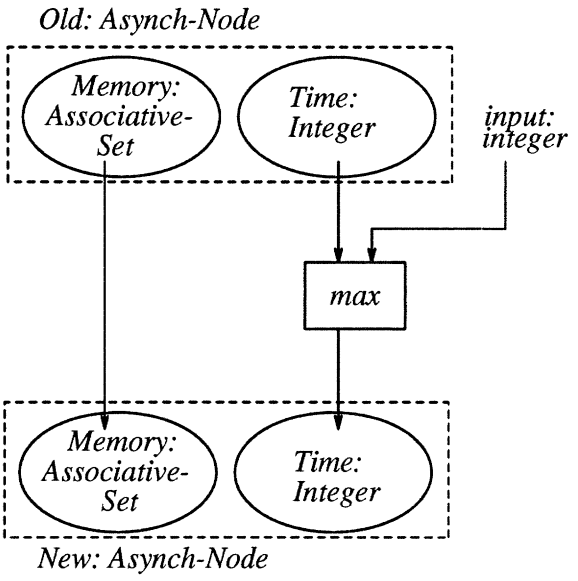


Figure 4-25: Plan definition for the Update-Node-Time cliché.

Unfortunately, it is difficult to do this within the graph parsing framework. It would require the Handle-Message non-terminal in the rule for Process-Event to derive an arbitrary flow graph. In general, it is difficult to express and match a cliché that is parameterized over non-primitive, non-clichéd functions. (This is the same problem we ran into in codifying the generation cliché in Section 4.1.3. See Section 5.2.3 for more discussion of this problem.)

Update-Node-Time

Update-Node-Time is a clichéd operation that synchronizes an `ASYNCH-NODE`'s Clock to the current "simulated time," which is the time of the most recent `EVENT` pulled from the `EVENT-QUEUE`. The operation takes a `ASYNCH-NODE` and the simulated time (an integer) and returns a new `ASYNCH-NODE` whose Clock is either the simulated time or the time of the input `ASYNCH-NODE`'s Clock, whichever is later. The plan definition of this operation is shown in Figure 4-25. An `ASYNCH-NODE` has two parts: a Memory (an Associative Set) and a Time (an Integer). This cliché takes an `ASYNCH-NODE` and an integer and creates a new `ASYNCH-NODE` whose Time part is the maximum of the input integer and Time part of the input `ASYNCH-NODE`. The Memory part of the output is the same as that of the input `ASYNCH-NODE`. The rule that encodes this plan definition is shown in Figure 4-26.

Enqueuing New Events

One of the actions of a processing node that is simulated as part of the simulation of message handling is the creation and sending of new messages. One of the constraints on the event-driven simulation algorithm is that whenever a message send is simulated, a new `EVENT`

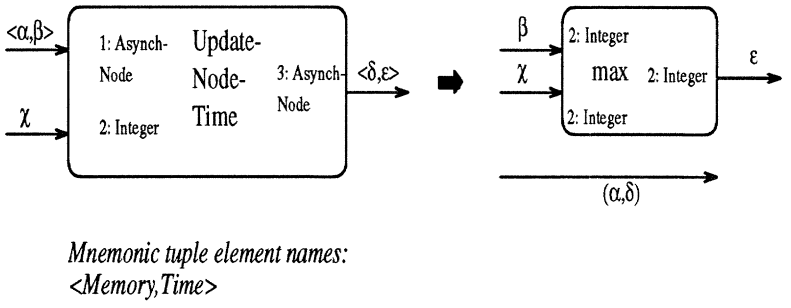


Figure 4-26: Grammar rule encoding the Update-Node-Time plan.

must be created and added to the **EVENT-QUEUE**. (Similarly, in the synchronous simulation algorithm, when the message handling simulation simulates the sending of a message, the **MESSAGE** that represents it must be added to the global **MESSAGE** buffer.)

Unfortunately, this constraint is difficult to express in the grammar rule encoding and to check in the simulator code. Partly this is because the node action simulation code is not guaranteed to be clichéd, so we have no context in which to express the constraint. Another reason is that the part of the simulation code that performs the activity of enqueuing new **EVENT**s (or **MESSAGE**s) is typically given as *input* to the simulator. So, it is not available for analysis. (As discussed in Section 2.2, `PiSim` takes as input a set of functions each of which specifies how to simulate the actions of a node in executing some machine operation. Some of these functions create new **EVENT**s and enqueue them.) These problems are discussed further in Section 5.2.4.

Although this constraint is difficult to express and check within the current graph parsing framework, it is not a hard constraint for a person to check. It might be easier to just ask the user whether the constraint holds. This question can be asked with reference to the particular locations in the program, corresponding to locations in the input graph where the Handle-Message operation is likely to occur. (This can be based on where the rest of Process-Event has been found.)

4.2 Architectural Details

This section fills in details of how flow graph parsing is used to solve the partial program recognition problem. Section 4.2.1 describes how textual source code is translated into an attributed flow graph. Section 4.2.2 discusses an additional monitor that tailors the parser to deal with a type of graph variation that is specific to the program recognition application. Section 4.2.3 describes how the Paraphraser presents the parser's results.

4.2.1 Translating Programs to Flow Graphs

A program is translated from source code to attributed flow graph in two stages. First, a plan representation of the source code is created. Then, an attributed flow graph is com-

puted from this intermediate representation. Creating the intermediate plan representation of the code facilitates the computation of attributes for the flow graph.

Source Code to Plan Diagram

The plan creation stage is itself composed of two stages: macro-expansion, followed by symbolic evaluation. The macro-expander translates the program into a simpler language of primitive forms. It does this by expanding any macro calls in the source program and by using a set of additional macro-like definitions to expand each complex construct in the source into a set of simpler forms. In particular, all of the control constructs are converted to simple conditional and unconditional branches. All of the data constructs are converted into bindings of or assignments to simple atomic variables.

The macro-expanded code is then symbolically evaluated. The evaluator follows all possible control paths of the program, starting with some topmost ("main") function of the program. It converts operations to boxes and places arcs between them, corresponding to data and control flow. Whenever a branch in control flow occurs, a test box is added. Similarly, when control flow comes back together, a join box is placed in the graph and all data representing the same variable are merged together.

Boxes for user-defined functions are replaced with the plans for their definitions, except for those within recursive functions. This flattening allows variability in the way programs to be analyzed are broken down into subroutines. The user may also advise that certain calls not be expanded for efficiency reasons. (Any unexpanded function whose name happens to be a non-terminal in the grammar is systematically renamed, unless the user specifies that the function is an instance of the cliché named by the non-terminal.)

The symbolic evaluator inserts explicit selector and constructor boxes into the plan diagram for each user-defined accessor and constructor.

The plan representation may be used as the target representation for many different languages. The flow analyzer used by GRASPR translates Lisp programs into plans. Similar analyzers were previously written not only for Lisp ([114, 137, 139]), but also for subsets of Cobol [42], Fortran [137], and Ada [139], but are not used in this system.

Plan Diagram to Attributed Flow Graph

Once the plan representation for the program is created, it is encoded as an attributed flow graph. The dataflow structure of the plan is retained in the flow graph. Control environment attributes are computed from the control flow structure. Joins are replaced with edges that fan in, annotated with ce-from attributes. Explicit accessors and constructors are also replaced by attributed edges. Each accessor and composition of accessors is treated as a Spread node and each constructor as a Make node. These Spreads and Makes are removed using the aggregation-removal transformations described in Section 3.4.2. The residual Spreads and Makes are then replaced with attributed fan-out and fan-in edges.

```
(defun Insert-Queue (Entry)
  (cond ((Empty-or-Low-Priority-Head? Entry *Event-Queue*)
         (push Entry *Event-Queue*))
        (t (let ((Next (cdr *Event-Queue*))
                 (Previous *Event-Queue*))
             ;; find spot to splice Entry in:
             (loop do
               (when (Empty-or-Low-Priority-Head? Entry Next)
                 (return))
               (setq Previous Next)
               (setq Next (cdr Next)))
             ;; perform the splice:
             (rplacd Previous (cons Entry Next))))))
```

Figure 4-27: Code that side effects the mutable data structure `*Event-Queue*`.

4.2.2 Additional Monitor to Handle Recursion Unfolding

One of the types of variations that can arise in recursive programs is that a loop in one can be unrolled in another, or more generally, a recursion can be unfolded. This variation arises in our program examples when we convert the impure programs to pure ones (having no side effects to mutable objects). In this situation, special cases of a recursion sometimes translate to the general recursive case. This means that the general case is redundantly performed once, before the recursion is called.

For example, the code in Figure 4-27 destructively inserts `Entry` into the ordered associative list `*Event-Queue*`. It first tests for the special case in which `Entry` belongs on the front of the list (either because the list is empty or its first element has a lower priority than `Entry`). In this case, it destructively places `Entry` on the front of `*Event-Queue*` using `push`. `Insert-Queue` then performs the general case in which `*Event-Queue*` is searched for the place to insert `Entry` and then `Entry` is spliced in at that place.

When this program is translated into its non-destructive version, shown in Figure 4-28, the special case head insertion becomes the same as the normal splice-in operation. `Insert-Queue-Pure` can be rewritten as `Folded-Insert-Queue`, shown in Figure 4-29, in which the recursion is folded back up.

To deal with this type of variation, we provided an additional monitor to the flow graph parser, which looks for an opportunity to view a program that contains an unfolded recursion as one in which the recursion is folded back up. By generating this alternative view, the parser is then able to recognize the program as if it did not have an unfolded recursion. This augmentation of the parser with a new monitor tailors it to solve a problem specific to its application to the program recognition problem. This section describes the new monitor and how the new view is generated.

```
(defun Insert-Queue-Pure (Entry)
  (setq *Event-Queue*
        (cond ((Empty-or-Low-Priority-Head? Entry *Event-Queue*)
               (cons Entry *Event-Queue*))
              (t (cons (car *Event-Queue*)
                       (Splice-in Entry (cdr *Event-Queue*)))))))

(defun Splice-In (Entry Next)
  (cond ((Empty-or-Low-Priority-Head? Entry Next)
         (cons Entry Next))
        (t (cons (car Next)
                 (Splice-In Entry (cdr Next))))))
```

Figure 4-28: Functional version of `Insert-Queue`.

```
(defun Folded-Insert-Queue (Entry)
  (setq *Event-Queue* (Splice-In Entry *Event-Queue*)))

(defun Splice-In (Entry Next)
  (cond ((Empty-or-Low-Priority-Head? Entry Next)
           (cons Entry Next))
        (t (cons (car Next)
                 (Splice-In Entry (cdr Next))))))
```

Figure 4-29: Version of `Insert-Queue-Pure` in which recursion is folded up.

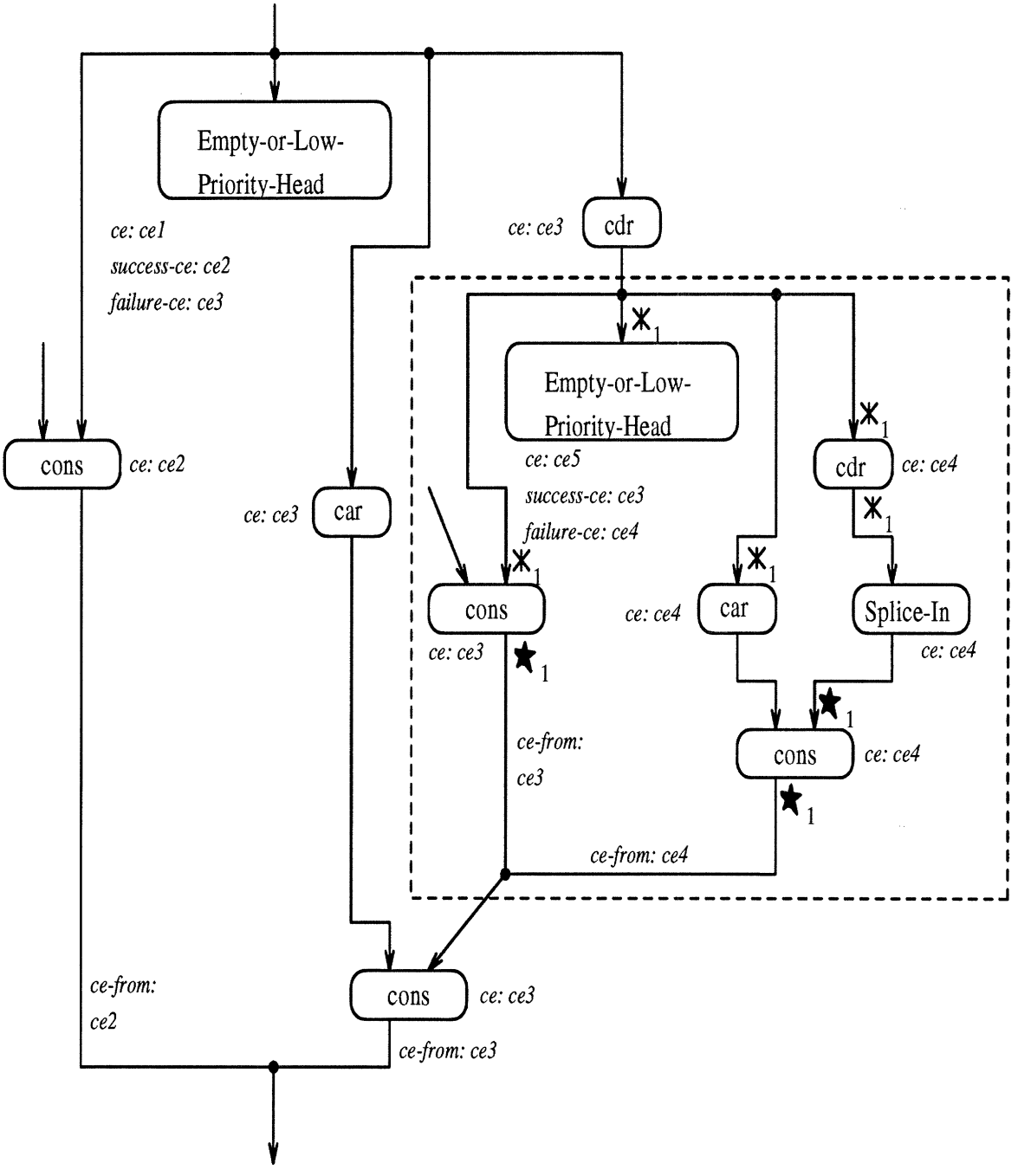


Figure 4-30: Flow graph representing `Insert-Queue-Pure`.

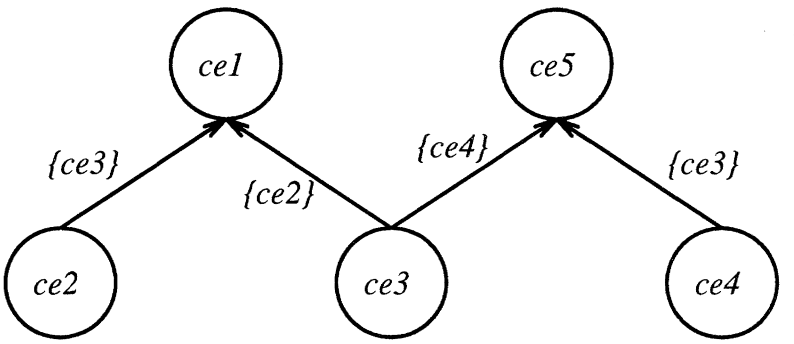


Figure 4-31: Partial ordering relationships between the control environments of `Insert-Queue-Pure`'s flow graph.

Figure 4-30 shows the flow graph representation of `Insert-Queue-Pure`. A dashed box is drawn around the boundary of the sub-flow graph representing its recursion. `GRASPR` generates an alternative view of this flow graph in which the recursion boundary is expanded outward and the redundant computation is collapsed together.

The way it works is based on the observation that when `GRASPR` tries to recognize an unfolded program, most of the constraints (structural as well as attribute conditions) are satisfied. The only ones that are not are those that refer to the program's recursion information (e.g., those constraining two ports to input-correspond or those referring to the feedback-ce of the recursion).

So, constraints are placed into two classes: *regular* and *recursion.* When an item fails only its recursion constraints, it is *suspended*, which means it is placed in a holding data structure used by the new monitor. The monitor watches for another complete item, called a *partner*, to be added to the chart that can *collapse* with the suspended item. An item I_s can collapse with another item I_p if they are recognizing the same non-terminal type in control environments that are *analogous.* (This relation is defined below.) Collapsing two items means creating a new item which is the same as the suspended item, but whose constraints are checked in the context of the partner item.

Intuitively, two control environments are *analogous* if they contain operations that would collapse together if the recursion were folded back up. For example, Figure 4-31 shows the partial ordering of the control environments and recursion information for `Insert-Queue-pure`. The analogous pairs of control environments are $(ce1, ce5)$, $(ce2, ce3)$, and $(ce3, ce4)$.

The analogy relations are symmetric, but not reflexive, or transitive. Analogy relations between control environments are computed from the surface plan during its translation to an attributed flow graph.

Once a suspended item is collapsed with a partner, the new "collapsed" item is added to the agenda. Its constraints are satisfied because they refer to attributes of the sub-flow

graph matched by the partner item. The collapsed item's left-hand side control environment attributes are computed by applying the rule's attribute-transfer rules in the context of the partner item and then translating them to the analogous control environment. (Attribute-transfer rules that use recursion information in their computation are handled specially. In particular, if the rule computes the *outside-ce* of the innermost recursion containing some node, the control environment analogous to the *recur-ce* of this recursion is transferred.)

When a collapsed item is used to extend another item, it imposes new edge connection constraints on the items for adjacent non-terminals. Suppose a collapsed item I_A, having partner I_p extends another item to create an item I_C, where I_A is representing the derivation of non-terminal A in the right-hand side of I_C's rule. If an item I_B for a non-terminal adjacent to A has a partner I_q, then I_p and I_q should be connected together in the same way as I_A and I_B.

The suspend-collapse-resume mechanism for recursion folding can be generalized to a "try-harder" technique for handling more types of near-misses besides those that fail recursion constraints. More classes of constraints can be identified. When an item fails certain classes of constraints, something might be done to cause them to be satisfied (e.g., changing an attribute) or weakened (e.g., changing a co-occurence condition between two nodes to a $\sqsubseteq$ condition). Then the item can be resumed simply by putting it back on the agenda. The changes can be reported as conditions or assumptions under which some cliché is recognized in the program.

4.2.3 Paraphraser

The output of the recognition process is a forest of design trees, representing the clichés found and how they relate to each other. One way to use this output is to automatically generate documentation for the program recognized. Paraphraser is a tool which takes the forest of design trees produced by `GRASPR` and generates textual documentation for each. Each cliché in our library has an associated schematized textual explanation fragment whose slots may be filled in with identifiers in the program. (This is based on earlier work by Cyphers [24] and Frank [45].)

Paraphraser starts at the root of a design tree and traverses it depth first, generating a hierarchical description based on the explanation fragments associated with each cliché encountered. It reports the relationships between each cliché in the tree and those immediately below it (e.g., `Queue-Insert is implemented by FIFO-Enqueue`, `Sum temporally abstracts Summing`). If an implementation relationship exists between two clichés and a data abstraction is uncovered, this is reported as well (e.g., `The Queue is implemented as a FIFO.`).

Variable names are included in the text to indicate the location of the cliché. Also, some slots in the explanation fragments are filled in with primitive operation types, such as `<` in `An element's priority P is higher than another's Q, if P < Q.` This often happens when generalized node types are used. In this case the generalized node type matched

any primitive predicate that was a comparator. Paraphraser is also able to compute some mappings from user-defined data structure part names to the part names of aggregate data clichés that are recognized. This is described below.

The user can select which design trees to document. By default, Paraphraser documents all of them, starting with those whose roots are at the highest level in the library. Currently, all clichés recognized are reported, including those that represent multiple views of some part of the program. No single best interpretation is preferred. We view the job of selecting views of the program and focusing on particular results of the recognition as the responsibility of a higher-level control mechanism which has information about how the results will be used and which view of the program is most useful.

Mapping Clichéd Aggregate Names to User-Defined Data Structure Names

Paraphraser heuristically computes mappings from the names of user-defined data structures and their parts to those of aggregate data clichés that are recognized in the program. However, the current implementation is not robust. The mappings are often incomplete and ambiguous. (This is an area requiring further work.)

The names of user-defined data structures and their parts are associated with edges in the program's flow graph in the form of *accessor* and *constructor* attribute values. Each accessor attribute has a value that describes how the data it carries to the edge's sink is a part of the data structure at the edge's source. Because data structure accesses and constructions can be composed, the values of these attributes are sets of *ordered lists* of tuples of the form `<structure-type part-name>`, where the order corresponds to the order of composition of the accesses or constructions. They are *sets* of ordered lists because an edge can represent dataflow from more than one output of a selector to more than one input of a constructor. For example, in the flow graph representing `(1+ (queue-length (node-queue (aref *nodes* i))))`, the edge from the output of "aref" to the input of "1+" has an accessor attribute of value `(<Node Queue> <Queue Length>)`.

Each ordered list can be seen as a "path" that describes how the source data structure is destructured to result in the piece of data at the sink. The path may be of arbitrary length, since the piece of data may be nested deeply within several data structures.

Similarly, each edge holds a *constructor* attribute that describes how the data it carries becomes part of some data structure. The value of the accessor and constructor attributes is undefined if the edge is not carrying data involved in some aggregation.

The edge attributes are used to create the mappings between names in clichéd structures and in user-defined ones. When an operation on a clichéd aggregate data structure is recognized, the parser has matched each part of the structure to an edge (or recursively to a tuple of sub-part matchings, if the part itself is an aggregation). This creates a tree representing the clichéd aggregate data structure's organization, with the leaves matching edges in the flow graph representing the program. Those accessor and constructor values

```
FIFO Dequeue is implemented as a Circular
Sequence Extract.  The FIFO is implemented as a CIS.
  Circular Indexed Sequence Extract extracts the
  first element from the Circular Indexed Sequence.
  The First part: (<NODE QUEUE> <QUEUE HEAD>)
  The Fill-Count part: (<NODE QUEUE> <QUEUE LENGTH>)
  The Size part: (<NODE QUEUE> <QUEUE DATA-SIZE>)
  The Base part: (<NODE QUEUE> <QUEUE DATA>)
```

Figure 4-32: Documentation containing a clichéd-to-user-defined name mapping.

that are defined are combined to form trees that represent the portions of the user-defined data structure organization. (There may be more than one if the recognition involves parts from more than one user-defined data structure.) The fringes of these trees are matched to the fringes of the clichéd organization tree. This generates mappings between the part names of the lowest level structures involved. Mappings between higher level nodes of the trees are heuristically computed. For example, if all parts of a clichéd data structure map to all parts of a user-defined structure, then the two data structures map to each other.

Equality constraints are imposed locally by the rules for cliché data structure operations. These require that each clichéd part name map consistently to the same programmer-defined part name (or set of names, if there is ambiguity in which attributes match).

Figure 4-32 gives an example of a mapping computed from the recognition of a CIS-Extract. The mapping is included in the documentation of this cliché. This mapping is incomplete in that the "Last" part of the Circular Indexed Sequence is not mapped to anything. This is because in the program, the optional unconstrained straight-through representing the "Last" part was not matched. Because not all of the parts of the clichéd data structure are mapped, the mapping cannot be refined. If Last were mapped to (<NODE QUEUE> <QUEUE TAIL>), then since the user-defined data structure QUEUE has no more parts, QUEUE can be mapped to CIS and each of the part mappings can be reduced from (<NODE QUEUE> <QUEUE x>) to (<QUEUE x>). If "Last" were mapped to (<NODE MAX-INDEX>), and NODE had only parts "Queue" and "Max-Index," then NODE would be mapped to CIS and the mappings would remain the same (i.e., not be reduced).

Ambiguity arises when an accessor or constructor attribute has a set of values that are mapped to some clichéd part. It also occurs when some part of a program is recognized as more than one data structure operation.

In addition to these local refinements to the mappings, global constraint propagation should be used to refine them further. Future research will focus on this. The results can be valuable not only in presenting the results of recognition, but also as a source of expectations which can be used to further guide and refine data structure recognition. (See Section 7.2.3.)

Chapter 5

Capabilities and Limitations

There are two parts of our analysis of the graph parsing approach. One is identifying its practical capabilities and limitations in the context of real-world programs. The other is studying the computational cost of this approach. This chapter discusses the first aspect, while Chapter 6 deals with the second. In this chapter, we consider both the robustness of our recognition technique under common program variations and the expressiveness of our graph grammar formalism for encoding programming clichés.

5.1 Variations Tolerated

Automated recognition of clichés must be robust under a wide range of variations in programs. We employ three basic strategies for achieving this goal. First, we use an *abstract representation* for programs and clichés. This representation suppresses many details which can vary across programs but which do not constitute significant differences between the clichés that exist in the programs. Our representation exposes the algorithmic and dataflow structure of the program, while abstracting away syntactic and organizational differences.

When some unimportant details are not suppressed by our representation (i.e., when two or more program variations are not represented the same), we try a second strategy. We provide ways for GRASPR to generate *cheap alternative views* of the program representation. These views are created by additional chart monitors during parsing, such as those that deal with redundancy.

It is possible to also handle this in a pre-processing stage (rather than during parsing) by choosing one variation as canonical and applying cheap transformations to canonicalize other variations with respect to this one. However, sometimes seeing the transformation opportunity requires performing recognition. For example, zipping up two instances of an abstract operation that each involve a different implementation requires recognition to view the redundant code as performing the same operation.

When a cliché exists in two programs that are not represented the same in our representation or cannot be cheaply viewed as the same, we fall back on our third strategy. This is

to *enumerate* the variations in our library. For example, we use this tactic to deal with implementation variation. However, when enumerating variations, we rely on our knowledge of the empirical frequency of occurrence of the variations. We do not collect every variation of a cliché we can think of, only those that are common. The hierarchical structure of the cliché library helps to make the enumeration concise.

These three tactics allow us to automate program recognition so that it is robust under the common program variations described in Section 2.3.1. Our abstract representation eliminates syntactic and organizational variation, as well as variation due to delocalization, unfamiliar code, and some function-sharing optimizations. This is discussed in more detail in Sections 5.1.1-5.1.5. By generating alternative views cheaply, GRASPR is able to deal with variation due to redundancy, as is discussed in Section 5.1.6. Because implementation variations are concisely enumerated in the cliché library, GRASPR is able to recognize the same abstract clichéd operation in programs that contain different implementations of the operation. This is discussed in Section 5.1.7.

5.1.1 Syntactic Variation

In Section 2.3.2, we showed two programs (in Figures 2-10 and 2-11) which GRASPR recognized as containing the same clichés, even though they differ syntactically. This is due to the fact that both programs are represented as the same flow graph, shown in Figure 5-1.

The figure does not show the complete flow graph. Some function calls are depicted as nodes for brevity. However, they are sub-flow graphs in the actual representation. These nodes are drawn with dotted lines to show that they hide some detail. Also, dashed lines are drawn around the sub-flow graph representing the recursive function Execute-Events. (Small filled-in circles indicate fan-in and fan-out. They are not special vertices in the flow graph. They are used to distinguish edges that share sinks or sources from those that merely cross each other.)

Accessor and constructor attributes on edges are not shown in the figure because they differ for the two programs. Instead, the edges for which these attributes have defined values (i.e., not undefined) are labeled <e1>, ...,<e7>. Figure 5-2 lists the actual attribute values for these edges for the programs of Figures 2-10, 2-11, as well as Figure 2-12.

The flow graph representation abstracts away syntactic differences between programs. Attributed dataflow edges explicitly represent the net effect of binding and control constructs, abstracting away such details as which constructs are used, which variables are bound, and whether data is passed through nested expressions or via bindings to intermediate variables.

Information concerning the names of user-defined data structures and their parts is relegated to edge attributes, so that differences due to explicit accessor and constructor functions do not arise in the structure of the graph.

Also, the representation captures only "essential" orderings of operations, which are

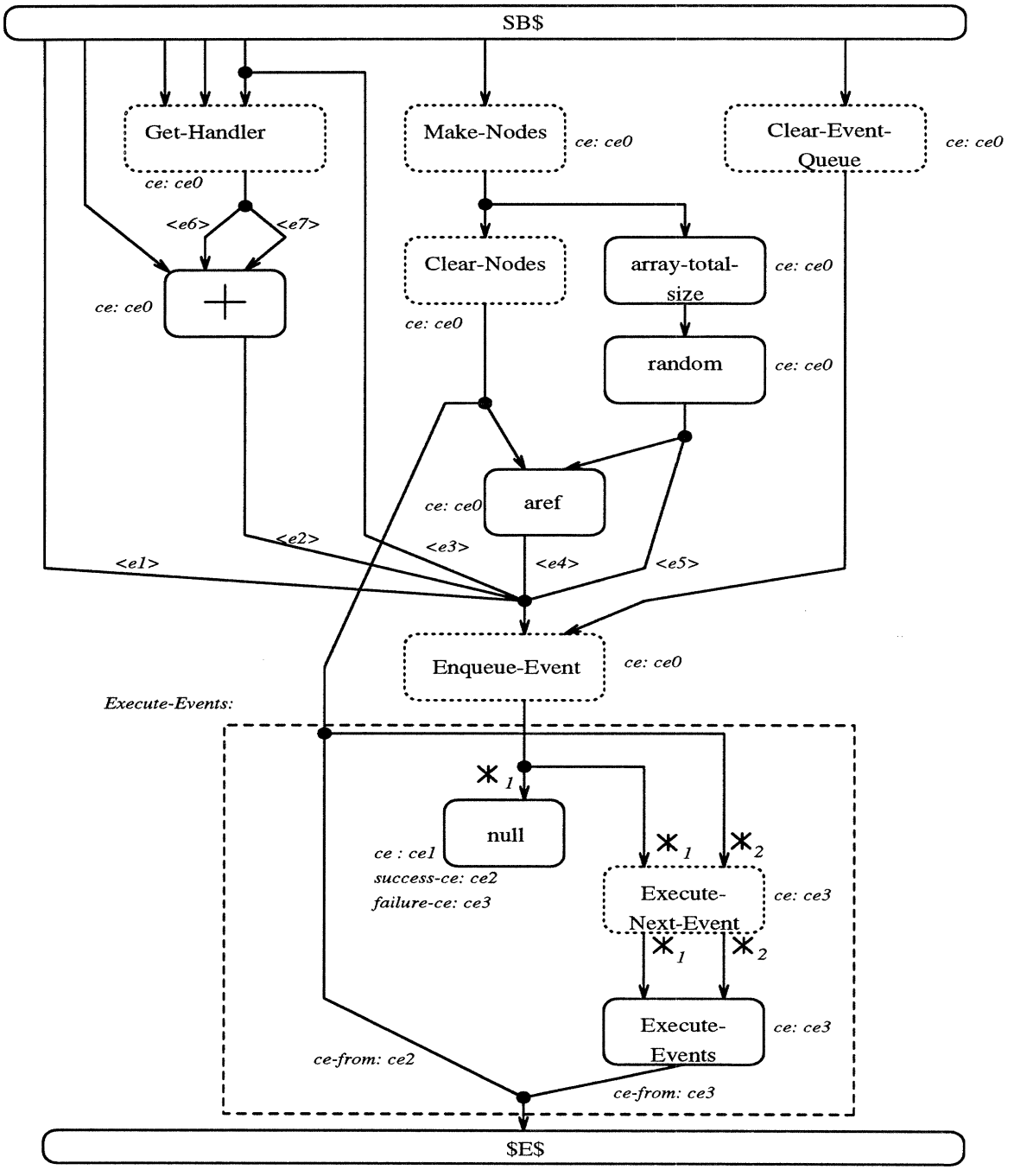


Figure 5-1: Flow graph representing the code in Figures 2-10, 2-11, and 2-12.

		a	b
<e1>:	*Accessor:*	*undefined*	*undefined*
	Constructor:	*{(<Message Arguments> <Event Object>)}*	*{(<Msg Args> <Event Object>)}*
<e2>:	*Accessor:*	*undefined*	*undefined*
	Constructor:	*{(<Message Length> <Event Object>)}*	*{(<Msg Storage-Length> <Event Object>)}*
<e3>:	*Accessor:*	*undefined*	*undefined*
	Constructor:	*{(<Message Type> <Event Object>)}*	*{(<Msg Type> <Event Object>)}*
<e4>:	*Accessor:*	*{(<Node Time>)}*	*{(<Node Time>)}*
	Constructor:	*{(<Event Time>)}*	*{(<Event Time>)}*
<e5>:	*Accessor:*	*undefined*	*undefined*
	Constructor:	*{(<Message Destination> <Event Object>)}*	*{(<Msg Dest-Addr> <Event Object>)}*
<e6>:	*Accessor:*	*{(<Handler Arity>)}*	*{(<Handler Arity>)}*
	Constructor:	*undefined*	*undefined*
<e7>:	*Accessor:*	*{(<Handler Number-of-Locals>)}*	*{(<Handler Number-of-Locals>)}*
	Constructor:	*undefined*	*undefined*

		c
<e1>:	*Accessor:*	*undefined*
	Constructor:	*{(<Handler-Data Arguments> <Msg Data>)}*
<e2>:	*Accessor:*	*undefined*
	Constructor:	*{(<Handler-Data Length> <Msg Data>)}*
<e3>:	*Accessor:*	*undefined*
	Constructor:	*{(<Handler-Data Type> <Msg Data>)}*
<e4>:	*Accessor:*	*{(<Node Time>)}*
	Constructor:	*{(<Msg Arrival-Time>)}*
<e5>:	*Accessor:*	*undefined*
	Constructor:	*{(<Msg Destination>)}*
<e6>:	*Accessor:*	*{(<Handler Arity>)}*
	Constructor:	*undefined*
<e7>:	*Accessor:*	*{(<Handler Number-of-Locals>)}*
	Constructor:	*undefined*

Figure 5-2: Attribute values for accessor and constructor attributes annotating the flow graphs representing the programs in Figures 2-10 (column a), 2-11 (column b), and 2-12 (column c).

those determined by dataflow dependencies. Dataflow graphs make dataflow dependencies explicit, imposing a partial ordering on the program's operations (rather than the linear, total ordering imposed by text). So programs which vary only in their ordering of independent computations will have the same flow graph representation.

The attributed flow graph representation also captures constraints on data and control flow, independent of the language in which they are expressed. This means the same library of clichés can be used to recognize clichés regardless of the language in which the program containing them is written. If the data and control flow of a program can be statically determined, then the program can be represented as an attributed flow graph. This is true for most imperative, sequential programs written in conventional languages, such as Fortran, Cobol, Lisp, and Ada.

Some examples of programs for which this is not true are those that contain nondeterministic or concurrent language features. Also, programs that take other programs as input cannot be fully modeled by our dataflow graph representation because part of their data and control flow information is hidden in their input. (This is discussed further in Section 5.2.)

The abstraction properties of the flow graph representation enable clichés to be recognized in programs without having to anticipate (and enumerate) all possible syntactic variations of each cliché and without relying on source-to-source transformations to canonicalize the code.

5.1.2 Organizational Variation

The flow graph representation is also the key to dealing with variation in how programs are decomposed into subroutines and how aggregate data structures are organized. In this representation, the subroutine structure is flattened. Each call to a subroutine is represented by the flow graph of the subroutine's body. In essence, the program is seen as completely open-coded. The key benefit of this is that instances of clichés which cross subroutine boundaries are recognized as easily as those that are within a boundary. The hierarchical organization of clichés built upon other clichés need not be reflected in the program's decomposition for the clichés to be recognized.

Of course, flattening all subroutine calls is not always advantageous. When a subroutine is used in several places throughout the code and contains clichés entirely within its boundaries, flattening it unnecessarily creates a large input flow graph and causes `GRASPR` to repeat work. For example, utility subroutines for basic data structures often contain general-purpose clichés entirely within their boundaries and they are usually called by several higher-level functions. In this case, the subroutines should be recognized independently. The results of recognition should then be duplicated and used wherever the subroutine was called. For example, if a subroutine is recognized as a cliché, calls to it in the program should be represented as an already-reduced non-terminal, which can be used in the recognition of

higher level clichés. This involves simply adding complete items to the chart, representing already-reduced non-terminals.

Besides eliminating variation due to subroutine decomposition, GRASPR also deals with variation in data structure organization. It does this by representing accessors and constructors as attributed edges, rather than as explicit nodes in the flow graph, as are other operations in the program. If the accessors and constructors were represented explicitly as nodes, then the representation would fail to eliminate variation between programs that aggregate the same data, but use different orderings of parts or different nesting of aggregations. (The problems with explicit representation of accessors and constructors as Spread and Make nodes were discussed in more detail in Section 3.4.2.)

The flow graph formalism was specifically designed to allow aggregation-equivalent flow graphs to be recognized. Programs are represented as minimally-aggregated flow graphs, with any internal residual Spreads and Makes replaced with attributed fan-out and fan-in edges. Clichés involving aggregate data structures are expressed in grammar rules in which the aggregation is specified in the embedding relation. The clichés are then recognized in programs by using the embedding relation to introduce the clichéd aggregation organization into the parsing process.

In Section 2.3.2, two organizational variations of `PiSim` are pointed out (in Figures 2-10 and 2-12). In one, the initialization and storage-requirements computations are found within `Inject`, while the other separates these computations out into the functions `Initialize-Simulator` and `Compute-Storage-Requirements`. The first aggregates four pieces of data into a `Message` data structure and then nests this inside an `Event` data structure, along with a Time part. The other aggregates three pieces of data into a `Handler-Data` data structure and then nests it inside a `Msg` data structure, along with a Destination and Arrival-Time part. Both aggregate the same pieces of data, but using different nesting organizations, ordering of parts, and names for structures and parts.

However, these two programs have the same basic flow graph representation, which is shown in Figure 5-1. The only difference between the two is in their edge attributes, as shown in Figure 5-2. (One program, `Inject`, iteratively calls a function `Execute-Next-Event`, while the other, `Start-Pisim`, calls `Process-Next-Message`. The flow graph representations of these two calls is the same for both. This flow graph is hidden in the dotted node labeled "Execute-Next-Event." Likewise, the dotted node labeled "Enqueue-Event" represents calls to the functions `Enqueue-Event` (by `Inject`) and `Enqueue-Message` (by `Start-Pisim`), which each have the same flow graph representation. Also, the recursive node shown in Figure 5-1 is labeled "Execute-Events," but in the flow graph for `Start-Pisim`, the recursive node is labeled "Process-Messages." This difference is not significant, since the recursive nodes are never expected to match any right-hand side node during parsing.)

5.1.3 Delocalized Clichés

Using the flow graph representation also addresses the problem that parts of a cliché may be scattered throughout the text of a program. Many clichés become much more localized in the flow graph than in the program text because only essential dataflow relationships are captured. For example, in Figure 2-13, a portion of the CST code is shown. Even though parts of a simulation cliché are separated by unrelated expressions in the source text, they are translated into neighboring nodes in the flow graph representation of the program. This representation is shown in Figure 5-3. The nodes that are unrelated to the simulation cliché are shaded.

5.1.4 Unrecognizable Code

GRASPR is able to recognize clichés despite the presence of unrecognizable code in the program. This is partly due to GRASPR's cliché localization abilities which helps to separate the familiar from the unfamiliar parts of the program. The clichéd sections of a program tend to become localized in sub-flow graphs of the program's flow graph representation.

The other aspect of GRASPR's approach that makes partial recognition possible is the bottom-up parsing strategy it uses. It recognizes and reports low-level clichés, even if it cannot reconstruct the higher level design that puts them together. All non-terminals are treated as start-types of the grammar, so that each instance of any non-terminal is reported.

GRASPR has been specifically designed to solve the partial program recognition problem, which is defined in Section 3.3.1: Given a program and a library of clichés, find all instances of the clichés in the program (i.e., determine which clichés are in the program and their locations). It formulates this problem in terms of the subgraph parsing problem, which is: Given a flow graph F and a flow graph grammar G, find all possible parses of all *sub-flow graphs* of F that are in the language of G.

In other words, when a program is partially recognized, one or more sub-flow graphs of the program's flow graph encoding are recognized as members of the graph grammar which encodes the cliché library. It follows from the definition of a sub-flow graph, that it is possible to ignore portions of a flow graph before and after a recognizable sub-flow graph, as well as portions that fan out from or into an internal port in the sub-flow graph.

What this means in terms of partially recognizing programs is that GRASPR can recognize a cliché in the presence of unrecognizable code or code that belongs to other clichés, as long as the cliché is localized into a sub-flow graph of the program's flow graph representation. It must be possible to separate the cliché from the rest of the flow graph by disconnecting a set of edges.

GRASPR is able to ignore unfamiliar code that "surrounds" a cliché (in that it sends dataflow to it and/or receives dataflow from it). See Figure 5-4b. It is also able to ignore unfamiliar code that is done conditionally (assuming that the control flow constraints do not require co-occurrence relations to hold between the component operations). See Figure

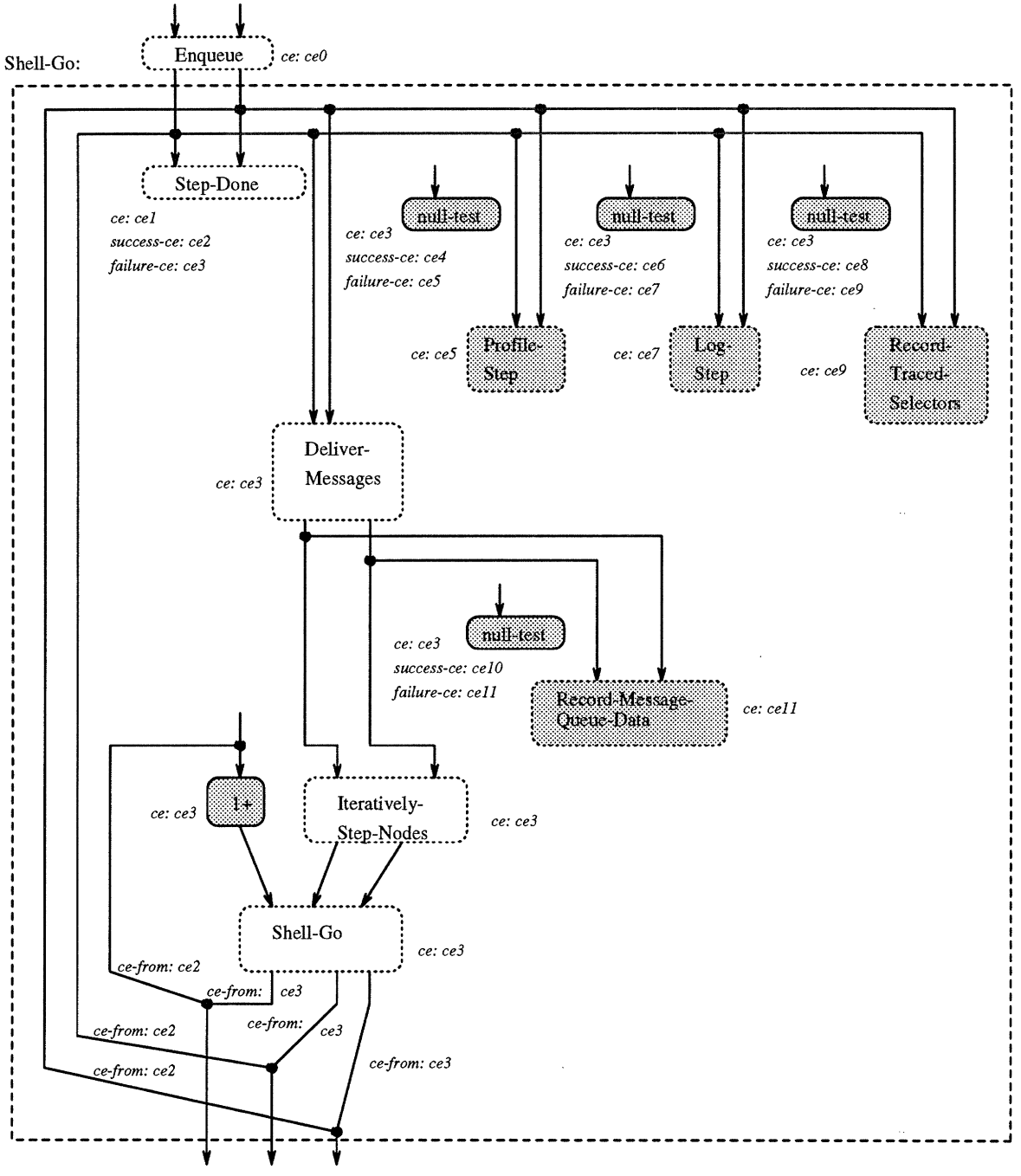


Figure 5-3: Flow graph representing the CST code of Figure 2-13.

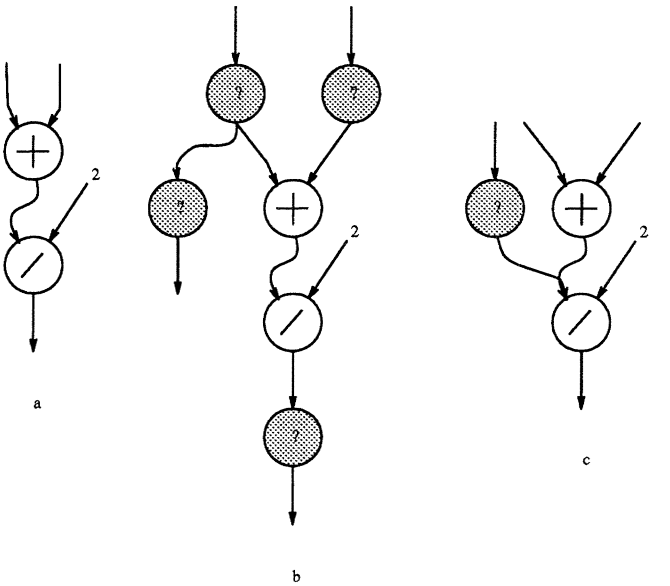


Figure 5-4: a) Average cliché. b-c) Some cases in which a program can be partially recognized.

5-4c.

GRASPR can partially recognize a program that not only has unfamiliar algorithmic fragments, but also has data structures that aggregate unfamiliar parts. It is able to ignore computation on unfamiliar parts of an aggregate data structure. This is a direct result of the parser's techniques for recognizing aggregation-equivalent flow graphs, as described in Sections 3.4.2 and 3.5.2. These techniques allow recognition of a clichéd data structure in a user-defined data structure even when the cliché aggregates only a *subset* of the parts aggregated by the user-defined structure.

For example, suppose the cliché library contained a cliché called Extract-Message, which is the common computation of looking up a SYNCH-NODE in an ADDRESS-MAP, given an integer index, dequeuing its Buffer part and updating the ADDRESS-MAP so that the integer index points to the new SYNCH-NODE. The rules encoding Extract-Message and the Local-Buffer-Dequeue cliché it contains as a part are shown in Figure 5-5.

This cliché is found in the program shown in Figure 5-6 which operates on a user-defined node data structure. The node consists of five parts, one of which (Queue) corresponds to the Buffer part of a SYNCH-NODE. The value of *nodes* corresponds to the ADDRESS-MAP. In addition to performing the Extract-Message operation, this program increments the Busy-Count part of the new node created. It also calls process-message on the msg dequeued, the ADDRESS-MAP, and *step-queue* (which is the global MESSAGE buffer).

GRASPR partially recognizes the node data structure as well as the program step. The flow graph representation of step is shown in Figure 5-7. (The dotted node labeled "Dequeue" is an abbreviation for a flow graph that is derived by the FIFO-Dequeue non-terminal.) The destructuring and construction of the user-defined node data structure is represented

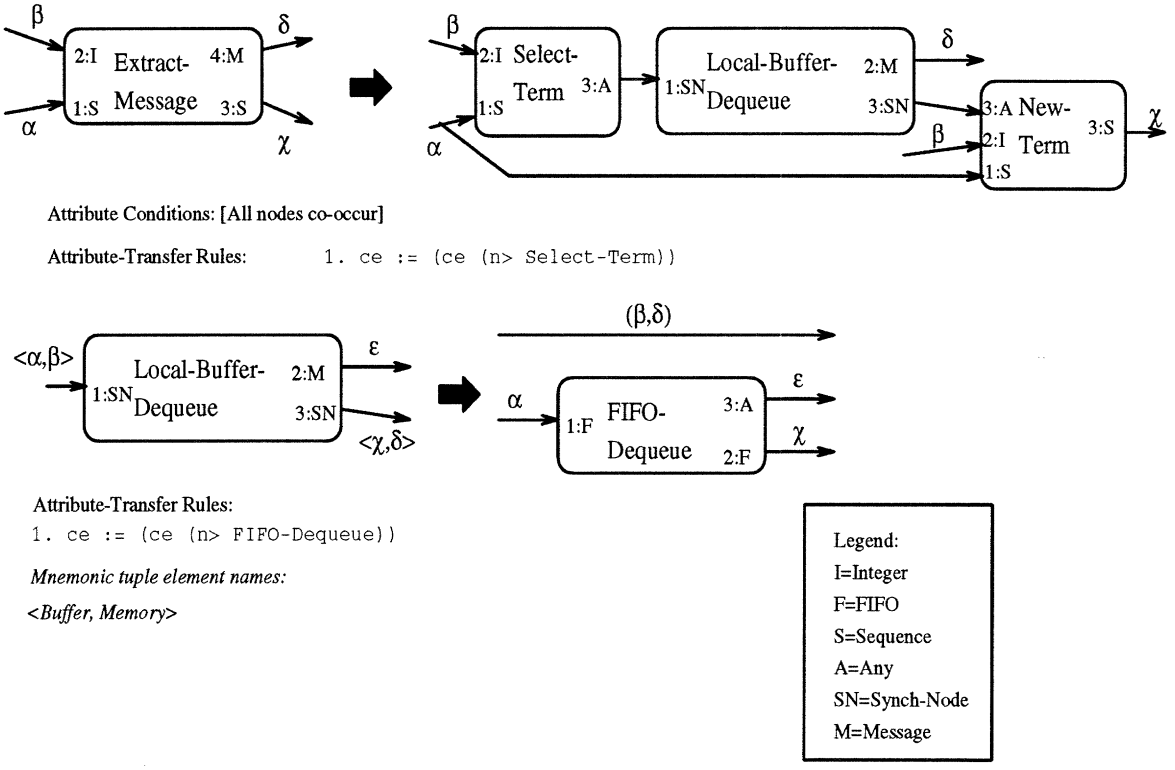


Figure 5-5: Rules for Extract-Message and Local-Buffer-Dequeue cliché.

```
(defun step (node-nr)
  (let* ((node (get-node node-nr))
         (q (node-queue node)))
    (multiple-value-bind (msg new-queue)
        (dequeue q)
      (setq node
            (make-node :queue new-queue
                       :objects (node-objects node)
                       :contexts (node-contexts node)
                       :busy-count (1+ (node-busy-count node))
                       :method-cache (node-method-cache node))))
      (setq *nodes* (copy-replace-elt node node-nr *nodes*))
      (multiple-value-bind (new-nodes new-step-queue)
          (process-message msg *nodes* *step-queue*)
        (setq *nodes* new-nodes *step-queue* new-step-queue)))))
```

Figure 5-6: Code containing a partially recognized data structure.

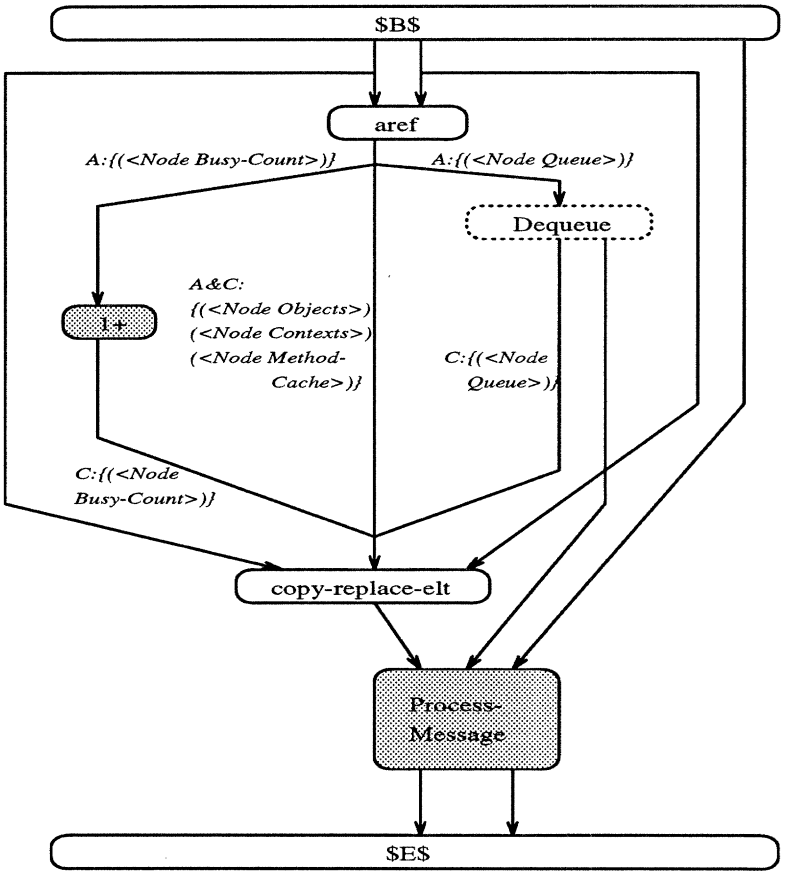


Figure 5-7: Flow graph representation for `step`.

in attributed fan-out and fan-in edges. This facilitates the separation of the unfamiliar computation (the increment of the `node`'s Busy-Count) from the familiar. It allows `GRASPR` to recognize Extract-Message by parsing the sub-flow graph that results from disconnecting the shaded portion of `step`'s flow graph from the rest of the flow graph.

5.1.5 Function-Sharing

The derivations generated for programs by the flow graph parser do not have to be strictly hierarchical. This means that `GRASPR` is able to recover the design of a program, even when parts of the implementation of two distinct abstract operations overlap as a result of an optimization. In effect, `GRASPR` "undoes" the optimization.

For example, in Section 2.3.2, Figures 2-19 and 2-21 show two programs that differ only in that one optimizes the other by enumerating the array `nodes` once instead of twice. The enumeration is shared between the two clichéd operations of advancing each `node` in `nodes` and computing the average length of their Queue parts.

`GRASPR` is able to recognize these two clichés in both programs, even though they overlap in one. `GRASPR` does not destructively reduce the input flow graph representing the program. It allows the recognition of a part of the flow graph to be seen as part of more than one higher-level cliché. The resulting design trees share a sub-tree, as is shown in Figure 2-22.

5.1.6 Redundancy

`GRASPR` is able to deal with variation due to redundancy which occurs when some part of a cliché appears more than once in the same instance of a cliché. There are two types of redundancy that we encountered in dealing with real programs.

One type is the repetition of some computation on the same set of inputs and/or producing outputs that are conditionally merged into the same consumer operation. An example of this is discussed in Section 2.3.2 and shown in Figure 2-23. In this example, the computation of accessing the first element of `Bucket-List` using `car` is performed twice. The parser's ability to recognize share-equivalent programs allows `GRASPR` to tolerate the variation due to this type of redundancy. In particular, the parser zips up the flow graph representation of the program, allowing it to recognize the cliché Ordered-Associative-List. That is, it generates an alternative view of the program in which the redundancy is removed.

The second type of redundancy occurs when a loop is unrolled or, more generally, a recursion is unfolded. This arises in our example programs when we convert the original programs, which contain destructive operations (causing side effects to mutable data structures), to their non-destructive versions. As described in Section 4.2.2, this is handled by an additional chart monitor that creates an alternative view in which the recursion is folded back up.

5.1.7 Implementation Variation

GRASPR is able to recognize two programs that perform the same clichéd abstract operation, even though they may use two different implementations of that operation. This is because the cliché library is encoded in a grammar that explicitly captures implementation relationships between the clichés. So GRASPR is able to view and describe structures on various levels of abstraction.

This enables it to produce the same high-level description of the two versions of the CST program shown in Figures 2-16 and 2-17 of Section 2.3.2, even though they differ on a lower level of abstraction in their implementation of the global message queue. GRASPR produces the design-trees shown in Figures 2-14 and 2-18 for the two versions. They differ only in the subtrees that are highlighted by dotted boxes in Figure 2-18.

It is impractical to enumerate all possible implementational variations of an abstract cliché in the cliché library as flat structures. However, the hierarchical organization of the cliché library allows implementation variation to be represented compactly.

5.2 Limitations

Our recognition approach is based primarily on dataflow graph matching and control flow constraint checking. The success of this approach depends on being able to:

1. faithfully capture the program's dataflow in our flow graph representation and the program's control flow in the attributes, and

2. express a programming cliché in an attributed graph grammar rule in terms of its data and control flow constraints (i.e., operation types and arity, dataflow connections, control environment relationships).

In general, the limitations of our approach arise when one or both of these are not possible to do. The first criterion is not possible when the dataflow or control flow of the program cannot be completely captured by static analysis or the dataflow is not made explicit (in that it is derived from intermediate computations). The second criterion is not satisfied for clichés that have loosely constrained data and control flow or that are defined by characteristics other than data and control flow.

This section gives specific situations in which we encountered these limitations in experimenting with the recognition of our example programs. It also suggests ways of dealing with these problems, e.g., by collaborating with other mechanisms or eliciting and accepting advice from a person. (There are additional limitations to the current recognition system that represent open research problems, rather than inherent difficulties with the approach. These are discussed in Section 7.2.)

5.2.1 Missing or Derived Dataflow

Our clichés are basically expressed as dataflow graphs. A cliché can be recognized only if a sub-flow graph of the flow graph representing the program is isomorphic to the cliché's flow graph representation. Unfortunately, sometimes a cliché exists in a program, but GRASPR fails to find it because dataflow links are derived or missing.

The principal cause of *missing* dataflow (and control flow) information in our example simulator programs is that they accept functions for simulating individual machine operations as input. This prevents data and control flow from being completely determined statically.

We found three common causes of *derived* dataflow links in our example programs. One is that a primary part of a clichéd data structure may correspond to a part of a data structure in the program that is a *handle*. The handle is used to look up the piece of data that actually corresponds to the cliché's primary part. For example, our Execution-Context data cliché contains a sequence of INSTRUCTIONs as a primary part. In the CST program, on the other hand, the corresponding data structure, called Context, has a "Code" part that is a symbol. This symbol is used to look up a Block, which is a sequence of INSTRUCTIONs, in a pooling structure containing all existing Blocks.

The problem with non-clichéd uses of handles is that they introduce intermediate computation which interrupts data flowing from one primitive operation to another. This computation looks up a piece of data using a handle into a pooling structure.

Unsimplified code is a second cause of obscured dataflow links. For example, in (F (Abs-val (G x))), where (G x) is always positive, there is always direct dataflow from G to F.

A third cause is that a program may *implicitly aggregate* heterogeneous pieces of data, rather than explicitly aggregating the data into a structure with named parts, using a structuring primitive (such as DEFSTRUCT in Common Lisp). In implicit aggregation, a primitive data structure, such as a list (in Common Lisp) or an array, is used to aggregate heterogeneous pieces of data, where the *position* in the data structure matters. For example, PiSim creates and uses an array whose first two elements cache information about a MESSAGE (Type and Storage-Requirements), while the rest of the array holds the MESSAGE's Arguments. This array should be treated as an aggregate data structure with three parts: Type (a symbol), Storage-Requirements (an integer), and Arguments (an array).

Implicitly aggregated data structures are accessed and constructed with primitive operations (such as aref) on the data structures at fixed indices. These operations are not converted to attributed edges, as are selectors and constructors for explicit aggregations.

There are two problems with this. One is that with explicit aggregation, the data from one operation to another is represented as a direct edge annotated with accessor and constructor attributes, but with implicit aggregation, this dataflow is interrupted by primitive operations that access or update at a fixed index. In other words, the explicit

dataflow link is replaced by a "derived" dataflow link.

The other problem is that it loses the benefit of our representation for explicit aggregation which facilitates the separation of familiar and unfamiliar computations on parts of a data structure. This separation allows partial recognition of the data structure and the computation on it. (This capability is discussed in Section 5.1.4.)

The underlying difficulty is that implicit aggregation hides the information that a certain primitive access or update at a fixed location is actually a selector or constructor involving a certain data structure and its parts. When data is explicitly aggregated (e.g., using `DEFSTRUCT`), the structuring primitive serves as a machine-readable comment that specifies that some pieces of data are aggregated and are only accessed and constructed using certain functions. It also provides information about which user-defined data structure and parts are involved in the selection or construction. Additionally, it represents the intent of the programmer to only use these accessors and constructors to manipulate the aggregation and never deal with it directly using primitive operations.

(Note that people find it hard to deal with implicit aggregation as well. It requires knowing how fixed locations in the data structure translate to the particular pieces of data being aggregated. It requires effort to perform this mapping during recognition.)

Solution Suggestions

To deal with the variation due to missing or derived dataflow, `GRASPR` would profit from advice from a user or collaboration with other automated techniques. For example, classical rewriting or partial evaluation techniques can be applied to simplify parts of the program. (See Letovsky [84] and Murray [95], for example.) By interleaving recognition with these other techniques, alternative views of the program can be generated to facilitate recognition. Recognition in turn can provide a more abstract view of the program and generate assertions about parts of it, based on the known properties associated with the clichés that have been recognized so far.

One way for `GRASPR` to elicit advice is by looking for "question-triggering" patterns (in addition to clichés) which point to the possibility that some dataflow is derived. For example, by looking for standard look up and update operations (such as associative-set clichés), `GRASPR` might uncover a use of a handle. Recognizing that each node created during initialization is put into `*NODES*` triggers asking the user if `*NODES*` always contains all the `NODE`s ever created. A fixed-position array or list access suggests an implicit aggregation is being used. These hypotheses can then be presented to the user or some expectation-driven component for confirmation. Once the use of a handle or an implicit aggregation is uncovered, `GRASPR` can generate an alternative view of the flow graph in which the derived links are made explicit attributed edges.

It can be more difficult for `GRASPR` to confirm its hypotheses on its own than for a human user to confirm them, since the user can take advantage of expectations generated

from the mnemonic names and documentation. For example, it can be easy for a person to tell whether a particular data structure is a pooling structure, just by its name: `*Nodes*` contains all `Node` data structures in `PiSim`, `*Blocks*` contains all `Block` structures in `CST`. (Alternatively, the user can give `GRASPR` advice about which structures are pooling structures up front, without waiting for `GRASPR` to ask for it).

A special (and common) case of implicit aggregation for which it is easy for a person to give advice is *manual abstraction.* In this case, functions are explicitly defined which perform the accesses and constructions involving fixed indices in an implicitly aggregated data structure. In other words, the programmer manually defines the accessor and constructor functions for an implicitly aggregated data structure. (These functions are defined automatically by explicit aggregation primitives (such as `DEFSTRUCT`).)

This is distinguished from general implicit aggregation in that the aggregation is explicit to people, even though it "looks" the same as implicit aggregation to `GRASPR`. The aggregation is expressed in the naming conventions the manual abstraction functions use. They also express the programmer's intent not to violate the abstraction by manipulating the aggregate directly using primitive operations. Since `GRASPR` does not take naming conventions into account, these functions are flattened just like any other function. However, a person can easily give `GRASPR` the information that certain functions should be seen as accessors and constructors for an aggregate data structure.

5.2.2 "Missing" Cliché Parts

Another common reason for an algorithmic cliché not to be recognized is because part of the cliché is replaced in the program by a special-case optimization. This optimization is not a clichéd one; it happens to be possible in the context in which the cliché is used.

A common instance of this occurs when some computation is avoided by using a value that equals the result of that computation. This can be an opportune equality or an intentionally cached value. For example, the cliché for polling the simulated nodes and stepping those that have work to do contains an enumeration of the collection of simulated nodes. The cliché for enumeration when the collection is implemented as a sequence has a part that computes the size of the sequence and then uses it to determine how many elements to enumerate. The instance of this cliché in the `CST` code does not compute the size of `*NODES*`, but instead uses `*NUMBER-NODES*` which is a global variable specifying the size of `*NODES*`. This variable is used during initialization to create `*NODES*`.

Sometimes part of a cliché is missing in the program because the general case represented by the cliché has been simplified in the context of the program. For example, a part of the Event-Driven Simulation cliché is a Priority-Queue Insert which adds an initial `EVENT` to the `Event-Queue`. Because the `Event-Queue` is empty at this point, the general case of this clichéd operation can be reduced to the computation done when the priority queue is empty. (For example, if the priority queue is implemented as an ordered associative list, the insertion

would simply **cons** the event onto the empty priority queue, without testing whether it is empty or providing actions for splicing it in if its not empty.) If the special-case version of the cliché is a common optimization, then it is included in the library along with the general case. However, when it is not, recognition of the cliché fails. (We cannot expect all possible optimizations in the context of use to be clichéd and we do not want to enumerate them all in the library.)

Solution Suggestions

What is needed for recognition to succeed in these cases is for the special-case computation and the general-case cliché to be seen as equivalent. In general, this cannot be done. However, it may be possible to apply limited reasoning techniques to uncover dataflow equalities or conditional simplifications in simple cases such as those discussed above.

Non-clichéd special-purpose optimizations often cause some, but not all of a cliché to be recognized. One way to elicit advice on whether some computation is a special-case optimization is to find maximally-sized near-misses (partial recognitions) of the cliché and then generate a hypothesis that the cached value used is equal to the result of the computation in the part of the cliché not yet matched.

Recognizing maximally-sized near-misses is costly (as is discussed in Section 6.2.7). However, we can generate them only for particular clichés and at particular locations in the program in order to reduce the cost. For example, we can choose only promising clichés, such as those for which some salient part has been recognized, and we can look for them only in the areas of the program that have not already been recognized as part of other unrelated clichés.

5.2.3 Expressing Clichés with Loose Constraints

In encoding clichés as constrained dataflow graphs in graph grammar rules we are required to specify exactly which operations (or classes of operations) make up a cliché, how dataflow connects them to each other, and their arity. For some clichés that we identified in our simulator domain, this is difficult to do.

There are three different cases in which we encounter difficulties. One is in expressing clichés that have as an integral part the application of an arbitrary, non-clichéd and non-primitive function. A second case is in compactly representing possible variations in the implementation of an algorithmic cliché whose parts may be combined in several possible valid configurations. The third case is in capturing a clichéd data and control flow pattern in which the operations and tests are not tightly constrained to be of particular types. The dataflow between them is only loosely constrained as well.

Arbitrary Function Application

We encountered two examples of types of clichés that are difficult to encode because a part of them is the application of an arbitrary function. They are second-order patterns, in that they are parameterized over arbitrary functions, which are non-clichéd and non-primitive.

One example arises in encoding iteration clichés, as discussed in Section 4.1.3. These clichés all contain applications of arbitrary functions or predicates in an iteration. However, we cannot encode these clichés without requiring the functions or predicates to be primitive operations (terminals) or clichéd functions (non-terminals). For example, it is not possible to recognize the generation cliché in the following code.

```
(defun f (l)
  ...
  (f (cdr (cdr l))))
```

This is because the generating function is an arbitrary composition of primitives (i.e., the generating function is `(lambda (x) (cdr (cdr x)))`).

Another example of this problem arises in trying to capture the simulation clichés without requiring that the code for simulating message handling be clichéd. In particular, we wanted to express the cliché for processing an event (in event-driven simulation) or advancing a node (in synchronous simulation) as having a part that applies some non-clichéd message handling simulation function.

Solution Suggestions

What is needed is a special-purpose mechanism (separate from the graph parser) to bundle up the sub-flow graph that satisfies certain constraints. This mechanism can make use of information about how much of the cliché has already been matched to focus on certain locations. It can also make use of information available in the cliché's constraints.

For example, in the iteration clichés, the input and output correspondence constraints place restrictions on which sub-flow graph can be bundled up. Waters [138] has developed general-purpose dataflow-based techniques for decomposing a program into temporally abstract fragments. It would be useful to incorporate these decomposition techniques into the recognition process to help bundle up possible functions. For instance, bundling up the composition of `cdr`s in our example above can be done by grouping together the sub-flow graph that is bounded by input and output ports that input-correspond.

In the case of bundling up message handling simulation code when no clichéd function for it is recognized (as in `CST`), it might be possible to ask for advice on which part of the program achieves this purpose. Also, based on the location of the rest of the cliché and which nearby parts of the program are unrecognizable, `GRASPR` might be able to hypothesize approximately which part of the program should be bundled up.

Implementational Variations

As we mentioned in Section 2.1.3, there are many variations of our synchronous simulation algorithm. On each iteration, the algorithm we described performs three actions in the following order: test for termination, deliver messages, and poll and advance nodes by one step. The other variations of this algorithm in which a different ordering is used also perform synchronous simulation.

However, each of these variations is represented by a different dataflow graph. For example, the algorithm described in Section 2.1.3 has the form shown in Figure 5-8a. (This is a sentential form of our current grammar which encodes the algorithm.) Two other valid configurations are shown in Figure 5-8b and 5-8c. In fact, all six permutations of the three actions are valid configurations.

The problem is that we must deal with these variations by enumerating them in the cliché library. This is because the flow graph encoding forces us to specify the exact dataflow connections between the three operations and therefore a particular ordering.

It is an open question whether there is a more compact representation for algorithmic clichés that vary in this way. (For example, reasoning about a program's functional semantics, as is done by Allemang's DUDU [4, 5], may help tolerate this variation.) In addition, more experience with encoding clichés is needed to tell how severe this problem is and how frequently it occurs in practice.

General Data and Control Flow Pattern

Because our formalism forces us to specify many details of dataflow, operation types, etc., it is sometimes hard to express some common data and control flow patterns that are not tightly constrained. One cliché we had difficulty expressing is a common type of conditional dispatch which occurs in program interpreters (particularly for the Lisp-like languages).

This cliché is the "Evaluate" part of an EVALUATE/APPLY recursion for interpreting statements in a language. The standard algorithm for this dispatches on the type of an expression to code for handling that expression. For some expression types, there are standard computations to perform. For example, for expressions that are constants, the expression is simply returned. For expressions that are applications of some operator to a set of arguments (which are themselves expressions), each argument is recursively evaluated and the operation is applied to the set of evaluated arguments.

However, instances of this cliché vary with the types of expressions that can be evaluated, which depends on the language of the program being interpreted. The number and type of test cases in the conditional dispatch vary. The actions that are dispatched to also vary. The dataflow connection constraints are flexible. The problem is that in our formalism, we must specify the number and types of tests and actions, and the exact dataflow between them. A more abstract language for expressing abstract data and control flow patterns is needed.

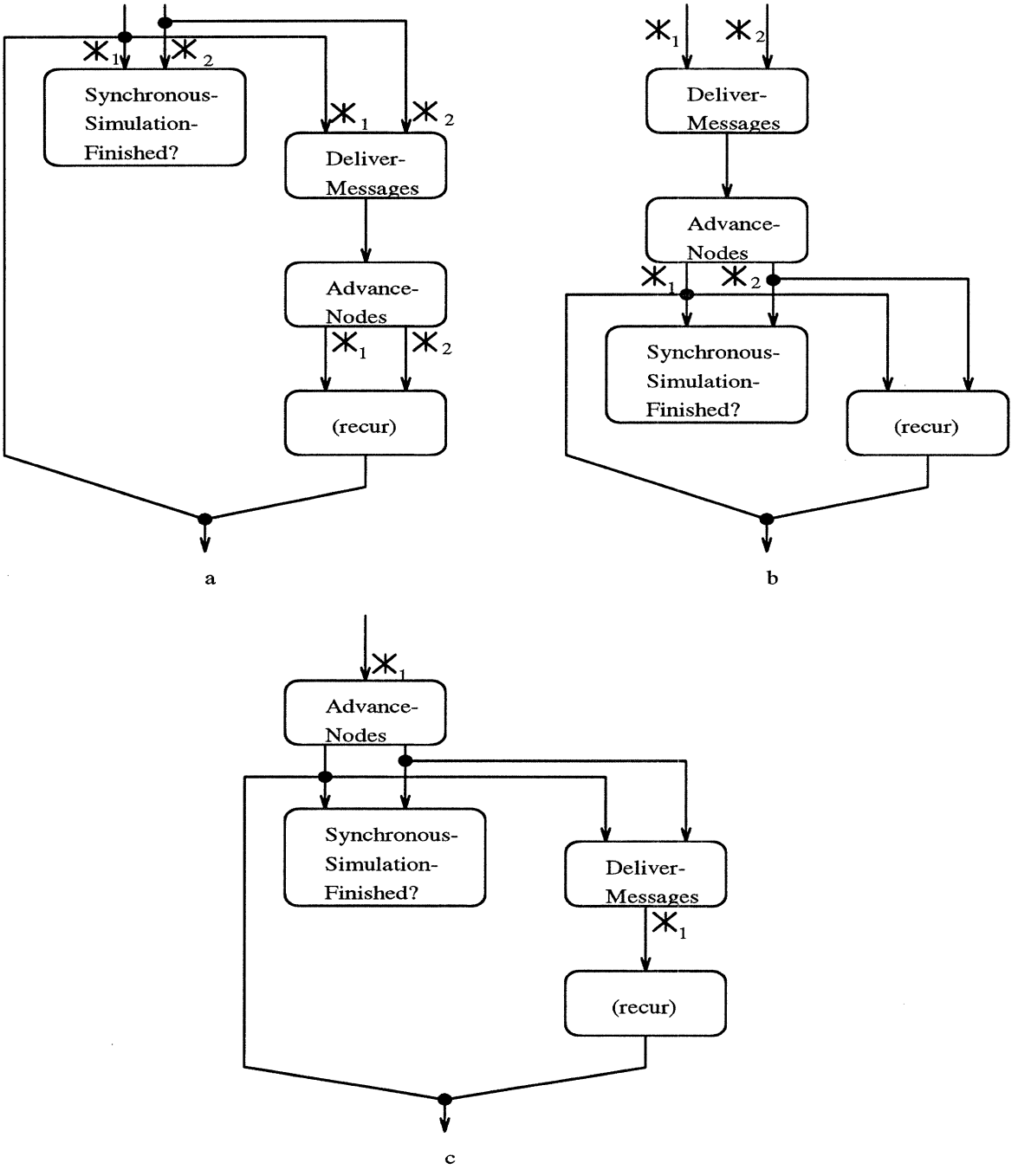


Figure 5-8: Some valid variations of Synchronous Simulation algorithm.

The point of this section and the previous is that although the flow graph formalism allows us to encode clichés on a high level of abstraction, the level of abstraction is still limited by the amount of detail that must be specified. Perhaps there are ways of combining this formalism with even more abstract formalisms that will allow looser dataflow constraints. For example, perhaps we can encode and recognize parts of clichés within the dataflow graph formalism, and then use a different encoding to express constraints on how these parts fit together.

5.2.4 Enqueuing New Messages and Events

This section deals with a problem that arises both as a result of not being able to fully determine the data and control flow of the example programs and of not being able to express and efficiently check certain constraints.

As mentioned in Section 4.1.4, one of the actions of a processing node that is simulated as part of the simulation of message handling is the creation and sending of new messages. One of the constraints on both simulation algorithms is that whenever a message send is simulated, a new `EVENT` or `MESSAGE` must be created and added to the event-queue or global message buffer, respectively.

We did not include this constraint in the grammar rule encoding of the rules for the synchronous and event-driven simulation clichés. There are three obstacles to expressing and checking this constraint within our graph parsing framework.

One is that the computation involved (enqueuing new `EVENT`s or `MESSAGE`s) is buried within the code for simulating a processing node's action. This code is not guaranteed to be clichéd, so we do not have grammar rules that derive all possible flow graphs representing this code. This means that we have no context in which to express the constraint.

Suppose it is clichéd, we still have a second problem which is that the part of the simulation code that performs the activity of enqueuing new `EVENT`s (or `MESSAGE`s) is typically given as *input* to the simulator. So, it is not available for analysis. The cliché models the application of functions for simulating a processing node's actions during an instruction execution. Since these functions are not part of what is analyzed, the exact data and control flow connecting the enqueuing operation to the rest of the cliché are not explicitly represented.

Finally, suppose we had the code available. That is, rather than accepting functions to simulate the actions of a processing node in executing some machine operation, suppose the simulator program contains a large conditional which dispatches on machine operation types to the code simulating operation execution. We encounter yet a third problem which is that in the current parsing framework, it is difficult to express and check the constraint that each time a message send is simulated, – i.e., a new `EVENT` (or `MESSAGE`) is created, – the new `EVENT` (or `MESSAGE`) is added to the event-queue (or global message buffer). It requires expressing and checking constraints that are quantified over instances of some computation.

A special-purpose global mechanism is needed to check this constraint, since the parser is currently only able to check constraints on individual instances. In addition, it requires some means of finding all instances of creating whatever user-defined data structure that corresponds to our clichéd aggregate EVENT (or MESSAGE). This requires unambiguous information about the mapping from clichéd data structures to user-defined ones. Also, since aggregate data structure creation is encoded in edge attributes, finding the instances of user-defined data structure creation cannot be done by recognizing a flow graph. Instead it must focus on patterns in edge attributes.

In summary, problems arise when:

- an integral part of cliché is non-clichéd and the constraint we want to express refers to this non-clichéd part,
- the data and control flow relating the constrained part of the cliché to the rest of the cliché are not completely and statically determined (e.g., because part of the program is read in as input), or
- the constraint quantifies over instances of some computation, particularly if the computation is a data structure creation or access, not the application of some primitive operations.

Solution Suggestions

Although the enqueuing constraint is difficult to express and check within the current graph parsing framework, it is not a hard constraint for a person to check. The person has the advantages of understanding mnemonic names which give clues about the purposes of machine operations. A person might also have expectations about which machine operations cause message sends, based on knowledge of the machine being simulated.

Rather than requiring that more code be given to GRASPR for analysis or extending the parser to quantify constraints over instances, it might be easier to just ask the user whether the constraint holds. The constraint should be expressed more generally as a condition on the code that simulates a node's action. If we are already eliciting advice on which part of the program handles a message (as suggested in Section 5.2.3), then we could also ask whether this general constraint holds. GRASPR might also ask for the simulator function that is called to perform the enqueuing and then can analyze that code to understand better how the event-queue (or global message buffer) is implemented.

5.2.5 Modifications to Example Programs

To enable GRASPR to recognize the example simulator programs, we made the following changes to the programs. Some avoid the inherent limitations of the graph parsing approach discussed in this section. Others help GRASPR deal with difficulties in the current system, which we expect to be addressed by extensions to GRASPR in the future. (For example,

these include recognizing programs that are multiply-recursive or that perform side effects to mutable objects. See Section 7.2). Appendix B contains the original versions of the two simulator programs, as well as their translations.

- We translated instances of implicit aggregation (including manual abstractions) to explicit aggregations. For example, we defined a `Task-Segment` data structure in `PiSim` to explicitly aggregate the Type, Storage-Requirements, and Arguments of a `MESSAGE`. In `CST`, we replaced the manual abstraction for `msg` with a `msg` structure definition.

- We simplified conditionals and canonicalized conditions involving `NOT`, `OR`, and `AND`. (See `step-done` and `enqueue` in `CST`, for example.)

- We manually undid special-case (nonclichéd) optimizations that take advantage of an opportune dataflow equality or a cached value. That is, we restored the computational part of a cliché that is avoided by an optimization. For example, in `CST`'s `step-nodes` function, which enumerates and steps the simulated nodes, the use of `*number-nodes*` is replaced by a call to `array-total-size`.

- To deal with the problem of encoding and recognizing loosely constrained clichés, we provided advice to `GRASPR` about where these clichés were located. (In a future hybrid system, we expect this advice to come from other recognition techniques that can deal with these types of clichés. See Section 7.2.2.) During the translation of the PiSim program to a plan, we advised the symbolic evaluator that the box representing the call to the function `Evaluate` not be expanded. This avoids a limitation in the current implementation of `GRASPR` which prevents it from translating multiply-recursive programs into meaningful attributed flow graphs. (See Section 7.2.1.) We also specified that the unexpanded call to `Evaluate` is an instance of the "Evaluate" cliché. (See Section 7.2.2.) Similarly, during the translation of the `CST` program, we specified that the `process-msg` function not be expanded and that it represents an instance of the `Handle-Message` non-terminal.

  When the symbolic evaluator creates the plan representation of a program (which is then translated to an attributed flow graph), it starts with some topmost function and recursively expands calls to user-defined functions into their plan representations. Only plans for functions whose calls are reached by the evaluator are included in the plan representation. This means the flow graphs for some functions in the example programs are not included as sub-flow graphs of the input graph parsed. In particular, those that are only called by `Evaluate` in `PiSim` and `process-msg` (or its subfunctions) in `CST` are not included. Also, functions in `PiSim` called by the Machine-Operation functions given as input to `PiSim` cannot be expanded into the program's plan representation. In addition, some logging and tracing functions in both programs are not expanded.

- We translated the programs into their functional versions by replacing destructive operations with their non-destructive counterparts. (See Section 7.2.4 for ideas on partially automating this translation.)

- All iterative computations are treated as tail-recursions by GRASPR. Currently, the translation from iterative to tail-recursive procedures is done manually, but it is well-known that this translation is straightforward to automate.

- Program breaks, errors, and non-local program exits are currently ignored in that they are treated as ordinary calls to primitive operations. The non-local control flow they cause is not modeled in our control flow attributes. Further research is needed to determine how best to model non-local flow. See [117], Section 3.4, for further discussion of this problem.

5.2.6 Conclusion

We have made observations of difficulties encountered in recognizing two programs. These might be relatively rare problems or they might be common. There is currently no natural partitioning of programs based on the difficult features they contain with respect to recognition. This report starts to point out some features that might distinguish programs that are hard to recognize from others (at least within the realm of recognition based on dataflow and control flow). Much more research is needed to map out this space of recognition difficulty.

Chapter 6

Analysis

Our flow graph parsing algorithm is worst-case exponential in both space and time. For each rule of the grammar, the parser is searching for a way to match each node of the rule's right-hand side to an instance of the node's type in the input graph. This search is inherently exponential. In fact, the flow graph recognition problem for flow graphs – given a flow graph F and a grammar G, determine whether or not F is in the language of G – is NP-complete. (Appendix A gives one proof of the NP-completeness of this problem.) The flow graph recognition problem is simpler than the flow graph parsing problem for flow graphs, so it is unlikely that there is a flow graph parsing algorithm that is not exponential in the worst case.

Nevertheless, we apply our flow graph parsing algorithm to the problem of partial recognition of programs and do not encounter the exponential behavior in practice. The reason is that we take advantage of constraints specific to the program domain which are strong enough to reduce the complexity and prevent the worst case from happening. (The application of the parser to other problem domains requires similar use of strong constraints.)

Efficiency is also gained by using a graph grammar that captures much of the commonality among the flow graphs the parser is searching for. This enables the parser to reuse results of exploring parts of the search space.

This chapter gives an expression for the time requirements of the parser, showing that they depend on the number of full and partial analyses the parser generates. It points out how the algorithm can be made to exhibit exponential behavior in the worst case. It then explains how constraints make it feasible for us to apply this inherently exponential process to practical program recognition. Weak constraints can arise in the general flow graph parsing case in the form of ambiguity and disconnected right-hand sides of graph grammar rules. However, additional program domain-specific constraints compensate for these weak structural constraints.

Empirical evidence supports these arguments and shows the effectiveness of the constraints used. The empirical results were obtained by experimenting with the recognition of the two example simulator programs, referred to as `CST` and `PISIM`. (These programs have

been modified from their original form (see Section 5.2.5) to get around the limitations of the current system that are discussed in Sections 5.2 and 7.2. Even with these modifications, the programs provide a realistic base for experimentation in that the modifications did not significantly affect the strength of constraints.) Further experimentation on more programs is needed to broaden our understanding of which constraints are crucial and which programs are inherently difficult to understand.

This chapter concludes with a few suggestions for improving the performance of the parser.

6.1 Cost

This section presents an expression for the time requirements of the parsing and constraint checking process which is at the heart of the recognition system. We first briefly describe the particular instantiation of the general chart parsing algorithm, which is used by the recognition system. The instantiation fixes the rule invocation strategy to be bottom-up. (This is the strategy used by the current recognition system for reasons described in Section 3.5. The top-down version of the algorithm for grammars with a simple embedding relation, which encodes no aggregation relationships, is equivalent to Brotsky's graph parsing algorithm. See [15], for an analysis. For the top-down string parsing case, see Earley's analysis [31, 32].)

We derive a formula for the average-case complexity of the bottom-up algorithm. The cost depends on the number of items that are created by the parser. Section 6.2 characterizes this number and shows how the worst-case exponential growth in the number of items is prevented by domain-specific constraints in practice.

In the complexity expression, the numbers of various types of items created by the parser are weighted by the costs of the parser's actions. Section 6.3 gives details of what the costs of these actions depend upon.

6.1.1 Brief Algorithm Description

For the purposes of our analysis, we need to describe a few additional details about the structure of items and graph grammars, so that we can refer to them.

Each rule in the grammar has an associated *node ordering*. This is a reflexive, anti-symmetric relation, that need not be transitive. We denote it as $\leq_n$. We distinguish node orderings in which all nodes are related in a *chain*, as *strict* node orderings. In these, there is exactly one *minimal node* n_1 (i.e., no other node is $\leq_n n_1$) and exactly one *maximal node* n_k (i.e., n_k is not $\leq_n$ any other node), all of the nodes are ordered from n_1 to n_k in a sequence $(n_1, ..., n_k)$ such that $n_i \leq_n n_{i+1}$ for $i = 1, ..., k-1$, and no other pair of nodes is related besides these. (The transitive closure of a strict node ordering is a total ordering.) We call non-strict node orderings *partial* node orderings. The transitive closure of a partial

node ordering is a partial ordering.

We call the node type that an item is recognizing its *label.* Each partial item has a grammar rule associated with it which is being used to recognize this node type. Also, each partial item contains a set of *needed* nodes which are nodes not yet matched in the item rule's right-hand side. We distinguish a subset of these as *immediately needed.* This subset is determined by the rule's node ordering. Initially, the immediately needed nodes are the minimal nodes. When a node x is matched, it is replaced in the immediately needed set by all other nodes not yet matched that x is less than in the ordering. (If a partial item's rule has a strict node ordering, the item will always have exactly one immediately needed node.)

The immediately needed set determines which nodes are allowed to be matched next. If a complete item for node-type A is added to the chart, only partial items that have immediately needed nodes of type A can be extended by the complete item. Similarly, if a partial item is added to the chart, it is only combined with complete items for those nodes in its immediately needed set.

Each item has a set of input and output mappings which specify the location of the node-type being recognized. For partial items, these might be empty. The location is specified in the form of a set of mappings of ports on a node (whose type is the item's label) to sets of location pointers (which may be nested due to aggregation, as described in Section 3.4.1). Each location pointer specifies some input graph edge.

We are now ready to describe the chart parsing algorithm which uses a bottom-up rule invocation strategy.

1. Initialization:

   - Add complete items to the agenda for each input graph node. The label of each item is the node label of the input graph node it represents.
   - For each rule, add an empty partial item to the agenda. The label of the item is the node-type of the rule's left-hand side. Make the item immediately need the set of nodes that are minimal in the rule's right-hand side node ordering.[1]

2. Until the agenda is empty, continually pull an item X from the agenda and if X is not a member of the chart, do the following:

   - Add X to the chart.
   - If X is a complete item and X's constraints are satisfied, then for each partial item P in the chart that is extendable by X, make a new item extending P with X and put it on the agenda.

[1]One or the other, but not both, of these initialization steps can add the items to the chart as an optimization. Also, the empty partial items can be added to the agenda as they are needed, as described in Section 3.5. To simplify the analysis, neither optimization is done here.

- If X is a partial item, then for each complete item C in the chart that can extend X, make a new item extending X with C and put it on the agenda.
- Apply the tests and operations of the additional monitors to the item. For example, for each complete item X whose constraints are satisfied, the zip-up monitor determines whether there are items that can zip up with X. If so, it performs the zip-ups and adds the results to the agenda.

To clarify, the check that "X is not a member of the chart" is checking that there is no item in the chart that represents the same analysis as X. If X is partial, then this checks that there is no other partial item that matches the same right-hand side nodes of some rule to the same input graph terminal nodes or non-terminal instances. If X is complete, then this checks that there is no other complete item with the same label at the same location as X.

There are two situations in which an item can be created that is a duplicate of an existing item. One occurs when there is *structural ambiguity* (i.e., there is more than one way to derive the same flow graph from the same non-terminal).

The other situation occurs when two complete or partial items are created as a result of a series of extensions, starting from the same partial item and involving the same set of complete items for the same right-hand side nodes, but occurring in two different orders.

Figure 6-1 gives an example. The partial item I_p immediately needs two nodes, n_1 of type A and n_2 of type B. Two complete items are formed, one for A and the other for B, such that both can extend I_p. I_p is extended to two new items I_{p1} and I_{p2}. Since the complete items for A and B are compatible in that they satisfy the binary constraints that I_p's rule imposes on n_1 and n_2, I_{p1} and I_{p2} are extended with the complete item for B and A, respectively. The two resulting items are duplicates of each other, since they have the same right-hand side nodes (n_1 and n_2) matched to the same non-terminal instances (represented by the complete items for A and B).

This can only happen if a partial item is able to have more than one immediately needed right-hand side node. Therefore, it occurs only when a rule has a partial node ordering.

Each complete and partial analysis created by the parser is added to the chart exactly once. This is guaranteed because before adding an item to the chart, the parser explicitly checks for a duplicate item already existing in the chart.

A grammar that is structurally ambiguous provides multiple ways to hierarchically view a subgraph. The multiple derivations are sometimes useful for understanding purposes. So, rather than simply throwing away duplicate complete items that represent different derivations, we can store them in an auxiliary structure to be accessed when presenting the parser's results.

Another clarification of the algorithm concerns the timing of constraint checking. Grammar rules place a number of constraints on the nodes and edges that match their right-hand sides. Some of these constraints are checked in the extendibility criterion (e.g., node type

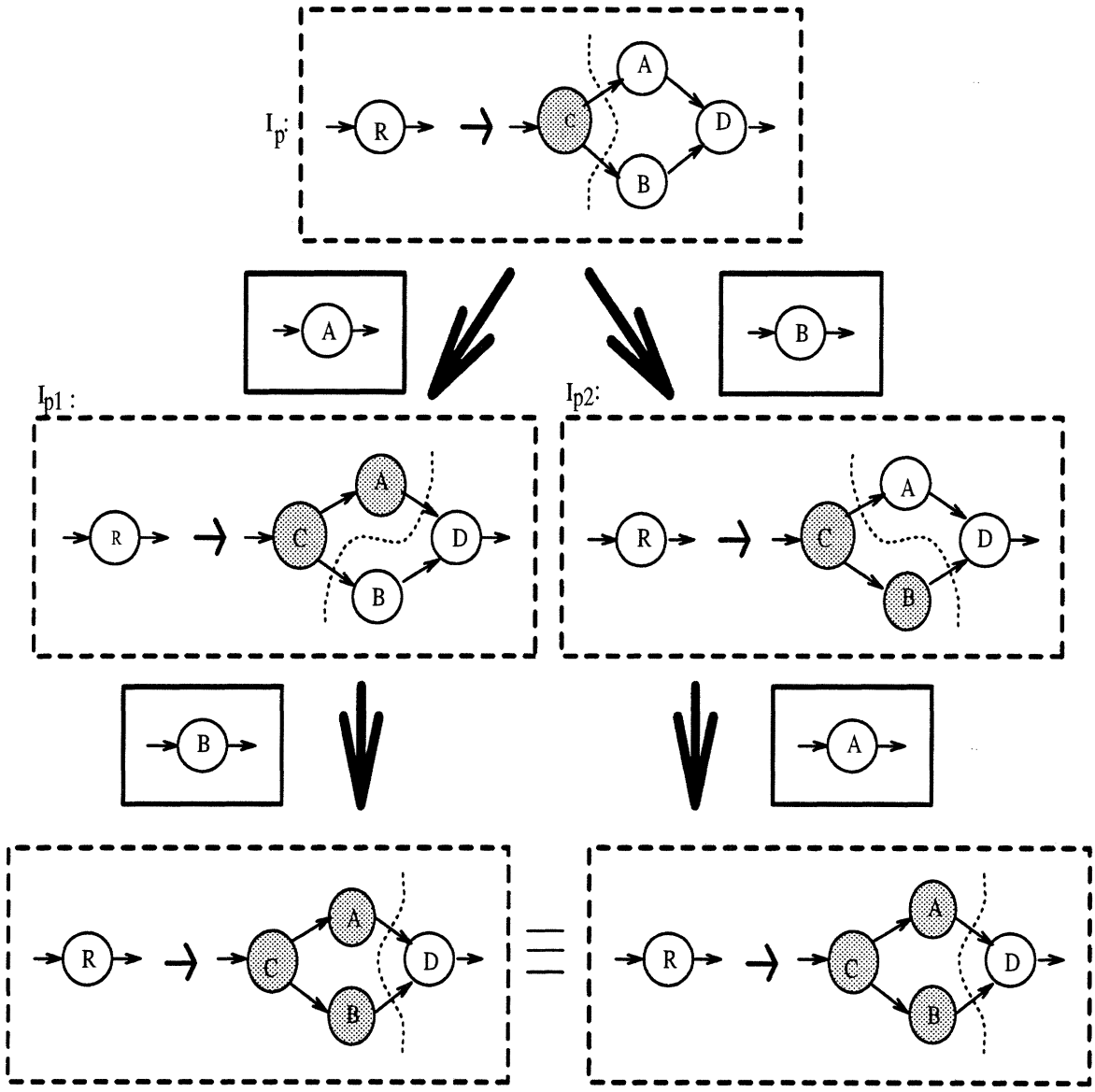


Figure 6-1: Two series of extensions resulting in duplicate items.

and edge connection constraints). Others (e.g., most attribute conditions) are checked when a complete item is added to the chart, before it is paired up with partial items to extend. Section 6.2.2 discusses the design decision concerning which constraints should be checked in the extendibility criterion and which should be postponed to apply to complete items alone.

Additional details of this algorithm will be fleshed out as needed. In particular, many of the details that are relevant to the actions of the parser, such as adding items to or looking up items in the chart, have not been presented. These will be described when the cost of each of these actions is considered.

6.1.2 Complexity

We can determine the cost of the parsing algorithm by considering the cost of each of its sub-operations and how often they are performed (i.e., the total number of items they act upon). To do this, it is useful to categorize the types of items created. We partition the full set of items ever created, denoted by I_T, in two ways. As shown in Figure 6-2a, one partitioning views I_T as consisting of four disjoint sets of items which are differentiated by how the items in the sets were created. (The relative sizes of the sets in the figure is *not* meant to reflect the relative sizes of the actual item sets.)

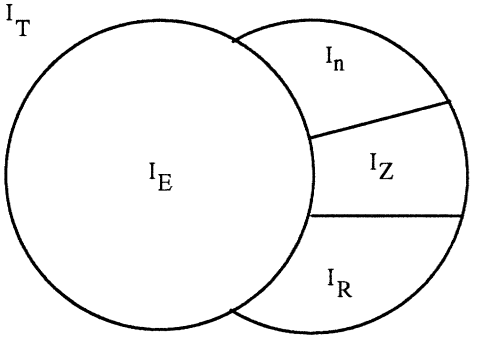
- I_n is the set of complete items created during initialization for each of the terminal nodes of the input graph.
- I_R is the set of empty partial items created during initialization for each rule.
- I_Z is the set of items created by zipping up two or more items.
- I_E contains all items created by extension.

The second partitioning breaks up I_T into two disjoint sets, as shown in Figure 6-2b:

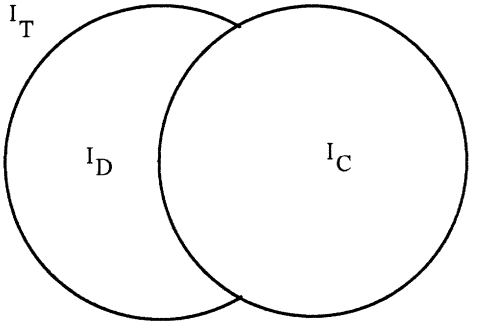
- I_D is the subset of I_E that contains duplicate items that were created but not added to the chart, and
- I_C is the set of items that are in the chart.

Figure 6-2c shows how the sets overlap across partitionings. We denote as I_f the subset of items in the chart which are complete items. I_f is shown in Figure 6-2c as the shaded portion.

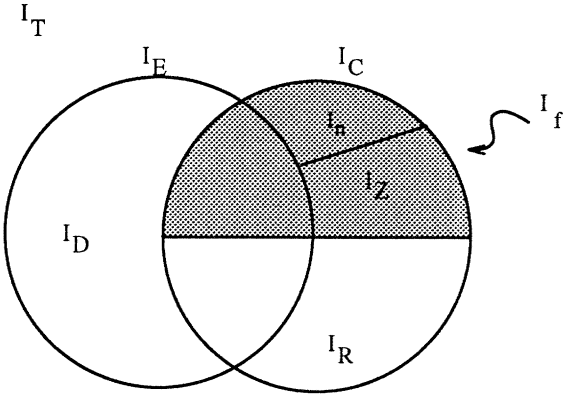
We can now characterize the overall cost of the parsing algorithm by considering the number of times each of the actions of the parser is applied. This can be expressed in terms of the sizes of the various sets of items described above. This is because each action of the parser acts upon a particular type of item and it is applied exactly once for each item of that type. There are no additional costs not accounted for. The overall cost is a sum of the action costs weighted by the number of items to which they apply.



a) Partitioning based on how items are created.



b) Partitioning based on whether items enter chart.



c) The relationship between the partitions.

Figure 6-2: Partitions of the total item set.

We consider which actions are applied to each of the items in each type of item set. Each action is followed by a variable denoting the run-time cost of performing this action on an item. These variables are used below in expressing the algorithm's complexity.

The following actions are taken upon each item ever created, whether or not it is added to the chart (i.e., for all $I \in I_T$):

- create it, which is one of these actions
  - if $I \in I_n$, create complete item for a terminal node ($C_{instantiate-terminal}$)
  - if $I \in I_R$, instantiate empty partial item ($C_{instantiate-empty}$)
  - if $I \in I_E$, create item by extension (C_{extend})
  - if $I \in I_Z$, create item by zipping up other items (C_{zip-up})
- add it to the agenda ($C_{agenda-add}$)
- pull it from the agenda ($C_{agenda-retrieve}$)
- look for a duplicate of it ($C_{duplicate-test}$).

Each item added to the chart (i.e., each item in I_C) additionally has the following actions applied to it. (For now, assume the only additional monitor is the zip-up monitor.)

- add it to the chart ($C_{chart-add}$),
- look up items to combine with it ($C_{combination-lookup}$),
- look up items to zip up with it ($C_{zip-up-lookup}$).

Each complete item in the chart (i.e., those in I_f) has its constraints checked ($C_{constraint-check}$).

The total run-time cost of this algorithm, in terms of the component action costs and the size of the item sets is:

$$
\begin{aligned}
&|I_T| * (C_{agenda-add} + C_{agenda-retrieve} + C_{duplicate-test}) + \\
&|I_E| * C_{extend} + \\
&|I_C| * (C_{chart-add} + C_{combination-lookup}) + \\
&|I_R| * C_{instantiate-empty} + \\
&|I_n| * C_{instantiate-terminal} + \\
&|I_Z| * C_{zip-up} + \\
&|I_f| * (C_{constraints-check} + C_{zip-up-lookup})
\end{aligned}
$$

The sizes of the component action costs are typically quite small. They depend polynomially upon the sizes of various parts of an item, such as the number of inputs or outputs. These costs are detailed in Section 6.3, where empirical averages are also presented.

In a typical recognition run, the dominant terms in the complexity formula are the first three. I_E is typically the largest of the item sets in the first partitioning. I_C is the largest in the second partitioning. It usually consists mostly of items that were created by extension as opposed to instantiation or zip-up (i.e., a majority of I_C overlaps with I_E).

The run-time space requirements of the parser also depend on the number of items created by the parser. The space cost is $O(|I_T|)$.

6.2 Counting Items

The algorithm's complexity (both time and space) depends on how much is recognized. This is a feature of the algorithm and is a consequence of the bottom-up rule invocation strategy used by the parser. The amount recognized can be measured by the number of items the parser creates, since each represents a partial or complete recognition of some sub-flow graph.

This section focuses primarily on characterizing the number of items that are created by the parser through extension. In practice, more items are created by extension than by instantiation or zip-up. Its size dominates the space cost, and the run-time cost of operations over this set dominates the parser's time complexity.

To simplify the presentation, we *temporarily* assume that no items are created by zipping up items. In this way, we avoid cluttering the discussion with details about zip-ups which might be irrelevant to other applications of the graph parser besides program recognition, which do not require parsing structure-sharing graph grammars. In Section 6.2.6, we consider the effect of zip-ups on the total item count.

We also simplify the discussion by assuming *for now* that the nodes of each rule's right-hand side are matched according to a *strict* node ordering. One effect of enforcing a strict node ordering is that the parser does not generate duplicate items representing the same analysis. That is, each item created by extension is unique in that there is no other item for the same rule R which has the same matches for each of R's right-hand side nodes.

To see this, suppose an item I_1 were created for which there is a duplicate item I_2. The two items would have to be created through a series of extensions involving the same complete items for the same right-hand side nodes, but the extensions would have to occur in different orders. This is because each partial and complete item is added to the chart at most once and they are combined with each other only once – when the second of the two is added to the chart. So, the same partial item cannot be extended more than once by the same complete item for the same node. Since the series of extensions must have occurred in different orders, some partial item must have been extended with complete items for more than one right-hand side node. This can only happen to a partial item that has more than one immediately needed node, which can only occur when partial node orderings are being used. Therefore, with strict node orderings, no duplicate items representing the same analysis will be created.

Another effect of using a strict node ordering is that fewer partial items are created. By the argument just given, strict node orderings permit only one possible series of partial items leading to a complete item through extension. Partial node orderings may allow several series of extensions, each involving a different set of partial items.

The reason we consider the case of using strict node orderings first is that this makes it easier to see the effect of constraints on reducing the parser's search. We want to study the growth in the number of items for a particular rule as the size of the items increases. This growth is affected by two things: the constraints that are acting on the right-hand side nodes matched so far and the number of immediately needed nodes an item can have. Strict node orderings force the number of immediately needed nodes of any partial item to be exactly one. So, imposing a strict node ordering on all rules allows us to study the effect of constraints on the growth of the number of items, independent of the effect of multiple immediately needed nodes.

Another reason we make this simplification is that parsing using a strict node ordering is one of the ways in which this parser is expected to be used. It is more efficient than parsing with partial node orderings since, in general, it allows fewer partial items to be created. (String chart parsing is a general case in which strict node ordering is typically used, where the "nodes" are string symbols.)

The analysis of the algorithm when partial node orderings are being used is an extension of the analysis of this simplified form. This is given in Section 6.2.7, where the advantages of using strict versus partial node orderings are also discussed.

The organization of this section is centered around the characterization of the number of items generated for a single rule through extension. The total number of items created by extension is the sum of this number over all the rules of the grammar. Section 6.2.1 defines *item trees*, which relate the items created by the parser in matching a rule's right-hand side. Sections 6.2.2 and 6.2.3 discuss the effect that constraints and the grammar have on the growth of these trees. Empirical observations of the shape of item trees (i.e., the growth of the number of items) created in two typical recognition runs are given in Section 6.2.4. In Section 6.2.5, we borrow a theoretical model presented by Grimson [49, 50] in his analysis of the constrained search object recognition technique, which is similar to the sub-flow graph matching subprocess performed by our parser. The model helps us to understand the role of constraints and suggests future research into ways of concretely measuring their effectiveness for a particular input flow graph and grammar. The final two sections (6.2.6 and 6.2.7) lift the two simplifying assumptions of suppressing zip-ups and using only strict node orderings and discuss the effects this has on the parser's complexity.

6.2.1 Item Trees

For each rule, the parser searches for a match of the rule's right-hand side nodes, such that the rule's constraints hold. Each right-hand side node is matched to some terminal node or

some non-terminal instance that has been found in the input graph. The rule's constraints are unary (such as node type constraints) or binary (such as edge connection constraints). The items for a rule R represent each of the stages in this search. The *size* of an item is the number of right-hand side nodes of the item's rule it has matched so far. The number of items created is an indication of the amount of search the parser is doing.

The items for a rule R can be viewed as vertices of an *item tree*. The root of the tree is the empty item for R. An item is the *child* of another item (called the *parent*) iff the parent was extended to the child during parsing.

A parent item can be extended to two children items if more than one instance of some right-hand side node type is found in the input graph and these instances satisfy the constraints imposed by the item's rule with respect to the matches of other nodes that have been made so far. (With partial node orderings, additional children are generated if an item has more than one immediately needed node, as is discussed in Section 6.2.7.)

The growth in the number of items that are created by extension can be modeled by these item trees. In the *worst* case, the number of items at the fringe of an item tree for a given rule R can be exponential in the number of nodes in R's right-hand side, k. In particular, if each node in the right-hand side can be matched to m instances of its node type, then the number of possible complete items (of size k) is m^k and the total number of items created in recognizing R's right-hand side is $\sum_{i=0}^{k} m^i$

Furthermore, in general, m can be much worse than linear in the number of nodes of the input graph because of the recursive nature of the matching process in parsing. Each of the complete items at the fringe of an item tree for a rule R represent instances of R's left-hand side node type. Since there can be an exponential number of them, m can be exponential. In the worst case, this exponential can build up as higher-level non-terminals are recognized. (Assuming the grammar contains no cycles, we define the *height* of a node type recursively as: the height of a terminal type is 0 and the height of non-terminal type A is one plus the maximum of the heights of all node types on the right-hand sides of the rules for A.)

As the worst case, suppose the following. All rules have right-hand sides of size k. Each non-terminal has only one rule for it. Each right-hand side has either only terminals or only non-terminals. Each terminal node can match n input graph nodes. Each non-terminal in the same right-hand side is at the same height in the grammar. Then, the number of complete items for a non-terminal at height h is n^{k^h}.

6.2.2 Constraints Prune Item Trees

It would be crazy to use this inherently exponential algorithm for program recognition if it were not that, in practice, constraints prune item trees considerably. For example, node type constraints alone are able to reduce the branching factor, which is the base of the exponential. In the program examples, there is a variety of terminal and non-terminal node-

types, with a fairly flat distribution of instances. In CST, the average number of instances of each node type is 3.6, with a median of 2. In PISIM, the average is 3.7, with median 2.

The exponential build-up of the number of instances of non-terminals as their height increases is not typically encountered, either. The number of instances of non-terminals is usually small and decreases as their height in the grammar increases. The reason is that the recognition of high-level non-terminals requires more constraints to be satisfied than for low-level non-terminals.

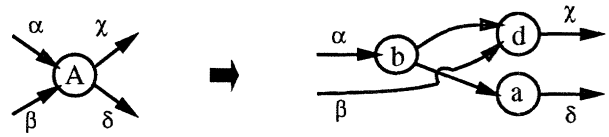
The worst-case exponential behavior of the parser is only encountered if the constraints imposed by the grammar rules are weak. This section explores the constraints used in applying the graph parser to program recognition and describes their effect on the growth of item trees in terms of empirical observations.

A complete item for a non-terminal A is one in which for some rule for A, all the rule's right-hand side nodes are matched to input graph nodes or non-terminal instances, such that the rule's unary and binary constraints are satisfied. The unary constraints are the *node-type* constraints that each node in the right-hand side imposes on the nodes matched with it. The binary constraints are the following:

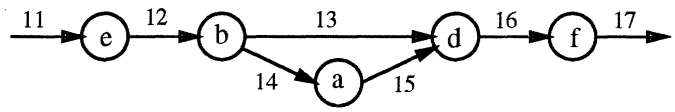
- *Edge connection* constraints between pairs of ports on nodes. (These include the constraints on aggregation organization discussed in Section 3.5.2.)
- *Attribute conditions*, which are binary relations on the attributes of nodes and edges.
- *Port precedence* restrictions, which are constraints on the edges in an input graph that can be mapped to the ports of a non-terminal. In particular, a transitive, irreflexive, and antisymmetric relation *precedes* imposes an ordering on the ports in the input graph. The source of each edge precedes the sink of the edge and the input ports of each node precede each of the node's output ports. The port precedence constraint is that no two input (or output) ports on a non-terminal can be mapped to a pair of input graph edges in which the sink of one precedes the source of the other.

The port precedence restrictions are used to avoid cyclic reductions, such as the one shown in Figure 6-3. The non-terminal A's top input port is mapped to the input graph edge with location pointer 12 coming into b, while A's bottom input port maps to the edge with location pointer 15 coming from a. This is illegal, since b's input precedes a's output. The reason cyclic reductions are prevented is that they are unnecessary:

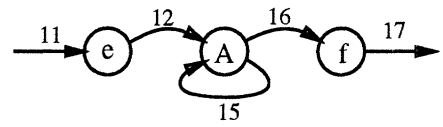
- flow graphs are acyclic,
- all sentential forms of a flow graph grammar are acyclic (i.e., you cannot derive a flow graph that is cyclic),
- a reduction step that creates a cyclic graph cannot be the inverse of any valid derivation step, so the cyclic graph will not be reduced further.



a) A simple grammar.



b) An input graph.



c) A cyclic reduction.

Figure 6-3: Grammar and input graph leading to an illegal, cyclic reduction.

Cyclic reductions do not cause any problems. They simply result in dead-end items that are not used by anyone. We avoid them simply because they waste time and space. This restriction can be lifted if a cyclic reduction is a useful interpretation to report and the flow graph formalism is extended to include cycles.

Some of these unary and binary constraints are applied incrementally to each partial item as the complete match is being built up. Since these are interleaved with the *matching* process, we refer to them as *match-interleaved* constraints. They are applied as soon as the portions of the right-hand side to which they refer are matched. These constraints are part of the extendibility criterion.

Other constraints are postponed until the match is complete (i.e., all nodes and edges of the right-hand side are paired with nodes and edges of the input graph). These are interleaved with the *parsing* process and are referred to as *parse-interleaved* constraints.

The decision about whether to match-interleave or parse-interleave a particular constraint depends on its effectiveness in pruning the search, the cost of applying it, and its degree of applicability. Ideally, the match-interleaved constraint should be satisfied by relatively few matches, be inexpensive to check, and apply to most nodes or pairs of nodes. The current recognition system match-interleaves node-type, edge connection, co-occurrence, and port precedence constraints. All attribute conditions besides co-occurrence constraints, are parse-interleaved. This section discusses how this decision was made and

node-type	number of instances
aref	6
mod	4
Increment-or-Decrement	12
Decrement	3

Table 6.1: Number of instances of CIS-Extract's node types.

describes the impact that match-interleaving of these constraints has on the complexity of matching right-hand sides in the two example simulator programs.

We are not only trying to show the advantages of match-interleaving some constraints versus parse-interleaving them. (The advantages are obvious.) We are mainly trying to show the effect that various constraints have on the complexity. The case in which a constraint is parse-interleaved is simply a base-line to which to compare the case in which the constraint is match-interleaved. The improvement is a measure of the effectiveness of that constraint.

For most rules, node type and edge connection constraints are strong. The strength of a node-type constraint depends on the number of instances of that node-type in the input graph. Since the distribution of node types is fairly flat in the flow graphs representing the two example programs, the node type constraint can usually significantly reduce the number of possible matchings between right-hand side nodes and node type instances in the input graph.

The strength of an edge connection constraint depends on the number of edges in the input graph. If this number is low, then few pairs of incorrect matches between nodes will satisfy the constraint. The flow graphs representing the two example programs had sparse edge sets. The average degree of the ports in `CST` is 1.3, with a median of 1. In `PISIM`, the average degree is 1.5, with a median of 1.

However, there is a class of rules for which node type and edge connection constraints are weak. In particular, in rules representing clichéd operations on aggregate data structures, the right-hand side graph is usually made up of disconnected nodes. The operations on aggregate data structures tend to be implemented using a set of less abstract operations that act on the parts of the structure independently. In addition, many of the aggregate operations are implemented by primitive operations that are relatively common in the program (e.g., +), as well as being common among the aggregate operations.

The plan for Circular-Indexed Sequence Extract is an example (see Figure 6-4). The rule encoding a plan like this imposes few structural constraints, since it has few edges between its nodes. It also contains nodes that are of relatively common node types. Table 6.1 shows the distribution of number of instances over these node types.

If no other constraints are interleaved with the matching process, a combinatorial explosion occurs in the number of items created in recognizing CIS-Extract. Figure 6-5 shows

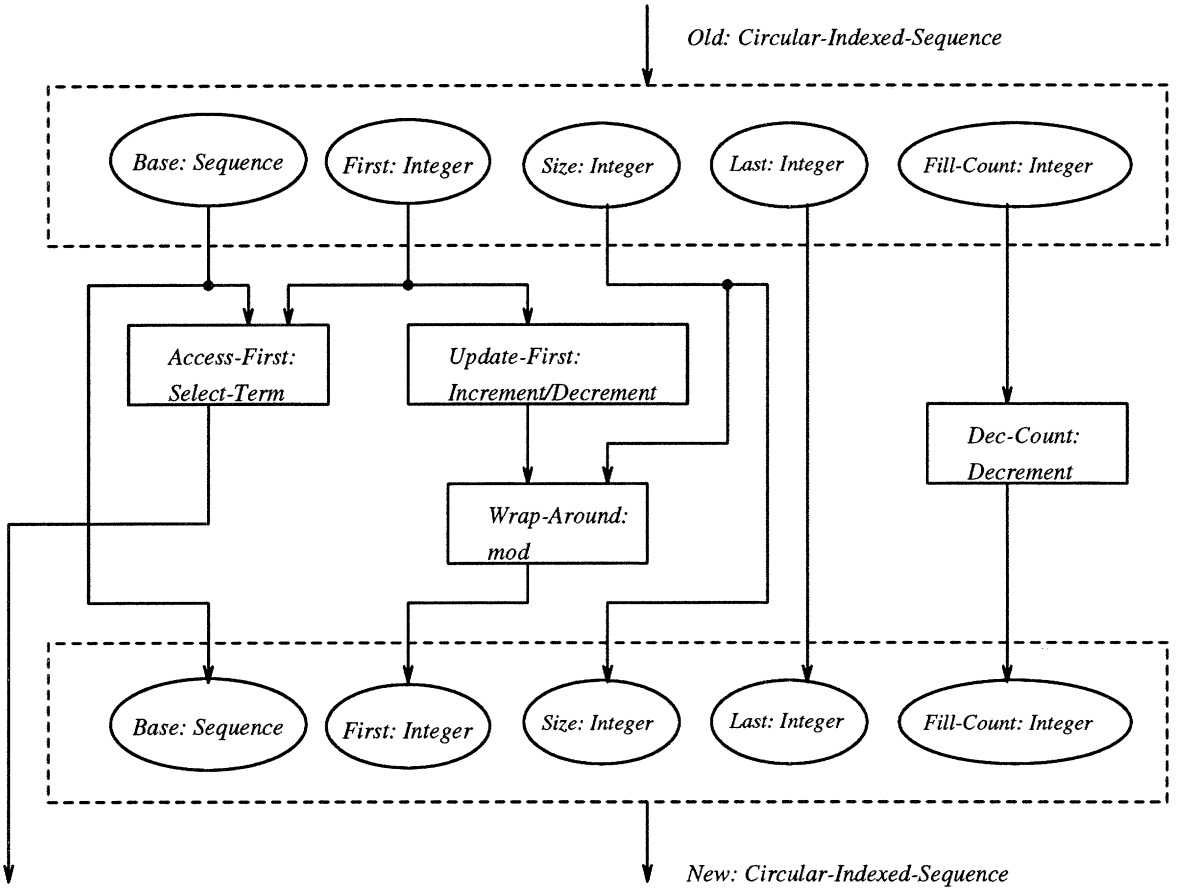


Figure 6-4: The plan for extracting from a Circular-Indexed Sequence.

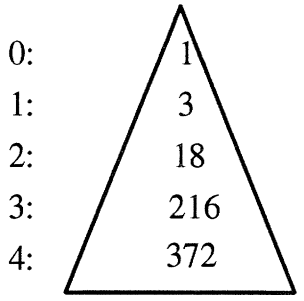


Figure 6-5: Bushy item tree produced in recognizing CIS-Extract with weak match-interleaved constraints.

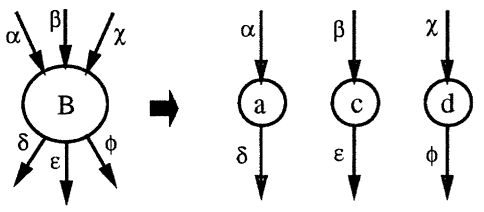
the bushy item tree created for CIS-Extract in this case. The items of size 1 are those created in extending the initial empty partial item with the complete items representing three instances of Decrement. Each of these are then extended with the six complete items for the **AREF** terminal nodes, yielding 18 items. Each of these is extended by the 12 complete items for Inc-or-Dec, yielding 216 items. Finally, the parser extends these with each of the four complete items for **MOD** for which the edge connection constraint is satisfied.

This shows how a lack of strong match-interleaved constraints causes the number of partial items to build up exponentially. In fact, flow graph parsing with a flow graph grammar whose rules impose no edge connection constraints or any other binary constraint is NP-complete. Appendix A shows that the problem of recognizing unordered context-free grammars (**UCFG**) can be reduced to flow graph parsing. **UCFG**s are context-free string grammars in which the symbols in the right-hand side string are considered unordered. (For example, given a **UCFG** containing the rule $S \rightarrow xyz$, S can be recognized in the strings xyz, yxz, zyx, etc.)

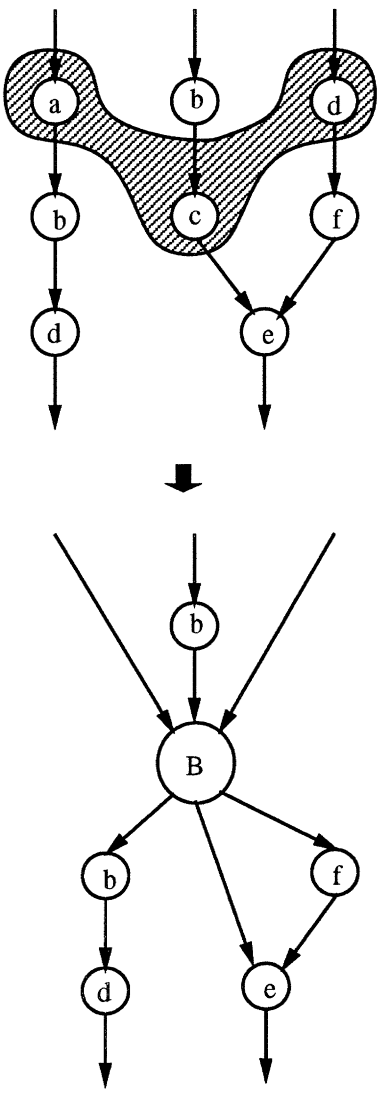
Fortunately, in applying the flow graph parser to program recognition, other constraints can be interleaved with the matching process to prune item trees early. These are the co-occurrence and port precedence constraints. (As described in Section 4.1.1, if two nodes in a right-hand side are constrained to co-occur, then they must match nodes that represent operations in the same control-environment.)

The precedence relation constraint enforces the condition that the data structure operation must cut across slices of dataflow, rather than allowing the disconnected pieces of the operation to be recognized vertically in the same slice. See Figure 6-6. Cyclic reduction avoidance prevents B from being recognized in the rightmost graph.

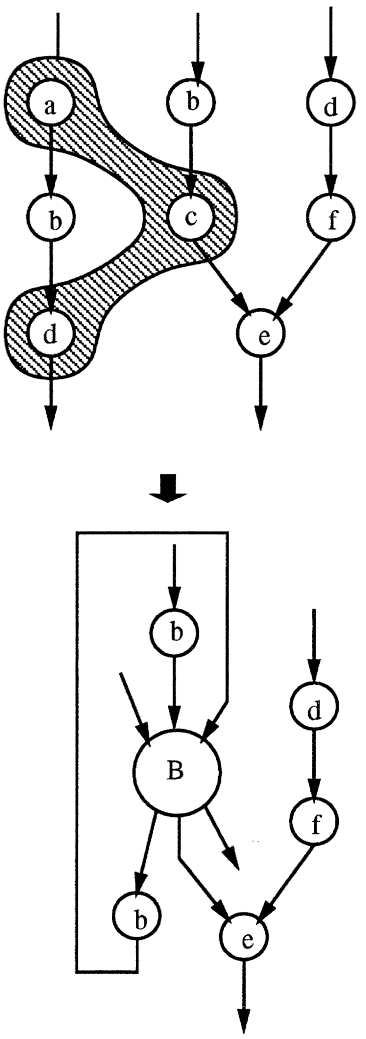
The advantage of match-interleaving these constraints can be seen by contrasting the parser's performance when match-interleaving the constraints to its performance when these constraints are parse-interleaved. In the parse-interleaving case, item trees for data structure operations are extremely bushy and can be exponential in the worst case. Most of the items at the leaves are killed by the co-occurrence and port precedence constraints when they are finally applied. For example, the item tree for CIS-Extract, shown in Figure 6-5, has



A grammar rule



A legal reduction.



An illegal reduction.

Figure 6-6: The restriction on legal instances imposed by the precedence relation constraint.

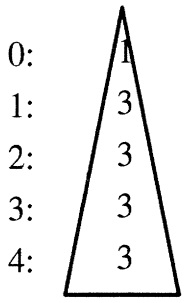


Figure 6-7: Skinny item tree produced in recognizing CIS-Extract with strong match-interleaved constraints.

372 items at height 4, but only 3 of these satisfy the co-occurrence and port precedence constraints.

With match-interleaving, the items trees are much shorter and skinnier, since the co-occurrence constraints are applied as early as possible. Figure 6-7 shows the item tree for CIS-Extract. As soon as the Decrement node is matched, the matches of all the other nodes are disambiguated to involve only nodes in the same control environment.

The influence that match-interleaving co-occurrence constraints has on reducing the parser's search can also be seen by contrasting the parser's time and space requirements when match-interleaving is performed versus when parse-interleaving is used. We do the same in order to study the influence of match-interleaved port precedence constraints. This helps us evaluate the effectiveness of each constraint in reducing the overall complexity of the parser and it allows us to compare the relative effectiveness of the two constraints.

Figure 6-8 shows the results of running the `CST` example under the following four conditions: a) parse-interleave both constraints, b) match-interleave co-occurrence, parse-interleave port precedence, c) parse-interleave co-occurrence, match-interleave port precedence, and d) match-interleave both.[2] In Figure 6-8, the number of items created by the parser is shown as the number of items of three different types. "Successful" items are complete items which satisfy all their rules' constraints. "Killed" items are complete or partial items that have failed their rules' constraints. "Extendable" items are partial items that have not yet failed any match-interleaved constraints and may be extended with complete items for their immediately needed nodes. (The relationship between these sets and the sets of complete and partial items is shown in Figure 6-9.)

The number of successful items remains the same over all the cases, as it should. The effect of the two constraints can be seen in the total number of killed and extendable items, which is reduced by more than 70% (from 2235 to 662) by match interleaving both constraints. This has the effect of dramatically speeding up the parser – when match-

[2]The run times for the experiments in this chapter were obtained by running the recognition system on a Sparc 2 in Lucid. These statistics were collected with zip-up creation being performed, since zip-ups are needed to recognize the simulator clichés. However, the number of zip-ups created in these runs is relatively small, as is discussed in Section 6.2.6.

a) Parse-Interleave Both

Time:	201	seconds
Successful:	329	
Killed:	1432	2235
Extendable:	803	2235

b) Match-Interleave Co-occur;
Parse-Interleave Precedence

Time:	86	seconds
Successful:	329	
Killed:	505	749
Extendable:	244	749

c) Parse-Interleave Co-occur;
Match-Interleave Precedence

Time:	190	seconds
Successful:	329	
Killed:	1230	1966
Extendable:	736	1966

d) Match-Interleave Both

Time:	86	seconds
Successful:	329	
Killed:	446	662
Extendable:	216	662

Figure 6-8: Results of running `CST` example with constraints parse-interleaved versus match-interleaved.

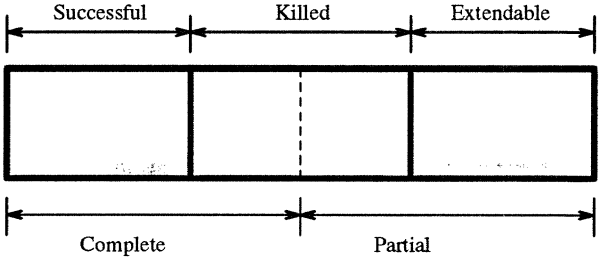


Figure 6-9: Relationship of the sets of successful, killed, and extendable item sets to the sets of complete and partial items.

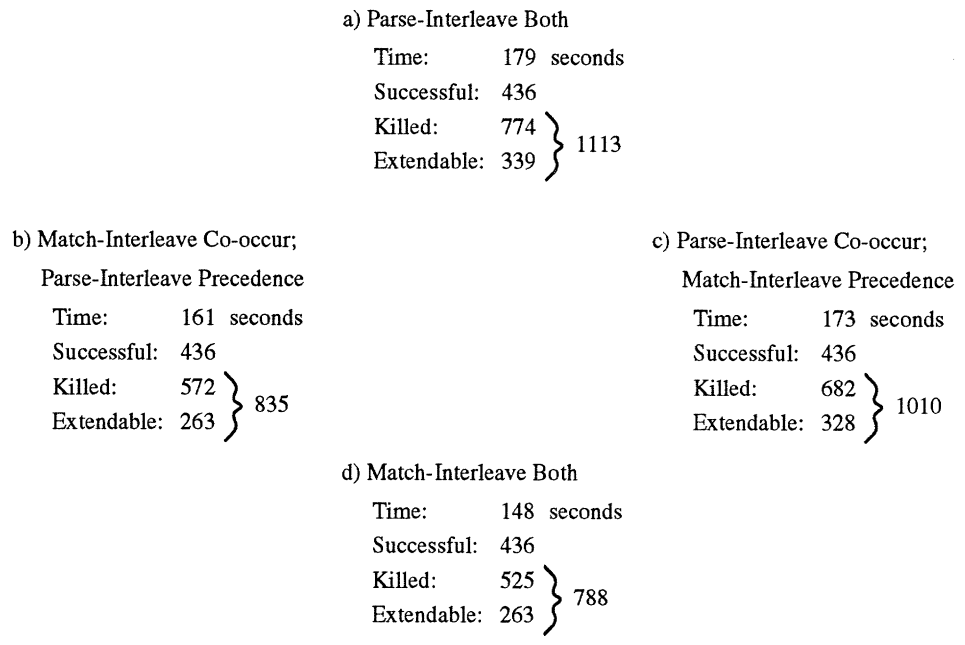


Figure 6-10: Results of running `PISIM` example with constraints parse-interleaved versus match-interleaved.

interleaving both constraints, the parser is 133% faster than when parse-interleaving them.[3] This is because partial items are killed earlier. Only 12% of the killed items had less than half of their rules' right-hand sides matched when the two constraints were parse-interleaved. However, when the constraints were match-interleaved, 53% of the killed items had less than half of their rules' right-hand sides matched. This causes fewer extendable items to be created, and therefore fewer killed items as well.

Most of the savings are the result of match-interleaving co-occurrence constraints which reduces the number of killed and extendable items by 66% (from 2235 to 749). Port precedence constraints have a more modest effect, reducing this number by only 12% (from 2235 to 1966).

In the `PISIM` example, match-interleaving has a less dramatic impact than in the `CST` example, but it still helps, as can be seen in Figure 6-10. Match-interleaving both constraints reduces the killed and extendable item count by 30% (from 1113 to 778). This is simply because the rules used in recognizing the clichés in `PISIM` had strong node type and edge connection constraints with respect to the input graph representing the `PISIM` program. There was not as much need to rely on co-occurrence or port precedence constraints to prune the search.

As in the `CST` example, match-interleaving co-occurrence constraints had more of an

[3] Performance is the reciprocal of execution time, so performance increase n (as in "X is n% faster than Y") is computed from the relationship: $1+\frac{n}{100} = \frac{Performance_X}{Performance_Y} = \frac{Execution_Y}{Execution_X}$. (See Hennessy and Patterson, Section 1.2 [57].)

effect than match-interleaving port precedence constraints. Match-interleaved co-occurrence checking reduces the number of killed and extendable items by 25% (from 1113 to 835), while match-interleaved port precedence checking only reduced the number by 9% (from 1113 to 1010).

The two experiments above allow us to evaluate the co-occurrence and port precedence constraints as candidates for match-interleaving, with respect to two particular input flow graphs and a specific graph grammar. Co-occurrence constraints are excellent candidates, in terms of their effectiveness, cost, and applicability. Co-occurrence constraints are effective as evidenced by the vast decrease in the number of items created when they are match-interleaved. They are particularly valuable when other binary constraints are weak which is the case in the rules representing aggregate data structure clichés that are activated in recognizing the `CST` example. Co-occurrence constraints can be checked cheaply by simply comparing two attribute values. Since all nodes have control environments, co-occurrence constraints are applicable to any pair of nodes in a right-hand side.

Port precedence constraints are also good candidates for match-interleaving, although not as good as co-occurrence constraints. They are modestly effective in reducing the number of items created. The cost of checking port precedence constraints incrementally is no more than the cost of checking them all at once when an item is complete. Their applicability is limited to only input ports of a right-hand side graph. That is, if they are included as part of the extendibility criterion, they only apply to pairs of partial and complete items in which the complete item is representing the recognition of a left-fringe node.

Implications for Chart Organization

The decision as to which constraints should be interleaved with the matching process concerns which constraints should be included as part of the extendibility criterion. The extendibility criterion is checked in two steps. Some parts of the extendibility criterion are enforced when a candidate item is retrieved from the chart. The rest are checked by filtering the candidate items that have been retrieved. The parts that are checked during candidate retrieval influence the design of the organization of the chart.

If a certain constraint is strong in that it can usually be satisfied by only a few items and this constraint refers to some attribute or part of an item, then it can be used as an index into the chart. Node type and edge connection constraints are very important in reducing the combinatorics of matching many right-hand sides. Currently, the chart is organized so that complete items are indexed by their label and location and partial items are indexed by the node types of their immediately needed nodes and the locations at which they are needed. Constraints on node type and location are therefore enforced during item retrieval. In the future, it might be beneficial to index on control-environment information as well.

6.2.3 Grammar Facilitates Reusing Sub-Search Space Exploration

In addition to constraints, the complexity of parsing can be reduced if the grammar captures the commonalities among the flow graphs being recognized in its hierarchical structure. The grammar may specify that a non-terminal derives some sub-flow graph that is common to several other flow graphs. When an instance of this non-terminal is found, the results of the recognition are reused in recognizing all the flow graphs that contain it, rather than repeatedly matching the common sub-flow graph.

In terms of item trees, the effect of a good grammar organization such as this is that it prevents multiple redundant sub-trees from being grown within each tree. In other words, if the grammar captures commonality, the parser can avoid exploring parts of the search space over and over.

6.2.4 Empirical Observations of Item Trees

In using the graph parser to recognize two example simulator programs, we have found the item trees to be typically sparse and skinny. This section summarizes statistics concerning the characteristics of the item trees that are created in recognizing the `CST` and `PISIM` programs.

In the recognition runs, both co-occurrence and port precedence constraints are match-interleaved. Also, zip-up creation was being performed by the parser, since it is needed to recognize the simulator clichés. Zip-up items increase the number of instances of particular node types. However, the number of zip-ups only negligibly increases the number of items created in parsing. Since there are so few of them, they do not significantly affect the node type distribution nor the branching factor of item trees. Section 6.2.6 characterizes the number of zip-up items created by the parser and gives empirical statistics for the actual number created in practice.

The "bushiness" of the item trees gives an indication of whether the parser is encountering exponential behavior. We measure this property of the trees in the following ways. We look at the *maximum width* of the item trees and observe how it changes as the *height of the item trees* increases. The maximum width of an item tree is the maximum, over all possible sizes of items, of the number of items in the tree of a particular size. (It is the same as the maximum number of items at a particular level in an item tree.) If the parser requires exponential space and time, the maximum width will increase exponentially with the height of the tree. The height of an item tree is the maximum size of the items in the tree.

We also look at the *branching factor* of the trees and how it varies as we increase the *height of the non-terminal* being recognized. This is done to detect an exponential buildup in the number of instances of non-terminals as their height in the grammar increases. (Recall the worst case of this can cause $O(n^{k^h})$ number of instances of a non-terminal at height h to be created using a rule whose right-hand side is of size k, as discussed at the beginning of

tree height	maximum maximum width	average maximum width	median maximum width
0	1	1.00	1
1	28	5.84	3
2	28	10.88	5
3	13	6.60	6
4	43	19.00	16
5	3	3.00	3

Table 6.2: Tree height versus maximum width statistics for item trees in `CST`.

tree height	maximum maximum width	average maximum width	median maximum width
0	1	1.00	1
1	24	5.77	4
2	43	8.09	5
3	9	6.00	6
4	38	13.25	4
5	0	0.00	0
6	0	0.00	0
7	32	32.00	32

Table 6.3: Tree height versus maximum width statistics for item trees in `PiSim`.

Section 6.2.) If the parser is experiencing an exponential explosion, the average branching factor over all the trees of non-terminals of a particular height in the grammar will increase as the height is increased. Otherwise, it will stay the same or decrease.

Maximum Width

For each item tree, we computed its maximum width, which is the maximum number of items on any level in the tree. Tables 6.2 and 6.3 show, for each tree height, the maximum, average, and median maximum width of the trees of that height.

As the tree height increases, none of the statistics for the maximum width of the trees increase exponentially. This includes the maximum of the maximum widths of the trees at each possible height, which would indicate the existence of even one bushy tree. For the trees over a particular height, the average maximum width is typically much smaller than the maximum maximum width and the median maximum width is even smaller. This means that there are few relatively wide trees among trees of a particular height.

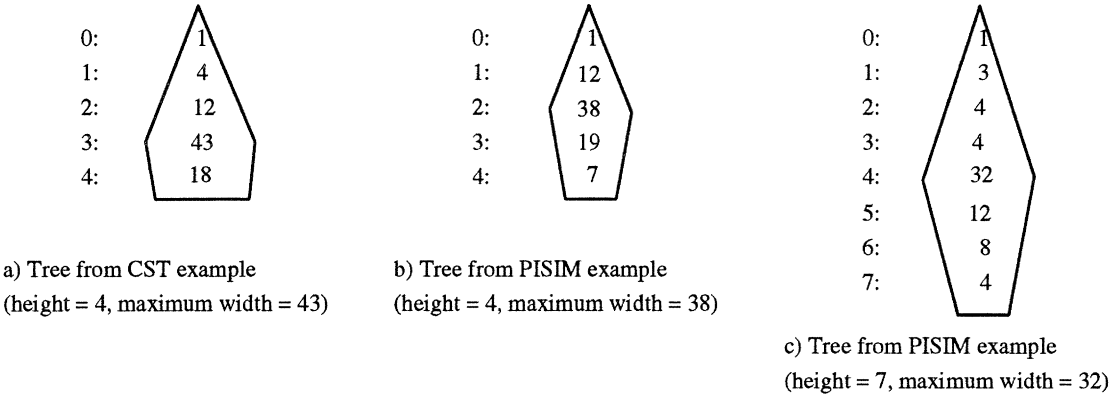


Figure 6-11: The shapes of item trees having maximum maximum width.

In general, for trees of height 4 to 7 the maximum width level of an item tree occurs in the middle of the tree. The width tapers off deeper in the tree, as constraints prune it. Figure 6-11 shows the shapes of trees of height 4 and 7 which have the maximum maximum width. The shapes are shown in terms of the width of each level.

Branching Factor

We now observe how the branching factor of an item tree changes as we vary the height of the non-terminal being recognized by the items in the item tree. Tables 6.4 and 6.5 show the maximum, average and median branching factor over all the item trees of each possible non-terminal height for `CST` and `PISIM`, respectively. In general, the branching factors of item trees produced in both examples decrease as the height of their non-terminal increases. So there is no exponential build-up occurring as non-terminals higher in the grammar are recognized.

For low-level non-terminals, the maximum branching factor is much worse than the average or median branching factors. This shows that the relatively bushy trees for these non-terminals are few in number. (For high-level non-terminals, the maximum branching factor is comparable to the average and median branching factor, which is small – only 1 for most high level non-terminals in the `CST` example!)

The table also includes the maximum maximum width of all the trees at each non-terminal height. This shows that in general the maximum width trees occur in recognizing low-level non-terminals.

These statistics show that the item trees produced in recognizing the two example programs are typically skinny. These examples represent two real programs, showing the good behavior of the parser in practice, despite its potential for worst case exponential performance. Further experimentation is need with other programs to see how typical this is and what additional constraints may be needed to keep the complexity under control.

non-terminal height	maximum branching factor	average branching factor	median branching factor	maximum maximum width
1	12.00	8.17	6.00	12
2	28.00	16.34	6.80	28
3	9.00	7.75	8.00	9
4	7.00	3.01	2.33	43
5	19.00	4.76	3.00	19
6	19.00	4.76	3.00	19
7	3.00	1.50	1.00	3
8	6.75	3.16	1.74	14
9	4.00	2.33	2.00	5
10	3.00	1.83	1.33	3
11	9.00	3.25	1.00	9
12	2.50	2.50	2.50	6
13	1.00	1.00	1.00	1
14	1.00	1.00	1.00	1
15	1.50	1.50	1.50	2
16	1.00	1.00	1.00	1
17	1.00	1.00	1.00	1
18	1.00	1.00	1.00	1
19	1.00	1.00	1.00	1
20	2.33	1.67	1.00	6
21	0.00	0.00	0.00	1
22	0.00	0.00	0.00	1
23	0.00	0.00	0.00	1

Table 6.4: CST: Branching factor statistics for item trees of non-terminals over the range of possible node-type heights.

non-terminal height	maximum branching factor	average branching factor	median branching factor	maximum maximum width
1	15.00	8.35	7.00	38
2	24.00	8.90	4.00	24
3	10.00	6.46	6.25	43
4	4.00	2.69	2.50	16
5	7.00	2.13	2.00	7
6	2.00	1.51	1.50	9
7	5.00	2.73	2.33	6
8	2.00	2.00	2.00	2
9	3.00	2.33	3.00	3
10	3.00	1.87	1.60	4
11	3.33	3.33	3.33	6
12	7.00	4.50	2.00	7
13	2.00	2.00	2.00	2
14	2.00	2.00	2.00	2
15	3.00	2.50	2.50	4
16	4.00	3.00	4.00	4
17	4.00	2.50	1.00	4
18	2.39	2.39	2.39	32
19	4.00	4.00	4.00	4
20	2.56	2.56	2.56	8
21	4.50	4.50	4.50	5
22	4.00	4.00	4.00	4
23	1.60	1.60	1.60	4

Table 6.5: `PiSim`: Branching factor statistics for item trees of non-terminals over the range of possible node-type heights.

6.2.5 Modeling Constraint Consistency

We can discuss the effect constraints have on the complexity of recognition in terms of a model of consistency Eric Grimson [49, 50] presented in analyzing his constrained search object recognition algorithm. (This in turn is based on general analyses of the consistent labeling problem of which constrained search and sub-flow graph matching are specializations.)

In constrained search, sensory data are searched for an object model, by incrementally building a tree of *interpretations*, which are lists of pairings of data and model features. Each node in the interpretation tree represents an interpretation of size k, where k is the level of the node in the tree. The size of the interpretation is the number of pairings it contains. Each of the children of a node that represents an interpretation I represent an augmentation of I with an additional pairing. At each step, the additional pairings are all between the same data fragment and each of the possible model features.

Interpretation trees are analogous to item trees that are produced when strict node orderings are used. However, the roles of model and data fragments correspond to the roles of the input graph and right-hand side graph, respectively. (At each step in the item tree, the partial items are all extended with complete items for the same right-hand side node, not the same input graph node.)

Unary and binary constraints are used to prune the interpretation trees. For example, these are edge length and relative distance constraints. Grimson's formulation captures the notion that as the size of an interpretation increases, the probability that a random matching of that size is consistent in terms of the constraints decreases. This means that if the unary and binary constraints are strong enough, the interpretation trees will tend to be sparse rather than bushy.

Grimson defines the number of analyses of a particular size in terms of the probability that an analysis of that size will be consistent in terms of the constraints.

The probability that a set of data-model pairings will satisfy unary and binary constraints even if they are not part of a correct interpretation depends on the strength of the constraints. This in turn depends on the properties of the data and models. In the flow graph parsing problem, several input graph nodes of the same type (ambiguity) will weaken the unary node type constraints of right-hand sides containing that node-type. This will make it more likely that a random pairing of an input graph node with a right-hand side node will satisfy this constraint even though the pairing is not part of a valid interpretation. Similarly, if the input graph is highly connected, edge connection constraints are more likely to be satisfied by random pairings.

Grimson relates this probability to properties of the object recognition problem, such as the amount of sensory error, the number of model fragments, and the model object's perimeter. He then proves that the expected amount of search to find a correct interpretation is quadratic in the parameters (when all the data belong to the same object and the

identity of the object is known).

In the future, it would be interesting to compute the analogous relationship of probabilities of consistency to properties of programs and clichés, such as node-type or control environment distributions or number of dataflow dependencies. The probabilities provide a measure of the effectiveness of the constraints. This information could then be used to automatically generate advice concerning the optimal order of application of constraints.

Grimson also provides interesting results that point out the need for good *indexing* and *selection* techniques to control the complexity of recognizing partially occluded objects in noisy, cluttered scenes. Indexing is the problem of selecting from the model object library a small number of model objects that are likely to be in the scene. Selection is the problem of grouping together data features that are likely to have come from the same object. These results carry over to the program recognition domain. They will be relevant to future work in applying our parser to the analogous task of near-miss recognition, which is the task of finding the "best" partial recognition of a cliché. (Currently, our recognition system is able to do partial recognition of programs, but does not generate maximally-sized partial recognitions of clichés.) Section 6.2.7 discusses this further.

6.2.6 Counting Zip-ups

The effect of zipping up complete items is that more instances of non-terminals may arise. This can cause the branching factor to increase in item trees for higher-level non-terminals. Usually, however, the binary constraints on the inputs and outputs of the zipped up items (especially the edge connection constraints) are powerful enough to quickly disambiguate the instances so the branching factor is not affected much.

The number of zip-ups depends on the number of instances of a non-terminal found at a particular location such that:

- either all of the edges specified in the candidates' input mappings share the same source ports or all of the edges in their output mappings share the same sink ports, or both,
- none of the input mappings of the candidates overlap (i.e., contain common edges) and neither do the output mappings, and
- the attribute values of the zipped up item's left-hand side are defined, with respect to the attribute combination function. (See Section 3.5.1.) In other words, zipping up the candidates makes sense in terms of the attributes of the resulting non-terminal instance.

To count the number of zip-ups for some non-terminal or terminal node-type, partition items for the node-type into maximally-sized groups of items that can be zipped up, according to the above definition. These groups may overlap. Within each group of items,

CST		PiSim	
height	number of zip-ups	height	number of zip-ups
0	3	0	7
1	4	1	10
2	3	2	5
3	1	3	0
4	0	4	0
5	1	5	0
≥ 6	0	≥ 6	0

Table 6.6: Distribution of zip-up count over height of node-type in grammar.

zip-ups are created from each subset of the group (for subsets of size greater than one). So, for a group g of items that can be zipped up, $2^{|g|} - |g| - 1$ items are created.

Empirical Observations

Zipping up is actually a rare occurrence in practice. The reason is that programmers tend not to write redundant code. Function-sharing is a common optimization employed to avoid redoing work – for the programmer in writing the code and for the machine in executing it. (Optimizations usually add to the complexity of recognition, but in this case, the function-sharing optimization actually helps.)

The need for zip-ups does occur, but relatively infrequently. Programmers cannot (or do not want to) share all common sub-computations. One reason is that sometimes it is cheap to recompute some value whenever it is used and the programmer does not want to go to the trouble of defining a local variable to hold the shared result. Another situation in which redundancy can occur is in writing conditionals in which some but not all of the branches contain common computations. The code is sometimes more understandable, and easier to write correctly if the computation is repeated, rather than shared. This situation is rare, since it is usually possible to combine the conditional cases that have the same consequence into a single case. Both of these situations normally involve small expressions, containing primitive functions. So the complete items that are typically zipped up are for terminals in the input graph or low-level non-terminals.

In the `CST` example, only 12 zip-ups were created (out of 991 total items) and they all were zip-ups of low level non-terminals. In `PISIM`, only 22 zip-ups were created (out of 1224 total items). In both cases, they all were zip-ups of items for terminals or low-level non-terminals, as the distribution of zip-up count over node-type height shows in Table 6.6. (Terminal node types have height 0.)

In both examples, the size of the group of candidate items being zipped up was either

two or three, with an average of 2.1 and a median of 2.

(Both examples were run with strict node orderings on the rules and match-interleaved co-occurrence and port-precedence constraints.)

6.2.7 Partial Node Orderings

When node orderings are not restricted to being strict, partial items can have more than one immediately needed node. This causes more partial items to be created. It also causes duplicate items to arise, which are worthless and are not added to the chart.

In terms of item trees, partial node orderings increase the branching factor of the trees. A partial item can be extended more than once with complete items for the same node (if there is ambiguity) and/or with complete items for more than one node (if the item has more than one immediately needed node). Section 6.2 explored the effect of ambiguity on the branching factor of item trees. This section discusses the effect of using partial node orderings.

The worst case partial node ordering is no ordering at all: no pair of right-hand side nodes is related. In this case, the number of different (non-duplicate) items created in recognizing a rule's right-hand side of size k nodes is at least 2^k. There is a partial item for each member of the power set of the rule's right-hand side nodes. (More than 2^k items are created if there is any ambiguity.) Contrast this with strict ordering in which only k items will be created if there is no ambiguity.

With no node ordering, there will be $m - 1$ duplicates of an item of size m. To see this, consider an item I_1 of size m. I_1's parent is one of m possible parents (since there are m ways of choosing a subset of size $m - 1$ of I_1's already matched nodes). All m possible parents have been created, since there is no node ordering. One is the parent of I_1. The other $m - 1$ are parents of duplicates of I_1.

So, with no node ordering, the total number of duplicate items created in recognizing a right-hand side flow graph of size k is

$$\sum_{m=1}^{k} (m-1) \left(\begin{array}{c} k \\ m \end{array} \right).$$

This section gives some empirical observations of the recognition of our example programs under the conditions of three different node orderings. It then discusses the advantages of using partial node orderings versus using strict node orderings, in terms of efficiency and recognition power. Finally, it discusses ways of choosing a rule's node ordering.

Empirical Results

To get a feel for how partial node orderings affect recognition performance, we perform recognition on our two example programs, using two different partial node orderings and compare the results to those obtained using strict node orderings.

One partial node ordering is *edge-based* in that a node n_1 is $\leq_n$ another n_2 if n_1 has an output connected to an input of n_2 and n_2 has no input that is an input of the right-hand side graph. The minimal nodes in this ordering are all the nodes in the right-hand side that are on the left-fringe (i.e., have input ports that are inputs to the right-hand side flow graph). When this node ordering is used, an empty partial item for recognizing some rule has all the left-fringe nodes of the rule's right-hand side as its initial set of immediately needed nodes. When a partial item is created by extending another partial item with a complete item for some node x, all nodes connected to x that have not already been matched are added to the immediately needed node set.

With the grammar used by the current system, an edge-based node ordering is an approximation of having no node ordering, which the current recognition system cannot handle because the current implementation is not flexible or robust enough. Edge-based orderings take advantage of the fact that many of the right-hand sides of rules in our grammar consist mostly of nodes that have at least one input that is an input of the right-hand side flow graph. These nodes will all be considered minimal nodes in the node ordering. If all nodes of a right-hand side have some input that is a right-hand side flow graph input, then none of the nodes will be ordered with respect to any other node.

The other node ordering considered is *topological*: a node n_1 is $\leq_n$ another n_2 if the two nodes are connected by an edge from n_1 to n_2 and there is no other node n_3 such that $n_1 \leq_n n_3$ and $n_3 \leq_n n_2$. (This is not exactly the same as a topological sort of a dag [21], since it does not completely linearize the partial order imposed by the edges of the flow graph. Nodes that have no edges connected to their inputs are not ordered with respect to each other.)

Each program was run with the edge-based node ordering and then with the topological node ordering. The results of these two runs can be compared to the results of recognizing the programs using a strict node ordering on the rules. The strict node orderings are optimal in that they are designed to match salient nodes first. They are manually assigned to the grammar rules.

Tables 6.7 and 6.8 show the results of the three experimental runs on the `CST` and `PISIM` programs, respectively. In the `CST` example, the strict node ordering is more than 200% faster than the edge-based ordering, reducing the total number of items by 62%, creating less than a third of the number of killed and extendable items. In fact, it creates less than one fourth the number of partial items that are not killed (i.e., are extendable). The strict node ordering does not save as much over the topological node ordering as it did over the edge-based ordering. However, it nearly halves the number of extendable items.

Similarly, in the `PISIM` example, using the strict node ordering allows the parser to run 238% faster than with the edge-based ordering and there is a reduction by more than 50% in the total number of items created with the edge-based ordering. Less than one fourth of the number of extendable items are produced. Again, there is only a slight difference in the number of items created in using the topological versus using strict node orderings.

items	edge-based	topological	strict
successful	329	329	329
killed	1296	491	446
extendable	994	418	216
total	2619	1238	991
killed+extendable	2290	909	662
time (seconds)	260	104	86

Table 6.7: Experimental runs with `CST` using three different types of node orderings.

items	edge-based	topological	strict
successful	436	436	436
killed	953	597	525
extendable	1073	356	263
total	2462	1389	1224
killed+extendable	2026	953	788
time (seconds)	501	187	148

Table 6.8: Experimental runs with `PiSim` using three different types of node orderings.

It is significant that the topological node ordering does nearly as well as the strict node ordering in terms of efficiency, since it is based on an easy, automatable ordering heuristic. The reason that the two node orderings yield comparable results is that the rules are typically long and skinny so that the partial topological node orderings are nearly strict node orderings. The strict node orderings can be seen as topological node orderings that are improved using saliency information.

The strict node orderings that were used in the example runs above were assigned manually and were designed to place node types early in the ordering that are salient with respect to the input graph. The measure of saliency of a node type is based on the number of instances of that node type there are in the input graph; lower instance counts mean higher saliency. This takes into consideration non-terminal node type counts, so this assignment of strict node orderings relies on knowledge of the input graph and results of prior recognition runs. Below, we discuss ways of approximately measuring the saliency of non-terminal node types automatically.

Partial Versus Strict Node Orderings

There is no doubt that using partial node orderings is more expensive than using strict node orderings. However, using partial node orderings has advantages in terms of flexibility and tolerance when a cliché is not entirely recognizable. Since it allows more than one order in which to match right-hand side nodes, if a portion is missing, an order in which the other portion is matched first can still yield useful partial information. With a strict node ordering, only one order of matching is tried, so if a node is missing, all nodes following it in the strict ordering will be prevented from being matched.

In other words, partial node orderings allows partial recognition of right-hand sides of rules. This is a type of partial recognition which is different from the partial recognition of the input graph. (In the program recognition domain, this is partial recognition of clichés, as opposed to partial recognition of programs, as defined in Section 3.3.1.) To distinguish it from partial recognition of the input graph, we use the term *near-miss recognition.*

Near-miss recognition is useful in being able to try harder. Pure near-miss recognition – using no node ordering – generates maximally-sized partial analyses. These can give clues as to which *small* set of constraints must be relaxed, suspended, or satisfied (e.g., by changing the input graph) in order for some cliché to be recognized. This has applications both in debugging programs (in which a programmer meant to use a cliché but did so incorrectly) and in learning new clichés.

In general, with partial node orderings, the partial analyses can become larger and more plentiful than with strict node orderings. This reveals a trade-off between the efficiency of strict node orderings, which cut off analyses as soon as constraints are violated, and the near-miss recognition power afforded by partial node orderings, which explores more of the search space, "tolerating" constraint violations to gather more information about the input

graph.

To do near-miss recognition efficiently, the parser's search must be focused on a small number of non-terminals at a small number of places in the input graph. Grimson provided theoretical confirmation of this in his study of constrained search. The mapping between constrained search and right-hand side matching makes his results applicable to near-miss recognition by flow graph parsing as well.

Grimson found that constrained search is efficient when indexing and selection are perfect, as discussed in Section 6.2.5. However, an exponential amount of work is needed to tell that a possibly partially occluded object model is not in a scene, even when good (but not perfect) selection techniques are performed. So it is important that indexing techniques are used to narrow down the library of models, rather than sequentially searching through the library and using the exponential process to rule out incorrect models. Also, an exponential amount of work is needed to find an object model in a cluttered scene if adequate selection techniques are not used to distinguish the object from the noise. This is the case even if perfect indexing is done. So both good indexing and good selection are needed to efficiently perform recognition of partially occluded objects in cluttered scenes.

A few program recognition researchers, such as Johnson [65], Lukey [87], and Murray [95], have worked on the problem of guiding the recognition system to a "best" partial analysis in the context of program debugging applications. They use heuristics based on saliency, mnemonic names, and partial analysis size, for example. Section 6.4 gives some suggestions for ways of incorporating other possible indexing and selection techniques into the current recognition system.

Choosing a Node Ordering

The node ordering of a rule determines the order in which individual unary and binary constraints are applied. The best order is one in which stronger constraints are applied first. An automatic assignment of node orderings to rules can look at the structure of the rules' right-hand sides and at the input graph to get clues as to which ordering is most likely to impose stronger constraints earlier.

Unary Constraints

The unary node-type constraints are strongest for salient node types. So a node-ordering in which salient nodes are matched first is best. There are two useful notions of saliency. One notion is a node type that is rare in the input graph. The other is a node type that only appears in a few grammar rules.

The unary node-type constraint for nodes that are salient with respect to the *input graph* is strong in that they reduce the branching factor of item trees. Applying them early can help disambiguate partial analyses while they are still small. (Reduction of branching is most beneficial near the top of item trees, since binary constraints can usually keep the

branching factor down at lower levels.)

Ideally, node orderings that are based on saliency of node types with respect to the input graph should take into account the number of instances of non-terminal as well as terminal node types in the input graph. However, this requires knowledge of the results of recognition.

We can use heuristics to automatically produce node orderings that approximate this ideal assignment. Given a right-hand side, we can compute a frequency number for each right-hand side node. The nodes of a rule's right-hand side are then ordered from smallest to largest frequency of their node-type, so that salient nodes are earlier in the ordering. (This is not necessarily a strict node ordering.)

For each terminal, the frequency number is the number of nodes in the input graph with the same type. For a non-terminal A, take each rule R for A and recursively compute the frequency numbers of the nodes in R's right-hand side, choosing the minimum frequency number as the frequency of A with respect to R. Finally, combine these frequency numbers over all the rules for A to get A's frequency. The combination function (e.g., sum, max, average) chosen depends on how conservative or optimistic we want the heuristic to be.

The advantage of matching nodes that are salient with respect to the *grammar* first is that the growth of an item tree for a rule does not begin until the salient node is found. This has the effect of only activating the matching process for a particular rule when it is worth it (i.e., when the rule's right-hand side or a near-miss of it is likely to exist in the input graph). This is a form of *indexing.* It helps speed up recognition and it also produces better partial analyses for near-miss recognition.

An issue that arises when using saliency measures based on the grammar is that as the parsing proceeds, the grammar is changing. As the set of item trees is pruned away, the set of grammar rules under consideration is effectively becoming smaller. Since the saliency of a node-type is relative to the grammar, saliencies change as the grammar changes. Matching a node that is salient with respect to an entire grammar might narrow down the grammar to a few rules that contain that node. Then, with respect to these rules, there are other salient node types (which might not have been salient with respect to the entire grammar). These salient node types should be matched first, to disambiguate between the possibilities, and so on. The point is that saliency with respect to a grammar changes as the grammar changes, so if we are basing our node orderings on it, we will have to change the node orderings dynamically as parsing proceeds.

Binary Constraints

Node orderings can also be created to force strong binary constraints to be checked earlier. For example, the topological partial node ordering used in the experimental runs was effective in reducing complexity. It ensured that no node was matched until all nodes preceding it in the right-hand side flow graph had been matched. This meant that when a node is

matched, there are edge connection constraints applicable to it and its preceding nodes. The partial items are always extended by complete items for nodes that can be constrained the most by the preceding nodes.

Another ordering heuristic is to match nodes earlier that have more binary constraints applied to them. For example, match those with more output edges, before those with few outputs, or match those that are constrained to co-occur, before those that are not. The advantage of these heuristics is that they require no knowledge of the input graph.

6.2.8 Summary of Item Count

Recall from Section 6.1.2 that the overall cost of the parsing algorithm is

$$
\begin{aligned}
&|I_T| * (C_{agenda-add} + C_{agenda-retrieve} + C_{duplicate-test}) + \\
&|I_E| * C_{extend} + \\
&|I_C| * (C_{chart-add} + C_{combination-lookup}) + \\
&|I_R| * C_{instantiate-empty} + \\
&|I_n| * C_{instantiate-terminal} + \\
&|I_Z| * C_{zip-up} + \\
&|I_f| * (C_{constraints-check} + C_{zip-up-lookup})
\end{aligned}
$$

The number of items created during initialization for the terminal nodes of the input graph ($|I_n|$) is n, the number of nodes in the input graph. The number of empty partial items also created during initialization ($|I_R|$) is the number of rules in the grammar ($|P|$). This section has discussed the number of items created by extension and zip-up and how constraints and node orderings influence the size of these sets ($|I_E|$ and $|I_Z|$). The number of items in the chart is $I_C = (|I_E| - |I_D|) + n + |P|$, where I_D is the set of duplicate items. If strict node orderings are used, then $|I_D| = 0$. The set of complete items that enter the chart (I_f) are those in I_n and I_Z and the subset of the complete items created by extension that contains no duplicate items. The total number of items $|I_T| = |I_E| + n + |P| + |I_Z| = |I_C| + |I_D|$.

We now detail the costs of the actions that are performed on each of these types of items.

6.3 Component Costs

The sizes of the various types of item sets are weighted in the complexity formula by the costs of applying the basic parser actions to each type of item. The terms in the formula are ordered by the typical size of the set of items in the term, based on the empirical study of recognizing `CST` and `PISIM`. The first three terms are dominant. It is best for the costs weighting them to be small. We will consider the cost of each of the parser's actions in the order in which it appears in the complexity formula.

The cost of adding to and retrieving an item, $C_{agenda-add}$ and $C_{agenda-retrieve}$, are small constants in the current implementation. They are implemented as simple queue operations. In general, however, they may be more complex operations, depending on the type of structure imposed on the agenda to implement more complicated search strategies.

$C_{duplicate-test}$ is the cost of testing whether an item is a duplicate of an existing item already in the chart. There are two different tests used, depending on whether the item is partial or complete.

To describe the test of partial items, we need to define two more parts of the structure of items. One is a set of *sub-items* which are complete items that represent the recognition of the nodes that have been matched so far in the item rule's right-hand side. These are the items that have successively extended partial items to ultimately result in this item. The other new part of items is a set of *super-items* which are items that resulted from extending a partial item with this item. Only complete items have super-items. An item might have more than one super-item if a sub-derivation is being shared between two derivation trees. (Super-items and sub-items of an item I_1 are different than the item's parent or children in item-trees. Links to super- and sub-items encode the structure of the derivation graphs generated by the parser. The links to parent and children items in an item tree show the history of extensions performed on items for the same rule.)

Each *partial* item will have a sub-item for each of the nodes of its rule's right-hand side that have been matched so far. If a duplicate I_d of a partial item I_p exists, I_d will share all of its sub-items with I_p. So, given any partial item I_p, we can tell if a duplicate of it exists by taking any one of its sub-items I_s and looking for one of its super-items (other than I_p) that has the same set of sub-items matched to the same nodes as I_p. If none is found, the partial item is not a duplicate. The average cost is polynomial in the average number of super-items an item can have and the number of sub-items being compared (which is the size of the partial item being tested and which is less than the size of its rule's right-hand side). The average number of super-items is 2.84 in `CST` and 2.07 in `PISIM`. Right-hand side sizes range from 1 to 7 nodes.

To test whether a duplicate of a *complete* item I_c exists, we look in the chart for items with the same label as I_c at the location of I_c. For each location pointer in the input and output mappings of I_c, the items for I_c's label at that location pointer are retrieved. The sets of items retrieved for the location pointers are intersected. The average cost is polynomial in the average number of location pointers per input or output mapping (3.21 in `CST`, 2.92 in `PISIM`) and the average number of items retrieved (2.91 in `CST`, 2.61 in `PISIM`).

The number of location pointers in the mappings is not the same as the number of inputs and outputs of the left-hand side non-terminal of an item's rule or the number of internal edges to immediately needed non-terminals. It depends on the degree of fan-out or fan-in of edges in the input graph, and on the bushiness of nested location pointers which represent aggregation. (In terms of the program recognition application, the size of the nested location pointers representing aggregation depends on the complexity of the clichéd

data structure – how many parts it has and how many its sub-parts have, and so on.)

The cost of extension C_{extend} is the sum of the cost of

- copying an item: linear in the sizes of its parts, such as lists of callers and sub-items.
- updating input and output mappings: polynomial in the number of location pointers in the input and output mappings of the complete item.
- comparing location pointer tuples on the inputs and outputs of adjacent non-terminals and propagating st-thru matches: polynomial in the number of edges in the right-hand side and the number of location pointers per right-hand side edge. (There may be more than one location pointer on an edge due to fan-in or fan-out and aggregation.) The average number of edges in a right-hand side is 0.53 and the average number of location pointers per edge is 2.63 in `CST` and 4.16 in `PISIM`.

The cost of recording an item (complete or partial) in the chart, $C_{chart-add}$, is linear in the number of location pointers in the input and output mappings of the item. This is because the item is recorded in the chart multiple times, once for each location pointer. (For partial items, the "output mappings" are the sets of location pointers on the edges to immediately needed non-terminals.) The chart is broken into two parts, one containing only complete items and the other containing only partial items. The set of complete items is indexed on the label of the item and on the location pointers of the item's input and output mappings. The set of partial items is indexed on the location pointers and node types of the item's immediately needed non-terminals. This makes it easier to look up all complete items for a particular node type at a particular location (to combine with a given partial item), and to look up all partial items needing a particular node type at a particular location (to combine with a given complete item). The average number of times an item is entered into the chart is 7.51 in `CST` and 6.35 in `PISIM`.

$C_{combination-lookup}$ is the cost of looking up partial or complete items to combine with an item that is entering the chart. Given a complete item for a non-terminal A, looking up partial items for it to extend involves taking each location pointer in the mappings of the complete item and looking up all partial items that immediately need A at the location pointer. The candidate items retrieved are organized by item and for each candidate, a validity check is performed. The validity check is an application of unary and binary constraints. So, the cost of looking up partial items is a polynomial in the number of location pointers in the mappings, the number of candidate items retrieved, and the cost of applying the unary and binary constraints.

Given a partial item that immediately needs non-terminals $A_1, ..., A_n$, a similar cost is incurred in looking up complete items for each of these non-terminals. This cost is summed over the sets of location pointers on the edges going to each of the immediately needed non-terminals.

The cost of checking parse-interleaved constraints $C_{constraint-check}$ is hard to characterize, since the constraint expressions can be arbitrarily complex. However, in the current system, the constraints applied are very simple and this term contributes little.

The cost of looking up items to zip up with a given item I_A is $C_{zip-up-lookup}$. This involves looking up each item I_c for I_A's label A that satisfies the following conditions:

- either all of the edges pointed to by the location pointers in I_c's and I_A's input mappings share the same source ports or all of the edges pointed to by the location pointers in their output mappings share the same sink ports, or both,
- none of the input mappings of either item overlap (i.e., contain common location pointers) and neither do the output mappings, and
- the attribute values of the zipped up item's left-hand side are defined, according to the attribute combination function.

The cost of doing this is polynomial in the number of location pointers contained in the input and output mappings of I_A, in the number of items retrieved per location pointer, and in the cost of applying the attribute combination function.

The costs of creating empty partial items, $C_{instantiate-empty}$, and complete items for terminal nodes, $C_{instantiate-terminal}$, during instantiation are both small constants.

The cost of zipping up a set of items C_{zip-up} is polynomial in the number of items being zipped up (for the example programs, the typical number is 2 or 3) and in the cost of zipping up the parts of the items (e.g., unioning sets of callers).

6.4 Other Performance Improvements

This section contains suggestions for improving the performance of the parser. These are useful when constraints are not strong enough to prune the parser's search adequately. They are also important if the parser is to be used for near-miss recognition in the future. Most of these can benefit from advice from an external agent.

6.4.1 Decomposition

Parsing smaller flow graphs can be easier than parsing larger ones if the smaller flow graphs are less ambiguous. Decomposing an input graph and then focusing the parser only on sub-flow graphs within the decomposition boundaries can speed up recognition.

John Hartman [55] demonstrates the advantage of decomposition in program recognition. He provides an efficient recognition technique for clichéd control concepts, which hierarchically decomposes a program represented as a control flow graph into propers (single entry/single exit control flow sub-graphs) and performs simple graph matching within the propers.

This section gives some examples of program domain-specific heuristic decompositions that can be used to focus our parser. They are all static decompositions that occur before parsing is begun. Section 6.4.3 discusses dynamic decompositions.

Subroutinization provides one type of heuristic decomposition. The parser can be forced to recognize non-terminals only within the boundaries of a subroutine or module. (When using this heuristic, there is no need to "flatten" the program by expanding out all subroutines within their callers. When the flow graph for an entire subroutine body is recognized as a non-terminal A, all nodes representing calls of that subroutine can be replaced by a node of type A.)

An analogous decomposition can be made based on data structure organization. The idea is to require a non-terminal to be recognized only in sub-flow graphs whose nodes all represent operations that are acting on parts of the same user-defined data structure. For example, `1+` and `AREF` occur all over the input graph, but we should not pair them up as an instance of the Stack-Pop cliché if one is applied to the `Tail` part of a user-defined structure `Queue` and the other is applied to the `Instructions` part of a `Handler`. Since our clichés are primarily based on dataflow, this partitioning seems natural. A single dataflow slice is not always the best unit of decomposition, since aggregate data structures typically involve a bundle of slices. This partitioning allows a bundle of slices to be considered as a unit.

Both of these decompositions work best if the programmer's decomposition of the program into procedural and data abstractions is very close to a typical way programs in that domain are decomposed.

The main problem with focusing the parser on each partition independently is that completeness can be lost if clichés occur across the partition boundaries. A more flexible partitioning technique is to augment the extendibility criterion of the parser with a binary *partitioning constraint* which requires that a complete item can only extend a partial item if all of the partial item's sub-items and the complete item represent the recognition of sub-flow graphs in the same partition. Combination attempts that fail this constraint can be postponed, rather than eliminated altogether. This allows certain combinations to be preferred over others, while allowing less favorable combinations to still be tried in a try-harder phase.

The drawback with this scheme is that more combinations between pairs of items will be attempted. When parsing is focused on sub-flow graphs independently, the combinations that cross boundaries are not even attempted.

An advantage of incorporating a partitioning constraint into the extendibility criterion is that it can be selectively applied. It would be like any other match-interleaved constraint in that it can be specified on a rule-by-rule basis to apply to certain (not necessarily all) nodes of each rule's right-hand side. The match-interleaved co-occurrence constraint currently used by the parser can be seen as a partitioning constraint that requires certain right-hand side nodes to occur within the same control-environment boundary.

Finally, the recognition system can make use of advice from an external agent, that has

access to more information about the program than is found in the source code. People can often break up the program into pieces that "go together" in that they provide a particular functionality or belong to the same abstract domain-specific concept. They base this decomposition on design documentation and program comments or even just names of subroutines and variables. (As part of the DESIRE project [12, 13] Josiah Hoskins has proposed a neural-network-based approach to automating this process.) This information can be used to focus the recognition system on particular sub-flow graphs and also to suggest clichés to look for within them (i.e., index into the cliché library – see the next section).

6.4.2 Indexing

Efficiency can be gained not only by reducing the focus of the parser to smaller sub-flow graphs, but also by reducing its focus to a smaller subset of the grammar. For large grammars, it is advantageous for recognition to be *sub-linear* in the size of the grammar.

The current parser makes use of indexing to some extent in that it only creates (non-empty) items for rules when part of the rule's right-hand side has been found in the input graph. The chart's structure allows the parser to index on the node type found to retrieve partial items that immediately need it. Heuristics have been discussed in Section 6.2.7 for choosing a node ordering that will force salient nodes to be matched first. This stunts the growth of item trees until it is likely that a non-terminal instance or a near-miss of one exists in the input graph.

Advice can also be given to the program recognition system from an external agent, based on expectations about which clichés are likely to be found in the program. This can be used to narrow down the grammar given to the parser.

6.4.3 Interleaved Decomposition and Indexing

We can also interleave indexing and decomposition (selection) techniques with the parsing process. The idea is to use strict node orderings first and then try harder later by giving certain partial items partial node orderings, expanding their immediately needed nodes based on the new orderings, and returning them to the agenda to continue parsing. Advice from an expectation-driven component or heuristics can be used to choose the partial items to "encourage". An example heuristic might be to choose partial items that have started recognizing non-terminals in an area of the input graph in which no cliché has been fully recognized. Another heuristic is to choose the partial items that have the salient nodes of their right-hand side matched already.

Interleaved indexing and decomposition techniques have an advantage over static techniques that are applied before recognition in that they can make use of deeper knowledge about the input graph based on the previous recognition results.

Hierarchically representing patterns in a graph grammar facilitates this process. If a "flat" pattern were searched for, using a strict node ordering, the search would end as

soon as the parser fails to match the "next" node in the ordering. With a hierarchical organization, more parts of the pattern can be recognized and used to make a more informed decision about which candidate partial analyses should be pursued further with a partial node ordering. This information can also be used to decide which node ordering to try.

6.4.4 Avoiding Unnecessary Copying

When a partial item is extendable by a complete one, a copy of the partial item is created and the copy is extended. The reason is that this helps the parser deal with ambiguity and allows it to perform partial recognition and incremental analysis. (See Section 3.5.) However, sometimes a large number of the copies made are unnecessary, either because the input graph is not ambiguous, it does not contain multiple instances of some node types, or it is expected to remain static. This section suggests ways of avoiding unnecessary copying.

We can identify unnecessary copies retrospectively by looking for partial items that have been extended with only one complete item for the same immediately needed node. In the `CST` example (using strict node orderings), the percentage of copies that were unnecessary is 13.5%. The percentage of the total number of items that are the results of unnecessary copies is 10.9%. In the `PISIM` example (using strict node orderings), the percentage of copies that were unnecessary is 14.7%. The number of items that are the result of an unnecessary copy as a percentage of the total number of items is 11.6%.

Unnecessary copies contribute to both the height and width of item trees. When strict node orderings are used, they contribute only to the height of trees.

The following are a few techniques for avoiding copying.

1. *Lazy copying*: Make a copy only when it is necessary. Extend partial items with complete items without copying. However, when an alternative complete item arises for an already matched node A in some item I_0, make a copy, I_1, of I_0 and restore it to the state I_0 was in before the old complete item I_{A1} was used to extend it. To do this, we remove any links it has to super-items (since only complete items can have super-items). We must also find out which sub-items of I_1 must be retracted. These are I_{A1} and all complete items that extended it after I_{A1}, which can be computed from the node ordering and a history of the immediately needed sets. These are removed from I_1's set of sub-items and all information associated with I_1 that was derived from them is removed. (This requires keeping track of dependencies of parts of an item on the sub-item parts, such as its inputs and outputs. It also requires allowing partial items to be indexed based on already matched nodes as well as immediately-needed nodes, so that new complete items can be paired up with them.) Once the retraction is finished, I_1 can be extended with the alternative complete item.

   This scheme is only worthwhile when the majority of copying is unnecessary. It can be applied selectively to certain extensions if the parser has been given advice

that certain node-types are not likely to be found more than once or in a partially ambiguous situation.

2. *Structure-sharing*: A common technique to avoid copying when there is little change between the original and the copy is to share the common structure. The parser can store one "original item" per rule plus a log of augmentations, representing the successive extensions. This is a more compact way to record intermediate states in the search. This technique is used in resolution theorem proving [14] and in unification-based grammar parsing [67, 104].

3. *Estimating Number of Instances*: We can heuristically count the maximum possible number of instances of a particular node type, based on the node type distribution of the input graph. As soon as the maximum number of instances of a node-type A are entered in the chart, if a partial item immediately needing A arises, the parser can tell whether there is more than one possible complete item for A that can extend it. If there is only one, then the partial item need not be copied before being extended. However, this scheme is only beneficial if the heuristic for counting instances is good[4] and most of the partial items that need a node-type A enter the chart after the maximum number of instances of A have been found. An alternative is to use a less conservative heuristic that computes a lower bound on the number of instances in conjunction with lazy copying. This allows copying to be prevented earlier, without sacrificing safety.

4. *Restricted Control Strategy*: The parser can be forced to produce all complete items for node-types of a particular height h in the grammar before going up to the next height $h+1$, starting with the terminal node types ($h = 0$). This guarantees that all instances of a node-type A have been found when a partial item immediately needing A enters the chart. The partial item need not be copied before being extended if only one complete item for A can extend it. The disadvantage is that the control of the parser is severely restricted.

The decision and technique used to avoid copying depends on the severity of the problem of unnecessary copying. In the two example programs, it is not severe enough to merit the overhead of these techniques.

6.5 Conclusion

This section has shown the following.

- Although flow graph parsing is exponential in the worst case, it is feasible to apply it to practical partial program recognition. Structural (node-type and edge connection)

[4] Perfectly counting the number of instances of a node-type is no easier than recognition itself.

constraints as well as program domain-specific constraints (e.g., co-occurrence) are able to control the complexity in practice.

- The type of node ordering imposed on the right-hand side nodes of rules affects the parser's efficiency. Strict node orderings focus the search, generating fewer partial analyses and duplicate items than partial node orderings. This reveals a trade-off between efficiency and recognition power. The choice of how to order nodes within a strict or partial node ordering also affects performance. This choice can be made with the help of external advice or heuristics. It may need to dynamically change as parsing proceeds.

- The capability of generating maximally-sized partial recognitions of clichés (i.e., near-miss recognition) is expensive. Future near-miss recognition capabilities must take advantage of advice and automated techniques for indexing and decomposition to be feasible. These techniques can be interleaved profitably with recognition, rather than being performed statically beforehand.

Chapter 7

Conclusions

We have developed and studied a graph parsing approach to program recognition in which programs are represented as attributed flow graphs and the clichéd library is encoded as an attributed graph grammar. Graph parsing is used to recognize clichés in the code. We have demonstrated that this graph parsing approach is a feasible and useful way to automate program recognition.

The approach has two key features. One is the representation shift it employs. The other is its exhaustive, systematic, but flexible control strategy. The graph representation is able to suppress many common forms of program variation which hinder recognition. This enables our recognition approach to be robust under syntactic, organizational, and implementational variation, as well as variation due to delocalization, unfamiliar code, and common function-sharing optimizations. Difficulties arise when a program's data and control flow are implicit or derived or cannot be determined statically.

The flow graph formalism is able to concisely encode algorithmic and data aggregation clichés whose constraints are primarily based on data and control flow. These include not only general-purpose programming clichés, but also clichés specific to the simulation domain. Limitations arise in capturing loosely constrained clichés. Although the flow graph formalism allows us to encode clichés on a high level of abstraction, the level of abstraction is still limited by the amount of detail that must be specified about the clichés (e.g., operation types and arity, dataflow connections, control environment relationships).

In studying the graph parsing approach, we have experimented with two real-world simulator programs. We empirically and analytically studied the computational cost of our recognition system with respect to these programs. We have found that although our graph parsing algorithm is exponential in the worst case, its complexity is reduced in its practical application to program recognition. Structural (node-type and edge connection) constraints as well as constraints which are specific to the program recognition application (e.g., co-occurrence) improve the parser's performance in practice. Section 7.1 discusses the need for more empirical study.

Section 7.2 discusses some open research issues that have not yet been fully explored.

An important future goal is to complement our code-driven technique with an expectation-driven technique that provides guidance based on such knowledge as the program's goals, problem domain, and documentation. With its flexibility, our recognition architecture forms a seed for this future hybrid program understanding system. It can make use of advice and guidance from external agents. In Section 7.2.5, we summarize our observations of typical forms of advice that would be helpful to our recognition system in controlling its complexity and its search for clichés.

Section 7.3 gives a comparative summary of related work in program recognition. Finally, in Section 7.4, we briefly discuss applications of program recognition and of our parsing formalism in general.

7.1 Empirical Studies

Our study is a step toward understanding a particular recognition technique in the context of real-world programs. It tries to break out of the "toy" program rut. Our example programs are medium-sized and not written by us. They start to give some indication of what is typical in terms of characteristics of real-world programs. They contain domain-specific clichés as well as general utility clichés. They also contain unfamiliar code. This allows us to study the ability of our parsing-based technique to perform various types of partial recognition.

However, it is important to keep the findings of our empirical studies with just two programs in perspective. We have made some general observations that we expect to be true of programs and libraries other than those studied here. For example, we point out general classes of variation that are handled, which types of constraints are effective in improving performance, and situations in which partial recognition can occur. On the other hand, we have also made specific observations about recognizing these programs using the current library. For example, we observed that recognition by graph parsing can be done efficiently in practice. We also discuss weaknesses of our representation and approach, but only those that we encountered in our study. This is not a complete list. These are interesting only if these programs and the library are typical.

Our example programs are still small, relative to real-world programs in the software industry. There are bound to be issues of scaling up to large programs that have not yet been encountered. More empirical studies are needed to:

- expand and refine the cliché library,
- identify more classes of variation that can or cannot be tolerated,
- determine how severe and common the limitations are that we have pointed out,
- identify other factors that affect efficiency,
- determine if our experiences with good performance were lucky or typical and,

- evaluate the ability of the existing system to recognize new programs.

7.2 Future

This section discusses areas in which additional research is needed.

7.2.1 Multiple Recursion

Currently, GRASPR can represent and recognize singly-recursive programs. In the future, we will extend its attribute language to capture the control flow information of multiply recursive programs as well. This involves a straightforward generalization of recursion information triples to hold more than one feedback-ce – one for each recursive call. To express constraints on the control environment attributes of these programs, we will need new ways of referring to particular feedback-ces. We can no longer refer simply to the "feedback-ce in the innermost recursion" containing a particular operation or test. We may need to identify common forms of multiple recursions, such as the familiar binary tree recursion, in which the feedback-ces are related in standard ways. Then individual feedback-ces can be referred to, based on their relationship to others in the multiple recursion.

In addition, more research is needed to extend the temporal abstraction techniques to abstract multiply recursive programs. There may be some common types of multiple recursion for which temporal abstraction is a straightforward generalization of the techniques for singly recursive programs. For example, Rich [110] (Section 9.4) briefly discusses temporal abstraction of binary tree recursions. In these programs, the feedback-ces are the same control environment. Other programs seem not to be amenable to temporal abstraction, such as those in which one feedback-ce is $\sqsubseteq$ the other. (This arises when two or more functions are mutually recursive and one calls itself, as in the familiar Evaluate/Apply recursion.)

Because the current implementation of GRASPR is not able to translate multiply-recursive programs into meaningful attributed flow graphs, we selectively flattened the Evaluate/Apply recursion within PiSim to avoid generating more than one recursive call. During the translation of the program to a plan, we specifically advised that the box representing the call to the function Evaluate not be expanded into a flow graph representing the function's body. The resulting flow graph contained only one recursive call, (in the iterative mapping of Evaluate over a list of Arguments to which an operation is to be applied). The function Evaluate in PiSim corresponds to what we would like to recognize as the "Evaluate" cliché.

7.2.2 Interfacing with Other Recognition Techniques

Recall from Section 5.2.3 that we had difficulty encoding the Evaluate cliché, due to its loose constraints on data and control flow. Suppose that we not only advise GRASPR not to expand the node representing the call to Evaluate, but we also specify that it is an instance of the "Evaluate" cliché. (Normally when a user specifies that a function is not to be

expanded whose name happens to be a non-terminal in the grammar, GRASPR systematically renames the function. We specify that the function is an instance of the "Evaluate" cliché by overriding this renaming and labeling the node "Evaluate.")

This can be seen as a way to use results from another recognition technique (in this case, performed by people), which applies more flexible constraints and can recognize the body of `Evaluate` as the "Evaluate" cliché. In other words, GRASPR uses results from another recognition technique in the form of an already reduced non-terminal "Evaluate" which the other technique inserted into the flow graph representing the program.

An alternative way for GRASPR to use recognition results from other techniques is for these techniques to create items representing the recognition results and add them directly to GRASPR's parser agenda. For example, rather than directly relabeling the node representing the call to `Evaluate`, a complete item can be created for the "Evaluate" non-terminal and added to the parser's agenda. This has the advantage that the program is not destructively modified by the insertion of the already-reduced non-terminal.

7.2.3 Disambiguating Data Structure Operation Instances

GRASPR has been designed to exhaustively and algorithmically recognize all clichés in a program. It does not employ global consistency checks to rule out some analyses or to disambiguate multiple views of the same part of a program. Its recognition process is "monotonic" in that new recognitions cannot invalidate previously recognized structures. Recognition of one cliché does not depend on the failure to recognize another cliché.

There are two main reasons for this. One is that the code-driven parsing approach is not best suited to perform the disambiguation of multiple views or global consistency checks. These should be done by a higher-level control mechanism that has access to information other than the program's data and control flow. It may have expectations about which interpretations are most likely. Also, the parsing approach does relatively local constraint checking. All consistency checks and disambiguation refer to individual instances of clichés that are parts of some larger cliché. A higher level mechanism can quantify over cliché instances that are not explicitly related by being part of some larger cliché.

The second reason that GRASPR generates multiple, possibly ambiguous analyses is that sometimes multiple views are useful in understanding a program. A higher-level control mechanism may require different views at different times, depending on how the recognition results are being used.

The interaction between GRASPR and a higher-level control mechanism would be particularly profitable in the recognition of aggregate data clichés. Data clichés are recognized by recognizing operations on them. These operations form groups, called "suites," each of which represents a globally consistent set of operations with respect to some data structure. For example, Figure 7-1 shows four different consistent pairs of operations for inserting and extracting elements from an indexed sequence. Each of these represent valid operations to

be used together in implementing a stack, since they maintain stack discipline. Each pair is a suite.

When GRASPR recognizes an individual clichéd data structure operation, it reports the recognition of the operation and the data cliché. Some of these may be locally ambiguous. For example, `zerop` and `null` can be empty tests for a variety of clichéd data structures. Also, some recognitions might not be globally consistent with the recognition of other operations on the same data elsewhere in the program. For example, recognizing one operation from a suite in Figure 7-1 does not necessarily mean a Stack is being used in the program. Another access or update to this same aggregate data structure elsewhere in the program might use an operation from another suite.

GRASPR does not attempt to disambiguate recognitions of data structure operations. Nor does it globally check that the data that has been recognized as the data cliché is always operated upon by operations in the same suite. The main reason is that GRASPR is not the one best suited for this task.

It is difficult to do these things in the flow graph parsing framework, based only on the data and control flow of the program. This is because instances of operations that act on the same aggregations of data are often difficult to group together, in order to apply consistency constraints (i.e., check that they are all in the same suite). As we discussed earlier, data and control flow cannot always be completely determined or made explicit. So, the operations are not always connected directly by dataflow. It may be possible to uncover direct dataflow in some cases (e.g., implicit aggregation might be made explicit). However, often aggregate data structures are collected in primitive data structures (e.g., lists or arrays) which do not represent implicit aggregations. (For example, `PiSim`'s `*Event-Queue*` is a homogeneous list of `Events`.) For these, the connections between operations on the aggregate structures must be derived.

In addition, negative constraints, such as that no other operations beside those in some suite act on certain pieces of data, are difficult to check in our recognition framework. This is particularly true when parts of the program are not available for analysis. For example, in `PiSim`, the function `Next-Instruction` takes a user-defined data structure `Task` (which corresponds to the `EXECUTION-CONTEXT` data cliché) and fetches an `INSTRUCTION` from an array of `INSTRUCTIONs` nested within the `Task` data structure. The function uses the current integer value of the `Task`'s "IP" part (which stands for "Instruction-Pointer") to index into the array. It then increments the "IP" part. GRASPR recognizes this function as a "Stack-Pop." However, in the machine operation simulation functions, which are given as input to `PiSim`, the "IP" part of a `Task` is sometimes updated to an arbitrary value (in the code for simulating branching operations), rather than being incremented or decremented.

Disambiguation and preferring recognitions may be done more easily by a higher-level control mechanism which has access to other information about the program. For example, user-defined part names provide a powerful clue to which structures an operation is acting upon. It is often the case that the operations acting on data that was selected using the

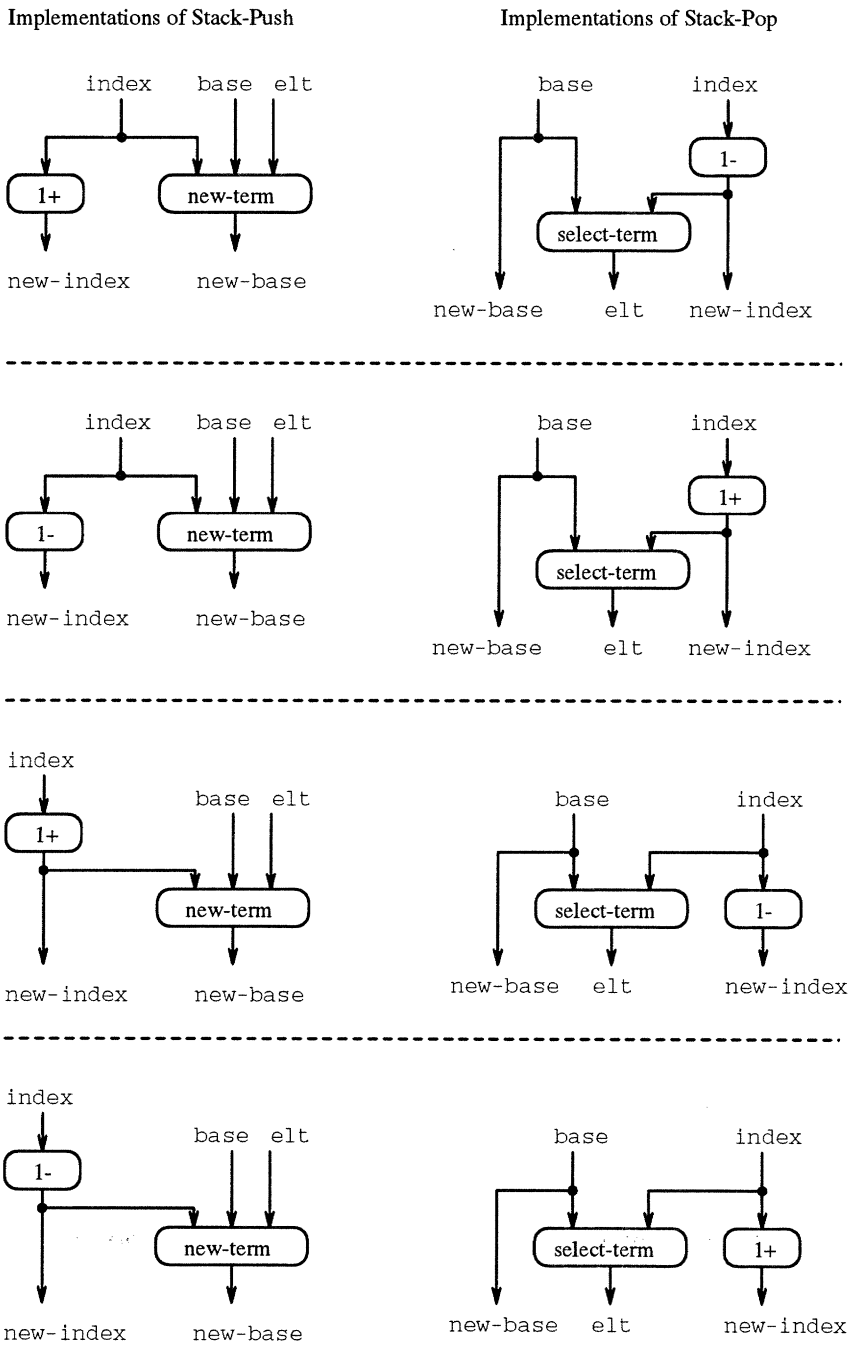


Figure 7-1: Four ways of implementing Stack-Push and Stack-Pop with the Stack implemented as an Indexed-Sequence.

same set of part names or generating data that's always stored in the same set of part names, are the only ones used to access or change those parts. Mnemonic variable names (including synonyms) and stylistic conventions (e.g., module decomposition) can also be a good source of expectations about how operations should be grouped. This information must be used heuristically and non-monotonically. (Section 4.2.3 discusses an initial attempt to map user-defined data structure and part names to clichéd structure names. However, these mappings are not always complete or unambiguous.)

When portions of a program are not available for analysis, there may be other information available about the interface between the unavailable code and the rest of the program, such as which functions of the program are called and which new data structures are created. This information can be used, for example, to determine that the "IP" part of a `Task` is not always updated using increment or decrement, but can be given an arbitrary integer value. The recognition process can be seen as giving as output the clichés recognized and a set of assumptions or invariants on which the recognition of those clichés is dependent.

7.2.4 Side Effects to Mutable Data Structures

We studied the recognition of aggregate data structures, independent of issues concerning side effects to mutable data structures. In order to do this, we manually translated our example programs to pure (functional) versions and recognized pure clichés in them. Fortunately, the translation was straightforward and much of it may be automatable.

An open problem for the future is dealing with programs that contain mutable data structures and destructive operations on them. The problem is modeling the dataflow correctly in representing our programs as dataflow graphs. This is complicated, of course, by aliasing. While we will not be able to automatically resolve all aliasing, it seems possible to use recognition to uncover common, stereotypical aliasing patterns. Complex aliasing patterns are not the norm [126, 127].

If recognition is interleaved with dataflow analysis, aliasing patterns might be recognized and used to help correctly translate a destructive operation into its non-destructive version.

There are two main classes of mutations to mutable data structures:

1. mutations to fixed, named parts (e.g., `(setf (queue-head queue) new-head)`).

2. mutations to a "derived" part (e.g., searching through a list for an element with some property or satisfying some predicate and then deleting that element).

When a change is made to a *fixed, named part* of a data structure, this destructive assignment should be replaced with non-destructive code which creates a new data structure containing the new value for the part and the old values for the rest of the parts. It must also recursively create new versions of the data structures within which this data structure is nested. For example, consider the following destructive operation which updates the `Time` part of a `Node` data structure, which is the value of the `Node` part of a given `Task`.

```
(defun Set-Time-Of (Task New-Time)
  (setf (Node-Time (Task-Node Task))
        New-Time))
```

The following non-destructive translation of this operation creates a copy of the `Task`'s `Node`, but giving the `Time` part the `New-Time`. It also creates a copy of the `Task`, with the new `Node` as its `Node` part. It also returns the new, updated structures so that the callers of `Set-Time-Of` can use them.

```
(defun Set-Time-Of (Task New-Time)
  (let ((Task-Node (Task-Node Task)))
    (setq Task-Node (Make-Node :Time New-Time
                               :ID (Node-ID Task-Node)
                               :Segments (Node-Segments Task-Node)
                               :Nodals (Node-Nodals Task-Node)))
    (values New-Time
            Task-Node
            (Make-Task :Handler (Task-Handler Task)
                       :Node Task-Node
                       :Segment (Task-Segment Task)
                       :IP (Task-IP Task)
                       :Status (Task-Status Task)))))
```

For nesting of fixed, named parts, it may be possible for the symbolic evaluator to keep track of how the structures are nested. The symbolic evaluator can treat the variables bound to data structures as bound to sets of "part variables," which are bound either to regular values or to other data structures (i.e., sets of part variables). When a part is modified, the part variables are traced backward to see what other objects are modified.

Aliasing is harder to uncover when mutations are made to *derived parts* because it's harder to prove that the part changed is the same as the part pointed to by something else. (In other words, the "nesting" relationships are derived.) However, these types of side effects usually occur in clichéd operations, such as searching through a list and modifying the element found or changing all elements of an array. If we heuristically (and nonmonotonically) assume that the aliasing pattern is localized and standard, we can transform the clichéd side effecting operation to the functional version.

For example, a common aliasing pattern occurs in splicing an element into a recursive data structure, such as a list. An example is in the following function which is used in PiSim to enqueue events on an event queue (which is a priority-queue).

```
(defun Insert-Event (New-Event Event-Queue)
  (if (or (null (cdr Event-Queue))
          (< (Event-Time New-Event)
             (Event-Time (second Event-Queue))))
      ;; push New-Event on (cdr Event-Queue)
```

```
(rplacd Event-Queue
        (cons New-Event (cdr Event-Queue)))
(Insert-Event New-Event (cdr Event-Queue))))
```

In this splice-in operation, the program "cdrs-down" the list `Event-Queue` until it finds a spot to insert the element `New-Event`. Then the new element is spliced in by destructively modifying the `cdr` of the current list. However, the current list is not only pointed to by the variable holding the current list, but also by the cons cell at the end of the sub-list already passed. This aliasing pattern is simple and localized within the recursive data structure and the variables used in the splice-in program. It is very common in our example programs.

Suppose `GRASPR` recognized the pattern of `cdr`-ing down a list and replacing the `cdr` (using `rplacd`) of the current list with a new list consisting of the new element followed by the old `cdr` of the current list. Then it may be possible to replace this pattern with the following non-destructive version in which the side effect is propagated up to the top of the data structure.

```
(defun Insert-Event (New-Event Event-Queue)
  (if (or (null (cdr Event-Queue))
          (< (Event-Time New-Event)
             (Event-Time (second Event-Queue))))
      (cons (car Event-Queue)
            (cons New-Event (cdr Event-Queue)))
      (cons (car Event-Queue)
            (Insert-Event New-Event (cdr Event-Queue)))))
```

In particular, the tail-recursive destructive program is replaced with a recursive non-destructive program and the list is `cdr`'d down as usual, but the elements passed on the way are remembered in the stack of recursive calls and are used to create a copy of the front of the list on the way back out of the recursion.

Another common type of aliasing involves pooling structures which contain all existing instances of some type of data structure. For example, the array `*Nodes*` contains all `NODE` structures. When a part "Time" of `NODE` is modified, this mutation should be replaced with non-destructive code that not only creates a new `NODE`, with the new value for the part "Time," but also creates a new `*Nodes*` array, with the new `NODE` in place of the old.

This update of the pooling structure requires knowing the inverse translation of an object to its pooling structure. This can be difficult to compute. However, we found that in our example programs, all of the objects contained in pooling structures had a part, such as an "ID" number or a "Tag" symbol, that held an index into the pooling structure. A useful form of advice is an identification of all pooling structures in the program (which is usually easy for a person to provide, based on mnemonic variable names and documentation) and an inverse mapping (if any) from the objects pooled to the pooling structure. As was suggested for dealing with variation due to handles, `GRASPR` can elicit advice about pooling structures by recognizing question-triggering patterns. (See Section 5.2.1.)

7.2.5 Advising GRASPR

We have presented a recognition architecture that has a flexible control structure in that it can accept advice to help control its complexity and to guide its search for recognitions. This advice can be given in a data-directed way, as opposed to modifying the parsing algorithm to build heuristics into the system. There are a variety of "control knobs" and parameters that are available to provide GRASPR with guidance.

- *Strict versus partial node orderings:* One form of advice that can be given to control the computational complexity of the recognition system is a specification of the type of node ordering that should be imposed on the right-hand side nodes of grammar rules. Strict node orderings are cheaper, since they generate fewer partial and duplicate items. However, partial node orderings provide more near-miss information, which is important in dealing with buggy programs and in eliciting more advice.

- *Node orderings:* Another form of advice is the choice of how to order nodes within a strict or partial node ordering. These can affect the order in which constraints are imposed, so that stronger constraints are imposed early. (For example, requiring salient nodes to be matched first imposes strong disambiguation constraints early.)

- *Selection of items from agenda:* Procedures can be provided which decide which items to pull from the current agenda and process. This is one way to control GRASPR's search strategy. For example, certain partial items might be pulled from the agenda, based on which part of the input program they have started to match or based on how much of their right-hand sides they have matched already.

- *Additional monitors:* Special-purpose monitors can be defined to watch the chart for particular types of items to enter. Additionally, rules for question-triggering patterns can be included in the grammar along with the rules for clichés. Monitors can watch for these patterns and then interact with outside agents. Monitors can also be defined to watch for opportunities to "try-harder" by generating alternative views or by weakening some constraints that make an analysis fail. The recursion folding monitor described in Section 4.2.2 is an example of monitoring for items that are failing certain constraints, but which might be made to complete by forcing certain constraints to be satisfied. The tasks set up by chart monitors can be prioritized so that those that are expensive or less likely to be effective can be postponed while quick, promising tasks are accomplished first.

- *Indexing partial analyses:* In addition to indexing into the chart to retrieve successful recognitions, it is possible to index into the chart to retrieve partial analyses that fail certain types of constraints. It is also possible to find out approximately how far the recognition of some cliché has gotten. GRASPR does this by taking the non-terminal representing the cliché and enumerating, in breadth-first fashion, the non-

terminals that this non-terminal is built upon in the grammar. For each non-terminal, it looks up all successful and failed recognitions of the non-terminal in the flow graph representing the program. It cuts off the breadth-first traversal whenever a successful or failed item is found for a non-terminal. These are collected and given as output. In other words, this finds the highest roots of the possible sub-derivation trees that can build up to the recognition of the cliché's non-terminal. This currently does not use any information about the location of the recognized non-terminals. It is best for high-level clichés whose parts occur infrequently in the input flow graph. Failed items contain information about which constraints they failed to satisfy. This is useful in determining what can be done to push the recognition through.

- *Partitioning constraints:* Section 6.4.1 described various heuristics for decomposing a program into partitions which can be used to focus the parser. This information can be used by augmenting the extendibility criterion with a binary partitioning constraint. This requires that a pair of complete and partial items that are candidates for combination represent the recognition of sub-flow graphs within the same partition. Combination attempts that fail this constraint can be postponed, rather than eliminated altogether. This allows certain combinations to be preferred over others, while allowing less favorable combinations to be available in a later try-harder phase. The advantage is that completeness will not be lost due to heuristic partitioning. Also, the partitioning constraint can be selectively applied on a rule-by-rule basis and to particular pairs of nodes in a rule's right-hand side.

While GRASPR has flexible control capabilities, the control knobs and parameters listed above form its current interface for accepting advice. More work is needed to develop a higher-level interface between GRASPR and the other agents it will interact with in the future hybrid system.

Other forms of advice that are useful to GRASPR include indications of which structures in the program are pooling structures (for side effect analysis, and uncovering the use of handles), and pointing out when implicit aggregation and manual abstraction are being used. These might be elicited during recognition (based on question-triggering patterns) or they might be given as machine-readable comments.

For GRASPR to intelligently ask questions of a user (e.g., based on recognizing question-triggering patterns), it must be able to refer to parts of the source text. When GRASPR represents programs as attributed flow graphs, it suppresses a great deal of detail. Although the information is still around in annotations, GRASPR currently has only limited facilities for efficiently mapping from one representation to another. (For example, it associates sets of variables to dataflow edges. It can also recreate small expressions in the program.)

Additionally, GRASPR is expected to interact with other reasoning components in the future, which will perform such things as conditional simplifications, reasoning about dataflow equalities, and data structure operation disambiguation and consistency checking. Multiple

representations of the program (including source text) will need to be maintained for GRASPR to interface with these other components.

Additional Code-Based Information Sources

Aside from eliciting advice from an external agent, some additional information can be gleaned from the leftover non-clichéd parts of the program, particularly in the program's error checking and its initialization procedures.

Error Conditions. Non-local exits are currently ignored. (The non-local control flow they represent is not modeled.) However, error conditions could be a useful form of machine-readable comment. They often give part of the specification for the program. For example, when a Handler is invoked for a message and a list of arguments, PiSim checks whether exactly the right number of arguments were given to the handler:

```
(when (not (= (Handler-Arity Handler) (length Arguments)))
  (error "PiSim error: arity mismatch")).
```

If a cliché is being looked for that has `(length Arguments)` as a subcomputation, but the program uses `(Handler-Arity Handler)` instead, then we can use the assertion from the error condition to push the recognition through.

A key advantage of error conditions is that they are easier to process and more up-to-date than textual comments.

Initialization. GRASPR normally does not recognize computations for program initialization or reading in input, since these are usually non-standard. They vary with the way the data is organized. However, we can extract information from this non-standard code about how data structures are organized. For example, the following code for Clear-Nodes tells how the parts of a Node interact. The part Nodals of a node is a key into the node's Segments part, which is a hash table. The elements of this hash table are Segment data structures, whose Data parts are arrays.

```
(defun Clear-Nodes ()
  (loop for Node being the array-elements of *Nodes*
        for Nodals-ID = (Node-Nodals Node)
        for Nodals = (Hash-Lookup (Node-Segments Node) Nodals-ID)
        doing (setf (Node-Time Node) 0)
        doing (Clear-Hash-Table (Node-Segments Node))
        doing (Hash-Insert (Node-Segments Node) Nodals-ID Nodals)
        doing (loop with Data = (Segment-Data Nodals)
                    for Index from 0 below (array-total-size Data)
                    doing (setf (aref Data Index) 'Unbound))))
```

7.3 Related Work

We can contrast our work on program recognition with that of other researchers along several lines. This section focuses mainly on the distinctions between the program and cliché representations and the recognition techniques used. Both affect how well the recognition systems can deal with variation, allow partial recognition, and fit into a hybrid system.

Our work is also distinguished from other program recognition research in that we analyze our approach, both empirically and analytically. Much of the early work in program recognition provides no analysis of the representations or techniques used. Some of the more recent research includes some empirical analysis of techniques. They typically study the accuracy of recognition and the recognition rates over sets of programs (usually student programs in program tutoring applications) [65, 95]. However, with the exception of Hartman's work [55], discussions of limitations have focused mainly on practical implementational limitations, rather than on general limitations of the approach. They also do not describe how additional information or guidance can help.

Our recognition work can also be compared to other work along the lines of the types of programs and clichés recognized. Our recognition system is able to recognize structured programs and clichés containing conditionals, loops with any number of exits, recursion, aggregate data structures, and simple side effects due to assignments. This allows GRASPR to recognize larger programs than existing recognition systems. It also enables encoding and recognition of domain-specific clichés as well as general-purpose ones, since many domain-specific clichés are aggregate data structure clichés. With the exception of CPU [84], existing recognition systems cannot handle aggregate data structure clichés and a majority do not handle recursion. Talus [95] heuristically handles some side effects to lists and arrays. The largest program recognized by any existing recognition system is a 300-line database program recognized by CPU. All other systems work with programs on the order of tens of lines. None deal with domain-specific clichés, except Laubsch's system [81, 82]. Hartman's UNPROG [55] is the only system that has demonstrated recognition of unstructured programs.

Our earlier work on the "Recognizer" [118, 144, 145] is typical of previous approaches to automating program recognition. It recognized small, contrived example programs, on the order of tens of lines. Its cliché library consisted exclusively of general-purpose, utility clichés. The Recognizer could deal with programs containing conditionals, loops, but not regular (non-tail) recursion or data aggregation. Like GRASPR, it used a dataflow graph representation for programs and clichés, but it employed a rigid control strategy. (It was based on a subgraph parsing algorithm that evolved from Brotsky's algorithm. See Section 3.5.) The development of the Recognizer was a feasibility study to demonstrate that graph parsing can be used to automate recognition, remove many types of variation, and create a useful description of a program. Our current work moves beyond studying feasibility by analyzing computational costs, studying GRASPR's tolerance (or vulnerability) to various

types of variation, identifying limits in graph grammar expressiveness for programming clichés, and studying how GRASPR can fit into a hybrid understanding system. GRASPR moves into the next level of maturity of recognition systems.

7.3.1 Representation

Johnson's PROUST [65], Ruth's system [122], Lukey's PUDSY [87], Looi's APROPOS2 [85] and Allemang's DUDU [4, 5] operate directly on the program text. This limits the variability and complexity of the structures that can be recognized, because these systems must wrestle directly with syntactic variations, performing source-to-source transformations to twist the code into a recognizable form. Most of these systems' effort is expended trying to canonicalize the syntax of the program, rather than concentrating on its semantic content. In addition, diffuse clichés pose a serious problem.

Because the types of patterns searched for in these systems are sets of statements, they limit the types of programs in which they can be found. In PUDSY, the group of statements matching a pattern must be contiguous, not scattered throughout the code. Ruth's system translates programs into a Lisp-like *model* language consisting of a small set of primitive operations. This representation abstracts away information about which particular binding and control constructs were used. However, it assumes program statements are totally ordered (by control flow as well as dataflow), rather than partially ordered (by data dependencies only). This prevents the system from recognizing that two programs that differ only in the order of execution of two independent statements are the same modulo this difference.

PROUST uses *plan-difference* rules to account for mismatches between the clichés (which Johnson calls "plans") it is looking for and the actual text of the program. These may allow the code to be transformed into an equivalent syntactic variation of the code or they may trigger the identification of a bug as being one listed in its bug catalog. Thus, allowable variations in code are limited to those accounted for by plan-difference rules. To be flexible and powerful, PROUST must have a large knowledge base of these rules. The number of rules could be reduced, however, if a more abstract representation for programs were used, or if the semantic equivalence of the mismatched code with the cliché could be confirmed using a theorem prover [95] or symbolic evaluation [87].

Allemang's DUDU (which stands for Debugging Using Device Understanding) [4, 5] attaches information about a program's functional semantics to its representation. DUDU's representation of clichés extends Johnson's text-based plan representation [65] to include not only goals and components for achieving them, but also causal links to show how the components achieve the goals. For example, an iterative cliché would be represented as a program template of statements with assertions that the loop invariants hold after initialization, after each iteration, and when the loop terminates, as well as assertions that the terminating conditions hold when the loop terminates.

The functional representation specifies which parts of a clichéd program's proof of correctness are supported by which parts of its plan representation. (Allemang uses the functional representation language of Sembugamoorthy and Chandrasekaran [125].) A key benefit gained by this representation is that it provides useful information that can make it easier to tolerate variation in how a function is achieved. Because it explicitly describes the purpose or function of each part of a cliché in the context of a larger proof of correctness, if some part of the cliché does not match the program, the functional representation describes the function of that part. It may then be possible to prove that the mismatched portion of the program still achieves this function. How much variation can be tolerated depends on the generality of the associate proof (e.g., how generally are the loop invariants and terminating conditions expressed).

Reasoning about functional semantics in this way requires that the recognition system know the intended function or purpose of a program. Like Proust, DUDU was developed in the context of debugging student programs, where this information is readily available. However, for purely code-driven recognition (as is usually required in maintenance situations), near-miss recognition of clichés must first be performed. This can be used to help generate expectations about which subset of clichés to try harder to recognize by proving that the functions of their unrecognized parts are still being achieved. However, this requires overcoming the expense of near-miss recognition (see Section 6.2.7) and defining preferences among near-misses.

One drawback of Allemang's representation is that it is limited by its text-based representation of clichés and programs. Since it directly extends Proust's text-based representation, it inherits Proust's problems with syntactic variation. This can be avoided by using a graph representation, such as ours, as the base upon which to attach the functional information (see [4], Section 7.4).

Adam and Laurent's LAURA [2] represents programs as graphs, thereby allowing some syntactic variability. However, the graph representation differs from ours in that dataflow is represented implicitly in the graph structure. Nodes represent assignments, tests, and input/output statements, rather than simply operations; arcs represent only control flow. Because of this, LAURA must rely on the use of program transformations to "standardize" the dataflow. (GRASPR need not perform these transformations, since the flow graph representation shows net dataflow explicitly.) LAURA debugs a program by comparing it to a given correct implementation, called the *program model*, of the algorithm which the program is supposed to be using. Only the program model's implementation is recognizable in the program; no implementational variation is allowed.

The system proposed by Fickas and Brooks [43] uses a Plan Calculus-like notation, called *program building blocks* (pbbs), for clichés. Each pbb specifies inputs, outputs, post-conditions, and pre-conditions. (Pbbs are equivalent to Water's segments [137].) The structure of the library is provided by *implementation plans*, which are like implementation overlays in the Plan Calculus. They decompose non-primitive pbbs into smaller pbbs, linked

by dataflow and purpose descriptions. However, on the lowest level of their library (unlike that used by GRASPR), the pbbs are mapped to language-specific code fragments which are matched directly against the program text. Thus, this system also falls prey to the syntactic variation problem.

Murray's Talus [95] uses an abstract frame representation (called an *E-frame*) for programs. The slots of an E-frame contain information about the program, including the type of recursion used, the termination criteria, and the data types of the inputs and outputs. This representation helps abstract away from the syntactic code structure by extracting semantic features from the program, allowing greater syntactic variability. However, listing all characteristics of the code in E-frame slots fails to expose constraints (such as dataflow constraints) in a way that facilitates recognition.

Bertels [11] defines a broad hierarchy of programming knowledge with programming primitives on the bottom, problem solving strategies at the top and clichés at successively higher levels of abstraction in between. The problem solving strategies are strategies for debugging (e.g., slicing), program understanding (e.g., conjecturing), and program synthesis (e.g., divide and conquer). Each level builds on the levels below it. Bertels' model of programming knowledge also includes rules of programming discourse [128] which are applicable at all levels in the hierarchy.

To represent clichés, Bertels uses *conceptual schemes*, which are essentially hierarchical semantic networks. Like our flow graph formalism, these schemes focus on data and control flow constraints. Each conceptual scheme hierarchically represents the decomposition of some goal into subgoals and the methods for achieving them. They can also represent multiple alternative methods for achieving some goal. Their hierarchical structure resembles the organization of clichés in our library, as shown in Figures 2-1, 2-3, and 2-4. Additional information included in the conceptual scheme identifies the roles and various characteristics of the pieces of data used by the methods (e.g., that some piece is a divisor and has a minimum value of 0). Dataflow connections are not explicitly represented.

At the lowest level, conceptual schemes are built out of "Semantically Augmented Programming Primitives" (or SAPPs). These are programming primitives that have been classified in terms of their role in the program on a slightly higher level of abstraction. For example, an assignment might be viewed as an increment and a predicate can be seen as a loop exit test or a filter. In general, it is difficult to unambiguously make this classification of primitives, but Bertels uses a very restricted unambiguous set of SAPPs. These correspond to our lowest level clichés.

Letovsky's Cognitive Program Understander (CPU) [84] uses a lambda calculus representation for programs. CPU uses transformations to standardize (i.e., make more canonical) the program's syntax and to simplify expressions. However, Letovsky generalizes canonicalization to be the entire means of program recognition. Canonicalization involves not only standardizing the syntax of the program, but also standardizing the expression of standard plans (i.e., clichés) in the program. Recognizing a plan that achieves a particular goal is

equivalent to canonicalizing the plan expression to the goal. So, CPU uses a single, general transformation mechanism for dealing with syntactic variability and for recognition. In contrast, GRASPR uses a special-purpose mechanism (the program-to-flow graph translator) to factor out most of the syntactic variability before recognition is attempted.

For CPU to localize clichés in a lambda expression so that a transformation rule can apply, numerous transformations need to be made to copy subexpressions and move them around the program. For example, function-inside-if ([84], p.109) copies functional applications to all branches of a conditional and stored expressions are copied to replace each corresponding variable reference. This is expensive both in the time it takes to apply transformations and in the exponential space blow-up that occurs as a result. In our representation, clichés are localized in the connectivity of the flow graphs. In addition, the ability of the parser to generate multiple analyses enables GRASPR to recognize two clichés whose implementations overlap without first copying the parts that are shared, as CPU must.

Another difference arising from the use of the lambda calculus formalism is in the types of clichés that can be expressed. The components of a cliché expressed in the lambda calculus must be connected in terms of dataflow interaction. CPU's assumption is that clichés are tied together by dataflow, otherwise there is nothing bringing the results together. (One exception to this is a data abstraction plan in which a non-lambda-calculus tupling operation is used to bind together multiple dataflows into a single value.) In flow graph grammar rules, clichés can contain components that are disconnected in terms of dataflow, but which are tied together by other constraints, such as control flow.

There is also a difference between CPU's transformations and our grammar rules. Simple transformations are similar to grammar rules, but complex transformations often specify procedurally how to change the program. For example, the loop analysis transformation is procedural. Loop clichés, such as filtering out certain elements from a list that is being enumerated, are transformed using a recursion elimination technique in which the patterns of dataflow in a loop are analyzed and classified as stream expressions. Then, based on dataflow dependencies, occurrences of primitive loop plans are identified and composed to represent the loop. (This is Waters' temporal abstraction technique [137, 138].) Our rules, on the other hand, are declarative. They can be used in both synthesis (generation) and analysis (parsing).

Laubsch and Eisenstadt [81, 82] and Lutz [88] use variations of the Plan Calculus. Laubsch and Eisenstadt's system differs from GRASPR in the recognition technique it employs. Lutz proposes using a program recognition approach similar to ours. See Section 3.6 for the relationship of Lutz's "flowgraphs" to our flow graphs. (Both of these approaches will be described further in the next section.)

Ning's PAT [100, 54] organizes its cliché library as a hierarchy of *event* classes. Each instance of a cliché is an object, which is an instance of an event class. Each object is a set of attribute-value pairs, representing information about an abstract clichéd operation. They specify the variables involved and lexical information (given in terms of statement line

numbers and block numbers) describing the control path leading to the event. Relationships between program components, such as calling, declaration, and data dependencies, are all encoded implicitly in the event object attributes. Interval logic (which is similar to Allen's temporal logic) is used to derive these relationships during recognition. Because these relationships are not made explicit in the representation, their derivation places a computational burden on the recognition process.

Hartman's UNPROG [55] uses a graphical representation, called a hierarchical program model, or HMODEL, that is roughly the dual of our dataflow graph representation. UNPROG recognizes clichéd patterns of control flow, called *control concepts*, such as "read-process loop", and "bounded linear search". The HMODEL representation consists of a hierarchically decomposed control flow graph and a type of dataflow graph. The nodes of the control flow graph are primitive actions, tests, joins, or other sub-HMODELS and its edges represent the control flow between them. The control flow graph is hierarchically partitioned by proper decomposition, which bundles up sub-graphs that are single-entry, single exit. This static partitioning is performed before recognition is attempted. The dataflow graph represents definition-use relations between the variable names referred to by the control flow graph nodes.

The HMODEL representation can be seen as an encoding of plan diagrams (see Section 4.1.2) in a graph representation which retains the control flow information in the graph structure, but which relegates the dataflow information to attributes (definition-use relations). However, unlike plan diagrams, HMODEL does not represent net dataflow: the definition and use of *variable names* is explicitly captured and assignment is considered a primitive action.

Due to its emphasis on control flow, the HMODEL representation is able to concisely represent general control flow patterns, which are more difficult to capture in our dataflow graphs. (See Section 5.2.3.) On the other hand, our dataflow graphs concisely capture constraints on patterns of dataflow that must exist for instances of algorithmic and data structure clichés to occur. The two representations are complementary. UNPROG and GRASPR could profitably co-operate as co-routines: UNPROG could quickly provide coarse-grain analysis of control patterns, which suggest the existence of certain algorithmic clichés, while GRASPR could focus on a more detailed recognition of these clichés in the parts of the program narrowed down by UNPROG.

7.3.2 Other Recognition Techniques

Besides representational differences, GRASPR differs from other current recognition systems in its technique for performing recognition. Existing recognition techniques differ from ours mainly in the flexibility of their control strategy, how they use heuristics, and how much knowledge about the purpose or goals of the program they *require* as input to help guide their search.

Our *recognition architecture* has a general, flexible control structure which can accept advice and guidance from external agents. Other existing recognition systems are committed to a rigid (often ad hoc) control strategy. Most search for a single best interpretation of the program, while permanently cutting off alternatives. This can cause clichés to be missed. They cannot try harder later to incrementally increase their power and find clichés that the heuristic recognition missed. They also cannot generate multiple views of the program when desired, nor provide partial information when only near-misses of clichés are present.

In addition, many of these systems have heuristics for controlling cost built in directly. These are are chosen on a trial-and-error basis. For example, they often evolve through experimentation with sets of student programs until a good level of performance is reached. Interesting future work with GRASPR will try to formulate probabilities of consistency for constraints (see Section 6.2.5), which can be computed and used to automatically tailor the recognition system to check certain constraints before others. This would dynamically prioritize constraints based on a given program and library of clichés, rather than statically prioritizing them for good performance over "typical" programs and clichés.

Many recognition techniques also take information about the goals and purpose of the program (in the form of a specification or model program). Some recognition systems can accept and respond to information from other non-recognition techniques (e.g., a theorem prover [95] or dynamic analysis of program executions [85]) with which they are integrated. While these techniques show the utility of these additional sources of information, they *rely* on this information being given as input, rather than accepting it and responding to it if it becomes available. Most of these systems have been developed in the context of intelligent tutoring systems for teaching programming skills. In this domain, the purpose of the program being analyzed is very well-defined. It can be used to provide reliable guidance to the program recognition process. However, in many other task applications, especially software maintenance, information about the purpose of the program and its design is rarely complete, accurate, or detailed enough to rely on as required input.

Johnson's PROUST [65] is a system that analyzes and debugs PASCAL programs written by novice programmers. It takes as input a description of the goals of the program and knowledge about how goals can be decomposed into subgoals, as well as the relationships between goals and the computational patterns (clichés) that achieve them. Based on this information, PROUST searches the space of goal decompositions, using heuristics to permanently prune the search. (For example, it uses heuristics about which goals and patterns are likely to occur together.) PROUST looks up the typical patterns that implement the goals and tries to recognize at least one in the code. The low level patterns that actually implement the goals are then found by simple pattern matching.

Ruth's system [122], like PROUST, is given a program to analyze and a description of the task that the program is supposed to perform. The system matches the code against several implementation patterns (clichés) that the system knows about for performing the task. Ruth's approach is similar to GRASPR's in that the system uses a grammar to describe a

class of programs and then tries to parse programs using that grammar. The differences are that Ruth's system makes use of knowledge about the purpose of the program (in the form of a task description) to narrow down its search and the program is analyzed in its textual form and is therefore parsed as a string. Another difference is that Ruth's system does no partial recognition. The entire program must be matched to an algorithm implementation pattern for the analysis to work.

Lukey's Program Understanding and Debugging System (PUDSY) [87] also takes as input information about the purpose of the program it is analyzing, in the form of a *program specification*, which describes the effects of the program. This description is not used, however, in guiding the search for clichés. Rather, PUDSY analyzes the program and then compares the results of the analysis to the program specification. Any discrepancy is pointed out as a bug. The analysis proceeds as follows. PUDSY first uses heuristics to segment the program into *chunks*, which are manageable units of code (e.g., a loop is a chunk). It then describes the flow of information (or interface) between the chunks by generating assertions about the values of the output variables of each chunk. These assertions are generated by recognizing familiar patterns of statements (called *schema*), similar to GRASPR's clichés, in the chunks. Associated with each schema are assertions describing their known effects on the values of variables involved. For chunks that have not been recognized, assertions are generated by symbolic evaluation.

Adam and Laurent's LAURA [2] receives information about the program to be analyzed and debugged in the form of a *model program*, which correctly performs the task that the program to be analyzed is supposed to accomplish. LAURA then compares the graphs of the two programs and treats any mismatches as bugs. Since nodes are really statements of the program, the graph matching is essentially statement-to-statement matching. The system works best for statements that are algebraic expressions because they can be normalized by unifying variable names, reducing sums and products, and canonicalizing their order. The system heuristically applies graph canonicalizing transformations to try to make the program graph better match the model graph. It can find low-level and localized bugs by identifying slight deviations of the program graph from the model graph.

The system proposed by Fickas and Brooks' [43] starts with a high-level cliché abstractly describing the purpose of the program. From this, it hypothesizes refinements and decompositions to subclichés, based on its implementation plans (analogous to overlays in the Plan Calculus). These hypotheses are verified by matching the code fragments of the clichés on the lowest level of the library with the code. While a hypothesis is being verified, other outstanding clues (called *beacons*) may be found that suggest the existence of other clichés. This leads to the creation, modification, and refinement of other hypotheses about the code.

Murray's Talus system [95] is given a student program to be analyzed and debugged, as well as a description of the task the program is supposed to perform. It has a collection of reference programs that perform various tasks that may be assigned to the student. The task description is used to narrow down the reference programs that need to be searched

to find one that best matches the student's possibly buggy program. Heuristic and formal methods are interleaved in Talus's control structure. Symbolic evaluation and case analysis methods detect bugs by pointing out mismatches between the reference program and the student's program. Heuristics are then used to form conjectures about where bugs are located. Theorem proving is used to verify or reject these conjectures. The virtue of this approach is that heuristics are used to pinpoint relatively small parts of the program where some (expensive) formal method (such as theorem proving) may be applied effectively. However, the success of the system depends heavily on the heuristics that identify the algorithm, find localized dissimilarities between the reference program and the student's program, and map the student's variables to reference variables.

Looi's APROPOS2 [85] uses a technique very close to Talus's. It matches a Prolog program against a set of possible algorithms for a particular task. Like Talus, it applies a heuristic best-first search of the algorithm space to find the best fit to the code.

Bertels' [11] Camus performs recognition of programs for the purposes of debugging student programs. It compares student programs against a model program as follows. Camus uses a knowledge base containing the knowledge necessary to analyze a program that is intended to solve the classic Noah Rainfall Problem [65]. The model and student programs are each analyzed using this knowledge base. The analysis converts each program into a "High Level Description" (HLD), containing the conceptual schemes that are found in the program. Camus first "augments" the programming primitives found in the program by classifying them in terms of their role on a slightly higher level of abstraction (i.e., it creates SAPPs – see Section 7.3.1). Based on these SAPPs, conceptual schemes are recognized in a bottom-up, heuristic fashion, using beacons as guides. The two HLD's are compared (currently by a straightforward manual process) and any inconsistency or incompleteness in the student HLD is reported as a bug.

There are a few other recognition techniques that, like GRASPR, are purely code-driven. These will be described in the remainder of this section.

Letovsky's CPU [84] uses a technique called *transformational analysis.* It takes as input a lambda calculus representation of the source code and a collection of correctness-preserving transformations between lambda expressions. Recognition is performed by opportunistically applying the transformations: when an expression matching a standard plan (cliché) is recognized, it is rewritten to an expression of the plan's goal. This is similar to the parsing performed by GRASPR, except that CPU does not find all possible analyses. Rather, it uses a simple recursive control structure in applying transformations: when more than one standard plan matches a piece of code, an arbitrary choice is made between them. The program is destructively reduced and the alternative is never explored further. Letovsky defines a well-formedness criterion for the library of clichéd plans which requires that no plan be a generalization of any other plan. If the library is well-formed, then this arbitrary choice will not matter, since recognizing one plan will not prevent the recognition of another.

However, this relies on the fact that CPU performs a great deal of copying: if two clichés overlap in a program (e.g., as a result of merging implementations as an optimization), their common subparts are copied so that each cliché can be recognized individually without interfering with the recognition of the other cliché. Unfortunately, this leads to the problem of severe "expression swell."

CPU is not able to generate multiple partial analyses of the program. There are situations in which it is better (or necessary) to carry along multiple possible analyses, while sometimes it is sufficient to generate just one analysis. For example, in verification applications, *any* analysis is all that is required. However, multiple analyses are often helpful for programs in which there are unrecognizable sections which lead to several useful ways of partially recognizing the program. Being able to generate partial (near-miss) recognitions is important in robustly dealing with buggy programs as well as in eliciting advice.

The value of our flexible control strategy is that we can tailor it to a particular application or input/output environment. GRASPR can be made to produce a single analysis, by allowing each complete item to extend at most one partial item. Unlike CPU, however, GRASPR can be made to generate more recognition results by exploring alternative analyses, trying harder to find certain clichés, and responding to incremental changes in the input program that may uncover more clichés and cause others to disappear.

Laubsch and Eisenstadt's system [81, 82] distinguishes between two types of clichés: standard (general programming knowledge) and domain-specific. Standard clichés are recognized in the program's plan diagram by nonhierarchical pattern matching (as opposed to parsing). Then the recognized clichés attach effect descriptions to the code in which they are found. Symbolic-evaluation of the program's plan diagram computes the effect-description associated with the entire program. Domain-specific library clichés are recognized by comparing the program's effect description to the effect descriptions of clichés in the library. This transforms the problem of program recognition into the problem of determining the equivalences of formulas. For the examples given, effect-descriptions are simple expressions. However, in general, proving the equivalence of formulas is extremely hard.

Lutz [88, 89] has developed his flowgraph parsing algorithm as a general tool for use in artificial intelligence. He proposes some applications which include program recognition. The examples he sketches use flowgraphs to represent plan diagrams, such as the one shown in Figure 4-6. He proposes using a program recognition process similar to GRASPR's. In addition, his system will use symbolic evaluation to deal with unrecognizable code. Our graph parsing algorithm evolved from the graph parsing algorithm Lutz developed [90] for this purpose. Our algorithm extends Lutz's to handle data aggregation.

Ning's PAT [54, 100] uses basically a bottom-up parsing approach, though not within a formal parsing framework. PAT uses a rule-based inference engine to recognize clichés (i.e., derive high-level program concepts, or events, from lower-level ones). Each rule consists of a trigger pattern of program events, which specifies the events (operations and data types) composing a cliché and how they are related by various types of dependencies and

lexical relationships. The action of the rule is an assertion that a particular higher-level event (cliché) exists in the program at a particular location. PAT can recognize overlapping as well as delocalized clichés and it can do partial recognition. Its rules also distinguish some events within patterns as "key" events, like beacons, that are searched for first. This helps to reduce the search. This is similar to specifying a node ordering in our graph grammar rules. The main difference between PAT's recognition architecture and GRASPR's chart-parser-based architecture is in GRASPR's flexibility of control. GRASPR has explicit data-directed mechanisms for guiding and advising the recognition process.

Hartman's UNPROG [55] performs a type of recognition that is complementary to ours. Hartman has identified a restricted class of clichés, called *control concepts*, that can be recognized efficiently. As mentioned earlier, UNPROG hierarchically models the program's flow of control by performing a proper decomposition on the program's control flow graph. Recognition is then performed by simple exact graph matching. This takes advantage of the fact that typically the implementations of control concepts are not interleaved with each other or with unrecognizable code within propers.

The difference between this technique and our parsing technique is that UNPROG's decomposition of the program is static and independent of the matching, while in parsing, the decomposition is dynamically driven by what is matched. The static, a priori decomposition yields efficiency and scalability advantages. The search is reduced because control concepts are localized within propers. There is no need to generate all partial matches of propers. There is no ambiguity about how to match inputs and outputs of clichéd control concept implementations to those of a proper, since all propers have one input and one output. Hartman's research shows the benefits of good decomposition techniques.

This technique works well for control concept recognition. However, in general, the danger of decomposing the program representation and then looking for particular clichés only within the partitions is that a cliché might be missed if it is not contained within some partition boundary. This technique works best if there are standard decompositions of clichés and the clichés appear in programs in these same organizations. Future research should look for other classes of clichés like control concepts and for methods of decomposition that allow them to be recognized efficiently.

One way GRASPR can benefit from the efficiency of a priori decomposition without sacrificing completeness is to use some sort of decomposition, such as subroutinization, or bundles of slices all contributing to the same user-defined, aggregate data structure to do an initial, quick recognition. Then "try-harder" later by looking for clichés that might cross the boundaries, e.g., in areas where no cliché was recognized or by extending partial items that are near-misses or have salient parts matched already. Section 6.4.1 discussed some of these ideas.

A novel type of recognition is being pursued by Soni [129, 130] as part of the development of a Maintainer's Assistant. This system will focus on recognizing *guidelines* which constrain the design components of a program and embody global interactions between

the components. For example, guidelines express relations between the slots of data structures and constraints on how they may be accessed or updated. This type of recognition is orthogonal to the recognition of clichés reported in this paper.

A completely different approach to recognition was proposed by Biggerstaff [12, 13]. A central part of his recognition system is a rich domain model. This model contains machine-processable forms of design expectations for a particular domain, as well as informal semantic concepts. It includes typical module structures and the typical terminology associated with programs in a particular problem domain. The goal of the recognition is to link these conceptual structures to parts of the program, based on the correlation (experientally acquired) between the structures and the mnemonic procedure and variable names used and the words used in the program's comments. A `grep`-like pattern recognition is performed on the program's text (including its comments) to cluster together parts of the program that are statistically related. (The Unix tool `grep` searches files for given regular expressions.)

The virtue of this type of recognition is that it quickly directs the user's attention to sections of the program where there may be computational entities related to a particular concept in the domain. While this technique cannot be extended to provide a deeper understanding, it provides a way of focusing the search of other more formal and complete recognition approaches, such as `GRASPR`'s. Like Soni's recognition, it is orthogonal and complementary to the recognition of clichés reported here.

7.4 Applications

Being able to automatically recognize existing code has applications in many areas of software development and maintenance, including software reuse, verification, debugging, optimization, program translation, and documentation. The ability to recognize clichés in a broad range of programs is also useful for computer-aided instruction of programmers. See Wills [144, 145] and Hartman [55] for discussions of these applications.

Two other applications of our flow graph formalism and parser, not related to programming, are automatic circuit verification and plan recognition. Circuit verification has been cast as a graph matching problem, with much work focusing on heuristic techniques for solving graph isomorphism [22, 108]. More recently, Bamji [8, 9] has shown how graph parsing can be applied to this problem. This gains the advantage of being able to encode an entire design methodology into a design grammar, so that a circuit can be verified with respect to a class of correct circuits, not just one. Our parsing algorithm is applicable in this area.

Plan recognition shares several difficulties with program recognition, such as dealing with variation due to loose temporal ordering constraints, interleaved steps, and shared steps among plans. Graphical nonlinear plan representations are amenable to the graph parsing technique we used to solve these problems in program recognition.

Appendix A

Flow Graph Recognition is NP-Complete

Barton, Berwick, and Ristad ([10], Chapter 7) give a clever reduction of the vertex cover problem to the problem of recognizing sentences according to an unordered context-free grammar (UCFG). A UCFG is a context-free string grammar in which the symbols in a right-hand side string are considered unordered. (So, for example, given a UCFG containing the rule $S \rightarrow xyz$, S can be recognized in the strings xyz, yxz, zyx, etc.)

Our flow graph parsing algorithm can be used to perform UCFG parsing (and the simpler recognition problem) on a special class of UCFGs, which I will call "fixed-UCFGs." Furthermore, the same reduction proof given by Barton, et al. can be used to prove that the fixed-UCFG recognition problem is NP-complete. This can be used to show that flow graph recognition is NP-complete.

The class of fixed-UCFGs is the class in which each non-terminal derives strings of a fixed length k, where k can be different for different non-terminals. For example, this grammar

```
S -> A B   |  C D E
A -> a | x
B -> b y | w z
C -> c
D -> d | f
E -> e | g | h
```

is a fixed-UCFG. S only derives strings of length three (such as awz or cfh), B only derives strings of length two, the rest of the non-terminals all derive strings of length one. This grammar

```
S -> A B
A -> a x | x y z
B -> b
```

is not a fixed-UCFG, since A can derive two different length strings.

The grammar constructed in Barton, et al.'s NP-completeness proof to encode the vertex cover existence question is always a fixed-UCFG. So, the same construction can be used to reduce the vertex cover problem to the fixed-UCFG recognition problem in polynomial-time.

We reduce the fixed-UCFG recognition problem to flow graph recognition as follows. For each non-terminal, we first compute the length k of the strings it derives. This can be done by imposing a partial ordering on the non-terminals, where non-terminal $A <$ non-terminal B if A appears on B's right-hand side.[1] Then the k's can be computed bottom-up through the partial ordering from the non-terminals that have only terminals on at least one of their rules' right-hand sides.

Next, for each rule in the fixed-UCFG, $A \rightarrow x_1 x_2 x_3 ... x_n$, deriving strings of length k, we create a graph grammar rule with

1. a left-hand side node of type A having k inputs and k outputs,

2. a right-hand side flow graph containing n nodes, where the i-th node has type x_i and each terminal node has a single input and a single output, while each non-terminal node has j inputs and j outputs, where j equals the length of strings derived by that non-terminal, and

3. the rule embedding function maps the i-th input (resp. output) of A to the i-th input (resp. output) of the right-hand side graph. (None of the right-hand sides have edges between ports.)

Finally, the input string is translated into a flow graph by creating a node for each symbol, with the type of the node being the symbol type. Each node has one input and one output. There are no edges between ports.

For example, Figures A-1a and b show a fixed-UCFG and the graph grammar into which it would be translated. Figure A-1c shows how the input string is translated into a flow graph.

Now, we can decide whether a particular input sentence is in the language generated by the fixed-UCFG simply by determining whether the flow graph is in the language generated by the flow graph grammar encoding of the fixed-UCFG. The flow graph is in the language of the flow graph grammar iff the input sentence is in the fixed-UCFG's language.

Since the NP-complete problem of fixed-UCFG recognition can be reduced to flow graph recognition, the flow graph recognition problem is also NP-complete.

Note that the type of flow graph recognition that we are showing to be NP-complete is simpler than the flow graph parsing problem. This in turn is even simpler than the subgraph parsing problem in which program recognition is cast. This means that even if we were just

[1] Cycles in the grammar can be handled, but I do not describe how here. Alternatively, we can do this NP-completeness proof with acyclic fixed-UCFGs.

S → A B | C D E

A → a | x

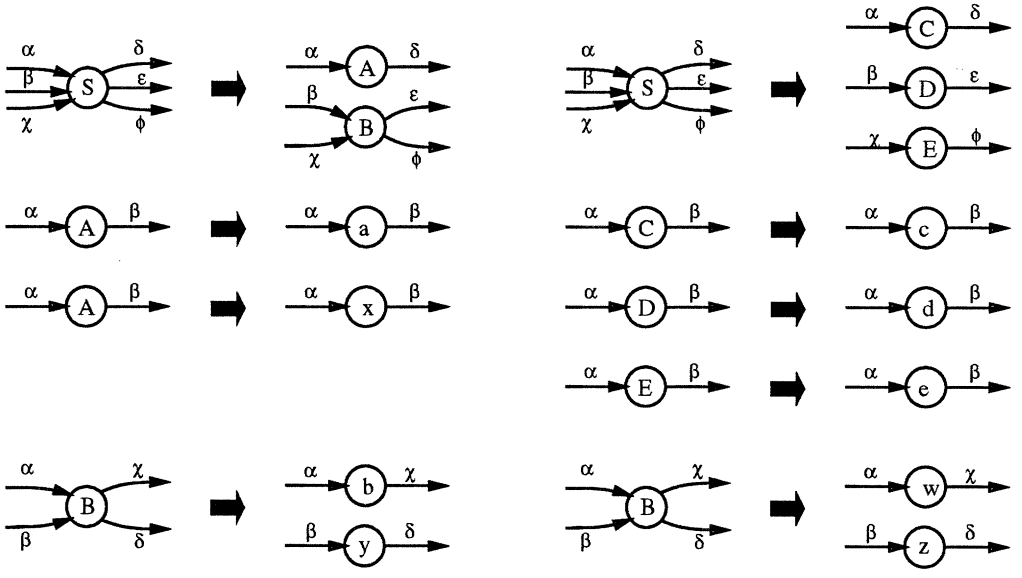
B → b y | w z

C → c

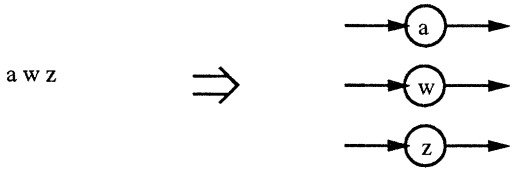
D → d

E → e

a) An Unordered Context-Free Grammar.



b) Graph grammar that the UCFG above is translated into.



c) An input string.

The flow graph it is translated into.

Figure A-1: Reducing fixed-UCFG recognition to flow graph recognition.

trying to recognize an entire program as a single cliché and even if we did not need to deal with fan-in or fan-out, we can still encounter exponential behavior.

Readers familiar with Brotsky's algorithm might contrast flow graph parsing (not subgraph parsing and not dealing with fan-in or fan-out or aggregation) with the parsing Brotsky's algorithm does in polynomial time. The same types of flow graphs are parsed, using the same types of flow graph grammars; no extension to the flow graph formalism is necessary. The crucial distinction is that Brotsky's parser takes an additional input besides the input flow graph and the flow graph grammar, which is a specification of how the inputs of the input graph match to the inputs of the start type of the grammar. This information is used to predict the start type at a particular location (i.e., a particular matching of inputs of the input graph to inputs of the start type). Our parser, on the other hand must figure out all the possible locations at which a non-terminal can be found. This increases the computational complexity of the problem.

Appendix B

The Example Programs

This appendix contains the original `PiSim` and `CST` source code, as well as their functional versions. Section 5.2.5 lists the changes made in translating between the original and functional versions. The original `PiSim` code is listed on pages 260 to 265. Its functional version is found on pages 266 to 274. The original `CST` code is on pages 275 to 280 and its functional version is on pages 281 to 288.

```
;;; -*- Syntax:Common-Lisp; Mode:LISP; Base:10; Package:USER -*-

;;=============================================================
;;   P i     S i m u l a t o r  -- original version

(in-package 'user)

(proclaim '(optimize (compilation-speed 0) (safety 3) (speed 3)))

;;=============================================================
;;                          Global Variables

(defconstant *Machine-Dimensions* '(4 4 4)
          "this is the machine dimensions")

(defvar *Event-Queue* nil
      "this is the global event queue")

(defvar *Nodes* nil
      "this is the node array")

(defvar *Global-Bindings* (Make-Hash-Table)
       "these are the bindings for nodals, constants, etc.")

(defvar *Nodal-Count* 0
      "This is the number of defined nodals")

(defvar *Debug-Level* 0
      "this is the debugging level")

(defvar *Log* nil
      "this is the logging information")

;;=============================================================
;;                           Structures

(defstruct Node
  (Time 0)
  (ID 0)
  (Segments (Make-Hash-Table))
  (Nodals nil))

(defstruct Segment
  (Type nil)
  (Data nil)
  (Size 0))

(defstruct Task
  (Handler nil)
  (Node nil)
  (Segment nil)
  (IP 0)
  (Status 'New))

(defstruct Message
  (Destination nil)
  (Length 0)
  (Type nil)
  (Arguments nil))

(defstruct Event
  (Time 0)
  (Object nil))

(defstruct Handler
  (Name nil)
  (Instructions nil)
  (Arity 0)
  (Number-Of-Locals 0)
  (Bindings (Make-Hash-Table)))

(defstruct D-Sync
  (Suspended-Tasks nil))

(defstruct B-Sync
  (Count 0)
  (Suspended-Tasks nil))

(defstruct Log
  (Type 'All)
  (Task-Status-Profile (Make-Hash-Table))
  (Task-Type-Profile (Make-Hash-Table))
  (Instruction-Type-Profile (Make-Hash-Table))
  (Operation-Type-Profile (Make-Hash-Table))
  (Concurrency-List nil)
  (Old-Logs nil))

(defstruct Delta
  (Time 0)
  (Value 0))

;;=============================================================
;;                           Nodes

;; This translates a node ID to a node.

(defun Translate-Node (Node-ID)
  (aref *Nodes* Node-ID))

;; This function returns the number of nodes.

(defun Number-Of-Nodes ()
  (array-total-size *Nodes*))

;; This function creates the node array according to the dimension
;; constant.

(defun Make-Nodes ()
  (loop with Number-Of-Nodes = (apply #'* *Machine-Dimensions*)
        with Nodes = (make-array Number-Of-Nodes)
        for ID from 0 below Number-Of-Nodes
        for Node = (Make-Node :ID ID)
        for Nodals-Segment = (Create-Read-Write-Segment 100)
        do (setf (aref Nodes ID) Node)
        do (setf (Node-Nodals Node)
                 (Add-Segment Nodals-Segment Node))
        finally (setq *Nodes* Nodes)))

;; This function resets the node time and clears the node segment.

(defun Clear-Nodes ()
  (loop for Node being the array-elements of *Nodes*
        for Nodals-ID = (Node-Nodals Node)
        for Nodals = (Translate-Segment-On-Node Nodals-ID Node)
        doing (setf (Node-Time Node) 0)
        doing (Clear-Hash-Table (Node-Segments Node))
        doing (Hash-Insert (Node-Segments Node) Nodals-ID Nodals)
        doing
        (loop with Data = (Segment-Data Nodals)
              for Index from 0 below (array-total-size Data)
              doing (setf (aref Data Index) 'Unbound))))

;;=============================================================
;;                           Segments

;; This adds a segment to the node's segment translations.  It
;; returns the unique segment ID.

(defun Add-Segment (Segment Node)
  (let ((Segment-ID (gensym "Segment-")))
    (Hash-Insert (Node-Segments Node)
                 Segment-ID
                 Segment)
    Segment-ID))

;; This removes a segment ID from the node's segment translations.

(defun Delete-Segment (Segment-ID Node)
  (Hash-Delete (Node-Segments Node)
               Segment-ID))

;; This translates a segment ID to a segment on the specified
;; task's node.

(defun Translate-Segment (Segment-ID Task)
  (Translate-Segment-On-Node Segment-ID
                             (Task-Node Task)))

;; This translates a segment ID on a specified node.

(defun Translate-Segment-On-Node (Segment-ID Node)
  (let ((Segment (Hash-Lookup (Node-Segments Node)
                              Segment-ID)))
    (if (null Segment)
        (break "PiSim error: missing segment")
        Segment)))

;; This function creates a read-write segment.

(defun Create-Read-Write-Segment (Size)
  (Make-Segment :Size Size
                :Type 'Read-Write
                :Data (make-array Size)))

;; This function creates an associative set segment.

(defun Create-Associative-Set-Segment (Size)
  (Make-Segment :Size Size
                :Type 'Associative-Set
                :Data (Make-Hash-Table Size)))

;; This function creates a cache segment.

(defun Create-Cache-Segment (Size)
  (Make-Segment :Size Size
                :Type 'Cache
                :Data (make-array Size)))
```

```
;; This function reads a read-write segment.

(defun Read-Segment (Segment Offset)
  (unless (equal (Segment-Type Segment)
                 'Read-Write)
    (break "PiSim error: incorrect access operation for ~
            segment type"))
  (aref (Segment-Data Segment) Offset))

;; This function writes a read-write segment.

(defun Write-Segment (Segment Offset New-Value)
  (unless (equal (Segment-Type Segment)
                 'Read-Write)
    (break "PiSim error: incorrect access operation for ~
            segment type"))
  (setf (aref (Segment-Data Segment) Offset)
      New-Value))

;; This function attempts to match a key in an associative set
;; or cache segment.

(defun Match-Segment (Segment Key)
  (case (Segment-Type Segment)
    (Associative-Set
      (Hash-Lookup (Segment-Data Segment) Key))
    (Cache
      (Match-Cache Key Segment))
    (otherwise
     (break "PiSim error: incorrect access operation for ~
             segment type"))))

;; This function inserts a key in an associative set or cache
;; segment.

(defun Insert-Segment (Segment Key New-Value)
  (case (Segment-Type Segment)
    (Associative-Set
      (Hash-Insert (Segment-Data Segment)
                   Key
                   New-Value))
    (Cache
      (Insert-Cache Key Segment New-Value))
    (otherwise
     (break "PiSim error: incorrect access operation for ~
             segment type"))))

;; This function removes a key from an associative set or cache
;; segment.

(defun Remove-Key-Segment (Segment Key)
  (case (Segment-Type Segment)
    (Associative-Set
      (Hash-Delete (Segment-Data Segment) Key))
    (Cache
      (Remove-Key-Cache Key Segment))
    (otherwise
     (break "PiSim error: incorrect access operation for ~
             segment type"))))

;; This function clears an associative set or cache segment.

(defun Clear-Segment (Segment)
  (case (Segment-Type Segment)
    (Associative-Set
      (Clear-Hash-Table (Segment-Data Segment)))
    (Cache
      (Clear-Cache Segment))
    (otherwise
     (break "PiSim error: incorrect access operation for ~
             segment type"))))

;;================================================================
;;                              Caches
;;
;; In PiSim, caches are implemented as direct mapped arrays.  A
;; hash function computes an index into an array.  Array entries
;; are cons cells are of the format: (Key . Value).

;; This is the hash function for caches.

(defun Cache-Hash (Key Size)
  (when (numberp Key)
    (setq Key (format nil "~a" Key)))
  (loop with String = (string Key)
      for Character being the array-elements of String
      summing (char-int Character)
        into Value
      finally (return (mod Value Size))))

;; This function attempts to match a key in a hash table.
;; If the key is found, the corresponding value is returned.
;; Otherwise, 'Miss is returned.

(defun Match-Cache (Key Segment)
  (let* ((Index (Cache-Hash Key (Segment-Size Segment)))
         (Entry (aref (Segment-Data Segment) Index)))
    (if (and (not (equal Entry 'Empty))
             (equal (first Entry) Key))
        (rest Entry)
        'Miss)))

;; This function writes an entry in the cache, possible overwriting
;; another value.

(defun Insert-Cache (Key Segment New-Value)
  (setf (aref (Segment-Data Segment)
              (Cache-Hash Key (Segment-Size Segment)))
        (cons Key New-Value)))

;; This function removes a key from a cache.  If the key is not present,
;; no action is taken.

(defun Remove-Key-Cache (Key Segment)
  (let* ((Index (Cache-Hash Key (Segment-Size Segment)))
         (Entry (aref (Segment-Data Segment) Index)))
    (when (and (not (equal Entry 'Empty))
               (equal (first Entry) Key))
      (setf (aref (Segment-Data Segment) Index)
            'Empty))))

;; This function clears a cache.

(defun Clear-Cache (Segment)
  (loop with Data = (Segment-Data Segment)
        for Index from 0 below (array-total-size Data)
        doing (setf (aref Data Index) 'Empty)))

;;================================================================
;;                                Tasks

;; This returns the node ID of the specified task's nodes.

(defun Node-Of (Task)
  (Node-ID (Task-Node Task)))

;; This returns the time of a task.  This is defined as the node
;; time for the specified task.

(defun Time-Of (Task)
  (Node-Time (Task-Node Task)))

;; This sets the time of the specified task (i.e. the time of
;; the node of the specified task).

(defun Set-Time-Of (Task New-Time)
  (setf (Node-Time (Task-Node Task))
        New-Time))

;; This increments the task time by the specified delta.

(defun Increment-Time-Of (Task Delta)
  (incf (Node-Time (Task-Node Task))
        Delta))

;; This returns the handler type of the task.

(defun Handler-Name-Of (Task)
  (Handler-Name (Task-Handler Task)))

;; This function creates a new task segment of the specified length.
;; The number of arguments and message length values are compared with
;; the handler arity and arity plus number of locals respectively.  Two
;; is added to the arity and number of locals to account for the message
;; length and type information stored in the segment.  The segment is
;; then initializes with the supplied arguments.

(defun Create-Task-Segment (Length Task-Type Arguments Handler)
  (let ((New-Segment (Create-Read-Write-Segment Length)))
    (when (not (= (Handler-Arity Handler)
                  (length Arguments)))
      (break "PiSim error: arity mismatch"))
    (when (not (= Length (+ (Handler-Arity Handler)
                            (Handler-Number-Of-Locals Handler)
                            2)))
      (break "PiSim error: length/ handler storage mismatch"))
    (Write-Segment New-Segment 0 Length)
    (Write-Segment New-Segment 1 Task-Type)
    (loop for Argument in Arguments
          for Index from 2
          doing (Write-Segment New-Segment Index Argument))
    New-Segment))

;; This function creates a new task for a message.  The handler and
;; node are determined.  A new segment is created and initialized.
;; After the new task is created, its segment is added to the task's
;; node.  Finally the new task is returned.
```

```
(defun Create-Task (Message)
  (let* ((Handler (Get-Handler (Message-Type Message)))
       (Node (Translate-Node (Message-Destination Message)))
       (New-Segment (Create-Task-Segment
                     (Message-Length Message)
                     (Message-Type Message)
                     (Message-Arguments Message)
                     Handler))
       (New-Segment-ID (Add-Segment New-Segment Node))
       (New-Task (Make-Task :Handler Handler
                            :Node Node
                            :Segment New-Segment-ID)))
    New-Task))

;; This function executes a task.  It executes instructions which
;; change a task's status.  If the status is 'Running, another
;; instruction is executed.

(defun Execute-Task (Task)
  (loop doing (Execute-Next-Instruction Task)
      while (equal (Task-Status Task) 'Running)))

;;=============================================================
;;                              Events
;;
;; This function enqueues an event in the global event queue.
;; Events are enqueued in order on increasing event time.
;; ** Note that when 2 events have the same time, the one sent
;; to Enqueue-Event first has higher priority.

(defun Enqueue-Event (New-Event)
  (if (or (null *Event-Queue*)
        (< (Event-Time New-Event)
           (Event-Time (first *Event-Queue*))))
      (push New-Event *Event-Queue*)
      (Insert-Event New-Event *Event-Queue*)))

;; This function is used to enqueue events inside the event queue.
;; It is part of a recursive, priority queue insert algorithm.

(defun Insert-Event (New-Event Event-Queue)
  (if (or (null (rest Event-Queue))
        (< (Event-Time New-Event)
           (Event-Time (second Event-Queue))))
      (push New-Event (rest Event-Queue))
      (Insert-Event New-Event (rest Event-Queue))))

;; This function dequeues and returns a event from the global
;; event-queue.  If the queue is empty, nil is returned.

(defun Dequeue-Event ()
  (pop *Event-Queue*))

;; This function clears the event queue.

(defun Clear-Event-Queue ()
  (setq *Event-Queue* nil))

;; This function dequeues and executes the next event in the event
;; queue.  If the event is a message, a new task is created.  The
;; node time is adjusted if the event time is later than node
;; time.  If a event is executed, t is returned.

(defun Execute-Next-Event ()
  (let* ((Event (Dequeue-Event))
         Task)
    (setq Task (Create-Task (Event-Object Event)))
    (Set-Time-Of Task
                 (if (> (Event-Time Event)
                        (Time-Of Task))
                     (Event-Time Event)
                     (Time-Of Task)))
    (Debug-Print 1
              "[start: task ~a node ~d time ~d old status ~a]~&"
              (Handler-Name-Of Task) (Node-Of Task)
              (Time-Of Task) (Task-Status Task))
    (Log-Task Task)
    (setf (Task-Status Task) 'Running)
    (Adjust-Concurrency-List (Time-Of Task) 1)
    (Execute-Task Task)
    (Adjust-Concurrency-List (Time-Of Task) -1)
    (Debug-Print 1
              "[stop: task ~a node ~d time ~d status ~a]~&"
              (Handler-Name-Of Task) (Node-Of Task)
              (Time-Of Task) (Task-Status Task))))

;;=============================================================
;;                            Handlers

;; This predicate tests if a statement is an instruction.

(defun Label? (Statement)
  (symbolp Statement))

;; This predicate tests if a statement is an instruction.

(defun Instruction? (Statement)
  (listp Statement))

;; This function inserts a binding into a handler's bindings.  If the
;; specified handler is 'Global, the binding is inserted in the global
;; bindings.

(defun Insert-Binding (Name Value Handler)
  (if (equal Handler 'Global)
      (Hash-Insert *Global-Bindings* Name Value)
      (Hash-Insert (Handler-Bindings Handler) Name Value)))

;; This function looks up the binding of a symbol in the handler.  If
;; it is not found there, the global bindings are checked.

(defun Lookup-Binding (Name Handler)
  (or (Hash-Lookup (Handler-Bindings Handler) Name)
      (Hash-Lookup *Global-Bindings* Name)))

;; This function returns the number of instructions in a handler.

(defun Number-Of-Instructions (Handler)
  (array-total-size (Handler-Instructions Handler)))

;; This function returns the handler object for the handler name.  If
;; the handler does not exist, an error message is printed.

(defun Get-Handler (Name)
  (let ((Handler (get Name 'Handler)))
    (if (null Handler)
        (break "PiSim error: unknown handler")
        Handler)))

;; This function determines the number of instructions in a sequence
;; of statements and builds a instruction array of the correct size.
;; It then reads each statement.  If it is an instruction, it is
;; inserted into the array.  If it is a label, the label and
;; statement index is inserted into the handler's bindings.

(defun Make-Instructions (Statements Handler)
  (let (Instructions)
    (loop for Statement in Statements
          unless (Label? Statement)
            count Statement
              into Number-Of-Statements
          finally (setf Instructions
                        (make-array Number-Of-Statements)))
    (loop with Index = 0
          for Statement in Statements
          when (Label? Statement)
            do (Insert-Binding Statement Index Handler)
          when (Instruction? Statement)
            do (setf (aref Instructions Index)
                     Statement)
               (incf Index))
    (setf (Handler-Instructions Handler)
          Instructions)))

;; This function indexes the parameters and locals in a handler.
;; This includes assigning a each parameter and value an index in the
;; handler segment.  These assignments are included in the handler's
;; bindings.  The arity and number of locals parameters are also set.

(defun Index-Parameters-And-Locals (Parameters Locals Handler)
  (loop for Parameter in Parameters
        for Index from 2
        doing (Insert-Binding Parameter Index Handler))
  (loop for Local in Locals
        for Index from (+ (length Parameters) 2)
        doing (Insert-Binding Local Index Handler))
  (setf (Handler-Arity Handler)
        (length Parameters))
  (setf (Handler-Number-Of-Locals Handler)
        (length Locals)))

;; This function reads a handler from an expression.  The resultant
;; handler is stored on the property list of the handler name.

(defun Read-Handler (Expression)
  (let ((Name (first Expression))
        (Parameters (second Expression))
        (Locals (third Expression))
        (Statements (nthcdr 3 Expression))
        (New-Handler (Make-Handler)))
    (setf (Handler-Name New-Handler) Name)
    (Index-Parameters-And-Locals Parameters Locals New-Handler)
    (Make-Instructions Statements New-Handler)
    (setf (get Name 'Handler) New-Handler)))

;; This allows the definition of handlers.  This should be part
;; of a more general reader.
```

```
(defun Define-Handler (&rest Expression)
  (Debug-Print 0 "~&loading handler ~a~&" (first Expression))
  (Read-Handler Expression)
  nil)

;;=============================================================
;;                            Nodals

;; This allows the definition of nodals (node variables).  An
;; index is assigned (using the number of existing nodals).  A
;; new global binding is added.

(defun Define-Nodal (Name)
  (Debug-Print 0 "~&defining nodal ~a~&" Name)
  (cond ((not (null (Hash-Lookup *Global-Bindings* Name)))
         (format t
                 "~&Warning: ~a has already been defined globally~&"
                 Name))
        (t
         (Insert-Binding Name *Nodal-Count* 'Global)
         (incf *Nodal-Count*))))

;;=============================================================
;;                            Constants

;; This allows the definition of global constants.  The binding
;; is added to the global bindings.

(defun Define-Constant (Name Value)
  (Debug-Print 0 "~&defining constant ~a~&" Name)
  (Insert-Binding Name Value 'Global))

;;=============================================================
;;                             Instructions

;; This function returns the next instruction of the handler to be
;; executed.  The current instruction pointer (IP) is obtained
;; from the task.  The instructions are obtained from the handler.
;; The task instruction pointer is incremented.  Note: the
;; instruction pointer is incremented AFTER the next instruction
;; is fetched.

(defun Next-Instruction (Task)
  (let ((IP (Task-IP Task)))
    (when (>= IP
              (Number-Of-Instructions (Task-Handler Task)))
      (break "PiSim error: IP out of range"))
    (incf (Task-IP Task))
    (aref (Handler-Instructions (Task-Handler Task))
          IP)))

;; This function executes a single instructions.  It first
;; locates the next instruction using the task instruction
;; pointer.  The instruction pointer is incremented. Then it
;; applies the operation to the arguments.

(defun Execute-Next-Instruction (Active-Task)
  (let ((Instruction (Next-Instruction Active-Task)))
    (Debug-Print 2 "   [executing instruction ~a]~&"
                 (first Instruction))
    (Log-Instruction Instruction)
    (Apply-Operation (first Instruction)
                     Active-Task
                     (rest Instruction))))

;;=============================================================
;;                            Operations

;; This function applies a processor operation to a list of
;; arguments.  Each argument is evaluated before the operation
;; is applied.  The apply only takes place if the task status
;; is 'RUNNING.

(defun Apply-Operation (Operation Active-Task Arguments)
  (let ((Argument-List
         (loop for Argument in Arguments
               collecting (Evaluate Active-Task Argument))))
    (when (equal (Task-Status Active-Task) 'RUNNING)
      (Log-Operation Operation)
      (push Active-Task Argument-List)
      (apply (Get-Operation Operation)
                     Argument-List))))

;; This function evaluates the expression and returns the results.
;; This is an evaluator appropriate for the limited expressions
;; in a Pi program.  Expressions are only evaluated if the task
;; status is 'RUNNING. The following expression types are
;; possible:
;;   A number or string returns the value of the number or string.
;;
;;   A symbol is looked up in the handler bindings.  If it is
;; present, the corresponding value is returned.  Otherwise, the
;; symbol is returned.
;;
;; A nested expression (a list) is in the form (symbol arg1 arg2...).
;; In this case, Apply-Operation is recursively called.

(defun Evaluate (Active-Task Expression)
  (when (equal (Task-Status Active-Task)
               'RUNNING)
    (typecase Expression
      ((or number string)
       Expression)
      (symbol
       (or (Lookup-Binding Expression (Task-Handler Active-Task))
           Expression))
      (list
       (Apply-Operation (first Expression)
                        Active-Task
                        (rest Expression)))
      (otherwise
       (break "PiSim error: unknown expression")))))

;; This function returns the operation function for the operation
;; name.  If the operation does not exist, an error message is
;; printed.

(defun Get-Operation (Name)
  (let ((Operation (get Name 'Operation)))
    (if (null Operation)
        (break "PiSim error: unknown operation")
        Operation)))

;; This is used to define processor operations.

(defmacro Define-Operation (Name &rest Rest)
  (setf (get ',Name 'Operation)
        #'(lambda ,@Rest)))

;;=============================================================
;;                              Debugging

;; This prints debug messages depending on the debug level.

(defmacro Debug-Print (Level Format &rest Arguments)
  `(when (<= ,Level *Debug-Level*)
     (format t ,Format ,@Arguments)))

;; This function sets the debug level.

(defun Set-Debug-Level (New-Level)
  (setq *Debug-Level* New-Level))

;;=============================================================
;;                              Logging

;; This predicate starts a new log, saving the current log.

(defun Start-New-Log ()
  (setq *Log*
        (Make-Log :Type (Log-Type *Log*)
                  :Old-Logs *Log*)))

;; This is used in a counting profile.  The category count is
;; incremented, or created, if non-existent.

(defun Collect-Profile (Category Profile)
  (if (Hash-Lookup Profile Category)
      (Hash-Insert Profile
                   Category
                   (1+ (Hash-Lookup Profile Category)))
      (Hash-Insert Profile Category 1)))

;; This predicate tests if logging is enabled.  If the log is nil, logging
;; is on.

(defun Logging? ()
  (not (or (null *log*)
           (equal (Log-Type *Log*) 'None))))

;; This function logs the specified task. Presently, profiles of task types
;; and status' are maintained.

(defun Log-Task (Task)
  (when (Logging?)
    (Collect-Profile (Task-Status Task)
                     (Log-Task-Status-Profile *Log*))
    (when (equal (Task-Status Task) 'New)
      (Collect-Profile (Handler-Name-Of Task)
                       (Log-Task-Type-Profile *Log*)))))

;; This function collects statistics on instruction types.

(defun Log-Instruction (Instruction)
  (when (Logging?)
    (cond ((not (equal (first Instruction) 'Write))
```

```
        (Collect-Profile (first Instruction)
                         (Log-Instruction-Type-Profile *Log*)))
       ((not (listp (fourth Instruction)))
        (Collect-Profile 'Initialize
                         (Log-Instruction-Type-Profile *Log*)))
       ((equal (first (fourth Instruction)) 'Read)
        (Collect-Profile 'Move
                         (Log-Instruction-Type-Profile *Log*)))
       (t
        (Collect-Profile (first (fourth Instruction))
                         (Log-Instruction-Type-Profile
                         *Log*))))))

;; This function creates an operation profile.

(defun Log-Operation (Operation)
  (when (Logging?)
    (Collect-Profile Operation
                     (Log-Operation-Type-Profile *Log*))))

;; This function searches down a sorted list of deltas looking
;; for an entry at a specified time.  If such an entry is found,
;; its value is adjusted by Change.  If no such value is found,
;; a new delta is created an inserted at the correct position
;; in the list.

(defun Adjust-Concurrency-List (Time Change)
  (when (Logging?)
    (let ((Concurrency-List (Log-Concurrency-List *Log*)))
      (cond ((or (null Concurrency-List)
                 (< Time
                    (Delta-Time (first Concurrency-List))))
             (push (Make-Delta :Time Time
                               :Value Change)
                   (Log-Concurrency-List *Log*)))
            ((= Time
                (Delta-Time (first Concurrency-List)))
             (incf (Delta-Value (first Concurrency-List))
                   Change))
            (t
             (Adjust-Rest-Of-Concurrency-List
               Time Change Concurrency-List))))))

;; This is the recursive part of Adjust-Concurrency-List.

(defun Adjust-Rest-Of-Concurrency-List (Time Change
                                        Concurrency-List)
  (cond ((or (null (rest Concurrency-List))
             (< Time (Delta-Time (second Concurrency-List))))
         (rplacd Concurrency-List
                 (cons (Make-Delta :Time Time
                                   :Value Change)
                       (rest Concurrency-List))))
        ((= Time
            (Delta-Time (second Concurrency-List)))
         (incf (Delta-Value (second Concurrency-List))
               Change))
        (t
         (Adjust-Rest-Of-Concurrency-List
           Time Change (rest Concurrency-List)))))

;; This function prints the information from the current log.

(defun Print-Log-Information ()
  (when (or (equal (Log-Type *Log*) 'All)
            (equal (Log-Type *Log*) 'Profile))
    (Print-Profile-Data))
  (when (or (equal (Log-Type *Log*) 'All)
            (equal (Log-Type *Log*) 'Plot))
    (Plot-Concurrency)))

;;=============================================================
;;                            Randoms

;; This function estimates the delivery delay of a message.  It
;; should be better than it is now.

(defun Delivery-Delay (Source Destination Length)
  (when (or (>= Source (Number-Of-Nodes))
            (minusp Source)
            (>= Destination (Number-Of-Nodes))
            (minusp Destination))
    (break "PiSim error: illegal node number"))
  (when (or (minusp Length)
            (zerop Length))
    (break "PiSim error: illegal message length"))
  (loop for Dimension in *Machine-Dimensions*
      collecting (mod Source Dimension)
        into Source-Components
      doing (setq Source (floor Source Dimension))
      collecting (mod Destination Dimension)
        into Destination-Components
      doing (setq Destination (floor Destination Dimension))
      finally (return
               (loop for Source-Component
                       in Source-Components
                     for Destination-Component
                       in Destination-Components
                     summing (abs (- Source-Component
                                     Destination-Component))
                       into Distance
                     finally (return (+ Distance (- Length 1)))))))

;; This function injects a starting message into the machine.  It
;; starts calculating the message length and destination.  The
;; message is then enqueued, and events are executed until the
;; event queue is empty.

(defun Inject (Type &rest Arguments)
  (Make-Nodes)
  (Clear-Nodes)
  (Clear-Event-Queue)
  (let* ((Handler (Get-Handler Type))
         (Length (+ (Handler-Arity Handler)
                    (Handler-Number-Of-Locals Handler)
                    2))
         (Destination (random (Number-Of-Nodes)))
         (Arrival-Time (Node-Time (Translate-Node Destination)))
         (Message (Make-Message :Destination Destination
                                :Length Length
                                :Type Type
                                :Arguments Arguments))
         (Event (Make-Event :Time Arrival-Time
                            :Object Message)))
    (Enqueue-Event Event)
    (loop
      (cond ((null *Event-Queue*)
             (return))
            (t
             (Execute-Next-Event))))))

;;=============================================================
;;                          Hash Table Functions

(defconstant MIN_HASH_TABLE_SIZE 11)

(defstruct Entry
  (Key   nil :type symbol)
  (Value nil :type any))

(defstruct HashTable
  (Num-Buckets nil :type integer)
  (Number-Entries nil :type integer)
  (Buckets nil :type array))

;;; This function inserts a entry into the hash table.  If a bucket
;;; collision occurs, the entry is inserted in the list in increasing key
;;; order.  If a key collision occurs, the older entry is overwritten.
;;; This function also increases the hash table size if necessary.

(defun Hash-Insert (Table Key Value)
  (let* ((Index (Hash-Function Key
                               (HashTable-Num-Buckets Table)))
         (Bucket-List (aref (HashTable-Buckets Table)
                            Index)))
    (cond ((or (null Bucket-List)
               (string< Key (Entry-Key (car Bucket-List))))
           (push (Make-Entry :Key Key
                             :Value Value)
                 (aref (HashTable-Buckets Table)
                       Index))
           (setf (HashTable-Number-Entries Table)
                 (1+ (HashTable-Number-Entries Table))))
          (t
           (let ((This-Entry (car Bucket-List)))
             (cond ((string= Key (Entry-Key This-Entry))
                    ;; if Key = key of This-Entry, then overwrite older
                    ;; bucket entry.  (New bucket has same Key as older
                    ;; Bucket entry, but new entry value.)
                    (format t "~&Bashing older bucket entry ~A."
                            This-Entry)
                    (setf (Entry-Value This-Entry)
                          Value))
                   (t
                    (Splice-In-Bucket
                      Key Value Bucket-List Table)))))))
    (if (>= (HashTable-Number-Entries Table)
            (HashTable-Num-Buckets Table))
        (Hash-Resize Table)
        Table)))

(defun Splice-In-Bucket (Key Value Bucket-List Table)
  (let* ((Next-List (cdr Bucket-List))
    (cond ((or (null Next-List)
               (string< Key (Entry-Key (car Next-List))))
           (rplacd Bucket-List
```

```
                (cons (Make-Entry :Key Key
                                  :Value Value)
                      Next-List))
          (setf (HashTable-Number-Entries Table)
                (1+ (HashTable-Number-Entries Table))))
         (t
           (let ((This-Entry (car Next-List)))
             (cond ((string= Key (Entry-Key This-Entry))
                    ;; if Key = key of This-Entry, then overwrite
                    ;; older bucket entry's value.
                    (format t "~&Bashing older bucket entry ~A."
                            This-Entry)
                    (setf (Entry-Value This-Entry)
                          Value))
                   (t
                     (Splice-In-Bucket
                       Key Value Next-List Table)))))))))

;;; This function resizes the hash table and rehashes the
;;; entries.  The hash table size is approximately doubled.

(defun Hash-Resize (Table)
  (let* ((Old-Buckets (HashTable-Buckets Table))
         (Old-Size (HashTable-Num-Buckets Table))
         (New-Size
           (Determine-Hash-Table-Size
             (* (HashTable-Num-Buckets Table) 2))))
    (setf (HashTable-Num-Buckets Table)
          New-Size)
    (setf (HashTable-Buckets Table)
          (Make-Hash-Buckets New-Size))
    (setf (HashTable-Number-Entries)
          0)
    (Copy-Over-Buckets 0 Old-Size Old-Buckets Table)
    Table))

(defun Copy-Over-Buckets (Index Old-Size Old-Buckets Table)
  (cond ((>= Index Old-Size)
         Table)
        (t
         (let ((Bucket-List (aref Old-Buckets Index)))
           (Copy-Over-Bucket Bucket-List Table)
           (Copy-Over-Buckets
             (1+ Index) Old-Size Old-Buckets Table)))))

(defun Copy-Over-Bucket (Bucket-List Table)
  (cond ((null Bucket-List) Table)
        (t
         (let ((This-Entry (car Bucket-list)))
           (Hash-Insert Table
                        (Entry-Key This-Entry)
                        (Entry-Value This-Entry))
           (Copy-Over-Bucket (cdr Bucket-List) Table)

;; This function creates a hash table having the specified # of
;; buckets.  Since the size of a hash table must be a prime
;; number, the specified number of buckets is rounded up to a
;; nearby prime.  The new table is then initialized.

(defun Make-Hash-Table (&optional Num-Buckets)
  (let ((Size (Determine-Hash-Table-Size
                (or Num-Buckets MIN_HASH_TABLE_SIZE))))
    (Make-HashTable :Num-Buckets Size
                    :Buckets (Make-Hash-Buckets Size)
                    :Number-Entries 0)))

;;This function creates and initializes a bucket array.

(defun Make-Hash-Buckets (Size)
  (make-array Size))

;;; This function looks up a key in the hash table.  If it is
;;; found, the entry pointer is returned.  Otherwise, nil is
;;; returned.

(defun Hash-Lookup (Table Key)
  (let* ((Index (Hash-Function
                  Key (HashTable-Num-Buckets Table)))
         (Bucket-List (aref (HashTable-Buckets Table)
                            Index)))
    (loop
      (cond ((or (null Bucket-List)
                 (string< Key
                          (Entry-Key (car Bucket-List))))
             (return nil))
            ((string= Key
                      (Entry-Key (car Bucket-List)))
             (return (Entry-Value (car Bucket-List))))
            (t
             (setq Bucket-List (cdr Bucket-List)))))))

;;; This function deletes an entry in the hash table.

(defun Hash-Delete (Table Key)
  (let* ((Index (Hash-Function Key
                               (HashTable-Num-Buckets Table)))
         (Bucket-List (aref (HashTable-Buckets Table)
                            Index)))
    (if (null Bucket-List)
        Table
        (let ((This-Entry (car Bucket-List)))
          (cond ((string> Key (Entry-Key This-Entry))
                 (Splice-Out-Bucket Key Bucket-List Table))
                ((string= Key (Entry-Key This-Entry))
                 (setf (aref (HashTable-Buckets Table)
                             Index)
                       (cdr Bucket-List))
                 (setf (HashTable-Number-Entries Table)
                       (1- (HashTable-Number-Entries Table))))
                (t ;; Key string< key of This-Entry, so Key isn't found
                  Table))))))

(defun Splice-Out-Bucket (Key Bucket-List Table)
  (let ((Next-List (cdr Bucket-List)))
    (if (null Next-List)
        Table  ;; fell off end of bucket list, Key not found
        (let ((This-Entry (car Next-List)))
          (cond ((string> Key (Entry-Key This-Entry))
                 (Splice-Out-Bucket Key Next-List Table))
                ((string= Key (Entry-Key This-Entry))
                 (rplacd Bucket-List
                         (cdr Next-List))
                 (setf (HashTable-Number-Entries Table)
                       (1- (HashTable-Number-Entries Table))))
                (t ;; Key string< Key of This-Entry, Key not found
                  Table))))))

;;; This function clears for all entries in the specified hash table.

(defun Clear-Hash-Table (Table)
  (let ((Size (HashTable-Num-Buckets Table)))
    (setf (HashTable-Num-Buckets Table) Size)
    (setf (HashTable-Number-Entries Table) 0)
    (setf (HashTable-Buckets Table) (Make-Hash-Buckets Size))))

;;; This function picks the first prime number greater then or equal to
;;; the specified size estimate. The minimum hash table size is enforced
;;; here.

(defun Determine-Hash-Table-Size (Size-Estimate &aux Size)
  (if (< Size-Estimate MIN_HASH_TABLE_SIZE)
      (setq Size MIN_HASH_TABLE_SIZE)
      (setq Size Size-Estimate))
  (if (= (mod Size 2) 0)
      (setq Size (1+ Size)))
  (loop
    (if (null (Prime-Number-Test Size))
        (setq Size (+ Size 2))
        (return)))
  Size)

(defun Prime-Number-Test (Number)
  (let ((Index 3))
    (cond ((= Number 2) t)
          ((= (mod Number 2) 0) nil)
          (t
           (loop
             (cond ((<= (Square Index) Number)
                    (if (= (mod Number Index) 0)
                        (return nil))
                    (setq Index (+ Index 2)))
                   (t (return t)))))))))

(defun Square (n)
  (* n n))

;;; This function calculates a hash table index from a key
;;; (symbol->string) and the hash table size.

(defun Hash-Function (Key Size)
  (let* ((Sum 0)
         (Key-String (string Key))
         (Length (1- (string-length Key-String))))
    (loop
      (cond ((< Length 0) (return))
            (t
             (setq Sum
                   (+ Sum (char-int (aref Key-String Length))))
             (setq Length (1- Length)))))
    (mod Sum Size)))
```

```
;;; -*- Syntax:Common-Lisp; Mode:LISP; Base:10; Package:USER -*-

;;============================================================
;;   P i     S i m u l a t o r  -- functional version

;;============================================================
;;                          Global Variables

(defconstant *Machine-Dimensions* '(4 4 4)
          "this is the machine dimensions")

(defvar *Event-Queue* nil
        "this is the global event queue")

(defvar *Nodes* nil
      "this is the node array")

(defvar *Global-Bindings* (Make-Hash-Table)
      "these are the bindings for nodals, constants, etc.")

(defvar *Nodal-Count* 0
      "This is the number of defined nodals")

(defvar *Debug-Level* 0
      "this is the debugging level")

(defvar *Log* nil
      "this is the logging information")

(defvar *Global-Plist* nil
      "The global property list.")

;;============================================================
;;                           Structures

(defstruct Node
  (Time 0)
  (ID 0)
  (Segments (Make-Hash-Table))
  (Nodals nil))

(defstruct Segment
  (Type nil)
  (Data nil)
  (Size 0))

(defstruct Task
  (Handler nil)
  (Node nil)
  (Segment nil)
  (IP 0)
  (Status 'New))

(defstruct Message
  (Destination nil)
  (Length 0)
  (Type nil)
  (Arguments nil))

(defstruct Event
  (Time 0)
  (Object nil))

(defstruct Handler
  (Name nil)
  (Instructions nil)
  (Arity 0)
  (Number-Of-Locals 0)
  (Bindings (Make-Hash-Table)))

(defstruct D-Sync
  (Suspended-Tasks nil))

(defstruct B-Sync
  (Count 0)
  (Suspended-Tasks nil))

(defstruct Log
  (Type 'All)
  (Task-Status-Profile (Make-Hash-Table))
  (Task-Type-Profile (Make-Hash-Table))
  (Instruction-Type-Profile (Make-Hash-Table))
  (Operation-Type-Profile (Make-Hash-Table))
  (Concurrency-List nil)
  (Old-Logs nil))

(defstruct Delta
  (Time 0)
  (Value 0))

(defstruct Task-Segment
  (Storage-Rqmts 0)
  (Type nil)
  (Arguments nil))

(defstruct Instruction
  (Op nil)
  (Args nil))

;;============================================================
;;                          Nodes

;; This translates a node ID to a node.

(defun Translate-Node (Node-ID)
  (aref *Nodes* Node-ID))

;; This function returns the number of nodes.

(defun Number-Of-Nodes ()
  (array-total-size *Nodes*))

(defun Copy-Replace-Node (New-Node ID Nodes)
  (Copy-Replace-Elt New-Node ID Nodes))

;; This function creates the node array according to the dimension
;; constant.

(defun Make-Nodes ()
  (let* ((Number-Of-Nodes (apply #'* *Machine-Dimensions*))
         (Nodes (make-array Number-Of-Nodes))
         (ID 0)
         (Node nil)
         (Nodals-Segment NIL))
    (Make-Nodes-1 Number-of-Nodes Nodes ID Node Nodals-Segment)))

(defun Make-Nodes-1 (Number-of-Nodes Nodes ID Node Nodals-Segment)
  (cond ((not (< ID Number-Of-Nodes))
         (setq *Nodes* Nodes))
        (t
         (setq Node (Make-Node :ID ID))
         (setq Nodals-Segment (Create-Read-Write-Segment 100))
         (setq Nodes (Copy-Replace-Node Node ID Nodes))
         (multiple-value-bind (Sgmt-ID Intermediate-Node)
             (Add-Segment Nodals-Segment Node)
           (setq Node
                 (Make-Node :Time (Node-Time Intermediate-Node)
                            :ID (Node-ID Intermediate-Node)
                            :Segments (Node-Segments Intermediate-Node)
                            :Nodals Sgmt-ID))
           (setq Nodes (Copy-Replace-Node Node (Node-ID Node) Nodes)))
         (Make-Nodes-1 Number-of-Nodes Nodes (+ ID 1) Node
                       Nodals-Segment))))

;; This function the node time and clears the node segment.

(defun Clear-Nodes ()
  (let ((Node nil)
        (Nodes-Index 0)
        (Nodals-Id nil)
        (Nodals nil)
        (End-Index (array-total-size *Nodes*)))
    (Clear-Nodes-1 Node Nodes-Index Nodals-Id Nodals End-Index)))

(defun Clear-Nodes-1 (Node Nodes-Index Nodals-Id Nodals End-Index)
  (cond ((not (< Nodes-Index End-Index))
         nil)
        (t
         (setq Node (aref *Nodes* Nodes-Index))
         (setq Nodals-Id (Node-Nodals Node))
         (setq Nodals (Translate-Segment-On-Node Nodals-Id Node))
         (setq Node (Make-Node :Time 0    ;; (setf (Node-Time Node) 0)
                               :ID (Node-ID Node)
                               :Segments (Node-Segments Node)
                               :Nodals (Node-Nodals Node)))
         (setq *Nodes* (Copy-Replace-Node Node (Node-ID Node) *Nodes*))
         (setq Node
               (Make-Node :Time (Node-Time Node)
                          :ID (Node-ID Node)
                          :Segments (Clear-Hash-Table (Node-Segments Node))
                          :Nodals (Node-Nodals Node)))
         (setq *Nodes* (Copy-Replace-Node Node (Node-ID Node) *Nodes*))
         (setq Node (Make-Node :Time (Node-Time Node)
                               :ID (Node-ID Node)
                               :Segments (Hash-Insert (Node-Segments Node)
                                                      Nodals-ID
                                                      Nodals)
                               :Nodals (Node-Nodals Node)))
         (setq *Nodes* (Copy-Replace-Node Node (Node-ID Node) *Nodes*))
         (let* ((Data (Segment-Data Nodals))
                (Index 0)
                (Data-Size (array-total-size Data)))
           (Clear-Nodes-2 Data Index Data-Size))
         (setq Nodes-Index (1+ Nodes-Index))
         (Clear-Nodes-1 Node Nodes-Index Nodals-Id Nodals End-Index))))
```

```
(defun Clear-Nodes-2 (Data Index Data-Size)
  (cond ((not (< Index Data-Size))
         nil)
        (t
         (setq Data (Copy-Replace-Elt 'UNBOUND Index Data))
         (setq Index (1+ Index))
         (Clear-Nodes-2 Data Index Data-Size))))

;;=============================================================
;;                          Segments

;; This adds a segment to the node's segment translations.  It
;; returns the unique segment ID.

(defun Add-Segment (Segment Node)
  (let* ((Segment-ID (gensym "Segment-"))
         (New-Segments
          (Hash-Insert (Node-Segments Node)
                       Segment-ID
                       Segment))
         (New-Node
           (Make-Node :Time (Node-Time Node)
                      :ID (Node-ID Node)
                      :Segments New-Segments
                      :Nodals (Node-Nodals Node))))
    (values Segment-ID New-Node)))

;; This removes a segment ID from the node's segment
;; translations.

(defun Delete-Segment (Segment-ID Node)
  (let* ((New-Segments
            (Hash-Delete (Node-Segments Node)
                         Segment-ID))
         (New-Node (Make-Node :Time (Node-Time Node)
                              :ID (Node-ID Node)
                              :Segments New-Segments
                              :Nodals (Node-Nodals Node))))
    New-Node))

;; This translates a segment ID to a segment on the specified
;; task's node.

(defun Translate-Segment (Segment-ID Task)
  (Translate-Segment-On-Node Segment-ID
                             (Task-Node Task)))

;; This translates a segment ID on a specified node.

(defun Translate-Segment-On-Node (Segment-ID Node)
  (let ((Segment (Hash-Lookup (Node-Segments Node)
                  Segment-ID)))
    (if (null Segment)
      (break "PiSim error: missing segment")
      Segment)))

;; This function creates a read-write segment.

(defun Create-Read-Write-Segment (Size)
  (Make-Segment :Size Size
                :Type 'Read-Write
                :Data (make-array Size)))

;; This function creates an associative set segment.

(defun Create-Associative-Set-Segment (Size)
  (Make-Segment :Size Size
                :Type 'Associative-Set
                :Data (Make-Hash-Table Size)))

;; This function creates a cache segment.

(defun Create-Cache-Segment (Size)
  (Make-Segment :Size Size
                :Type 'Cache
                :Data (make-array Size)))

;; This function reads a read-write segment.

(defun Read-Segment (Segment Offset)
  (unless (equal (Segment-Type Segment)
                 'Read-Write)
    (break
     "PiSim error: incorrect access operation for segment type"))
  (aref (Segment-Data Segment) Offset))

;; This function writes a read-write segment.

(defun Write-Segment (Segment Offset New-Value)
  (unless (equal (Segment-Type Segment)
                 'Read-Write)
    (break
     "PiSim error: incorrect access operation for segment type"))
  (values New-Value
          (Make-Segment :Size (Segment-Size Segment)
                        :Type (Segment-Type Segment)
                        :Data (Copy-Replace-Elt New-Value
                                                Offset
                                                (Segment-Data Segment)))))

;; This function attempts to match a key in an associative set or cache
;; segment.

(defun Match-Segment (Segment Key)
  (case (Segment-Type Segment)
    (Associative-Set
      (Hash-Lookup (Segment-Data Segment) Key))
    (Cache
      (Match-Cache Key Segment))
    (otherwise
      (break "PiSim error: incorrect access operation for segment type"))))

;; This function inserts a key in an associative set or cache segment.

(defun Insert-Segment (Segment Key New-Value)
  (case (Segment-Type Segment)
    (Associative-Set
      (values
        (Make-Segment :Type (Segment-Type Segment)
                      :Data (Hash-Insert (Segment-Data Segment)
                                         Key
                                         New-Value)
                      :Size (Segment-Size Segment))
        New-Value))
    (Cache
      (Insert-Cache Key Segment New-Value))
    (otherwise
      (break "PiSim error: incorrect access operation for segment type"))))

;; This function removes a key from an associative set or cache segment.

(defun Remove-Key-Segment (Segment Key)
  (case (Segment-Type Segment)
    (Associative-Set
      (Make-Segment :Type (Segment-Type Segment)
                    :Data (Hash-Delete (Segment-Data Segment) Key)
                    :Size (Segment-Size Segment)))
    (Cache (Remove-Key-Cache Key Segment))
    (otherwise
      (break "PiSim error: incorrect access operation for segment type"))))

;; This function clears an associative set or cache segment.

(defun Clear-Segment (Segment)
  (case (Segment-Type Segment)
    (Associative-Set
      (Make-Segment :Type (Segment-Type Segment)
                    :Data (Clear-Hash-Table (Segment-Data Segment))
                    :Size (Segment-Size Segment)))
    (Cache
      (Clear-Cache Segment))
    (otherwise
      (break "PiSim error: incorrect access operation for segment type"))))

;;=============================================================
;;                             Caches
;;
;; In PiSim, caches are implemented as direct mapped arrays.  A hash
;; function computes an index into an array.  Array entries are cons
;; cells are of the format: (Key . Value).

;; This is the hash function for caches.

(defun Cache-Hash (Key Size)
  (when (numberp Key)
    (setq Key (format nil "~a" Key)))
  (let* ((String (string Key))
         (Character nil)
         (Value 0)
         (Index 0)
         (End-Index (array-total-size String)))
    (Cache-Hash-1 String Character Value Size Index End-Index)))

(defun Cache-Hash-1 (String Character Value Size Index End-Index)
  (cond ((not (< Index End-Index))
         (mod Value Size))
        (t
         (setq Character (aref String Index))
         (setq Value (+ (char-int Character) Value))
         (setq Index (1+ Index))
         (Cache-Hash-1 String Character Value Size Index End-Index))))

;; This function attempts to match a key in a hash table.  If the key
;; is found, the corresponding value is returned.  Otherwise, 'Miss is
;; returned.
```

```
(defun Match-Cache (Key Segment)
  (let* ((Index (Cache-Hash Key (Segment-Size Segment)))
         (Entry (aref (Segment-Data Segment) Index)))
    (if (and (not (equal Entry 'Empty))
             (equal (first Entry) Key))
      (rest Entry)
      'Miss)))

;; This function writes an entry in the cache, possibly
;; overwriting another value.

(defun Insert-Cache (Key Segment New-Value)
  (let* ((Value (cons Key New-Value))
       (New-Segment-Data
         (Copy-Replace-Elt Value
                           (Cache-Hash Key
                                       (Segment-Size Segment))
                           (Segment-Data Segment))))
    (values (Make-Segment :Type (Segment-Type Segment)
                          :Data New-Segment-Data
                          :Size (Segment-Size Segment))
          Value)))

;; This function removes a key from a cache.  If the key is not
;; present, no action is taken.

(defun Remove-Key-Cache (Key Segment)
  (let* ((Index (Cache-Hash Key (Segment-Size Segment)))
       (Entry (aref (Segment-Data Segment) Index)))
    (if (and (not (equal Entry 'Empty))
           (equal (first Entry) Key))
      (values
        (Make-Segment :Type (Segment-Type Segment)
                      :Data (Copy-Replace-Elt 'Empty
                                              Index
                                              (Segment-Data
                                               Segment))
                      :Size (Segment-Size Segment))
        'Empty)
      (values Segment nil))))

;; This function clears a cache.

(defun Clear-Cache (Segment)
  (let* ((Data (Segment-Data Segment))
       (Index 0)
       (End-Index (array-total-size Data)))
    (Clear-Cache-1 Data Index End-Index Segment)))

(defun Clear-Cache-1 (Data Index End-Index Segment)
  (cond ((not (< Index End-Index))
       Segment)
      (t
        (setq Data (Copy-Replace-Elt 'EMPTY Index Data))
       (setq Segment (Make-Segment :Type (Segment-Type Segment)
                                   :Data Data
                                   :Size (Segment-Size Segment)))
       (setq Index (1+ Index))
       (Clear-Cache-1 Data Index End-Index Segment))))

;;=============================================================
;;                              Tasks

;;; This returns the node ID of the specified task's nodes.

(defun Node-Of (Task)
  (Node-ID (Task-Node Task)))

;;; This returns the time of a task.  This is defined as the node
;;; time for the specified task.

(defun Time-Of (Task)
  (Node-Time (Task-Node Task)))

;;; This sets the time of the specified task (i.e. the time of
;;; the node of the specified task).

(defun Set-Time-Of (Task New-Time)
  (let ((Task-Node (Task-Node Task)))
    (setq Task-Node (Make-Node :Time New-Time
                               :ID (Node-ID Task-Node)
                               :Segments (Node-Segments Task-Node)
                               :Nodals (Node-Nodals Task-Node)))
    (values New-Time
            Task-Node
            (Make-Task :Handler (Task-Handler Task)
                       :Node Task-Node
                       :Segment (Task-Segment Task)
                       :IP (Task-IP Task)
                       :Status (Task-Status Task)))))

;;; This increments the task time by the specified delta.

(defun Increment-Time-Of (Task Delta)
  (let* ((Task-Node (Task-Node Task))
         (New-Time (+ (Node-Time Task-Node) Delta)))
    (setq Task-Node (Make-Node :Time New-Time
                               :ID (Node-ID Task-Node)
                               :Segments (Node-Segments Task-Node)
                               :Nodals (Node-Nodals Task-Node)))
    (values New-Time
            Task-Node
            (Make-Task :Handler (Task-Handler Task)
                       :Node Task-Node
                       :Segment (Task-Segment Task)
                       :IP (Task-IP Task)
                       :Status (Task-Status Task)))))

;; This returns the handler type of the task.

(defun Handler-Name-Of (Task)
  (Handler-Name (Task-Handler Task)))

;;; This function creates a new task segment of the specified length.
;;; The number of arguments and message length values are compared with
;;; the handler arity and arity plus number of locals respectively.  Two
;;; is added to the arity and number of locals to account for the message
;;; length and type information stored in the segment.  The segment is
;;; then initializes with the supplied arguments.

(defun Write-Arguments (Arguments Index New-Segment)
  (cond ((null Arguments)
         New-Segment)
        (t
         (multiple-value-bind (New-Value Written-Segment)
             (Write-Segment New-Segment Index (car Arguments))
           (Write-Arguments (cdr Arguments)
                            (1+ Index)
                            Written-Segment)))))

(defun Create-Task-Segment (Length Task-Type Arguments Handler)
  (let ((New-Segment (Create-Read-Write-Segment Length)))
    (when (not (= (Handler-Arity Handler)
                  (length Arguments)))
      (break "PiSim error: arity mismatch"))
    (when (not (= Length (+ (Handler-Arity Handler)
                            (Handler-Number-Of-Locals Handler)
                            2)))
      (break "PiSim error: length/ handler storage mismatch"))
    (Make-Task-Segment
      :Storage-Rqmts Length
      :Type Task-Type
      :Arguments (Write-Arguments Arguments 2 New-Segment))))

;;; This function creates a new task for a message.  The handler and
;;; node are determined.  A new segment is created and initialized.
;;; After the new task is created, its segment is added to the task's
;;; node.  Finally the new task is returned.

(defun Create-Task (Message)
  (let* ((Handler (Get-Handler (Message-Type Message)))
         (Node (Translate-Node (Message-Destination Message))))
    (Make-Task :Handler Handler
               :Node Node)))

;;; This function executes a task.  It executes instructions which
;;; change a task's status.  If the status is 'Running, another
;;; instruction is executed.

(defun Execute-Task (Task)
  (multiple-value-bind (Value New-Task)
      (Execute-Next-Instruction Task)
    (setq Task New-Task))
  (if (equal (Task-Status Task) 'Running)
      (Execute-Task Task)))


;;=============================================================
;;                              Events
;;
;; This function enqueues an event in the global event queue.
;; Events are enqueued in order on increasing event time.
;; ** Note that when 2 events have the same time, the one sent to
;; Enqueue-Event first has higher priority.

(defun Enqueue-Event (New-Event)
  (if (or (null *Event-Queue*)
          (< (Event-Time New-Event)
             (Event-Time (first *Event-Queue*))))
      (setq *Event-Queue*
            (cons New-Event *Event-Queue*))
      (setq *Event-Queue*
            (Insert-Event New-Event *Event-Queue*))))

;; This function is used to enqueue events inside the event queue.
;; It is part of a recursive, priority queue insert algorithm.
```

```
(defun Insert-Event (New-Event Event-Queue)
  (if (or (null (rest Event-Queue))
          (< (Event-Time New-Event)
             (Event-Time (second Event-Queue))))
      (cons (car Event-Queue)
            (cons New-Event (rest Event-Queue)))
      (cons (car Event-Queue)
          (Insert-Event New-Event (rest Event-Queue)))))

;; This function dequeues and returns a event from the global
;; event queue.  If the queue is empty, nil is returned.

(defun Dequeue-Event ()
  (let ((Event (car *Event-Queue*)))
    (setq *Event-Queue* (cdr *Event-Queue*))
    Event))

;; This function clears the event queue.

(defun Clear-Event-Queue ()
  (setq *Event-Queue* nil))

;; This function dequeues and executes the next event in the
;; event queue.  If the event is a message, a new task is
;; created.  The node time is adjusted if the event time is
;; later than node time.  If a event is executed, t is returned.

(defun Execute-Next-Event ()
  (let* ((Event (Dequeue-Event))
         Task)
    (setq Task (Create-Task (Event-Object Event)))
    (multiple-value-bind (New-Time Task-Node New-Task)
        (Set-Time-Of Task
                     (if (> (Event-Time Event)
                            (Time-Of Task))
                         (Event-Time Event)
                         (Time-Of Task)))
      (setq *Nodes*
            (Copy-Replace-Node
             Task-Node
             (Translate-Node
               (Message-Destination (Event-Object Event)))
             *Nodes*))
      (setq Task New-Task))
    (let* ((Message (Event-Object Event))
           (Node (Translate-Node (Message-Destination Message)))
           (New-Segment (Create-Task-Segment
                         (Message-Length Message)
                         (Message-Type Message)
                         (Message-Arguments Message)
                         (Task-Handler Task))))
      (multiple-value-bind (New-Segment-ID New-Node)
          (Add-Segment New-Segment Node)
        (setq Node New-Node)
        (setq *Nodes* (Copy-Replace-Node
                       Node
                       (Message-Destination Message)
                       *Nodes*))
        (setq Task (Make-Task :Handler (Task-Handler Task)
                              :Node Node
                              :Segment New-Segment-ID
                              :IP (Task-IP Task)
                              :Status (Task-Status Task)))))
    (Debug-Print 1
      "[start: task ~a node ~d time ~d old status ~a]~&"
      (Handler-Name-Of Task) (Node-Of Task)
      (Time-Of Task) (Task-Status Task))
    (Log-Task Task)
    (setq Task
        (Make-Task :Handler (Task-Handler Task)
                   :Node (Task-Node Task)
                   :Segment (Task-Segment Task)
                   :IP (Task-IP Task)
                   :Status 'Running))
    (Adjust-Concurrency-List (Time-Of Task) 1)
    (Execute-Task Task)
    (Adjust-Concurrency-List (Time-Of Task) -1)
    (Debug-Print 1 "[stop: task ~a node ~d time ~d status ~a]~&"
                 (Handler-Name-Of Task) (Node-Of Task)
                 (Time-Of Task) (Task-Status Task))))

;;=============================================================
;;                          Handlers

;; This predicate tests if a statement is an instruction.

(defun Label? (Statement)
  (symbolp Statement))

;; This predicate tests if a statement is an instruction.

(defun Instruction? (Statement)
  (listp Statement))

;; This function inserts a binding into a handler's bindings.  If the
;; specified handler is 'Global, the binding is inserted in the global
;; bindings.

(defun Insert-Binding (Name Value Handler)
  (cond ((equal Handler 'Global)
         (setq *Global-Bindings*
               (Hash-Insert *Global-Bindings* Name Value))
         (values Value Handler))
        (t
         (setq Handler
               (Make-Handler :Name (Handler-Name Handler)
                             :Instructions (Handler-Instructions Handler)
                             :Arity (Handler-Arity Handler)
                             :Number-of-Locals
                             (Handler-Number-of-Locals Handler)
                             :Bindings
                             (Hash-Insert (Handler-Bindings Handler)
                                          Name
                                          Value)))
         (values Value Handler))))

;; This function looks up the binding of a symbol in the handler.  If
;; it is not found there, the global bindings are checked.

(defun Lookup-Binding (Name Handler)
  (or (Hash-Lookup (Handler-Bindings Handler) Name)
      (Hash-Lookup *Global-Bindings* Name)))

;; This function returns the number of instructions in a handler.

(defun Number-Of-Instructions (Handler)
  (array-total-size (Handler-Instructions Handler)))

;; This function returns the handler object for the handler name.  If
;; the handler does not exist, an error message is printed.

(defun Get-Handler (Name)
  (let ((Handler (get Name 'Handler)))
    (if (null Handler)
        (break "PiSim error: unknown handler")
        Handler)))

;; This function determines the number of instructions in a sequence
;; of statements and builds a instruction array of the correct size.
;; It then reads each statement.  If it is an instruction, it is
;; inserted into the array.  If it is a label, the label and
;; statement index is inserted into the handler's bindings.

(defun Make-Instructions (Statements Handler)
  (let (Instructions)
    (let ((Temp-Stmts Statements)
          (Statement nil)
          (Number-Of-Statements 0))
      (setq Instructions
            (Make-Instructions-1 Instructions Temp-Stmts Statement
                                 Number-of-Statements)))
    (let ((Index 0)
          (Statement nil)
          (Temp-Stmts Statements))
      (multiple-value-bind (Instructions New-Handler)
          (Make-Instructions-2 Instructions Temp-Stmts Statement
                               Index Handler)
        (setq Handler New-Handler))
      (setq Handler
            (Make-Handler :Name (Handler-Name Handler)
                          :Instructions Instructions
                          :Arity (Handler-Arity Handler)
                          :Number-of-Locals (Handler-Number-of-Locals
                                             Handler)
                          :Bindings (Handler-Bindings Handler)))
      (values Instructions Handler))))

(defun Make-Instructions-1 (Instructions Temp-Stmts Statement
                            Number-of-Statements)
  (cond ((null Temp-Stmts)
         (setq Instructions (make-array Number-Of-Statements)))
        (t
          (setq Statement (car Temp-Stmts))
          (setq Temp-Stmts (cdr Temp-Stmts))
          (cond ((not (Label? Statement))
                 (if Statement
                     (setq Number-Of-Statements
                           (1+ Number-Of-Statements)))))
          (Make-Instructions-1 Instructions Temp-Stmts Statement
                               Number-of-Statements))))

(defun Make-Instructions-2 (Instructions Temp-Stmts Statement Index Handler)
  (cond ((null Temp-Stmts)
         (values Instructions Handler))
        (t (setq Statement (car Temp-Stmts))
           (setq Temp-Stmts (cdr Temp-Stmts))
           (cond ((Label? Statement)
                  (multiple-value-bind (Value New-Handler)
```

```
                (Insert-Binding Statement Index Handler)
                (setq Handler New-Handler)))
            ((Instruction? Statement)
             (progn
               (setq Instructions
                     (Copy-Replace-Elt
                      Statement Index Instructions))
               (setq Index (1+ Index)))))
          (Make-Instructions-2
           Instructions Temp-Stmts Statement Index Handler))))

;; This function indexes the parameters and locals in a handler.
;; This includes assigning a each parameter and value an index
;; in the handler segment.  These assignments are included in
;; the handler's bindings.  The arity and number of locals
;; parameters are also set.

(defun Index-Parameters-And-Locals (Parameters Locals Handler)
  (let ((Parameter nil)
        (Temp-Parameters Parameters)
        (Index 2))
    (setq Handler
          (Index-Parameters-And-Locals-1
           Parameter Temp-Parameters
           Index Handler)))
  (let ((Local nil)
        (Temp-Locals Locals)
        (Index (+ (length Parameters) 2)))
    (setq Handler
          (Index-Parameters-And-Locals-2 Local Temp-Locals Index
                                         Handler)))
  (setq Handler (Make-Handler :Name (Handler-Name Handler)
                              :Instructions (Handler-Instructions
                                             Handler)
                              :Arity (length Parameters)
                              :Number-of-Locals
                              (Handler-Number-of-Locals Handler)
                              :Bindings (Handler-Bindings
                                         Handler)))
  (setq Handler (Make-Handler :Name (Handler-Name Handler)
                              :Instructions (Handler-Instructions
                                             Handler)
                              :Arity (Handler-Arity Handler)
                              :Number-of-Locals (length Locals)
                              :Bindings (Handler-Bindings
                                         Handler)))
  Handler)

(defun Index-Parameters-And-Locals-1 (Parameter Temp-Parameters
                                      Index Handler)
  (cond ((null Temp-Parameters) Handler)
        (t
          (setq Parameter (car Temp-Parameters))
          (setq Temp-Parameters (cdr Temp-Parameters))
        (multiple-value-bind (Value New-Handler)
            (Insert-Binding Parameter Index Handler)
          (setq Handler New-Handler))
          (setq Index (1+ Index))
        (Index-Parameters-And-Locals-1 Parameter Temp-Parameters
                                       Index Handler))))

(defun Index-Parameters-And-Locals-2 (Local Temp-Locals
                                      Index Handler)
  (cond ((null Temp-Locals) Handler)
        (t
          (setq Local (car Temp-Locals))
          (setq Temp-Locals (cdr Temp-Locals))
        (multiple-value-bind (Value New-Handler)
            (Insert-Binding Local Index Handler)
          (setq Handler New-Handler))
          (setq Index (1+ Index))
        (Index-Parameters-And-Locals-2 Local Temp-Locals Index
                                       Handler))))

;; This function reads a handler from an expression.  The
;; resultant handler is stored on the property list of the
;; handler name.

(defun Read-Handler (Expression)
  (let ((Name (first Expression))
        (Parameters (second Expression))
        (Locals (third Expression))
        (Statements (nthcdr 3 Expression))
        (New-Handler (Make-Handler)))
    (setq New-Handler
          (Make-Handler :Name Name
                        :Instructions (Handler-Instructions
                                       New-Handler)
                        :Arity (Handler-Arity New-Handler)
                        :Number-of-Locals
                        (Handler-Number-of-Locals New-Handler)
                        :Bindings (Handler-Bindings New-Handler)))
    (setq New-Handler
          (Index-Parameters-And-Locals Parameters Locals New-Handler))
    (multiple-value-bind (Instructions Newer-Handler)
        (Make-Instructions Statements New-Handler)
      (setq New-Handler Newer-Handler))
    (setq *Global-Plist*
          (Update-Plist Name 'Handler New-Handler))))

;; This allows the definition of handlers.  This should be part
;; of a more general reader.

(defun Define-Handler (&rest Expression)
  (Debug-Print 0 "~&loading handler ~a~&" (first Expression))
  (Read-Handler Expression)
  nil)

;;============================================================
;;                              Nodals

;; This allows the definition of nodals (node variables).  An
;; index is assigned (using the number of existing nodals).  A new
;; global binding is added.

(defun Define-Nodal (Name)
  (Debug-Print 0 "~&defining nodal ~a~&" Name)
  (cond ((not (null (Hash-Lookup *Global-Bindings* Name)))
         (format t "~&Warning: ~a has already been defined globally~&"
                 Name))
        (t
         (multiple-value-bind (Value Handler)
             (Insert-Binding Name *Nodal-Count* 'Global))
         (setq *Nodal-Count* (1+ *Nodal-Count*)))))

;;============================================================
;;                              Constants

;; This allows the definition of global constants.  The binding
;; is added to the global bindings.

(defun Define-Constant (Name Value)
  (Debug-Print 0 "~&defining constant ~a~&" Name)
  (multiple-value-bind (Value Handler)
      (Insert-Binding Name Value 'Global)))

;;============================================================
;;                                Instructions

;; This function returns the next instruction of the handler to be
;; executed.  The current instruction pointer (IP) is obtained from
;; the task.  The instructions are obtained from the handler.  The
;; task instruction pointer is incremented.  Note: the instruction
;; pointer is incremented AFTER the next instruction is fetched.

(defun Next-Instruction (Task)
  (let ((IP (Task-IP Task)))
    (when (>= IP
              (Number-Of-Instructions (Task-Handler Task)))
      (break "PiSim error: IP out of range"))
    (setq Task (Make-Task :Handler (Task-Handler Task)
                          :Node (Task-Node Task)
                          :Segment (Task-Segment Task)
                          :IP (1+ (Task-IP Task))
                          :Status (Task-Status Task)))
    (values (aref (Handler-Instructions (Task-Handler Task))
                  IP)
            Task)))

;; This function executes a single instructions.  It first locates the
;; next instruction using the task instruction pointer.  The
;; instruction pointer is incremented. Then it applies the operation
;; to the arguments.

(defun Execute-Next-Instruction (Active-Task)
  (multiple-value-bind (Instruction New-Task)
      (Next-Instruction Active-Task)
    (setq Active-Task New-Task)
    (Debug-Print 2 "   [executing instruction ~a]~&"
                 (Instruction-Op Instruction))
    (Log-Instruction Instruction)
    (multiple-value-bind (Value New-Task)
        (Apply-Operation (Instruction-Op Instruction)
                         Active-Task
                         (Instruction-Args Instruction))
     (setq Active-Task New-Task)
     (values Value Active-Task))))

;;============================================================
;;                              Operations

;; This function applies a processor operation to a list of arguments.
;; Each argument is evaluated before the operation is applied.  The
;; apply only takes place if the task status is 'RUNNING.

(defun Apply-Operation (Operation Active-Task Arguments)
```

```
  (multiple-value-bind (Argument-List New-Nodes
                        New-Task New-Event-Queue)
      (Evaluate-Arguments Arguments Active-Task)
    (setq *Nodes* New-Nodes
        Active-Task New-Task
        *Event-Queue* New-Event-Queue)
    (cond ((equal (Task-Status Active-Task)
                  'RUNNING)
           (Log-Operation Operation)
          (multiple-value-bind (Result New-Nodes New-Task
                                New-Event-Queue)
              (apply (Get-Operation Operation)
                     Argument-List
                     *Nodes*
                     Active-Task
                     *Event-Queue*)
            (setq Active-Task New-Task
                  *Nodes* New-Nodes
                  *Event-Queue* New-Event-Queue)
            (values Result Active-Task)))
          (t (values nil Active-Task)))))

(defun Evaluate-Arguments (Arguments Active-Task)
  (let ((Argument nil))
    (Evaluate-Arguments-1 Argument Arguments *Nodes*
                          Active-Task *Event-Queue*)))

(defun Evaluate-Arguments-1 (Argument Arguments Nodes
                                      Active-Task Event-Queue)
  (cond ((null Arguments)
         (values nil Nodes Active-Task Event-Queue))
        (t
         (setq Argument (car Arguments))
         (setq Arguments (cdr Arguments))
         (multiple-value-bind (Value New-Nodes New-Task
                               New-Event-Queue)
             (Evaluate Active-Task Argument)
           (multiple-value-bind (Argument-List Newer-Nodes
                                 Newer-Task Newer-Event-Queue)
               (Evaluate-Arguments-1 Argument Arguments New-Nodes
                                     New-Task New-Event-Queue)
             (values (cons Value Argument-List)
                     Newer-Nodes Newer-Task Newer-Event-Queue)))))))

;; This function evaluates the expression and returns the
;; results.  This is an evaluator appropriate for the limited
;; expressions in a Pi programs.  Expressions are only evaluated
;; if the task status is 'RUNNING. The following expression
;; types are possible:
;;
;; A number or string returns the value of the number or string.
;;
;; A symbol is looked up in the handler bindings.  If it is
;; present, the corresponding value is returned.  Otherwise, the
;; symbol is returned.
;;
;; A nested expression (a list) in the form (symbol arg1 arg2..).
;; In this case, Apply-Operation is recursively called.

(defun Evaluate (Active-Task Expression)
  (when (equal (Task-Status Active-Task)
               'RUNNING)
    (values
      (typecase Expression
      ((or number string)
       Expression)
      (symbol
       (or (Lookup-Binding Expression (Task-Handler Active-Task))
           Expression))
      (list
       (multiple-value-bind (Value New-Task)
           (Apply-Operation (Instruction-Op Expression)
                            Active-Task
                            (Instruction-Args Expression))
         (setq Active-Task New-Task)
         Value))
      (otherwise
       (break "PiSim error: unknown expression")))
      Active-Task)))

;; This function returns the operation function for the operation
;; name.  If the operation does not exist, an error message is
;; printed.

(defun Get-Operation (Name)
  (let ((Operation (get Name 'Operation)))
    (if (null Operation)
        (break "PiSim error: unknown operation")
      Operation)))

;; This is used to define processor operations.

(defmacro Define-Operation (Name &rest Rest)
  (setq *Global-Plist*
        '(Update-Plist ,Name 'Operation #'(lambda ,@Rest))))

;;============================================================
;;                          Debugging

;; This prints debug messages depending on the debug level.

(defmacro Debug-Print (Level Format &rest Arguments)
  '(when (<= ,Level *Debug-Level*)
     (format t ,Format ,@Arguments)))

;; This function sets the debug level.

(defun Set-Debug-Level (New-Level)
  (setq *Debug-Level* New-Level))

;;============================================================
;;                          Logging

;; This predicate starts a new log, saving the current log.

(defun Start-New-Log ()
  (setq *Log* (Make-Log :Type (Log-Type *Log*)
                        :Old-Logs *Log*)))

;; This is used in a counting profile.  The category count is
;; incremented, or created, if non-existent.

(defun Collect-Profile (Category Profile)
  (cond ((Hash-Lookup Profile Category)
         (let ((New-Value (1+ (Hash-Lookup Profile Category))))
           (setq Profile
                 (Hash-Insert Profile Category New-Value))
           (values New-Value Profile)))
        (t
         (values 1 (Hash-Insert Profile Category 1)))))

;; This predicate tests if logging is enabled.  If the log is nil,
;; logging is on.

(defun Logging? ()
  (not (or (null *log*)
           (equal (Log-Type *Log*) 'None))))

;; This function logs the specified task. Presently, profiles of task
;; types and status' are maintained.

(defun Log-Task (Task)
  (when (Logging?)
    (multiple-value-bind (New-Value New-Profile)
        (Collect-Profile (Task-Status Task)
                         (Log-Task-Status-Profile *Log*))
      (setq *Log*
            (Make-Log :Type (Log-Type *Log*)
                      :Task-Status-Profile New-Profile
                      :Task-Type-Profile (Log-Task-Type-Profile *Log*)
                      :Instruction-Type-Profile
                      (Log-Instruction-Type-Profile *Log*)
                      :Operation-Type-Profile
                      (Log-Operation-Type-Profile *Log*)
                      :Concurrency-List (Log-Concurrency-List *Log*)
                      :Old-Logs (Log-Old-Logs *Log*)))
      (when (equal (Task-Status Task) 'New)
        (multiple-value-bind (New-Value New-Profile)
            (Collect-Profile (Handler-Name-Of Task)
                             (Log-Task-Type-Profile *Log*))
          (setq *Log*
                (Make-Log :Type (Log-Type *Log*)
                          :Task-Status-Profile
                          (Log-Task-Status-Profile *Log*)
                          :Task-Type-Profile New-Profile
                          :Instruction-Type-Profile
                          (Log-Instruction-Type-Profile *Log*)
                          :Operation-Type-Profile
                          (Log-Operation-Type-Profile *Log*)
                          :Concurrency-List (Log-Concurrency-List *Log*)
                          :Old-Logs (Log-Old-Logs *Log*))))))))

;; This function collects statistics on instruction types.

(defun Log-Instruction (Instruction)
  (when (Logging?)
    (cond ((not (equal (first Instruction) 'Write))
           (multiple-value-bind (New-Value New-Profile)
               (Collect-Profile (first Instruction)
                                (Log-Instruction-Type-Profile *Log*))
             (setq *Log*
                   (Make-Log :Type (Log-Type *Log*)
                             :Task-Status-Profile
                             (Log-Task-Status-Profile *Log*)
                             :Task-Type-Profile (Log-Task-Type-Profile
                                                 *Log*)
```

```
                        :Instruction-Type-Profile New-Profile
                        :Operation-Type-Profile
                        (Log-Operation-Type-Profile *Log*)
                        :Concurrency-List
                        (Log-Concurrency-List *Log*)
                        :Old-Logs (Log-Old-Logs *Log*)))))
        ((not (listp (fourth Instruction)))
         (multiple-value-bind (New-Value New-Profile)
             (Collect-Profile 'Initialize
                              (Log-Instruction-Type-Profile
                               *Log*))
             (setq *Log*
                 (Make-Log :Type (Log-Type *Log*)
                           :Task-Status-Profile
                           (Log-Task-Status-Profile *Log*)
                           :Task-Type-Profile
                           (Log-Task-Type-Profile *Log*)
                           :Instruction-Type-Profile New-Profile
                           :Operation-Type-Profile
                           (Log-Operation-Type-Profile *Log*)
                           :Concurrency-List
                           (Log-Concurrency-List *Log*)
                           :Old-Logs (Log-Old-Logs *Log*)))))
        ((equal (first (fourth Instruction)) 'Read)
         (multiple-value-bind (New-Value New-Profile)
             (Collect-Profile
              'Move (Log-Instruction-Type-Profile *Log*))
             (setq *Log*
                 (Make-Log :Type (Log-Type *Log*)
                           :Task-Status-Profile
                           (Log-Task-Status-Profile *Log*)
                           :Task-Type-Profile
                           (Log-Task-Type-Profile *Log*)
                           :Instruction-Type-Profile New-Profile
                           :Operation-Type-Profile
                           (Log-Operation-Type-Profile *Log*)
                           :Concurrency-List
                           (Log-Concurrency-List *Log*)
                           :Old-Logs (Log-Old-Logs *Log*)))))
        (t
         (multiple-value-bind (New-Value New-Profile)
             (Collect-Profile (first (fourth Instruction))
                              (Log-Instruction-Type-Profile
                               *Log*))
             (setq *Log*
                 (Make-Log :Type (Log-Type *Log*)
                           :Task-Status-Profile
                           (Log-Task-Status-Profile *Log*)
                           :Task-Type-Profile
                           (Log-Task-Type-Profile *Log*)
                           :Instruction-Type-Profile New-Profile
                           :Operation-Type-Profile
                           (Log-Operation-Type-Profile *Log*)
                           :Concurrency-List
                           (Log-Concurrency-List *Log*)
                           :Old-Logs (Log-Old-Logs *Log*)))))))))

;; This function creates an operation profile.

(defun Log-Operation (Operation)
  (when (Logging?)
    (multiple-value-bind (New-Value New-Profile)
        (Collect-Profile Operation
                         (Log-Operation-Type-Profile *Log*))
      (setq *Log*
            (Make-Log :Type (Log-Type *Log*)
                      :Task-Status-Profile
                      (Log-Task-Status-Profile *Log*)
                      :Task-Type-Profile
                      (Log-Task-Type-Profile *Log*)
                      :Instruction-Type-Profile New-Profile
                      :Operation-Type-Profile
                      (Log-Operation-Type-Profile *Log*)
                      :Concurrency-List
                      (Log-Concurrency-List *Log*)
                      :Old-Logs (Log-Old-Logs *Log*)))))))

;; This function searches down a sorted list of deltas looking
;; for an entry at a specified time.  If such an entry is found,
;; its value is adjusted by Change.  If no such value is found,
;; a new delta is created an inserted at the correct position in
;; the list.

(defun Adjust-Concurrency-List (Time Change)
  (when (Logging?)
    (let ((Concurrency-List (Log-Concurrency-List *Log*)))
      (cond ((or (null Concurrency-List)
                 (< Time (Delta-Time (first Concurrency-List))))
             (let ((New-Delta (Make-Delta :Time Time
                                          :Value Change)))
               (setq *Log*
                     (Make-Log :Type (Log-Type *Log*)
                               :Task-Status-Profile
                               (Log-Task-Status-Profile *Log*)
                               :Task-Type-Profile
                               (Log-Task-Type-Profile *Log*)
                               :Instruction-Type-Profile
                               (Log-Instruction-Type-Profile *Log*)
                               :Operation-Type-Profile
                               (Log-Operation-Type-Profile *Log*)
                               :Concurrency-List
                               (cons New-Delta
                                     (Log-Concurrency-List *Log*))
                               :Old-Logs (Log-Old-Logs *Log*)))
               New-Delta))
            ((= Time (Delta-Time (first Concurrency-List)))
             (let* ((First-Delta (first Concurrency-List))
                    (New-Delta
                      (Make-Delta :Time (Delta-Time First-Delta)
                                  :Value (+ (Delta-Value First-Delta)
                                            Change))))
               (setq *Log*
                     (Make-Log :Type (Log-Type *Log*)
                               :Task-Status-Profile
                               (Log-Task-Status-Profile *Log*)
                               :Task-Type-Profile
                               (Log-Task-Type-Profile *Log*)
                               :Instruction-Type-Profile
                               (Log-Instruction-Type-Profile *Log*)
                               :Operation-Type-Profile
                               (Log-Operation-Type-Profile *Log*)
                               :Concurrency-List
                               (cons New-Delta
                                     (cdr (Log-Concurrency-List *Log*)))
                               :Old-Logs (Log-Old-Logs *Log*)))
               (Delta-Value New-Delta)))
            (t
               (setq *Log*
                     (Make-Log :Type (Log-Type *Log*)
                               :Task-Status-Profile
                               (Log-Task-Status-Profile *Log*)
                               :Task-Type-Profile
                               (Log-Task-Type-Profile *Log*)
                               :Instruction-Type-Profile
                               (Log-Instruction-Type-Profile *Log*)
                               :Operation-Type-Profile
                               (Log-Operation-Type-Profile *Log*)
                               :Concurrency-List
                               (Adjust-Rest-Of-Concurrency-List
                                 Time Change Concurrency-List)
                               :Old-Logs (Log-Old-Logs *Log*)))))))))

;; This is the recursive part of Adjust-Concurrency-List.

(defun Adjust-Rest-Of-Concurrency-List (Time Change Concurrency-List)
  (cond ((or (null (rest Concurrency-List))
             (< Time (Delta-Time (second Concurrency-List))))
         (cons (car Concurrency-List)
               (cons (Make-Delta :Time Time :Value Change)
                     (rest Concurrency-List))))
        ((= Time (Delta-Time (second Concurrency-List)))
         (cons (car Concurrency-List)
               (cons (Make-Delta :Time (Delta-Time
                                        .  (second Concurrency-List))
                                 :Value
                                 (+ (Delta-Value (second Concurrency-List))
                                    Change))
                     (cdr (rest Concurrency-List)))))
        (t
         (cons (car Concurrency-List)
               (Adjust-Rest-Of-Concurrency-List
                 Time Change (rest Concurrency-List))))))

;; This function prints the information from the current log.

(defun Print-Log-Information ()
  (when (or (equal (Log-Type *Log*) 'All)
            (equal (Log-Type *Log*) 'Profile))
    (Print-Profile-Data))
  (when (or (equal (Log-Type *Log*) 'All)
            (equal (Log-Type *Log*) 'Plot))
    (Plot-Concurrency)))

;;=============================================================
;;                              Randoms

;; This function estimates the delivery delay of a message.  It
;; should be better than it is now.

(defun Delivery-Delay (Source Destination Length)
  (when (or (>= Source (Number-Of-Nodes))
            (minusp Source)
            (>= Destination (Number-Of-Nodes))
            (minusp Destination))
    (break "PiSim error: illegal node number"))
  (when (or (minusp Length)
```

```
        (zerop Length))
    (break "PiSim error: illegal message length"))
  (let ((Dimension nil)
      (Temp-Dimensions *Machine-Dimensions*)
      (Source-Components nil)
      (Destination-Components nil))
    (Delivery-Delay-1 Dimension Temp-Dimensions
                      Source-Components Destination-Components
                      Source Destination Length)))

(defun Delivery-Delay-1 (Dimension Temp-Dimensions
                         Source-Components
                         Destination-Components
                         Source Destination Length)
  (cond ((null Temp-Dimensions)
         (let ((Source-Component nil)
               (Destination-Component nil)
               (Distance 0))
           (Delivery-Delay-2
             Source-Component
             Destination-Component Distance Length
             Source-Components Destination-Components)))
        (t
          (setq Dimension (car Temp-Dimensions))
          (setq Temp-Dimensions (cdr Temp-Dimensions))
          (setq Source-Components
                (Put-on-End (mod Source Dimension)
                            Source-Components))
          (setq Source (floor Source Dimension))
          (setq Destination-Components
                (Put-on-End (mod Destination Dimension)
                            Destination-Components))
          (setq Destination (floor Destination Dimension))
          (Delivery-Delay-1
            Dimension Temp-Dimensions Source-Components
            Destination-Components Source Destination
            Length))))

(defun Put-on-End (X List)
  (cond ((null List)
         (list X))
        (t (cons (car List)
                 (Put-on-End X (cdr List))))))

(defun Delivery-Delay-2 (Source-Component Destination-Component
                         Distance Length Source-Components
                         Destination-Components)
  (cond ((null Source-Components)
         (+ Distance (- Length 1)))
        (t
          (setq Source-Component (car Source-Components))
          (setq Source-Components (cdr Source-Components))
          (cond ((null Destination-Components)
                 (+ Distance (- Length 1)))
                (t
                 (setq Destination-Component
                       (car Destination-Components))
                 (setq Destination-Components
                       (cdr Destination-Components))
                 (setq Distance
                       (+ (abs (- Source-Component
                                  Destination-Component))
                          Distance))
                   (Delivery-Delay-2
                   Source-Component Destination-Component
                   Distance Length Source-Components
                   Destination-Components))))))

;; This function injects a starting message into the machine.  It
;; starts calculating the message length and destination.  The
;; message is then enqueued, and events are executed until the
;; event queue is empty.

(defun Inject (Type &rest Arguments)
  (Make-Nodes)
  (Clear-Nodes)
  (Clear-Event-Queue)
  (let* ((Handler (Get-Handler Type))
       (Length (+ (Handler-Arity Handler)
                  (Handler-Number-Of-Locals Handler)
                  2))
       (Destination (random (Number-Of-Nodes)))
       (Arrival-Time (Node-Time (Translate-Node Destination)))
       (Message (Make-Message :Destination Destination
                              :Length Length
                              :Type Type
                              :Arguments Arguments))
       (Event (Make-Event :Time Arrival-Time
                          :Object Message)))
    (Enqueue-Event Event)
    (Execute-Events)))

(defun Execute-Events ()
  (cond ((null *Event-Queue*)
         (values *Event-Queue* *Nodes*))
        (t (Execute-Next-Event)
           (Execute-Events))))

;;============================================================
;;                          Hash Table Functions

(defconstant MIN_HASH_TABLE_SIZE 11)

(defstruct Entry
  (Key   nil :type symbol)
  (Value nil :type any))

(defstruct HashTable
  (Num-Buckets nil :type integer)
  (Number-Entries nil :type integer)
  (Buckets nil :type array))

(defun Hash-Insert (Table Key Value)
  (let* ((Index (Hash-Function Key (HashTable-Num-Buckets Table)))
         (New-Table
           (multiple-value-bind (New-Bucket-List Number-Entries)
               (Splice-In-Bucket Value
                                 Key
                                 (aref (HashTable-Buckets Table) Index)
                                 (HashTable-Number-Entries Table))
             (Make-HashTable
               :Num-Buckets (HashTable-Num-Buckets Table)
               :Buckets (Copy-Replace-Elt New-Bucket-List
                                          Index
                                          (HashTable-Buckets Table))
               :Number-Entries Number-Entries))))
    (if (>= (HashTable-Number-Entries New-Table)
            (HashTable-Num-Buckets New-Table))
        (Hash-Resize New-Table)
        New-Table)))

(defun Splice-In-Bucket (Value Key Bucket-List Number-Entries)
  (cond ((or (null Bucket-List)
             (string< Key (Entry-Key (car Bucket-List))))
         (values (cons (Make-Entry :Key Key
                                   :Value Value)
                       Bucket-List)
                 (1+ Number-Entries)))
        (t (let ((This-Entry (car Bucket-List)))
             (cond ((string= Key (Entry-Key This-Entry))
                    (format t "~&Bashing older bucket entry ~A."
                            This-Entry)
                    (values
                      ;; if Key = key of This-Entry, then overwrite the older
                      ;; bucket entry.  (New bucket has same Key as older
                      ;; Bucket entry, but new entry value.)
                      (cons (Make-Entry :Key Key
                                        :Value Value)
                            (cdr Bucket-List))
                      Number-Entries))
                   (t (multiple-value-bind (New-Bucket-List Num-Entries)
                          (Splice-In-Bucket Value
                                            Key
                                            (cdr Bucket-List)
                                            Number-Entries)
                        (values
                          (cons This-Entry New-Bucket-List)
                          Num-Entries)))))))))

(defun Hash-Resize (Table)
  (let* ((Old-Buckets (HashTable-Buckets Table))
         (Old-Size  (HashTable-Num-Buckets Table))
         (New-Size (Determine-Hash-Table-Size
                     (* (HashTable-Num-Buckets Table) 2)))
         (New-Table (Make-HashTable :Num-Buckets New-Size
                                    :Number-Entries 0
                                    :Buckets (Make-Hash-Buckets New-Size))))
    (Copy-Over-Buckets 0 Old-Size Old-Buckets New-Table)))

(defun Copy-Over-Buckets (Index Old-Size Old-Buckets New-Table)
  (cond ((>= Index Old-Size) New-Table)
        (t (let ((Bucket-List (aref Old-Buckets Index)))
             (Copy-Over-Buckets (1+ Index)
                                Old-Size
                                Old-Buckets
                                (Copy-Over-Bucket Bucket-List New-Table))))))
(defun Copy-Over-Bucket (Bucket-List New-Table)
  (cond ((null Bucket-List) New-Table)
        (t (let ((This-Entry (car Bucket-list)))
             (Copy-Over-Bucket (cdr Bucket-List)
                               (Hash-Insert New-Table
                                            (Entry-Key This-Entry)
                                            (Entry-Value This-Entry)))))))

;;This function creates a hash table having the specified # of buckets.
```

```
;; Since the size of a hash table must be a prime number, the
;; specified number of buckets is rounded up to a nearby prime.
;; The new table is then initialized.

(defun Make-Hash-Table (&optional Num-Buckets)
  (let ((Size (Determine-Hash-Table-Size
               (or Num-Buckets MIN_HASH_TABLE_SIZE))))
    (Make-HashTable :Num-Buckets Size
                    :Buckets (Make-Hash-Buckets Size)
                    :Number-Entries 0)))

;;This function creates and initializes a bucket array.

(defun Make-Hash-Buckets (Size)
  (make-array Size))

;;; This function looks up a key in the hash table.  If it is
;;; found, the entry pointer is returned.  Otherwise, nil is
;;; returned.

(defun Hash-Lookup (Table Key)
  (let* ((Index (Hash-Function Key
                               (HashTable-Num-Buckets Table)))
         (Bucket-List (aref (HashTable-Buckets Table)
                            Index)))
    (Hash-Lookup-1 Bucket-List Key)))

(defun Hash-Lookup-1 (Bucket-List Key)
  (cond ((or (null Bucket-List)
             (string< Key
                      (Entry-Key (car Bucket-List))))
         nil)
        ((string= Key
                  (Entry-Key (car Bucket-List)))
         (Entry-Value (car Bucket-List)))
        (t
         (Hash-Lookup-1 (cdr Bucket-List) Key))))

;;; This function deletes an entry in the hash table.

(defun Hash-Delete (Table Key)
  (let ((Index (Hash-Function Key
                              (HashTable-Num-Buckets Table))))
    (multiple-value-bind (New-Bucket-List Number-Entries)
      (Splice-Out-Bucket
        Key
        (aref (HashTable-Buckets Table) Index)
        (HashTable-Number-Entries Table))
      (Make-HashTable
        :Num-Buckets (HashTable-Num-Buckets Table)
      :Buckets (Copy-Replace-Bucket New-Bucket-List
                                    Index
                                    (HashTable-Buckets Table))
      :Number-Entries Number-Entries))))

(defun Splice-Out-Bucket (Key Bucket-List Number-Entries)
  (if (null Bucket-List)
      (values nil Number-Entries)  ;; fell off end of bucket list
      (let ((This-Entry (car Bucket-List)))
      (cond ((string> Key (Entry-Key This-Entry))
             (multiple-value-bind (New-Bucket-List Num-Entries)
                 (Splice-Out-Bucket Key
                                    (cdr Bucket-List)
                                    Number-Entries)
               (values
                 (cons This-Entry New-Bucket-List)
                 Num-Entries)))
            ((string= Key (Entry-Key This-Entry))
             (values (cdr Bucket-List)
                     (1- Number-Entries)))
            (t ; Key string< Key of This-Entry => Key isn't found
             (values nil Number-Entries))))))

;;; This function clears for all entries in the specified hash
;;; table.

(defun Clear-Hash-Table (Table)
  (let ((Size (HashTable-Num-Buckets Table)))
    (Make-HashTable :Num-Buckets Size
                    :Number-Entries 0
                    :Buckets (Make-Hash-Buckets Size))))

;;; This function picks the first prime number greater then or
;;; equal to the specified size estimate. The minimum hash table
;;; size is enforced here.

(defun Determine-Hash-Table-Size (Size-Estimate &aux Size)
  (if (< Size-Estimate MIN_HASH_TABLE_SIZE)
      (setq Size MIN_HASH_TABLE_SIZE)
      (setq Size Size-Estimate))
  (if (= (mod Size 2) 0)
      (setq Size (1+ Size)))
  (Determine-Hash-Table-Size-1 Size))

(defun Determine-Hash-Table-Size-1 (Size)
  (if (null (Prime-Number-Test Size))
      (Determine-Hash-Table-Size-1 (+ Size 2))
      Size))

(defun Prime-Number-Test (Number)
  (let ((Index 3))
    (cond ((= Number 2) t)
          ((= (mod Number 2) 0) nil)
          (t (Prime-Number-Test-1 Index Number)))))

(defun Prime-Number-Test-1 (Index Number)
  (cond ((<= (Square Index) Number)
         (if (= (mod Number Index) 0)
             nil)
         (setq Index (+ Index 2))
         (Prime-Number-Test-1 Index Number))
        (t t)))

(defun Square (n) (* n n))

;;; This function calculates a hash table index from a key
;;; (symbol->string) and the hash table size.

(defun Hash-Function (Key Size)
  (let* ((Sum 0)
         (Key-String (string Key))
         (Length (1- (string-length Key-String))))
    (setq Sum (Hash-Function-1 Sum Key-String Length))
    (mod Sum Size)))

(defun Hash-Function-1 (Sum Key-String Length)
  (cond ((< Length 0)
         Sum)
        (t
          (setq Sum
                (+ Sum (char-int (aref Key-String Length))))
          (setq Length (1- Length))
          (Hash-Function-1 Sum Key-String Length))))
```

```
;;; -*- Syntax: Common-lisp; Base: 10.; Package: USER -*-

;;; CST simulator -- original version

;;; queue stuff

(defvar *default-queue-size* 16
      "Initial Queue Size")

(defstruct queue
  (head 0)
  (tail 0)
  (length 0)
  (data-size *default-queue-size*)
  (data (make-array *default-queue-size*)))

(defun queue-first (queue)
  (if (> (queue-length queue) 0)
      (aref (queue-data queue)
            (queue-head queue))))

(defun queue-empty? (queue)
  (zerop (queue-length queue)))

(defun queue-list (queue)
  (if (queue-empty? queue)
      '()
      (let ((data (queue-data queue))
          (head (queue-head queue))
          (tail (queue-tail queue)))
      (if (< head tail)
          (loop for index from head below tail
                collect (aref data index))
          (nconc (loop for index from head
                       below (queue-data-size queue)
                       collect (aref data index))
                 (loop for index from 0 below tail
                       collect (aref data index)))))))

(defun enqueue (queue obj)
  (let* ((tail (queue-tail queue))
       (length (queue-length queue))
       (data (queue-data queue))
       (old-size (queue-data-size queue)))
    (if (< length (- old-size 2))
      (progn
          (setf (aref data tail) obj)
        (setf (queue-tail queue)
              (mod (1+ (queue-tail queue))
                   old-size))
        (incf (queue-length queue)))
      (progn
          (adjust-array data (* old-size 2))
        (setf (queue-data-size queue)
              (* old-size 2))
        (let ((head (queue-head queue)))
          (if (> head tail)     ;; other case requires no copy
              (progn
                (loop for index from head below old-size
                      do (setf (aref data (+ old-size index))
                               (aref data index)))
                (setf (queue-head queue)
                      (+ old-size head)))))
        (enqueue queue obj)))))

(defun dequeue (queue)
  (if (queue-empty? queue)
      (error "~&Attempt to dequeue from an empty queue ~S" queue)
      (progn
        (let ((elt (aref (queue-data queue)
                         (queue-head queue))))
          (setf (queue-head queue)
              (mod (1+ (queue-head queue))
                   (queue-data-size queue)))
          (decf (queue-length queue))
        elt))))

;;; code to access a node descriptor
;;; node =  queue X  objects X contexts X method-cache

(defstruct node
  (queue (make-queue))
  (objects (make-array 32))
  (contexts (make-array 32))
  (method-cache (make-array *method-cache-size*))
  (busy-count 0))

(defvar *nodes*)

(defvar *contexts*)

(defvar *nr-nodes* 256 "Must also change nrnodes in CST world")

(defvar *step-queue*)           ; holds messages awaiting delivery

(defvar *step-nr*)

(defvar *profile*)      ; profiling flag, statistics recorded when true

(defvar *profile-list*)

(defvar *log* '() "Message Logging Enable")

(defvar *trace* '() "Whether or not we're tracing")

(defvar *trace-selectors* '()
        "list of selectors we're tracing")

(defvar *method-cache* t)

(defvar *method-cache-size* 10)

(defvar *method-cache-trace* '()
        "Switch for method cache tracing")

(defvar *method-cache-trace-list* '()
        "Global MC Trace list")

(defvar *meter-message-queues* '()
        "Enable message queue size tracing")

(defvar *message-queue-trace* '())

(defun get-node (node-nr)
  (aref *nodes* node-nr))

;;; code to access a message
;;; msg is of the form (msg node-nr header selector obj-id args)

(defun new-msg (node-nr header selector receiver args)
  (if (listp args)
      (append '(msg ,node-nr ,header ,selector ,receiver) args)
      '(msg ,node-nr ,header ,selector ,receiver ,args)))

(defun msg-node (msg)
  (cadr msg))

(defun msg-header (msg)
  (caddr msg))

(defun msg-slotn (n msg)
  (nth (+ n 3) msg))

(defun msg-selector (msg)
  (msg-slotn 0 msg))

(defun msg-receiver (msg)
  (msg-slotn 1 msg))

(defun msg-args (msg)
  (nthcdr 5 msg))

(defun msg-argn (n msg)
  (nth n (msg-args msg)))

(defun is-msg (msg)
  (eq (car msg) 'msg))

(defun msg-length (msg)
  (1- (length msg)))

(defun deliver-msgs ()
  (do ()
      ((queue-empty? *step-queue*))
    (let* ((msg (dequeue *step-queue*))
           (node-nr (msg-node msg))
           (node (get-node node-nr))
           (q (node-queue node)))
      (enqueue q msg))))

;;; step-nodes walks through the nodes and attempts to run a
;;; message on each node

(defun step-nodes ()
  (when *profile*
    (profile-step))
  (when *log*
    (log-step))
  (when *trace*
    (record-traced-selectors *trace-selectors*))
  (deliver-msgs)
  (when *meter-message-queues*
    (record-message-queue-data))
  (dotimes (x *nr-nodes*)
    (step-node x))
  (incf *step-nr*))
```

```
;; Run until no more work.

(defun step-done ()
  (if (queue-empty? *step-queue*)
      (do ((i 0 (+ i 1)))
        ((or (= i *nr-nodes*)
             (not (queue-empty? (node-queue (get-node i)))))
         (= i *nr-nodes*)))))

(defun step-node (node-nr)
  (let* ((node (get-node node-nr))
         (q (node-queue node)))
    (if (not (queue-empty? q))
      (let ((msg (dequeue q)))
        (incf (node-busy-count node))
        (process-msg msg)))))

(defun send-msg (msg)
  (enqueue *step-queue* msg))

(defun cst-start (init-msg)
  (send-msg init-msg)
  (shell-go))

(defun shell-go ()
  (cond ((step-done)
         nil)
        (t (step-nodes)
           (shell-go))))

(defun process-msg (msg)
  (if *profile*
      (setq *nr-msgs-received*
            (+ 1 *nr-msgs-received*)))
  (let ((header (msg-header msg)))
    (case header
      (send (process-send msg))
      (call (process-call msg))
      (new (process-new msg))
      (newco (process-newco msg))
      (reply (process-reply msg)))
    nil))

;;; new creates a new object on a node
;;; new is of the form (new class reply-context reply-slot)
;;; or if the object is distributed, a count may be appended
;;; for distributed objects, new-co messages are sent in a fanout
;;; tree to all constituents.
;;; <??>

(defun process-new (msg)
  (let* ((class-name (msg-slotn 0 msg))
         (reply-context (msg-slotn 1 msg))
         (reply-slot (msg-slotn 2 msg))
         (dist (class-dist (get-class class-name)))
         (id (new-object class-name (msg-node msg))))
    (if dist
      (let ((size (msg-slotn 3 msg)))
        (init-distributed-object id size (msg-node msg)
                                 reply-context reply-slot))
      (reply-to-context reply-context reply-slot id))))

(defun init-distributed-object (id size node reply-context
                                reply-slot)
  (let* ((size (if size
                   (min size *nr-nodes*)
                   default-distobj-size*))
         (did (new-did node size)))
    (send-dist-init node id did 0 size node reply-context
                    reply-slot)))

(defun send-dist-init (node id did index size root reply-context
                       reply-slot)
  (let ((msg (new-msg node 'send 'newco id
                      (list index size root reply-context
                            reply-slot))))
    (set-object-did (get-object (ref-id id)) did)
    (send-msg msg)))

;;; the newco message is a hack to allow distributed object to be
;;; created.

(defun process-newco (msg)
  (let* ((class-name (msg-slotn 0 msg))
         (did (msg-slotn 1 msg))
         (index (msg-slotn 2 msg))
         (size (msg-slotn 3 msg))
         (root (msg-slotn 4 msg))
         (reply-context (msg-slotn 5 msg))
         (reply-slot (msg-slotn 6 msg))
         (id (new-object class-name (msg-node msg))))
    (send-dist-init (msg-node msg) id did index size
                    root reply-context reply-slot)))

;;; on a reply, stuff data into slot and resume context
;;; message is (reply context-nr slot-nr data)
;;; if value is a value, must allocate copy

(defun process-reply (msg)
  (let* ((context-nr (msg-slotn 0 msg))
         (slot (msg-slotn 1 msg))
         (data (msg-slotn 2 msg))
         (context (get-context context-nr)))
    (if context
        (progn
          (set-slot slot context data)
          (resume-context context-nr)))))

;;; code to send a reply

(defun reply-to-context (context-nr slot value)
    (let ((msg (new-msg (context-to-node context-nr)
                        'reply context-nr slot (list value))))
      (send-msg msg)))

;;;<??> handle did receiver
;;; send creates a new context and executes the first statement
;;; if receiver is not atomic, look up class
;;; ids are referred to like '(id 3) to distinguish them from the integer 3.

(defun process-send (msg)
  (let* ((receiver (msg-receiver msg))
         (node (msg-node msg)))
    (cond ((is-did receiver)
           (let* ((id (did-on-node receiver node)))
             (if id
                 (process-normal-send msg id)
                 (forward-did-message node msg receiver))))
          ((is-co receiver)
           (let ((id (did-on-node `(did ,(second receiver)) node)))
             (process-normal-send msg id)))
          ((is-block receiver)
           (process-block-send msg))
          (t
           (process-normal-send msg receiver)))))

(defun process-normal-send (msg receiver)
  (let* ((selector (msg-selector msg))
         (args (msg-args msg)))
    (if (is-id receiver)
        (let* ((id (second receiver))
               (obj (get-object id))
               (class-name (object-class obj))
               (code (method-lookup selector class-name)))
          (start-code code msg receiver args))
        (let* ((class-name
                  (cond ((integerp receiver) 'integer)
                        ((floatp receiver) 'float)
                        ((symbolp receiver) 'symbol)))
               (code (method-lookup selector class-name)))
          (start-code code msg receiver args))))))

(defun forward-did-message (node msg receiver)
  (setf (second msg) (id-to-node receiver))
  (send-msg msg))

(defun process-block-send (msg)
  (let ((block (get-block (blkid-get-id (msg-receiver msg))))
        (selector (msg-selector msg))
        (args (msg-args msg)))
    (if (eq selector 'value)
        (start-code block msg nil args)
        (cst-error "~&Block message other than value ~S" msg))))

(defun start-code (code msg receiver args)
  (if code
      (let ((nr-args (block-nr-args code)))
        (cond ((= (+ nr-args 2)
                  (length args))
               (start-method (msg-node msg) code receiver args))
              (t
               (progn
                 (cst-error "~&Wrong number of arguments in ~S" msg)
                 (cst-error "~&~S actuals, to match ~S formals"
                            args nr-args)))))))

;;; create a context, copy args from message, execute to first send

(defun start-method (node code receiver args)
  (let ((context-nr (ref-id (new-context node code receiver))))
    (copy-args args context-nr)
    (advance-context context-nr)))

(defun copy-args (args context-nr)
  (let ((context (get-context context-nr)))
    (loop for arg in args
          for i from 0 do
          (set-context-slot context i arg))))
```

```
;;; advances context over next action

(defun advance-context (context-nr)
  (let ((next (execute-instruction context-nr)))
    (when *profile*
        (incf *nr-icodes-executed*))
    (when *method-cache*
      (let* ((node-nr (context-node (get-context context-nr)))
           (node (get-node node-nr))
           (block (context-code (get-context context-nr))))
        (when *method-cache-trace*
            (let ((prev (first *method-cache-trace-list*)))
              (if (not (and (equal (first prev)
                                   *step-nr*)
                            (equal (second prev)
                                   node-nr)))
                  (push `(,*step-nr* ,node-nr ,(block-id block)
                         ,(length (block-insts block)))
                        *method-cache-trace-list*))))
        (when (not (method-cache-present-p
                    block
                    (node-method-cache node)))
        (progn
          (incf *nr-blocks-loaded*)
          (method-cache-insert block
                               (node-method-cache node))))))
    (case next
      (suspend nil)
      (back-up (back-up-context context-nr))
      (continue (advance-context context-nr))
      (dispose (remove-context context-nr))
      (otherwise
        (cst-error "~&Illegal value in advance context:~S"
                   next)))))

;;; <??> other opcodes

(defun execute-instruction (context-nr)
  (let* ((inst (fetch-instruction context-nr))
       (opcode (car inst)))
    (if *profile*
        (setq *nr-insts-executed*
            (+ (- (length inst) 1)
               *nr-insts-executed*)))
    (execute-instruction-1 inst opcode context-nr)))

(defun execute-instruction-1 (inst opcode context-nr)
  (case opcode
      (move
        (execute-move context-nr inst))
      ((send csend forward)
       (execute-send context-nr inst))
      ((falsejump jump)
       (execute-jump context-nr inst))
      (label
       'continue)
      ((reply reply-x)
       (execute-reply context-nr inst))
      ((return return-x)
       (execute-return context-nr inst))
      ;; implement return icodes
      (reply-console
       (execute-reply-console context-nr inst))
      (echo-console
       (execute-echo-console context-nr inst))
      (newco
       (execute-newco context-nr inst))
      (new
       (execute-new context-nr inst))
      (touch
       (execute-touch context-nr inst))
      (suspend
       'suspend)
      (exit
       'dispose)))

(defun execute-touch (context-nr inst)
  (let* ((context (get-context context-nr))
       (ref (second inst)))
    (if (equal (get-slot ref context) 'c-fut)
      'back-up
      'continue)))

;;; sends away for a new object

(defun execute-new (context-nr inst)
  (let* ((context (get-context context-nr))
       (class-name (caddr inst))
       (dest (cadr inst))
       (size (get-slot (cadddr inst) context)))
    (if (eq class-name 'array)
      (progn
        (set-slot dest context
                                  new-array (context-node context) size)
        'continue)
      (progn
        (set-slot dest context 'c-fut)
        (cst-new class-name context-nr dest size)
        'suspend))))

;;; creates a constitutent of a distributed object

(defun execute-newco (context-nr inst)
  (let* ((context (get-context context-nr))
         (slot (cadr inst))
         (args (mapcar #'(lambda (x)
                                (get-slot x context))
                       (cddr inst)))
         (object (get-object (ref-id (context-receiver context))))
         (class (object-class object))
         (did (object-did object))
         (msg (new-msg (car args) 'newco class did
                       (append (cdr args) (list context-nr slot)))))
    (set-slot slot context 'c-fut)
    (send-msg msg)
    'continue))

(defun execute-jump (context-nr inst)
  (let* ((opcode (car inst)))
    (case opcode
      (falsejump
        (if (eq (get-slot (cadr inst)
                          (get-context context-nr))
                'false)
            (do-jump context-nr (caddr inst))
            'continue))
      (jump
        (do-jump context-nr (cadr inst))))))

(defun do-jump (context-nr target)
  (let* ((context (get-context context-nr))
         (code (block-insts (context-code context))))
    (set-context-ip context
                    (find-jump-target code target 0))
    'continue))

(defun find-jump-target (code target nr)
  (if code
      (let* ((stat (car code)
             (type (car stat)))
        (if (and (eq type 'label)
                 (= (cadr stat) target))
            nr
            (find-jump-target (cdr code) target (+ nr 1))))))

;;; does a primop or sends a message

(defun execute-send (context-nr inst)
  (let* ((opcode (first inst))
         (context (get-context context-nr))
         (operation
           (let ((oper (third inst)))
             (if (symbolp oper)
                 oper
                 (get-slot oper (get-context context-nr)))))
         (rargs (cdddr inst))
         (reply-to
           (case opcode
             ((send csend)
              (cons context-nr (second inst)))
             (forward
              (get-slot (second inst) context)))))
    (basic-send opcode context-nr operation rargs reply-to)))

;;  if the operation is primitive, do it and continue
;;  otherwise, actually do a message send

(defun basic-send (opcode context-nr operation rargs reply-to)
  (let* ((context (get-context context-nr))
         (all-args (mapcar #'(lambda (x)
                               (get-slot x context))
                           rargs))
         (node (context-node context))
         (dest (cdr reply-to))
         (op (is-primitive operation all-args)))
    (if (member 'c-fut all-args)
        'back-up
        (if (and op
                 (equal (car reply-to) context-nr))
            (progn
              (set-slot dest context (apply op all-args))
              'continue)
            (progn
              (cst-send  node (car all-args)
                         operation (cdr all-args)
                         (car reply-to) (cdr reply-to))
```

```
            (case opcode
              (send
                (set-slot dest context 'c-fut)
                'suspend)
              (csend
                (set-slot dest context 'c-fut)
                'continue)
              (forward
                'continue)))))))

(defun execute-move (context-nr inst)
  (let* ((context (get-context context-nr))
         (dest (second inst))
         (src (third inst)))
    (set-slot dest context (get-slot src context))
    'continue))

;;; Reply sends the result and exits the context

(defun execute-reply (context-nr inst)
  (let* ((context (get-context context-nr))
         (reply-context (context-reply-context context))
         (reply-slot (context-reply-slot context))
         (value (get-slot (cadr inst) context)))
    (if reply-context
      (case reply-context
        (console
          (cst-display value))
        (otherwise
          (when reply-slot
            (reply-to-context reply-context reply-slot value)))))
    'dispose))

;;; Return sends the result and continues to run in the context

(defun execute-return (context-nr inst)
  (let* ((context (get-context context-nr))
         (reply-context (context-reply-context context))
         (reply-slot (context-reply-slot context))
         (value (get-slot (cadr inst) context)))
    (if reply-context
      (case reply-context
        (console
          (cst-display value))
        (otherwise
          (when reply-slot
            (reply-to-context reply-context reply-slot value)))))
    'continue))

(defun execute-reply-console (context-nr inst)
  (let* ((context (get-context context-nr))
         (value (get-slot (cadr inst) context)))
    (cst-display value)
    'dispose))

(defun execute-echo-console (context-nr inst)
  (let* ((context (get-context context-nr))
         (val-list
           (loop for val in (rest inst)
                 collecting (get-slot val context))))
    (cst-display-list val-list))
  'continue)

;;; returns a numerical offset into a context's arg/var list

(defun compute-slot (slot context)
  (let ((type (car slot))
        (index (cadr slot))
        (code (context-code context)))
    (case type
      (var
        (+ index
           2
           (block-nr-args code)))
      (arg
        index)
      (temp
        (+ index
           2
           (block-nr-args code)
           (block-nr-vars code)))
      (otherwise
        (cst-error "~&Slot must be temp, var, or arg: ~S"
                   slot)))))

;;; gets a slot e.g., (ivar 0)
;;; <??> fix const and global

(defun get-slot (slot context)
  (if (listp slot)
      (let ((type (car slot))
            (index (cadr slot)))
        (case type
          (ivar
            (object-ivar
              (get-object (ref-id (context-receiver context)))
              index))
          ((arg var temp)
            (let ((n (compute-slot slot context))
              (context-slot context n)))
          (block
            slot)
          (global
            (get-global index))
          (const
            index)))
      (case slot
        (self
          (context-receiver context))
        (group
          (object-did
            (get-object (ref-id (context-receiver context)))))
        (requester
          (cons (context-reply-context context)
                (context-reply-slot context))))))

;;; sets a slot

(defun set-slot (slot context value)
  (let ((type (car slot))
        (index (cadr slot)))
    (case type
      ((arg var temp)
        (let ((n (compute-slot slot context)))
          (set-context-slot context n value)))
      (ivar
        (set-object-ivar
          (get-object (ref-id (context-receiver context)))
          index
          value))
      (global
        (set-global index value))
      ('()
        '())  ;; do nothing if it's nil
      (otherwise
        (cst-error "~&Slot error ~S" slot)))))

;;; <??> - temporary hack to implement globals need to generate
;;; code to send and receive

(defun set-global (name value)
  (let* ((cell (assoc name *globals*)))
    (if cell
        (rplacd (cdr cell) value)
        (cst-error "~&unknown global ~S" name))))

(defun get-global (name)
  (let* ((cell (assoc name *globals*)))
    (if cell
        (cddr cell)
        (cst-error "~&unknown global ~S" name))))

(defun fetch-instruction (context-nr)
  (let* ((context (get-context context-nr))
         (ip (context-ip context))
         (inst (block-inst ip (context-code context))))
    (set-context-ip context (+ 1 ip))
    inst))

(defun next-instruction (context)
  (let ((ip (context-ip context)))
        (block-inst ip (context-code context))))

(defun back-up-context (context-nr)
  (let* ((context (get-context context-nr))
         (ip (context-ip context)))
    (set-context-ip context (- ip 1))))

;;; resumes a suspended context

(defun resume-context (context-nr)
  (advance-context context-nr))

(defun init-nodes ()
  (setq *step-queue* (make-queue))
  (setq *nodes* (make-array *nr-nodes*))
  (dotimes (x *nr-nodes*)
    (setf (aref *nodes* x) (make-node))))

(defun is-node (node)
  (node-p node))

(defun random-node ()
  (random *nr-nodes*))

(defun print-node (node-nr)
```

```
  (let ((node (get-node node-nr)))
    (format *standard-output*
            "~&NODE ~S QUEUE ~S OBJECTS ~S CONTEXTS ~S"
            node-nr (node-queue node)
            (node-objects node) (node-contexts node))))

(defun init-contexts ()
  (setf *contexts* (make-array *init-nr-contexts* :adjustable t))
  (setf *nr-contexts* *init-nr-contexts*)
  (setf *next-context* 0)
  (setf *free-contexts* (make-stack))
  (setf *context-state-resource* (make-array-resource)))

(defun initial-context (nr-slots)
  (get-array *context-state-resource* nr-slots))

(defun context-nr (context)
  (nth 1 context))

(defun context-node (context)
  (nth 2 context))

(defun context-code (context)
  (nth 3 context))

(defun context-ip (context)
  (nth 4 context))

(defun set-context-ip (context x)
  (setf (nth 4 context) x))

(defun context-state (context)
  (nth 5 context))

(defun context-receiver (context)
  (nth 6 context))

(defun context-slot (context n)
  (aref (context-state context) n))

(defun set-context-slot (context n x)
  (setf (aref (context-state context) n) x))

(defun context-reply-context (context)
  (context-slot context
                (block-nr-args (context-code context))))

(defun set-context-reply-context (context x)
  (set-context-slot context
                    (block-nr-args (context-code context))
                    x))

(defun context-reply-slot (context)
  (context-slot context
                (+ 1 (block-nr-args (context-code context)))))

(defun set-context-reply-slot (context x)
  (set-context-slot context
                    (+ 1 (block-nr-args (context-code context)))
                    x))

(defun get-context (context-nr)
  (aref *contexts* context-nr))

(defun context-to-node (context-nr)
  (context-node (get-context context-nr)))

(defun find-context (c-nr c-list)
  (loop for context in c-list
      until (= c-nr (context-nr context))
        finally (return context)))

(defun live-contexts ()
  (loop for index from 0 below (length *contexts*)
      when (aref *contexts* index)
        collect (aref *contexts* index)))

(defun context-method (context)
  (block-method (block-id (context-code context))))

;;
;;  A block identifier abstraction
;;  a block id is (block blksymbol)
;;

(defun make-blkid ()
  (gensym "BLOCK"))

(defun blkid-get-id (blkid)
  (cadr blkid))

(defun is-blkid (id)
  (equal (car id) 'block))

(defun block-method (blkid)
  (loop for method in *methods*
        when (eq (caddr method) blkid)
          return method))

(defvar *blocks* '()
        "Icode blocks")

(defun get-block (block-tag)
  (assoc block-tag *blocks*))

(defun block-id (block)
  (car block))

(defun block-nr-args (block)
  (cadr block))

(defun block-nr-vars (block)
  (caddr block))

(defun block-nr-temps (block)
  (cadddr block))

(defun block-insts (block)
  (nth 4 block))

(defun block-inst (n block)
  (nth n (block-insts block)))

;;; returns the code

(defun method-lookup (selector class-name)
  (let ((method (method-lookup1 selector class-name)))
    (if (null method)
        (progn
          (format *standard-output*
                  "~&message ~S not implemented for class ~S"
                  selector class-name)
          '())
        method)))

(defun method-lookup1 (selector class-name)
  (let* ((class (get-class class-name)))
    (if class
        (let* ((supers (class-supers class))
               (methods (class-methods class))
               (method (assoc selector methods)))
          (if method
              (get-block (caddr method))
              (if (or (not (listp supers))
                      (eq class-name 'object)
                      (eq class-name nil))
                  '()
                  (method-lookup1 selector (car supers))))))))

(defvar *classes* '()
        "Class Structure and methods")

(defun get-class (class-name)
  (let ((class (assoc class-name *classes*)))
    (if class
        class
        (cst-error "~&Undefined Class ~S" class-name))))

(defun class-name (class)
  (car class))

(defun class-supers (class)
  (cadr class))

(defun class-vars (class)
  (caddr class))

(defun class-methods (class)
  (cadddr class))

(defun class-dist (class)
  (fifth class))

(defvar *objects* nil)

(defun get-object (id)
  (aref *objects* id))

(defun object-id (obj)
  (second obj))

(defun object-did (obj)
  (third obj))

(defun set-object-did (obj x)
  (setf (third obj) x))
```

```
(defun object-node (obj)
  (fourth obj))

(defun object-class (obj)
  (fifth obj))

(defun object-state (obj)
  (sixth obj))

(defun object-ivar (obj n)
  (nth n (object-state obj)))

(defun set-object-ivar (obj n x)
  (setf (nth n (object-state obj)) x))

(defun is-object (obj)
  (eq (car obj) 'object))

(defun is-id (ref)
  (and (listp ref)
       (eq (car ref) 'id)))

(defun is-did (ref)
  (and (listp ref)
       (eq (car ref) 'did)))

(defun is-co (ref)
  (and (listp ref)
       (eq (car ref) 'co)))

(defun is-block (ref)
  (and (listp ref)
       (eq (car ref) 'block)))

(defun ref-id (ref)
  (cadr ref))

(defun cst-error (string &rest args)
  (apply #'format *standard-output* string args)
  nil)

(defun cst-display-list (alist)
   (format *standard-output* "~&~3D: " *step-nr*)
   (loop for val in alist
       do (cst-display-1 val)))

(defun cst-display (value)
  (format *standard-output* "~&~3D: " *step-nr*)
  (cst-display-1 value))

(defun cst-display-1 (value)
  (cond ((listp value)
         (let ((type (car value))
               (index (cadr value)))
           (case type
             (id
              (format *standard-output* " ~S" (get-object index)))
             (otherwise
              (format *standard-output* " ~S" value)))))
        ((arrayp value)
         (display-array value))
        (t
         (format *standard-output* " ~S" value))))

(defun display-array (value)
  (let ((y nil))
    (dotimes (x (length value))
      (setq y (cons (aref value x) y)))
    (format *standard-output* " ~S" (reverse y))))

;; statistics functions

(defvar *log-list* '()
        "Log of Messages")

;;
;;  log all messages this step
;;
(defun log-step ()
  (push (list *step-nr*
              (copy-list (queue-list *step-queue*)))
        *log-list*))

(defvar *trace-list* '()
      "Messages we've recorded")

;;
;;  record traced messages this step
;;
(defun record-traced-selectors (traced)
  (let ((new-msgs
         (selectively-copy-traced traced
                                  (queue-list *step-queue*))))
    (when new-msgs
      (push (list *step-nr* new-msgs) *trace-list*))))

;;
;;  Filter out the traced selectors
;;
(defun selectively-copy-traced (sel-list msglist)
  (loop for msg in msglist
    when (member (msg-selector msg) sel-list) collect msg into result
        finally (return result)))

(defvar *nr-msgs-received* 0
        "Number of msgs received in the current time step")

(defvar *nr-insts-executed* 0
        "Insts executed, current time step")

(defvar *nr-icodes-executed* 0
        "Icodes, current time step")

(defvar *nr-blocks-loaded* 0
        "Number of Method Cache misses, current time step" )

(defun profile-step ()
  (push (make-profile-frame *step-nr*
                            (queue-length *step-queue*)
                            *nr-msgs-received*
                            *nr-insts-executed*
                            *nr-icodes-executed*
                            *nr-blocks-loaded*
                            (avg-queue-length)
                            (total-message-length))
        *profile-list*)
  (setf *nr-insts-executed* 0)
  (setf *nr-icodes-executed* 0)
  (setf *nr-blocks-loaded* 0)
  (setf *nr-msgs-received* 0))

(defun make-profile-frame (time-step msgs-new msgs-done
                                     insts-exec icodes-exec blocks-loaded
                                     avg-q-length msgs-words)
  (list time-step msgs-new msgs-done
        insts-exec icodes-exec blocks-loaded
        avg-q-length msgs-words))

(defun record-message-queue-data ()
  (push (cons *step-nr*
              (loop for index from 0 below *nr-nodes*
                    with mqlen = 0
                    unless (zerop
                             (setf mqlen
                                   (loop for message
                                         in (queue-list
                                              (node-queue (get-node index)))
                                         sum (msg-length message))))
                    collect (list index mqlen)))
        *message-queue-trace*))

(defun avg-queue-length ()
  (let ((tql 0))
    (dotimes (x *nr-nodes*)
      (setq tql
            (+ tql
               (queue-length (node-queue (get-node x))))))
    (/ tql *nr-nodes*)))

(defun total-message-length ()
  (reduce #'+
          (mapcar #'message-length (queue-list *step-queue*))))

(defun message-length (message)
  (- (length message) 2))
```

```
;;; -*- Syntax: Common-lisp; Base: 10.; Package: USER -*-
;;; CST simulator -- functional version

;;; queue stuff

(defvar *default-queue-size* 16 "Initial Queue Size")

(defstruct queue
  (head 0)
  (tail 0)
  (length 0)
  (data-size *default-queue-size*)
  (data (make-array *default-queue-size* :adjustable t)))

(defun queue-first (queue)
  (if (> (queue-length queue) 0)
      (aref (queue-data queue) (queue-head queue))))

(defun queue-empty? (queue)
  (= (queue-length queue) 0))

(defun queue-list (queue)
  (if (queue-empty? queue)
      '()
      (let ((data (queue-data queue))
            (head (queue-head queue))
            (tail (queue-tail queue)))
        (if (< head tail)
            (let ((index head)
                  (list nil)
                  (end-index tail))
              (queue-list-1 index end-index data list))
            (append
              (let ((index head)
                    (list nil)
                    (end-index (queue-data-size queue)))
                (queue-list-1 index end-index list))
              (let ((index 0)
                    (list nil)
                    (end-index tail))
                (queue-list-1 index end-index list)))))))

(defun queue-list-1 (index end-index data list)
  (cond ((not (< index end-index))
         list)
        (t (setq list (cons (aref data index) list))
           (setq index (1+ index))
           (queue-list-1 index end-index data list))))

(defun enqueue (queue obj)
  (let* ((length (queue-length queue))
         (old-size (queue-data-size queue))
         (big-enough-queue
           (if (< length (1- old-size))
               queue
               (grow-queue queue))))
    (enqueue-base big-enough-queue obj)))

(defun enqueue-base (queue obj)
  (let ((old-size (queue-data-size queue)))
    (setq queue
          (make-queue :head (queue-head queue)
                      :tail (queue-tail queue)
                      :length (queue-length queue)
                      :data-size (queue-data-size queue)
                      :data (copy-replace-elt obj
                                              (queue-tail queue)
                                              (queue-data queue))))
    (setq queue
        (make-queue :head (queue-head queue)
                    :tail (mod (1+ (queue-tail queue))
                               old-size)
                    :length (queue-length queue)
                    :data-size (queue-data-size queue)
                    :data (queue-data queue)))
    (setq queue
          (make-queue :head (queue-head queue)
                      :tail (queue-tail queue)
                      :length (1+ (queue-length queue))
                      :data-size (queue-data-size queue)
                      :data (queue-data queue)))
    queue))

(defun grow-queue (queue)
  (let* ((old-size (queue-data-size queue))
         (new-size (* old-size 2))
         (old-data (queue-data queue))
         (new-data (make-array new-size))
         (head (queue-head queue))
         (number-elements (queue-length queue)))
    (setq new-data
          (copy-over-elts
           old-data new-data head old-size number-elements))
    (setq queue
          (make-queue :head 0
                      :tail (queue-tail queue)
                      :length (queue-length queue)
                      :data-size (queue-data-size queue)
                      :data (queue-data queue)))
    (setq queue
          (make-queue :head (queue-head queue)
                      :tail number-elements
                      :length (queue-length queue)
                      :data-size (queue-data-size queue)
                      :data (queue-data queue)))
    (setq queue
          (make-queue :head (queue-head queue)
                      :tail (queue-tail queue)
                      :length number-elements
                      :data-size (queue-data-size queue)
                      :data (queue-data queue)))
    (setq queue
          (make-queue :head (queue-head queue)
                      :tail (queue-tail queue)
                      :length (queue-length queue)
                      :data-size (* old-size 2)
                      :data (queue-data queue)))
    (setq queue
          (make-queue :head (queue-head queue)
                      :tail (queue-tail queue)
                      :length (queue-length queue)
                      :data-size (queue-data-size queue)
                      :data new-data))))

(defun copy-over-elts (old-data new-data from old-size number-elements)
  (copy-over-elts-1 old-data new-data 0 from old-size number-elements))

(defun copy-over-elts-1 (old-data new-data new-index from old-size
                         number-elements)
  (cond ((>= new-index number-elements)
         new-data)
        (t (copy-over-elts-1
            old-data
            (copy-replace-elt
              (aref old-data (mod (+ from new-index) old-size))
              new-index
              new-data)
            (1+ new-index)
            from
            old-size
            number-elements))))

(defun dequeue (queue)
  (let ((elt (aref (queue-data queue)
                   (queue-head queue))))
    (setq queue (make-queue :head (mod (1+ (queue-head queue))
                                       (queue-data-size queue))
                            :tail (queue-tail queue)
                            :length (queue-length queue)
                            :data-size (queue-data-size queue)
                            :data (queue-data queue)))
    (setq queue
          (make-queue :head (queue-head queue)
                      :tail (queue-tail queue)
                      :length (1- (queue-length queue))
                      :data-size (queue-data-size queue)
                      :data (queue-data queue)))
    (values elt queue)))

;;; code to access a node descriptor
;;; node =  queue X  objects X contexts X method-cache

(defstruct node
  (queue (make-queue))
  (objects (make-array 32))
  (contexts (make-array 32))
  (method-cache (make-array *method-cache-size*))
  (busy-count 0))

(defstruct msg
  (node nil) ;; a node number
  (header nil)
  (selector nil)
  (receiver nil)
  (args nil)) ;; a list

(defstruct context
  (nr nil)
  (node nil)
  (code nil)
  (ip nil)
  (state nil)
  (receiver nil))

(defstruct block
  (id nil)
```

```
  (nr-args nil)
  (nr-vars nil)
  (nr-temps nil)
  (insts nil))

(defstruct class
  (name nil)
  (supers nil)
  (vars nil)
  (methods nil)
  (dist nil))

(defstruct object
  (id nil)
  (did nil)
  (node nil)
  (class nil)
  (state nil))

(defun object-ivar (obj n)
  (nth n (object-state obj)))

(defun is-object (obj)
  (object-p obj))

(defun block-inst (n block)
  (nth n (block-insts block)))

(defvar *nodes*)

(defvar *contexts*)

(defvar *step-queue*)

(defvar *step-nr*)

(defvar *nr-nodes* 256 "Must also change nrnodes in CST world")

(defvar *profile*);profiling flag, statistics recorded when true.

(defvar *profile-list*)

(defvar *log* '())  ; message logging enable

(defvar *trace* '() "whether or not we're tracing"))

(defvar *trace-selectors* '() "List of selectors we're tracing")

(defvar *method-cache* t)

(defvar *method-cache-size* 10)

(defvar *method-cache-trace* '()
      "Switch for method cache tracing")

(defvar *method-cache-trace-list* '()
      "Global MC Trace list")

(defvar *meter-message-queues* '()
      "Enable message queue size tracing")

(defvar *message-queue-trace* '())

(defvar *blocks* '()
      "Icode blocks")

(defvar *classes* '()
      "Class Structure and methods")

(defvar *objects*)

(defun get-node (node-nr)
  (aref *nodes* node-nr))

(defun get-block (block-tag)
  (assoc block-tag *blocks*))

(defun get-class (class-name)
  (let ((class (assoc class-name *classes*)))
    (if class
      class
      (cst-error "~&Undefined Class ~S" class-name))))

(defun get-object (id)
  (aref *objects* id))

(defun msg-argn (n msg)
  (nth n (msg-args msg)))

(defun msg-length (msg)
  (if (listp (msg-args msg))
      (+ 4 (length (msg-args msg)))
      5))
```

```
(defun deliver-msgs ()
  (cond ((queue-empty? *step-queue*)
         nil)
        (t (multiple-value-bind (msg new-step-queue)
               (dequeue *step-queue*)
             (setq *step-queue* new-step-queue)
             (let* ((node-nr (msg-node msg))
                    (node (get-node node-nr))
                    (q (node-queue node))
                    (new-q (enqueue q msg))
                    (new-node
                      (make-node :queue new-q
                                 :objects (node-objects node)
                                 :contexts (node-contexts node)
                                 :method-cache
                                 (node-method-cache node)
                                 :busy-count
                                 (node-busy-count node))))
               (setq *nodes*
                     (copy-replace-elt new-node node-nr *nodes*))))
           (deliver-msgs))))

;;; step-nodes walks through the nodes and attempts to run a message
;;; on each node

(defun step-nodes ()
  (when *profile*
    (profile-step))
  (when *log*
    (log-step))
  (when *trace*
    (record-traced-selectors *trace-selectors*))
  (deliver-msgs)
  (when *meter-message-queues*
    (record-message-queue-data))
  (iteratively-step-nodes 0)
  (setq *step-nr* (1+ *step-nr*)))

(defun iteratively-step-nodes (x)
  (if (>= x (array-total-size *nodes*))
      nil
      (step-node x)
      (iteratively-step-nodes (1+ x))))

;; Run until no more work.

(defun step-done ()
  (if (queue-empty? *step-queue*)
      (nodes-unemployed? 0)
      nil))

(defun nodes-unemployed? (i)
  (cond ((>= i (array-total-size *nodes*))
         t)
        ((queue-empty? (node-queue (get-node i)))
         (nodes-unemployed? (+ i 1)))
        (t nil)))

(defun step-node (node-nr)
  (let* ((node (get-node node-nr))
         (q (node-queue node)))
    (if (queue-empty? q)
        nil
        (multiple-value-bind (msg new-queue)
            (dequeue q)
          (setq node
                (make-node :queue new-queue
                           :objects (node-objects node)
                           :contexts (node-contexts node)
                           :busy-count (1+ (node-busy-count node))
                           :method-cache (node-method-cache node)))
          (setq *nodes*
                (copy-replace-elt node node-nr *nodes*))
          (multiple-value-bind (new-nodes new-step-queue)
              (process-msg msg *nodes* *step-queue*)
            (setq *nodes* new-nodes
                  *step-queue* new-step-queue)))))))

(defun send-msg (msg)
  (setq *step-queue* (enqueue *step-queue* msg)))

(defun cst-start (init-msg)
  (send-msg init-msg)
  (shell-go))

(defun shell-go ()
  (cond ((step-done)
         nil)
        (t (step-nodes)
           (shell-go)))))

(defun process-msg (msg)
  (if *profile*
```

```
      (setq *nr-msgs-received*
          (+ 1 *nr-msgs-received*)))
  (let ((header (msg-header msg)))
    (case header
      (send (process-send msg))
      (call (process-call msg))
      (new (process-new msg))
      (newco (process-newco msg))
      (reply (process-reply msg)))
    nil))

;;; new creates a new object on a node
;;; new is of the form (new class reply-context reply-slot)
;;; or if the object is distributed, a count may be appended
;;; for distributed objects, new-co messages are sent in a
;;; fanout tree to all constituents.
;;; <??>

(defun process-new (msg)
  (let* ((class-name (msg-selector msg))
       (reply-context (msg-receiver msg))
       (reply-slot (first (msg-args msg)))
       (dist (class-dist (get-class class-name)))
       (id (new-object class-name (msg-node msg))))
    (if dist
      (let ((size (second (msg-args msg))))
        (init-distributed-object id size (msg-node msg)
                                 reply-context reply-slot))
      (reply-to-context reply-context reply-slot id))))

(defun init-distributed-object (id size node reply-context
                                reply-slot)
  (let* ((size (if size
                 (min size *nr-nodes*)
                 default-distobj-size*))
       (did (new-did node size)))
    (send-dist-init node id did 0 size node reply-context
                    reply-slot)))

(defun send-dist-init (node id did index size root reply-context
                       reply-slot)
  (let ((msg
       (make-msg :node node
                 :header 'send
                 :selector 'newco
                 :receiver id
                 :args
                 (list index size root reply-context reply-slot)))
      (object (get-object (ref-id id))))
    (setq *objects*
          (copy-replace-elt
        (make-object :id (object-id object)
                     :did did
                     :node (object-node object)
                     :class (object-class object)
                     :state (object-state object)
                     :ivar (object-ivar object))
        (ref-id id)
        *objects*))
    (send-msg msg)))

;;; the newco message is a hack to allow distributed object to be
;;; created.

(defun process-newco (msg)
  (let* ((class-name (msg-selector msg))
       (did (msg-receiver msg))
       (index (first (msg-args msg)))
       (size (second (msg-args msg)))
       (root (third (msg-args msg)))
       (reply-context (fourth (msg-args msg)))
       (reply-slot (fifth (msg-args msg)))
       (id (new-object class-name (msg-node msg))))
    (send-dist-init (msg-node msg) id did index size
                    root reply-context reply-slot)))

;;; on a reply, stuff data into slot and resume context
;;; message is (reply context-nr slot-nr data)
;;; if value is a value, must allocate copy

(defun process-reply (msg)
  (let* ((context-nr (msg-selector msg))
       (slot (msg-receiver msg))

       (data (first (msg-args msg)))
       (context (get-context context-nr)))
    (if context
      (progn
        (set-slot slot context data)
          (resume-context context-nr)))))

;;; code to send a reply

(defun reply-to-context (context-nr slot value)
  (let ((msg
          (make-msg :node (context-to-node context-nr)
                    :header 'reply
                    :selector context-nr
                    :receiver slot
                    :args (list value))))
    (send-msg msg)))

;;;<??> handle did receiver
;;; send creates a new context and executes the first statement
;;; if receiver is not atomic, look up class
;;; ids are referred to like '(id 3) to distinguish them from the integer 3.

(defun process-send (msg)
  (let* ((receiver (msg-receiver msg))
         (node (msg-node msg)))
    (cond ((is-did receiver)
           (let* ((id (did-on-node receiver node)))
             (if id
                 (process-normal-send msg id)
                 (forward-did-message node msg receiver))))
          ((is-co receiver)
           (let ((id (did-on-node '(did ,(second receiver)) node)))
             (process-normal-send msg id)))
          ((is-block receiver)
           (process-block-send msg))
          (t
           (process-normal-send msg receiver)))))

(defun process-normal-send (msg receiver)
  (let* ((selector (msg-selector msg))
         (args (msg-args msg)))
    (if (is-id receiver)
        (let* ((id (second receiver))
               (obj (get-object id))
               (class-name (object-class obj))
               (code (method-lookup selector class-name)))
          (start-code code msg receiver args))
        (let* ((class-name
                  (cond ((integerp receiver) 'integer)
                        ((floatp receiver) 'float)
                        ((symbolp receiver) 'symbol)))
               (code (method-lookup selector class-name)))
          (start-code code msg receiver args)))))

(defun forward-did-message (node msg receiver)
  (setq msg
        (make-msg :node (id-to-node receiver)
                  :header (msg-header msg)
                  :selector (msg-selector msg)
                  :receiver (msg-receiver msg)
                  :args (msg-args msg)))
  (send-msg msg))

(defun process-block-send (msg)
  (let ((block (get-block (blkid-get-id (msg-receiver msg))))
        (selector (msg-selector msg))
        (args (msg-args msg)))
    (if (eq selector 'value)
        (start-code block msg nil args)
        (cst-error "~&Block message other than value ~S" msg))))

(defun start-code (code msg receiver args)
  (if code
      (let ((nr-args (block-nr-args code)))
        (cond ((= (+ nr-args 2)
                  (length args))
               (start-method (msg-node msg) code receiver args))
              (t
               (progn
                 (cst-error "~&Wrong number of arguments in ~S" msg)
                 (cst-error "~&~S actuals, to match ~S formals"
                            args nr-args)))))))

;;; create a context, copy args from message, execute to first send

(defun start-method (node code receiver args)
  (let ((context-nr (ref-id (new-context node code receiver))))
    (copy-args args context-nr)
    (advance-context context-nr)))

(defun copy-args (args context-nr)
  (let ((context (get-context context-nr)))
    (let ((arg nil)
          (i 0))
      (copy-args-1 arg args i context))))

(defun copy-args-1 (arg args i context)
  (cond ((null args)
         nil)
        (t
         (setq arg (car args))
```

```
        (multiple-value-bind (value new-context)
            (set-context-slot context i arg)
          (setq context new-context))
        (setq args (cdr args))
        (setq i (1+ i))
        (copy-args-1 arg args i context))))

;;; advances context over next action

(defun advance-context (context-nr)
  (let ((next (execute-instruction context-nr)))
    (when *profile*
          (setq *nr-icodes-executed*
              (1+ *nr-icodes-executed*)))
    (when *method-cache*
      (let* ((node-nr (context-node (get-context context-nr)))
             (node (get-node node-nr))
             (block (context-code (get-context context-nr))))
        (when *method-cache-trace*
            (let ((prev (first *method-cache-trace-list*)))
              (if (not (and (equal (first prev)
                                   *step-nr*)
                            (equal (second prev)
                                   node-nr)))
                  (setq *method-cache-trace-list*
                        (cons (list *step-nr* node-nr
                                    (block-id block)
                                    (length (block-insts block)))
                              *method-cache-trace-list*)))))
        (when (not (method-cache-present-p
                    block
                    (node-method-cache node)))
          (progn
            (setq *nr-blocks-loaded*
                  (1+ *nr-blocks-loaded*))
            (method-cache-insert block
                                 (node-method-cache node))))))
    (case next
      (suspend nil)
      (back-up (back-up-context context-nr))
      (continue (advance-context context-nr))
      (dispose (remove-context context-nr))
      (otherwise
        (cst-error "~&Illegal value in advance context:~S"
                   next)))))

;;; <??> other opcodes

(defun execute-instruction (context-nr)
  (let* ((inst (fetch-instruction context-nr))
         (opcode (car inst)))
    (if *profile*
        (setq *nr-insts-executed*
            (+ (- (length inst) 1)
               *nr-insts-executed*)))
    (execute-instruction-1 inst opcode context-nr)))

(defun execute-instruction-1 (inst opcode context-nr)
  (case opcode
      (move
        (execute-move context-nr inst))
      ((send csend forward)
       (execute-send context-nr inst))
      ((falsejump jump)
       (execute-jump context-nr inst))
      (label
       'continue)
      ((reply reply-x)
       (execute-reply context-nr inst))
      ((return return-x)
       (execute-return context-nr inst))
      ;; implement return icodes
      (reply-console
       (execute-reply-console context-nr inst))
      (echo-console
       (execute-echo-console context-nr inst))
      (newco
       (execute-newco context-nr inst))
      (new
       (execute-new context-nr inst))
      (touch
       (execute-touch context-nr inst))
      (suspend
       'suspend)
      (exit
       'dispose)))

(defun execute-touch (context-nr inst)
  (let* ((context (get-context context-nr))
         (ref (second inst)))
    (if (equal (get-slot ref context) 'c-fut)
        'back-up
        'continue)))

;;; sends away for a new object

(defun execute-new (context-nr inst)
  (let* ((context (get-context context-nr))
         (class-name (caddr inst))
         (dest (cadr inst))
         (size (get-slot (cadddr inst) context)))
    (if (eq class-name 'array)
        (progn
          (set-slot dest context
                    new-array (context-node context) size)
          'continue)
        (progn
          (set-slot dest context 'c-fut)
          (cst-new class-name context-nr dest size)
          'suspend))))

;;; creates a constitutent of a distributed object

(defun execute-newco (context-nr inst)
  (let* ((context (get-context context-nr))
         (slot (cadr inst))
         (args (mapcar #'(lambda (x)
                           (get-slot x context))
                       (cddr inst)))
         (object (get-object (ref-id (context-receiver context))))
         (class (object-class object))
         (did (object-did object))
         (msg
           (make-msg :node (car args)
                     :header 'newco
                     :selector class
                     :receiver did
                     :args
                     (append (cdr args) (list context-nr slot)))))
    (set-slot slot context 'c-fut)
    (send-msg msg)
    'continue))

(defun execute-jump (context-nr inst)
  (let* ((opcode (car inst)))
    (case opcode
      (falsejump
        (if (eq (get-slot (cadr inst)
                          (get-context context-nr))
                'false)
            (do-jump context-nr (caddr inst))
            'continue))
      (jump
        (do-jump context-nr (cadr inst))))))

(defun do-jump (context-nr target)
  (let* ((context (get-context context-nr))
         (code (block-insts (context-code context))))
    (setq *contexts*
          (copy-replace-elt
            (make-context :nr (context-nr context)
                          :node (context-node context)
                          :code (context-code context)
                          :ip (find-jump-target code target 0)
                          :state (context-state context)
                          :receiver (context-receiver context))
            context-nr
            *contexts*))
    'continue))

(defun find-jump-target (code target nr)
  (if code
      (let* ((stat (car code))
             (type (car stat)))
        (if (and (eq type 'label) (= (cadr stat) target))
            nr
            (find-jump-target (cdr code) target (+ nr 1))))))

;;; does a primop or sends a message

(defun execute-send (context-nr inst)
  (let* ((opcode (first inst))
         (context (get-context context-nr))
         (operation
           (let ((oper (third inst)))
             (if (symbolp oper)
                 oper
                 (get-slot oper (get-context context-nr)))))
         (rargs (cdddr inst))
         (reply-to
           (case opcode
             ((send csend)
              (cons context-nr (second inst)))
             (forward
              (get-slot (second inst) context)))))
    (basic-send opcode context-nr operation rargs reply-to)))
```

```
;;  if the operation is primitive, do it and continue
;;  otherwise, actually do a message send
;;

(defun basic-send (opcode context-nr operation rargs reply-to)
  (let* ((context (get-context context-nr))
       (all-args (mapcar #'(lambda (x)
                             (get-slot x context))
                         rargs))
       (node (context-node context))
       (dest (cdr reply-to))
       (op (is-primitive operation all-args)))
    (if (member 'c-fut all-args)
      'back-up
      (if (and op
               (equal (car reply-to) context-nr))
          (progn
            (set-slot dest context (apply op all-args))
            'continue)
          (progn
            (cst-send  node (car all-args)
                       operation (cdr all-args)
                       (car reply-to) (cdr reply-to))
            (case opcode
              (send
                (set-slot dest context 'c-fut)
                'suspend)
              (csend
                (set-slot dest context 'c-fut)
                'continue)
              (forward
                'continue)))))))

(defun execute-move (context-nr inst)
  (let* ((context (get-context context-nr))
       (dest (second inst))
       (src (third inst)))
    (set-slot dest context (get-slot src context))
    'continue))

;;; Reply sends the result and exits the context

(defun execute-reply (context-nr inst)
  (let* ((context (get-context context-nr))
       (reply-context (context-reply-context context))
       (reply-slot (context-reply-slot context))
       (value (get-slot (cadr inst) context)))
    (if reply-context
      (case reply-context
        (console
          (cst-display value))
        (otherwise
          (when reply-slot
              (reply-to-context reply-context reply-slot
                                value)))))
    'dispose))

;;; Return sends the result and continues to run in the context

(defun execute-return (context-nr inst)
  (let* ((context (get-context context-nr))
       (reply-context (context-reply-context context))
       (reply-slot (context-reply-slot context))
       (value (get-slot (cadr inst) context)))
    (if reply-context
      (case reply-context
        (console
          (cst-display value))
        (otherwise
          (when reply-slot
            (reply-to-context reply-context reply-slot value)))))
    'continue))

(defun execute-reply-console (context-nr inst)
  (let* ((context (get-context context-nr))
       (value (get-slot (cadr inst) context)))
    (cst-display value)
    'dispose))

(defun execute-echo-console (context-nr inst)
  (let* ((context (get-context context-nr))
       (val-list
         (let ((val nil))
           (execute-echo-console-1 val (rest inst) context))))
    (cst-display-list val-list))
  'continue)

(defun execute-echo-console-1 (val vals context)
  (cond ((null vals)
       nil)
      (t
       (setq val (car vals))
       (setq vals (cdr vals))
       (cons (get-slot val context)
             (execute-echo-console-1 val vals context)))))

;;; returns a numerical offset into a context's arg/var list

(defun compute-slot (slot context)
  (let ((type (car slot))
        (index (cadr slot))
        (code (context-code context)))
    (case type
      (var
        (+ index
           2
           (block-nr-args code)))
      (arg
        index)
      (temp
        (+ index
           2
           (block-nr-args code)
           (block-nr-vars code)))
      (otherwise
        (cst-error "~&Slot must be temp, var; or arg: ~S" slot)))))

;;; gets a slot e.g., (ivar 0)
;;; <??> fix const and global

(defun get-slot (slot context)
  (if (listp slot)
      (let ((type (car slot))
            (index (cadr slot)))
        (case type
          (ivar
            (object-ivar
              (get-object (ref-id (context-receiver context)))
              index))
          ((arg var temp)
            (let ((n (compute-slot slot context))
              (context-slot context n))))
          (block
            slot)
          (global
            (get-global index))
          (const
            index)))
      (case slot
        (self
          (context-receiver context))
        (group
          (object-did
            (get-object (ref-id (context-receiver context)))))
        (requester
          (cons (context-reply-context context)
                (context-reply-slot context))))))

;;; sets a slot

(defun set-slot (slot context value)
  (let ((type (car slot))
        (index (cadr slot)))
    (case type
      ((arg var temp)
        (let ((n (compute-slot slot context)))
          (multiple-value-bind (value new-context)
              (set-context-slot context n value)
            value)))
      (ivar
        (let* ((id (ref-id (context-receiver context)))
               (object (get-object id))
          (setq *objects*
                (copy-replace-elt
                  (make-object :id (object-id object)
                               :did (object-did object)
                               :node (object-node object)
                               :class (object-class object)
                               :state
                               (replace-nth index
                                            (object-state object)
                                            value))
                  id
                  *objects*))
          value))
      (global
        (set-global index value))
      ('()
        '())  ;; do nothing if it's nil
      (otherwise
        (cst-error "~&Slot error ~S" slot)))))

(defun replace-nth (n list value)
  (cond ((null list)
         nil)
        ((= n 0)
```

```
       (cons value (cdr list)))
      (t
       (cons (car list)
             (replace-nth (1- n)
                          (cdr list)
                          value)))))

;;; <??> - temporary hack to implement globals need to generate
;;; code to send and receive

(defun set-global (name value)
  (let* ((cell (assoc name *globals*)))
    (if cell
        (setq *globals*
            (replace-global
              name
              (cons (car cell) value)
              *globals*))
      (cst-error "~&unknown global ~S" name))))

(defun replace-global (name cell globals)
  (cond ((null globals)
         nil)
        ((eql name (car (car globals)))
         (cons (cons name cell)
               (cdr globals)))
        (t
         (cons (car globals)
               (replace-global name cell (cdr globals))))))

(defun get-global (name)
  (let* ((cell (assoc name *globals*)))
    (if cell
      (cddr cell)
      (cst-error "~&unknown global ~S" name))))

(defun fetch-instruction (context-nr)
  (let* ((context (get-context context-nr))
       (ip (context-ip context))
       (inst (block-inst ip (context-code context))))
    (setq *contexts*
        (copy-replace-elt
          (make-context :nr (context-nr context)
                        :node (context-node context)
                        :code (context-code context)
                        :ip (+ 1 ip)
                        :state (context-state context)
                        :receiver (context-receiver context))
          context-nr
          *contexts*))
    inst))

(defun next-instruction (context)
  (let ((ip (context-ip context)))
      (block-inst ip (context-code context))))

(defun back-up-context (context-nr)
  (let* ((context (get-context context-nr))
       (ip (context-ip context))
       (new-ip (- ip 1)))
    (setq *contexts*
        (copy-replace-elt
          (make-context :nr (context-nr context)
                        :node (context-node context)
                        :code (context-code context)
                        :ip new-ip
                        :state (context-state context)
                        :receiver (context-receiver context))
          context-nr
          *contexts*))
    new-ip))

;;; resumes a suspended context

(defun resume-context (context-nr)
  (advance-context context-nr))

(defun init-nodes ()
  (setq *step-queue* (make-queue))
  (setq *nodes* (make-array *nr-nodes*))
  (let ((x 0))
    (init-nodes-1 x *nr-nodes*)))

(defun init-nodes-1 (x n)
  (cond ((not (< x n))
         nil)
        (t
         (setq *nodes*
               (copy-replace-elt (make-node) x *nodes*))
         (setq x (1+ x))
         (init-nodes-1 x n))))
```

```
(defun is-node (node)
  (node-p node))

(defun random-node ()
  (random *nr-nodes*))

(defun print-node (node-nr)
  (let ((node (get-node node-nr)))
    (format *standard-output* "~&NODE ~S QUEUE ~S OBJECTS ~S CONTEXTS ~S"
            node-nr (node-queue node)
            (node-objects node) (node-contexts node))))

(defun init-contexts ()
  (setf *contexts* (make-array *init-nr-contexts* :adjustable t))
  (setf *nr-contexts* *init-nr-contexts*)
  (setf *next-context* 0)
  (setf *free-contexts* (make-stack))
  (setf *context-state-resource* (make-array-resource)))

(defun initial-context (nr-slots)
  (get-array *context-state-resource* nr-slots))

(defun context-slot (context n)
  (aref (context-state context) n))

(defun set-context-slot (context n x)
  (let ((new-context
         (make-context :nr (context-nr context)
                       :node (context-node context)
                       :code (context-code context)
                       :ip (context-ip context)
                       :state (copy-replace-elt
                                x n (context-state context))
                       :receiver (context-receiver context))))
    (setq *contexts*
          (copy-replace-elt
            new-context
            (context-nr context)
            *contexts*))
    (values x new-context)))

(defun context-reply-context (context)
  (context-slot context
                (block-nr-args (context-code context))))

(defun set-context-reply-context (context x)
  (set-context-slot context
                    (block-nr-args (context-code context))
                    x))

(defun context-reply-slot (context)
  (context-slot context
                (+ 1 (block-nr-args (context-code context)))))

(defun set-context-reply-slot (context x)
  (set-context-slot context
                    (+ 1 (block-nr-args (context-code context)))
                    x))

(defun get-context (context-nr)
  (aref *contexts* context-nr))

(defun context-to-node (context-nr)
  (context-node (get-context context-nr)))

(defun find-context (c-nr c-list)
  (let ((context nil))
    (find-context-1 context c-nr c-list)))

(defun find-context-1 (context c-nr c-list)
  (cond ((null c-list)
         context)
        (t
         (setq context (car c-list))
         (cond ((= c-nr (context-nr context))
                context)
               (t
                 (setq c-list (cdr c-list))
                 (find-context-1 context c-nr c-list))))))

(defun live-contexts ()
  (let ((index 0)
        (limit (length *contexts*)))
    (live-contexts-1 index limit)))

(defun live-contexts-1 (index limit)
  (cond ((not (< index limit))
         nil)
        (t
         (setq index (1+ index))
         (let ((rest-live-contexts
                 (live-contexts-1 index limit)))
           (if (aref *contexts* index)
```

```
            (cons (aref *contexts* index)
                  rest-live-contexts)
            rest-live-contexts)))))

(defun context-method (context)
  (block-method (block-id (context-code context))))

;;
;;  A block identifier abstraction
;;  a block id is (block blksymbol)
;;

(defun make-blkid ()
  (gensym "BLOCK"))

(defun blkid-get-id (blkid)
  (cadr blkid))

(defun is-blkid (id)
  (equal (car id) 'block))

(defun block-method (blkid)
  (let ((method nil)
        (methods *methods*))
    (block-method-1 method methods blkid)))

(defun block-method-1 (method methods blkid)
  (cond ((null methods)
         nil)
        (t
         (setq method (car methods))
         (setq methods (cdr methods))
         (if (eq (caddr method) blkid)
             method
             (block-method-1 method methods blkid)))))

;;; returns the code

(defun method-lookup (selector class-name)
  (let ((method (method-lookup1 selector class-name)))
    (if (null method)
        (progn
          (format *standard-output*
                  "~&message ~S not implemented for class ~S"
                  selector class-name)
          '())
        method)))

(defun method-lookup1 (selector class-name)
  (let* ((class (get-class class-name)))
    (if class
        (let* ((supers (class-supers class))
               (methods (class-methods class))
               (method (assoc selector methods)))
          (if method
              (get-block (caddr method))
              (if (or (not (listp supers))
                      (eq class-name 'object)
                      (eq class-name nil))
                  '()
                  (method-lookup1 selector (car supers))))))))

(defun is-id (ref)
  (and (listp ref)
       (eq (car ref) 'id)))

(defun is-did (ref)
  (and (listp ref)
       (eq (car ref) 'did)))

(defun is-co (ref)
  (and (listp ref)
       (eq (car ref) 'co)))

(defun is-block (ref)
  (and (listp ref)
       (eq (car ref) 'block)))

(defun ref-id (ref)
  (cadr ref))

(defun cst-error (string &rest args)
  (apply #'format *standard-output* string args)
  nil)

(defun cst-display-list (alist)
  (format *standard-output* "~&~3D: " *step-nr*)
  (let ((val nil))
    (cst-display-list-1 val alist)))

(defun cst-display-list-1 (val alist)
  (cond ((null alist)
         nil)
        (t
         (setq val (car alist))
         (setq alist (cdr alist))
         (cst-display-1 val)
         (cst-display-list-1 val alist))))

(defun cst-display (value)
  (format *standard-output* "~&~3D: " *step-nr*)
  (cst-display-1 value))

(defun cst-display-1 (value)
  (cond ((listp value)
         (let ((type (car value))
               (index (cadr value)))
           (case type
             (id
              (format *standard-output* " ~S" (get-object index)))
             (otherwise
              (format *standard-output* " ~S" value)))))
        ((arrayp value)
         (display-array value))
        (t
         (format *standard-output* " ~S" value))))

(defun display-array (value)
  (let ((y nil)
        (x 0)
        (limit (length value)))
    (setq y (display-array-1 x limit y value))
    (format *standard-output* " ~S" (reverse y))))

(defun display-array-1 (x limit y value)
  (cond ((not (< x limit))
         y)
        (t
         (setq y (cons (aref value x) y))
         (setq x (1+ x))
         (display-array-1 x limit y value))))

;; statistics functions

(defvar *log-list* '()
        "Log of Messages")

;;
;;  log all messages this step
;;
(defun log-step ()
  (setq *log-list*
        (cons (list *step-nr*
                    (copy-list (queue-list *step-queue*)))
              *log-list*)))

(defvar *trace-list* '()
        "Messages we've recorded")

;;
;;  record traced messages this step
;;
(defun record-traced-selectors (traced)
  (let ((new-msgs
          (selectively-copy-traced traced (queue-list *step-queue*))))
    (when new-msgs
      (setq *trace-list*
            (cons (list *step-nr* new-msgs)
                  *trace-list*)))))

;;  Filter out the traced selectors

(defun selectively-copy-traced (sel-list msglist)
  (let ((msg nil))
    (selectively-copy-traced-1 msg sel-list msglist)))

(defun selectively-copy-traced-1 (msg sel-list msglist)
  (cond ((null msglist)
         nil)
        (t
         (setq msg (car msglist))
         (setq msglist (cdr msglist))
         (let ((rest-of-result
                 (selectively-copy-traced-1 msg sel-list msglist)))
           (if (member (msg-selector msg) sel-list)
               (cons msg rest-of-result)
               rest-of-result)))))

(defvar *nr-msgs-received* 0
        "Number of msgs received in the current time step")

(defvar *nr-insts-executed* 0
        "Insts executed, current time step")

(defvar *nr-icodes-executed* 0
        "Icodes, current time step")
```

```
(defvar *nr-blocks-loaded* 0
      "Number of Method Cache misses, current time step" )

(defun profile-step ()
  (setq *profile-list*
      (cons (make-profile-frame
              *step-nr*
              (queue-length *step-queue*)
              *nr-msgs-received*
              *nr-insts-executed*
              *nr-icodes-executed*
              *nr-blocks-loaded*
              (avg-queue-length)
              (total-message-length))
            *profile-list*))
  (setf *nr-insts-executed* 0)
  (setf *nr-icodes-executed* 0)
  (setf *nr-blocks-loaded* 0)
  (setf *nr-msgs-received* 0))

(defun make-profile-frame (time-step msgs-new msgs-done
                                  insts-exec icodes-exec
                                  blocks-loaded
                                  avg-q-length msgs-words)
  (list time-step msgs-new msgs-done
      insts-exec icodes-exec blocks-loaded
      avg-q-length msgs-words))

(defun record-message-queue-data ()
  (setq *message-queue-trace*
        (cons
          (cons *step-nr*
                (let ((index 0)
                      (limit *nr-nodes*)
                      (mqlen 0))
                  (record-message-queue-data-1
                    index limit mqlen)))
          *message-queue-trace*)))

(defun record-message-queue-data-1 (index limit mqlen)
  (cond ((not (< index limit))
         nil)
        (t
         (setq mqlen
               (let ((message nil)
                     (messages (queue-list
                                 (node-queue (get-node index))))
                     (sum 0))
                 (record-message-queue-data-2 message messages
                                              sum)))
         (let ((rest-queue-data (record-message-queue-data-1
                                  (1+ index) limit 0)))
           (if (not (zerop mqlen))
               (cons (list index mqlen)
                     rest-queue-data)
               rest-queue-data)))))

(defun record-message-queue-data-2 (message messages sum)
  (cond ((null messages)
         sum)
        (t
         (setq message (car messages))
         (setq messages (cdr messages))
           (setq sum (+ sum (msg-length message)))
         (record-message-queue-data-2 message messages sum))))

(defun avg-queue-length ()
  (let ((tql 0))
    (setq tql (sum-queue-lengths 0 tql))
    (/ tql (array-total-size *nodes*))))

(defun sum-queue-lengths (x tql)
  (if (>= x (array-total-size *nodes*))
      tql
      (sum-queue-lengths
        (1+ x)
        (+ tql (queue-length (node-queue (get-node x)))))))

(defun total-message-length ()
  (let ((sum 0))
    (total-message-length-1
      sum
      (mapcar #'message-length (queue-list *step-queue*)))))

(defun total-message-length-1 (sum lengths)
  (cond ((null lengths)
         sum)
        (t
         (setq sum (+ sum (car lengths)))
         (setq lengths (cdr lengths))
         (total-message-length-1 sum lengths))))

(defun message-length (message)
  (if (listp (msg-args message))
      (+ 3 (length (msg-args message)))
      4))
```

Appendix C

The Grammar Encoding the Cliché Library

This appendix contains the grammar that encodes our cliché library. It is an extraction of key parts of the grammar rules, showing their graph structure and the documentation associated with the clichés they represent. Due to space limitations, non-structural constraints are not included.

The syntax of a grammar rule is as follows:

```
(Defrule <lhs node type>
  <cliche name>
  :RHS-Node-Types
  <node label-type pairs>
  :Edge-List
  <source-sink pairs>
  :Input-Embedding
  <lhs-to-rhs mappings>
  :Output-Embedding
  <lhs-to-rhs mappings>
  :St-Thrus
  <lhs-to-lhs mappings>
  :L-R-Link <cliche relationship>
  :Doc
  (<documentation string> <documentation arguments>))
```

The non-terminal node type of the rule's left-hand side is given by `<lhs node type>`. The name of the cliché represented by this non-terminal type is given by `<cliche name>`.

The keywords `:RHS-Node-Types` and `:Edge-List` specify the right-hand side flow graph. `:RHS-Node-Types` describes the right-hand side nodes. The `<node label-type pairs>` is a list of pairs of the form (`<node-label>` . `<node-type>`), each of which specifies the label of a right-hand side node and its type. `:Edge-List` indicates which ports are connected by a directed edge. The `<source-sink pairs>` is a list of pairs of the form (`<source port`

specification> . <sink port specification>), where each port specification is of the form (<node label> <numeric port identifier>).

The keywords :Input-Embedding, :Output-Embedding, and :St-Thrus specify the embedding relation of the rule. The <lhs-to-rhs mappings> in the input and output embeddings is a list of mappings of the form (<lhs port specification> <rhs port specification> [<data part or overlay name>]). The pair of port specifications describes the correspondence between a port on the left-hand side node and a port on a right-hand side node. The <data part or overlay name> is optional. It can name either a part of a clichéd aggregate data structure or a data overlay. For example, in the rule for CIS-Extract, there is the lhs-to-rhs mapping ((CIS-Extract 1) (Access-Base 1) Base). This maps the Base part of the CIS aggregate data structure represented by port 1 of the left-hand side node CIS-Extract to port 1 of the right-hand side node Access-Base. An example of a lhs-to-rhs mapping that includes a data overlay name is found in a rule for FIFO-Dequeue: ((FIFO-Dequeue 1) (Extract-CIS-First 1) Circular-Indexed-Sequence>FIFO). This maps the first ports of the left-hand side and right-hand side nodes to each other and it specifies that they are related by a data overlay that views a Circular-Indexed-Sequence as a FIFO queue. Similarly, the <lhs-to-lhs mappings> following the :St-Thrus keyword is a list of mappings of the form (<lhs input port specification> <lhs output port specification> [<data part or overlay name>]). Such a mapping specifies that the two left-hand side ports correspond, i.e., the rule contains a st-thru.

The <cliche relationship> given with the :L-R-Link keyword describes how the clichéd operation represented by the left-hand side node is related to the clichéd operation(s) represented by the right-hand side node(s). This information is used in annotating the links of a design tree and in generating documentation.

The explanation fragment associated with a cliché is given in the :Doc keyword, whose value consists of a <documentation string> with slots that are filled in by the <documentation arguments>. The arguments are in the form of expressions that are evaluated in the context in which the right-hand side of the rule is reduced to the left-hand side during parsing.

If a rule has been depicted in a figure in the document, then the figure's number is given in a comment preceding the rule. (There is an index of the list of figures following this appendix.)

The grammar rules are followed by an alphabetical list of the non-terminal node types and the types of their ports. For example, a node of type ABC, having three ports of type Integer, Symbol, and Queue, respectively, is listed as: (ABC 1:Integer 2:Symbol 3:Queue). The number preceding each node type specifies the page on which the rules for the node type begin.

```
(Defrule SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM
  "Sequential Simulation of Parallel Message-Passing System"
  :RHS-Node-Types
  ((SIMULATE-ASYNCHRONOUSLY . EVENT-DRIVEN-SIMULATION))
  :Input-Embedding
  (((SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM 1)
    (SIMULATE-ASYNCHRONOUSLY 3))
   ((SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM 2)
    (SIMULATE-ASYNCHRONOUSLY 1)))
  :Output-Embedding
  (((SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM 3)
    (SIMULATE-ASYNCHRONOUSLY 4)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("sequentially simulates a parallel message-passing system."))

(Defrule SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM
  "Sequential Simulation of Parallel Message-Passing System"
  :RHS-Node-Types
  ((SIMULATE-SYNCHRONOUSLY . SYNCHRONOUS-SIMULATION))
  :Input-Embedding
  (((SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM 1)
    (SIMULATE-SYNCHRONOUSLY 1))
   ((SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM 2)
    (SIMULATE-SYNCHRONOUSLY 2)))
  :Output-Embedding
  (((SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM 3)
    (SIMULATE-SYNCHRONOUSLY 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("sequentially simulates a parallel message-passing system."))

;;; Figure 4-21.

(Defrule EVENT-DRIVEN-SIMULATION
  "Event-Driven Simulation"
  :RHS-Node-Types
  ((INSERT-INITIAL-EVENT . PQ-INSERT)
  (GENERATE-EVQ+NODES . GENERATE-EVENT-QUEUES-AND-NODES)
   (ED-FINISHED? . CO-EARLIEST-EDS-FINISHED))
  :Edge-List
  (((INSERT-INITIAL-EVENT 3) . (GENERATE-EVQ+NODES 1))
   ((GENERATE-EVQ+NODES 4) . (ED-FINISHED? 2))
   ((GENERATE-EVQ+NODES 3) . (ED-FINISHED? 1)))
  :Input-Embedding
  (((EVENT-DRIVEN-SIMULATION 1) (INSERT-INITIAL-EVENT 1))
   ((EVENT-DRIVEN-SIMULATION 2) (INSERT-INITIAL-EVENT 2))
   ((EVENT-DRIVEN-SIMULATION 3) (GENERATE-EVQ+NODES 2)))
  :Output-Embedding
  (((EVENT-DRIVEN-SIMULATION 4) (ED-FINISHED? 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("asynchronously simulates a collection of processing nodes ~
    handling messages, using an event-driven algorithm.  An ~
    event queue ~A of events is maintained.  To start, an ~
    initial event ~A is inserted in the event-queue.  On each ~
    step, an event is pulled off and processed, which may ~
    create new events to be added to the event-queue. ~
    The asynchronous nodes (which represent processing nodes) ~
    are collected in an address-map, called ~A."
   (INPUT-PORT-NAME> (DOC-BP> (EVENT-DRIVEN-SIMULATION 2)))
   (INPUT-PORT-NAME> (DOC-BP> (EVENT-DRIVEN-SIMULATION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (EVENT-DRIVEN-SIMULATION 3)))))

;;; Figure 4-21.

(Defrule GENERATE-EVENT-QUEUES-AND-NODES
  "Generate Event Queues and Nodes"
  :RHS-Node-Types
  ((EVENT+NODE-GEN-F . DEQUEUE-AND-PROCESS-GENERATION))
  :Input-Embedding
  (((GENERATE-EVENT-QUEUES-AND-NODES 1) (EVENT+NODE-GEN-F 1))
   ((GENERATE-EVENT-QUEUES-AND-NODES 2) (EVENT+NODE-GEN-F 2)))
  :Output-Embedding
  (((GENERATE-EVENT-QUEUES-AND-NODES 3) (EVENT+NODE-GEN-F 3))
   ((GENERATE-EVENT-QUEUES-AND-NODES 4) (EVENT+NODE-GEN-F 4)))
 :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("generates event queues and address-maps by repeatedly ~
    dequeuing the current event queue and processing the event ~
    dequeued.  Processing an event causes new events to be ~
    added to the event queue and a new address-map to be ~
    created.  The initial event queue is ~A and the initial ~
    address-map is ~A.~%~
    The outputs of this operation are 2 series:~%~
    one is the series of event queues and the other is the ~
    series of address-maps created."
   (INPUT-PORT-NAME>
     (DOC-BP> (GENERATE-EVENT-QUEUES-AND-NODES 1)))
   (INPUT-PORT-NAME>
     (DOC-BP> (GENERATE-EVENT-QUEUES-AND-NODES 2)))))

;;; Figure 4-21.

(Defrule DEQUEUE-AND-PROCESS-GENERATION
  "Dequeue and Process Generation"
  :RHS-Node-Types
  ((DQ-EVENT . PQ-EXTRACT)
   (PROCESS-THE-EVENT . PROCESS-EVENT))
  :Edge-List
  (((DQ-EVENT 3) . (PROCESS-THE-EVENT 2))
   ((DQ-EVENT 2) . (PROCESS-THE-EVENT 1)))
  :Input-Embedding
  (((DEQUEUE-AND-PROCESS-GENERATION 1) (DQ-EVENT 1))
   ((DEQUEUE-AND-PROCESS-GENERATION 2) (PROCESS-THE-EVENT 3)))
  :St-Thrus
  (((DEQUEUE-AND-PROCESS-GENERATION 2) (DEQUEUE-AND-PROCESS-GENERATION 4))
   ((DEQUEUE-AND-PROCESS-GENERATION 1) (DEQUEUE-AND-PROCESS-GENERATION 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("dequeues the event queue ~A and processes the event dequeued,~%~
    using the address-map ~A."
   (INPUT-PORT-NAME> (DOC-BP> (DEQUEUE-AND-PROCESS-GENERATION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (DEQUEUE-AND-PROCESS-GENERATION 2)))))

;;; Figure 4-22.

(Defrule CO-EARLIEST-EDS-FINISHED
  "Co-Earliest Event-Driven Simulation Finished"
  :RHS-Node-Types
  ((EDS-FINISHED? . CO-ITERATIVE-EDS-FINISHED))
  :Input-Embedding
  (((CO-EARLIEST-EDS-FINISHED 1) (EDS-FINISHED? 1))
   ((CO-EARLIEST-EDS-FINISHED 2) (EDS-FINISHED? 2)))
  :Output-Embedding
  (((CO-EARLIEST-EDS-FINISHED 3) (EDS-FINISHED? 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("takes a sequence of event-queues and a sequence of address-maps and ~%~
    returns the address-map in the sequence of address-maps that ~%~
    corresponds to the first empty event-queue in the sequence of ~%~
    event-queues."))

;;; Figure 4-22.

(Defrule CO-ITERATIVE-EDS-FINISHED
  "Co-Iterative Event-Driven Simulation Finished"
  :RHS-Node-Types
  ((TERMINATE-EDS? . PQ-EMPTY))
  :Input-Embedding
  (((CO-ITERATIVE-EDS-FINISHED 1) (TERMINATE-EDS? 1)))
  :St-Thrus
  (((CO-ITERATIVE-EDS-FINISHED 2) (CO-ITERATIVE-EDS-FINISHED 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("terminates the simulation when the current event-queue (~A)~%~
    is empty, returning the current value of the address-map (~A).~%~
    The event-queue is implemented as a Priority Queue."
   (INPUT-PORT-NAME> (DOC-BP> (CO-ITERATIVE-EDS-FINISHED 1)))
   (INPUT-PORT-NAME> (DOC-BP> (CO-ITERATIVE-EDS-FINISHED 2)))))

;;; Figure 4-24.

(Defrule PROCESS-EVENT
  "Process Event"
  :RHS-Node-Types
  ((GET-DEST . LOOKUP-DESTINATION)
   (TIME-UPDATE . UPDATE-NODE-TIME)
   (RECORD-DEST . RECORD-AT-DESTINATION)
   (PROCESS-THE-MSG . HANDLE-MESSAGE))
  :Edge-List
  (((GET-DEST 3) . (TIME-UPDATE 1))
   ((TIME-UPDATE 3) . (RECORD-DEST 1))
   ((RECORD-DEST 4) . (PROCESS-THE-MSG 2)))
  :Input-Embedding
  (((PROCESS-EVENT 1) (PROCESS-THE-MSG 1)
    OBJECT)
   ((PROCESS-EVENT 1) (RECORD-DEST 2)
    OBJECT)
   ((PROCESS-EVENT 1) (GET-DEST 2)
    OBJECT)
   ((PROCESS-EVENT 1) (TIME-UPDATE 2)
    TIME)
   ((PROCESS-EVENT 2) (PROCESS-THE-MSG 3))
   ((PROCESS-EVENT 3) (RECORD-DEST 3))
   ((PROCESS-EVENT 3) (GET-DEST 1)))
  :Output-Embedding
  (((PROCESS-EVENT 4) (PROCESS-THE-MSG 5))
   ((PROCESS-EVENT 5) (PROCESS-THE-MSG 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("processes the event ~A whose object ~A is a Message,~%~
    using the asynchronous node that is the destination of the message.~%~
    First the time of this node is updated with respect to the~%~
    time of the event's object ~A.  Then the node~%~
```

```
    handles the message, creating a new address-map and event ~
    queue."
   (INPUT-PORT-NAME> (DOC-BP> (PROCESS-EVENT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (PROCESS-EVENT 1) OBJECT))
   (INPUT-PORT-NAME> (DOC-BP> (PROCESS-EVENT 1) TIME))))

;;; Figure 4-26.

(Defrule UPDATE-NODE-TIME
  "Update Node Time"
  :RHS-Node-Types
  ((FIND-MAX . MAX))
  :Input-Embedding
  (((UPDATE-NODE-TIME 1) (FIND-MAX 1)
    TIME)
   ((UPDATE-NODE-TIME 2) (FIND-MAX 2)))
  :Output-Embedding
  (((UPDATE-NODE-TIME 3) (FIND-MAX 3)
    TIME))
  :St-Thrus
  (((UPDATE-NODE-TIME 1) (UPDATE-NODE-TIME 3)
    MEMORY))
  :L-R-Link COMPOSITION
  :Doc
  ("updates the time of the asynchronous node ~A~%~
    to be the maximum of its current time ~A~%~
    and the input time ~A."
   (INPUT-PORT-NAME> (DOC-BP> (UPDATE-NODE-TIME 1)))
   (INPUT-PORT-NAME> (DOC-BP> (UPDATE-NODE-TIME 1) TIME))
   (INPUT-PORT-NAME> (DOC-BP> (UPDATE-NODE-TIME 2)))))

(Defrule LOCAL-BUFFER-NQ
  "Local Buffer Enqueue"
  :RHS-Node-Types
  ((BUFFER-MSG-LOCALLY . FIFO-ENQUEUE))
  :Input-Embedding
  (((LOCAL-BUFFER-NQ 1) (BUFFER-MSG-LOCALLY 1))
   ((LOCAL-BUFFER-NQ 2) (BUFFER-MSG-LOCALLY 2)
    LOCAL-BUFFER))
  :Output-Embedding
  (((LOCAL-BUFFER-NQ 3) (BUFFER-MSG-LOCALLY 3)
    LOCAL-BUFFER))
  :St-Thrus
  (((LOCAL-BUFFER-NQ 2) (LOCAL-BUFFER-NQ 3)
    MEMORY))
  :L-R-Link COMPOSITION
  :Doc
  ("enqueues the Message ~A on the local buffer of the ~
    synchronous node ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOCAL-BUFFER-NQ 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LOCAL-BUFFER-NQ 2)))))

;;; Figure 5-5.

(Defrule LOCAL-BUFFER-DQ
  "Local Buffer Dequeue"
  :RHS-Node-Types
  ((EXTRACT-MSG . FIFO-DEQUEUE))
  :Input-Embedding
  (((LOCAL-BUFFER-DQ 1) (EXTRACT-MSG 1)
    LOCAL-BUFFER))
  :Output-Embedding
  (((LOCAL-BUFFER-DQ 2) (EXTRACT-MSG 2))
   ((LOCAL-BUFFER-DQ 3) (EXTRACT-MSG 3)
    LOCAL-BUFFER))
  :St-Thrus
  (((LOCAL-BUFFER-DQ 1) (LOCAL-BUFFER-DQ 3)
    MEMORY))
  :L-R-Link COMPOSITION
  :Doc
  ("dequeues the first message (if any) from the local buffer ~
    of the Synch-Node ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOCAL-BUFFER-DQ 1)))))

(Defrule LOOKUP-NODE+NQ+UPDATE
  "Lookup Node, Enqueue Message, and Update Node Map"
  :RHS-Node-Types
  ((LOOKUP-DEST-NODE . LOOKUP-DESTINATION)
   (NQ-MSG . LOCAL-BUFFER-NQ)
   (UPDATE-MAP . RECORD-AT-DESTINATION))
  :Edge-List
  (((LOOKUP-DEST-NODE 3) . (NQ-MSG 2))
   ((NQ-MSG 3) . (UPDATE-MAP 1)))
  :Input-Embedding
  (((LOOKUP-NODE+NQ+UPDATE 1) (UPDATE-MAP 2))
   ((LOOKUP-NODE+NQ+UPDATE 1) (NQ-MSG 1))
   ((LOOKUP-NODE+NQ+UPDATE 1) (LOOKUP-DEST-NODE 2))
   ((LOOKUP-NODE+NQ+UPDATE 2) (UPDATE-MAP 3))
   ((LOOKUP-NODE+NQ+UPDATE 2) (LOOKUP-DEST-NODE 1)))
  :Output-Embedding
  (((LOOKUP-NODE+NQ+UPDATE 3) (UPDATE-MAP 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("looks up the synchronous node at the address in the ~
    Destination Address part of message ~A in the global address-map ~
    ~A.  It then creates a new node w/ the message on the front of the ~
    new node's local buffer.  The new node is added to the global ~
    address-map."
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-NODE+NQ+UPDATE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-NODE+NQ+UPDATE 2)))))

(Defrule DELIVER-MESSAGE
  "Deliver Message"
  :RHS-Node-Types
  ((MAKE-DELIVERY . LOOKUP-NODE+NQ+UPDATE))
  :Input-Embedding
  (((DELIVER-MESSAGE 1) (MAKE-DELIVERY 1))
   ((DELIVER-MESSAGE 2) (MAKE-DELIVERY 2)))
  :St-Thrus
  (((DELIVER-MESSAGE 2) (DELIVER-MESSAGE 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("iteratively delivers the message ~A to the node addressed by the~%~
    message's Destination-Address part."
   (INPUT-PORT-NAME> (DOC-BP> (DELIVER-MESSAGE 1)))))

(Defrule DELIVER-MESSAGE-ACCUMULATE
  "Deliver Message Accumulate"
  :RHS-Node-Types
  ((THE-DELIVERY . DELIVER-MESSAGE))
  :Input-Embedding
  (((DELIVER-MESSAGE-ACCUMULATE 1) (THE-DELIVERY 1))
   ((DELIVER-MESSAGE-ACCUMULATE 2) (THE-DELIVERY 2)))
  :Output-Embedding
  (((DELIVER-MESSAGE-ACCUMULATE 3) (THE-DELIVERY 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("accumulates the new nodes created by delivering the message in the~%~
    series from ~A into a new address-map ~A."
   (INPUT-PORT-NAME> (DOC-BP> (DELIVER-MESSAGE-ACCUMULATE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (DELIVER-MESSAGE-ACCUMULATE 2)))))

(Defrule ENUMERATE-AND-DELIVER-MESSAGES
  "Enumerate and Deliver Messages"
  :RHS-Node-Types
  ((ENUMERATE-MESSAGES . DESTRUCTIVE-QUEUE-ENUMERATION)
   (DELIVER-THE-MESSAGES . DELIVER-MESSAGE-ACCUMULATE))
  :Edge-List
  (((ENUMERATE-MESSAGES 2) . (DELIVER-THE-MESSAGES 1)))
  :Input-Embedding
  (((ENUMERATE-AND-DELIVER-MESSAGES 1) (ENUMERATE-MESSAGES 1))
   ((ENUMERATE-AND-DELIVER-MESSAGES 2) (DELIVER-THE-MESSAGES 2)))
  :Output-Embedding
  (((ENUMERATE-AND-DELIVER-MESSAGES 3) (DELIVER-THE-MESSAGES 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the messages in the global message buffer ~A ~
    and delivers each one to the nodes addressed by the message's ~
    Destination Address part.  The new nodes created during delivery ~
    are accumulated into a global address-map, implemented as a ~
    sequence, whose initial value is ~A.~%~
    The new (accumulated) global address-map is returned."
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-AND-DELIVER-MESSAGES 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-AND-DELIVER-MESSAGES 2)))))

(Defrule DELIVER-MESSAGES
  "Deliver Messages"
  :RHS-Node-Types
  ((ENUMERATE-AND-DELIVER . ENUMERATE-AND-DELIVER-MESSAGES))
  :Input-Embedding
  (((DELIVER-MESSAGES 1) (ENUMERATE-AND-DELIVER 1))
   ((DELIVER-MESSAGES 2) (ENUMERATE-AND-DELIVER 2)))
  :Output-Embedding
  (((DELIVER-MESSAGES 3) (ENUMERATE-AND-DELIVER 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("delivers the messages in the global message buffer ~A, creating ~%~
    new nodes, which are accumulated into a global address-map ~%~
    whose initial value is ~A."
   (INPUT-PORT-NAME> (DOC-BP> (DELIVER-MESSAGES 1)))
   (INPUT-PORT-NAME> (DOC-BP> (DELIVER-MESSAGES 2)))))

(Defrule LOCAL-BUFFER-EMPTY?
  "Local Buffer Empty Test"
  :RHS-Node-Types
  ((CHECK-BUFFER . FIFO-EMPTY?))
  :Input-Embedding
  (((LOCAL-BUFFER-EMPTY? 1) (CHECK-BUFFER 1) LOCAL-BUFFER))
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether the local buffer of synchronous node ~A is empty."
   (INPUT-PORT-NAME> (DOC-BP> (LOCAL-BUFFER-EMPTY? 1)))))

(Defrule LOCAL-BUFFER-NONEMPTY?
  "Local Buffer Nonempty Test"
  :RHS-Node-Types
  ((CHECK-BUFFER . FIFO-EMPTY?))
```

```
  :Input-Embedding
  (((LOCAL-BUFFER-NONEMPTY? 1) (CHECK-BUFFER 1)
    LOCAL-BUFFER))
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether the local buffer of synchronous node ~A is ~
    nonempty."
   (INPUT-PORT-NAME> (DOC-BP> (LOCAL-BUFFER-NONEMPTY? 1)))))

(Defrule LOCAL-BUFFERS-ALWAYS-EMPTY?
  "Local Buffer Always Empty Test"
  :RHS-Node-Types
  ((CONTINUOUS-CHECK . LOCAL-BUFFER-NONEMPTY?))
  :Input-Embedding
  (((LOCAL-BUFFERS-ALWAYS-EMPTY? 1) (CONTINUOUS-CHECK 1)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("continually checks that each node in the input series of ~
    nodes ~A has an empty local buffer."
   (INPUT-PORT-NAME> (DOC-BP> (LOCAL-BUFFERS-ALWAYS-EMPTY? 1)))))

(Defrule ENUM-NODES+CHECK-BUFFERS
  "Enumerate Nodes and Check Buffers"
  :RHS-Node-Types
  ((ENUMERATE-NODES . SEQUENCE-ENUMERATION)
   (BUFFER-ALWAYS-EMPTY . LOCAL-BUFFERS-ALWAYS-EMPTY?))
  :Edge-List
  (((ENUMERATE-NODES 2) . (BUFFER-ALWAYS-EMPTY 1)))
  :Input-Embedding
  (((ENUM-NODES+CHECK-BUFFERS 1) (ENUMERATE-NODES 1)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the sequence of nodes ~A and checks that each ~
    node has an empty local buffer."
   (INPUT-PORT-NAME> (DOC-BP> (ENUM-NODES+CHECK-BUFFERS 1)))))

(Defrule LOCAL-BUFFERS-EMPTY?
  "Local Buffers Empty"
  :RHS-Node-Types
  ((CHECK-ALL-NODE-BUFFERS . ENUM-NODES+CHECK-BUFFERS))
  :Input-Embedding
  (((LOCAL-BUFFERS-EMPTY? 1) (CHECK-ALL-NODE-BUFFERS 1)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("checks that all nodes in ~A have an empty local buffer."
   (INPUT-PORT-NAME> (DOC-BP> (LOCAL-BUFFERS-EMPTY? 1)))))

(Defrule GLOBAL-AND-LOCAL-BUFFERS-EMPTY?
  "Global and Local Buffers Empty Test"
  :RHS-Node-Types
  ((CHECK-LOCAL-NODE-BUFFERS . LOCAL-BUFFERS-EMPTY?)
   (CHECK-GLOBAL-BUFFER . QUEUE-EMPTY?))
  :Input-Embedding
  (((GLOBAL-AND-LOCAL-BUFFERS-EMPTY? 1)
    (CHECK-LOCAL-NODE-BUFFERS 1))
   ((GLOBAL-AND-LOCAL-BUFFERS-EMPTY? 2)
    (CHECK-GLOBAL-BUFFER 1)))
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether the local buffers of the synchronous nodes in ~
    are all empty and the global message buffer ~A is also empty."
   (INPUT-PORT-NAME>
     (DOC-BP> (GLOBAL-AND-LOCAL-BUFFERS-EMPTY? 1)))
   (INPUT-PORT-NAME>
     (DOC-BP> (GLOBAL-AND-LOCAL-BUFFERS-EMPTY? 2)))))

(Defrule SYNCHRONOUS-SIMULATION-FINISHED?
  "Synchronous Simulation Finished?"
  :RHS-Node-Types
  ((CHECK-ALL-BUFFERS . GLOBAL-AND-LOCAL-BUFFERS-EMPTY?))
  :Input-Embedding
  (((SYNCHRONOUS-SIMULATION-FINISHED? 1) (CHECK-ALL-BUFFERS 1))
   ((SYNCHRONOUS-SIMULATION-FINISHED? 2) (CHECK-ALL-BUFFERS 2)))
  :St-Thrus
  (((SYNCHRONOUS-SIMULATION-FINISHED? 1)
    (SYNCHRONOUS-SIMULATION-FINISHED? 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether a synchronous simulation is finished by ~
    testing whether the global buffer and all of the nodes' ~
    local buffers are empty."))

(Defrule EXTRACT-AND-HANDLE-FIRST-MESSAGE
  "Extract and Handle First Message"
  :RHS-Node-Types
  ((HAS-WORK? . LOCAL-BUFFER-NONEMPTY?)
   (EXTRACT-FIRST-MSG . LOCAL-BUFFER-DQ)
   (RECORD-WORKING-NODE . NEW-TERM)
   (HANDLE-THE-MESSAGE . HANDLE-MESSAGE))
  :Edge-List
  (((EXTRACT-FIRST-MSG 2) . (HANDLE-THE-MESSAGE 1))
   ((EXTRACT-FIRST-MSG 3) . (RECORD-WORKING-NODE 1))
   ((RECORD-WORKING-NODE 4) . (HANDLE-THE-MESSAGE 2)))
  :Input-Embedding
  (((EXTRACT-AND-HANDLE-FIRST-MESSAGE 1) (EXTRACT-FIRST-MSG 1))
   ((EXTRACT-AND-HANDLE-FIRST-MESSAGE 1) (HAS-WORK? 1))
   ((EXTRACT-AND-HANDLE-FIRST-MESSAGE 2) (RECORD-WORKING-NODE 2))
   ((EXTRACT-AND-HANDLE-FIRST-MESSAGE 3) (RECORD-WORKING-NODE 3))
   ((EXTRACT-AND-HANDLE-FIRST-MESSAGE 4) (HANDLE-THE-MESSAGE 3)))
  :Output-Embedding
  (((EXTRACT-AND-HANDLE-FIRST-MESSAGE 5) (HANDLE-THE-MESSAGE 4))
   ((EXTRACT-AND-HANDLE-FIRST-MESSAGE 6) (HANDLE-THE-MESSAGE 5)))
  :St-Thrus
  (((EXTRACT-AND-HANDLE-FIRST-MESSAGE 4)
    (EXTRACT-AND-HANDLE-FIRST-MESSAGE 6))
   ((EXTRACT-AND-HANDLE-FIRST-MESSAGE 3)
    (EXTRACT-AND-HANDLE-FIRST-MESSAGE 5)))
  :L-R-Link COMPOSITION
  :Doc
  ("extracts the first message from the local buffer of synchronous node~%~
    ~A if the node has work, i.e., messages queued up.  The message is~%~
    then processed, which may generate new messages.  The new messages ~%~
    are collected on the message queue."
   (INPUT-PORT-NAME> (DOC-BP> (EXTRACT-AND-HANDLE-FIRST-MESSAGE 1)))))

(Defrule DO-WORK-ACCUMULATION
  "Do Work Accumulation"
  :RHS-Node-Types
  ((EXTRACT-AND-HANDLE . EXTRACT-AND-HANDLE-FIRST-MESSAGE))
  :Input-Embedding
  (((DO-WORK-ACCUMULATION 1) (EXTRACT-AND-HANDLE 1))
   ((DO-WORK-ACCUMULATION 2) (EXTRACT-AND-HANDLE 2))
   ((DO-WORK-ACCUMULATION 3) (EXTRACT-AND-HANDLE 3))
   ((DO-WORK-ACCUMULATION 4) (EXTRACT-AND-HANDLE 4)))
  :St-Thrus
  (((DO-WORK-ACCUMULATION 4) (DO-WORK-ACCUMULATION 6))
   ((DO-WORK-ACCUMULATION 3) (DO-WORK-ACCUMULATION 5)))
  :L-R-Link COMPOSITION
  :Doc
  ("iteratively receives a synchronous node ~A, extracts and handles its~
    first message if it has one in its local buffer, and accumulates the~
    new messages that this generates in a global message buffer ~A.  This~
    also creates new nodes, which are accumulated in an address-map, whose~
    initial value is ~A."
   (INPUT-PORT-NAME> (DOC-BP> (DO-WORK-ACCUMULATION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (DO-WORK-ACCUMULATION 4)))
   (INPUT-PORT-NAME> (DOC-BP> (DO-WORK-ACCUMULATION 3)))))

(Defrule DO-WORK-ACCUMULATE
  "Do Work Accumulate"
  :RHS-Node-Types
  ((DW-ACCUMULATION . DO-WORK-ACCUMULATION))
  :Input-Embedding
  (((DO-WORK-ACCUMULATE 1) (DW-ACCUMULATION 1))
   ((DO-WORK-ACCUMULATE 2) (DW-ACCUMULATION 2))
   ((DO-WORK-ACCUMULATE 3) (DW-ACCUMULATION 3))
   ((DO-WORK-ACCUMULATE 4) (DW-ACCUMULATION 4)))
  :Output-Embedding
  (((DO-WORK-ACCUMULATE 5) (DW-ACCUMULATION 5))
   ((DO-WORK-ACCUMULATE 6) (DW-ACCUMULATION 6)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("takes a series of nodes and simulates them taking one step (i.e.,~
    handling one message a piece from their local buffers).  It ~
    accumulates the new nodes that this creates in an address-map, which ~
    is given as output.  It also accumulates all new messages generated ~
    during the node stepping in a global message buffer, which it also ~
    produces as output.  The initial value of the address-map is ~A and ~
    of the global message buffer is ~A."
   (INPUT-PORT-NAME> (DOC-BP> (DO-WORK-ACCUMULATION 3)))
   (INPUT-PORT-NAME> (DOC-BP> (DO-WORK-ACCUMULATION 4)))))

(Defrule POLL-NODES-AND-DO-WORK
  "Poll Nodes and Do Work"
  :RHS-Node-Types
  ((POLL-NODES . SEQUENCE-AND-INDEX-ENUMERATION)
   (WORK . DO-WORK-ACCUMULATE))
  :Edge-List
  (((POLL-NODES 3) . (WORK 2))
   ((POLL-NODES 2) . (WORK 1)))
  :Input-Embedding
  (((POLL-NODES-AND-DO-WORK 1) (WORK 3))
   ((POLL-NODES-AND-DO-WORK 1) (POLL-NODES 1)))
  :Output-Embedding
  (((POLL-NODES-AND-DO-WORK 2) (WORK 5))
   ((POLL-NODES-AND-DO-WORK 3) (WORK 6)))
  :L-R-Link COMPOSITION
  :Doc
  ("polls all nodes in ~A and for each node that has messages on its ~
    local queue, it handles one of the messages."
   (INPUT-PORT-NAME> (DOC-BP> (POLL-NODES-AND-DO-WORK 1)))))

(Defrule ADVANCE-NODES
  "Advance Nodes"
  :RHS-Node-Types
  ((STEP-NODES . POLL-NODES-AND-DO-WORK))
```

```
  :Input-Embedding
  (((ADVANCE-NODES 1) (STEP-NODES 1)))
  :Output-Embedding
  (((ADVANCE-NODES 2) (STEP-NODES 2))
   ((ADVANCE-NODES 3) (STEP-NODES 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("steps each node in ~A that has work by processing 1 message ~
    each."
   (INPUT-PORT-NAME> (DOC-BP> (ADVANCE-NODES 1)))))

(Defrule EARLIEST-SIMULATION-FINISHED
  "Earliest Simulation Finished"
  :RHS-Node-Types
  ((FINISHED-TEST . SYNCHRONOUS-SIMULATION-FINISHED?))
  :Input-Embedding
  (((EARLIEST-SIMULATION-FINISHED 1) (FINISHED-TEST 1))
   ((EARLIEST-SIMULATION-FINISHED 2) (FINISHED-TEST 2)))
  :Output-Embedding
  (((EARLIEST-SIMULATION-FINISHED 3) (FINISHED-TEST 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("takes two input sequences: a sequence of address-maps, ~
    starting with ~A, and a sequence of global message buffers, ~
    starting with ~A.  It outputs the first address-map in the ~
    input sequence of address-maps that satisfies the predicate ~
    that all nodes in the address-map have empty local buffers ~
    and the corresponding global message buffer is empty."
   (INPUT-PORT-NAME> (DOC-BP> (EARLIEST-SIMULATION-FINISHED 1)))
   (INPUT-PORT-NAME> (DOC-BP> (EARLIEST-SIMULATION-FINISHED 2)))))

(Defrule DELIVER-MESSAGES-AND-STEP-NODES
  "Generate by Message Delivery and Node Stepping"
  :RHS-Node-Types
  ((DELIVER-ALL-MSGS . DELIVER-MESSAGES)
   (STEP-ALL-NODES . ADVANCE-NODES))
  :Edge-List
  (((DELIVER-ALL-MSGS 3) . (STEP-ALL-NODES 1)))
  :Input-Embedding
  (((DELIVER-MESSAGES-AND-STEP-NODES 1) (DELIVER-ALL-MSGS 2))
   ((DELIVER-MESSAGES-AND-STEP-NODES 2) (DELIVER-ALL-MSGS 1)))
  :St-Thrus
  (((DELIVER-MESSAGES-AND-STEP-NODES 2)
    (DELIVER-MESSAGES-AND-STEP-NODES 4))
   ((DELIVER-MESSAGES-AND-STEP-NODES 1)
    (DELIVER-MESSAGES-AND-STEP-NODES 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("generates address-maps and global message buffers by ~
    repeatedly delivering all messages in the global message ~
    buffer ~A and advancing the nodes ~A by one step each. ~
    This causes more messages to be generated and added to the ~
    global message buffer and a new address-map to be created ~
    on each iteration.  The outputs of this operation are 2 ~
    series: one is the series of address-maps created and the ~
    other is the series of global message buffers."
   (INPUT-PORT-NAME>
     (DOC-BP> (DELIVER-MESSAGES-AND-STEP-NODES 2)))
   (INPUT-PORT-NAME>
     (DOC-BP> (DELIVER-MESSAGES-AND-STEP-NODES 1)))))

(Defrule GENERATE-GLOBAL-BUFFERS-AND-NODES
  "Generate Global Message Buffer and Nodes"
  :RHS-Node-Types
  ((GEN-BUFFER-AND-NODES . DELIVER-MESSAGES-AND-STEP-NODES))
  :Input-Embedding
  (((GENERATE-GLOBAL-BUFFERS-AND-NODES 1)
    (GEN-BUFFER-AND-NODES 1))
   ((GENERATE-GLOBAL-BUFFERS-AND-NODES 2)
    (GEN-BUFFER-AND-NODES 2)))
  :Output-Embedding
  (((GENERATE-GLOBAL-BUFFERS-AND-NODES 3)
    (GEN-BUFFER-AND-NODES 3))
   ((GENERATE-GLOBAL-BUFFERS-AND-NODES 4)
    (GEN-BUFFER-AND-NODES 4)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("generates address-maps and global message buffers by ~
    repeatedly delivering all messages in the global message ~
    buffer ~A and advancing the synchronous nodes in ~A by one ~
    step each."
   (INPUT-PORT-NAME>
     (DOC-BP> (GENERATE-GLOBAL-BUFFERS-AND-NODES 2)))
   (INPUT-PORT-NAME>
     (DOC-BP> (GENERATE-GLOBAL-BUFFERS-AND-NODES 1)))))

(Defrule SYNCHRONOUS-SIMULATION-W-GLOBAL-MESSAGE-BUFFER
  "Synchronous Simulation using Global Message Buffer"
  :RHS-Node-Types
  ((INITIAL-INSERT . QUEUE-INSERT)
   (SIMULATION-STEP . GENERATE-GLOBAL-BUFFERS-AND-NODES)
   (SIMULATION-FINISHED? . EARLIEST-SIMULATION-FINISHED))
  :Edge-List
  (((INITIAL-INSERT 3) . (SIMULATION-STEP 2))
   ((SIMULATION-STEP 4) . (SIMULATION-FINISHED? 2))
   ((SIMULATION-STEP 3) . (SIMULATION-FINISHED? 1)))
  :Input-Embedding
  (((SYNCHRONOUS-SIMULATION-W-GLOBAL-MESSAGE-BUFFER 1) (SIMULATION-STEP 1))
   ((SYNCHRONOUS-SIMULATION-W-GLOBAL-MESSAGE-BUFFER 2) (INITIAL-INSERT 1)))
  :Output-Embedding
  (((SYNCHRONOUS-SIMULATION-W-GLOBAL-MESSAGE-BUFFER 3)
    (SIMULATION-FINISHED? 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("iteratively advances each synchronous node in ~A by handling one ~
    message a piece.  It uses a global message buffer to ensure that ~
    nodes advance in lock-step.  The global buffer's initial value is ~
    ~A.  The simulation starts by adding an initial message ~A to ~A. ~
    The simulation ends when no node has work to do (i.e., no more ~
    messages to handle) and the global message buffer ~A is empty. ~
    As messages are handled, new messages are created which are ~
    buffered on the global message buffer."
   (INPUT-PORT-NAME>
     (DOC-BP> (SYNCHRONOUS-SIMULATION-W-GLOBAL-MESSAGE-BUFFER 1)))
   (INPUT-PORT-NAME> (DOC-BP> (INITIAL-INSERT 2)))
   (INPUT-PORT-NAME>
     (DOC-BP> (SYNCHRONOUS-SIMULATION-W-GLOBAL-MESSAGE-BUFFER 2)))
   (INPUT-PORT-NAME> (DOC-BP> (INITIAL-INSERT 2)))
   (INPUT-PORT-NAME> (DOC-BP> (INITIAL-INSERT 2)))))

(Defrule SYNCHRONOUS-SIMULATION
  "Synchronous Simulation using Global Buffer"
  :RHS-Node-Types
  ((SIMULATE-W-BUFFER . SYNCHRONOUS-SIMULATION-W-GLOBAL-MESSAGE-BUFFER))
  :Input-Embedding
  (((SYNCHRONOUS-SIMULATION 1) (SIMULATE-W-BUFFER 1))
   ((SYNCHRONOUS-SIMULATION 2) (SIMULATE-W-BUFFER 2)))
  :Output-Embedding
  (((SYNCHRONOUS-SIMULATION 3) (SIMULATE-W-BUFFER 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("synchronously simulates a collection of processing nodes handling ~
    messages.  The synchronous nodes (which represent the processing ~
    nodes) are collected in an address-map, called ~A.  Each node ~
    maintains a local buffer of pending messages to handle."
   (INPUT-PORT-NAME> (DOC-BP> (SYNCHRONOUS-SIMULATION 1)))))

(Defrule ENUMERATE-NODES+COMPUTE-AVERAGE
  "Enumerate Nodes and Compute Average"
  :RHS-Node-Types
  ((ENUM-NODES . SEQUENCE-AND-INDEX-ENUMERATION)
   (COMPUTE-BUFFER-SIZE . SUM)
   (SIZE-OF-SEQUENCE . SEQUENCE-SIZE)
   (COMPUTE-AVG . DIVIDE))
  :Edge-List
  (((ENUM-NODES 2) . (COMPUTE-BUFFER-SIZE 1))
   ((COMPUTE-BUFFER-SIZE 2) . (COMPUTE-AVG 1))
   ((SIZE-OF-SEQUENCE 2) . (COMPUTE-AVG 2)))
  :Input-Embedding
  (((ENUMERATE-NODES+COMPUTE-AVERAGE 1) (SIZE-OF-SEQUENCE 1))
   ((ENUMERATE-NODES+COMPUTE-AVERAGE 1) (ENUM-NODES 1)))
  :Output-Embedding
  (((ENUMERATE-NODES+COMPUTE-AVERAGE 2) (COMPUTE-AVG 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates all nodes in ~A and computes the average of the sizes ~%~
    of their local buffers."
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-NODES+COMPUTE-AVERAGE 1)))))

(Defrule AVERAGE-LOCAL-BUFFER-SIZE
  "Average Local Buffer Size"
  :RHS-Node-Types
  ((AVG-LB-SIZE . ENUMERATE-NODES+COMPUTE-AVERAGE))
  :Input-Embedding
  (((AVERAGE-LOCAL-BUFFER-SIZE 1) (AVG-LB-SIZE 1)))
  :Output-Embedding
  (((AVERAGE-LOCAL-BUFFER-SIZE 2) (AVG-LB-SIZE 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("computes the average of the local buffer sizes of all nodes in ~A."
   (INPUT-PORT-NAME> (DOC-BP> (AVERAGE-LOCAL-BUFFER-SIZE 1)))))

(Defrule DESTRUCTIVE-QUEUE-ENUMERATION
  "Destructive Queue Enumeration"
  :RHS-Node-Types
  ((ENUM-PQ . PQ-ENUMERATION))
  :Input-Embedding
  (((DESTRUCTIVE-QUEUE-ENUMERATION 1) (ENUM-PQ 1)
    PRIORITY-QUEUE>QUEUE))
  :Output-Embedding
  (((DESTRUCTIVE-QUEUE-ENUMERATION 2) (ENUM-PQ 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("destructively enumerates the Queue ~A, which is implemented~%~
    as a Priority Queue."
   (INPUT-PORT-NAME> (DOC-BP> (DESTRUCTIVE-QUEUE-ENUMERATION 1)))))
```

```
(Defrule DESTRUCTIVE-QUEUE-ENUMERATION
  "Destructive Queue Enumeration"
  :RHS-Node-Types
  ((ENUM-FIFO . FIFO-DESTRUCTIVE-ENUMERATION))
  :Input-Embedding
  (((DESTRUCTIVE-QUEUE-ENUMERATION 1) (ENUM-FIFO 1)
    FIFO>QUEUE))
  :Output-Embedding
  (((DESTRUCTIVE-QUEUE-ENUMERATION 2) (ENUM-FIFO 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("destructively enumerates the Queue ~A, which is ~
    implemented as a FIFO."
   (INPUT-PORT-NAME>
     (DOC-BP> (DESTRUCTIVE-QUEUE-ENUMERATION 1)))))

(Defrule DESTRUCTIVE-QUEUE-ENUMERATION
  "Destructive Queue Enumeration"
  :RHS-Node-Types
  ((ENUM-STACK . STACK-ENUMERATION))
  :Input-Embedding
  (((DESTRUCTIVE-QUEUE-ENUMERATION 1) (ENUM-STACK 1)
    STACK>QUEUE))
  :Output-Embedding
  (((DESTRUCTIVE-QUEUE-ENUMERATION 2) (ENUM-STACK 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("destructively enumerates the Queue ~A, which is ~
    implemented as a Stack."
   (INPUT-PORT-NAME>
     (DOC-BP> (DESTRUCTIVE-QUEUE-ENUMERATION 1)))))

(Defrule STACK-ENUMERATION
  "Stack Enumeration"
  :RHS-Node-Types
  ((ENUM-LL-DESTRUCTIVELY . LE))
  :Input-Embedding
  (((STACK-ENUMERATION 1) (ENUM-LL-DESTRUCTIVELY 1)
    LINKED-LIST>STACK))
  :Output-Embedding
  (((STACK-ENUMERATION 2) (ENUM-LL-DESTRUCTIVELY 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("destructively enumerates the Stack ~A, which is ~
    implemented as a Linked-List."
   (INPUT-PORT-NAME> (DOC-BP> (STACK-ENUMERATION 1)))))

(Defrule STACK-ENUMERATION
  "Stack Enumeration"
  :RHS-Node-Types
  ((ENUM-IS-DESTRUCTIVELY . INDEXED-SEQUENCE-ENUMERATION))
  :Input-Embedding
  (((STACK-ENUMERATION 1) (ENUM-IS-DESTRUCTIVELY 1)
    INDEXED-SEQUENCE>STACK))
  :Output-Embedding
  (((STACK-ENUMERATION 2) (ENUM-IS-DESTRUCTIVELY 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("destructively enumerates the Stack ~A, which is ~
   implemented as an Indexed Sequence."
   (INPUT-PORT-NAME> (DOC-BP> (STACK-ENUMERATION 1)))))

(Defrule QUEUE-EXTRACT
  "Queue Extract"
  :RHS-Node-Types
  ((EXTRACT-FROM-PQ . PQ-EXTRACT))
  :Input-Embedding
  (((QUEUE-EXTRACT 1) (EXTRACT-FROM-PQ 1)
    PRIORITY-QUEUE>QUEUE))
  :Output-Embedding
  (((QUEUE-EXTRACT 2) (EXTRACT-FROM-PQ 2))
   ((QUEUE-EXTRACT 3) (EXTRACT-FROM-PQ 3)
    PRIORITY-QUEUE>QUEUE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("extracts an element from the queue ~A, which is ~
    implemented  as a Priority Queue."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-EXTRACT 1)))))

(Defrule QUEUE-EXTRACT
  "Queue Extract"
  :RHS-Node-Types
  ((EXTRACT-FROM-FIFO . FIFO-DEQUEUE))
  :Input-Embedding
  (((QUEUE-EXTRACT 1) (EXTRACT-FROM-FIFO 1)
    FIFO>QUEUE))
  :Output-Embedding
  (((QUEUE-EXTRACT 2) (EXTRACT-FROM-FIFO 2))
   ((QUEUE-EXTRACT 3) (EXTRACT-FROM-FIFO 3)
    FIFO>QUEUE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("extracts an element from the queue ~A, which is ~
    implemented as a FIFO."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-EXTRACT 1)))))

(Defrule QUEUE-EXTRACT
  "Queue Extract"
  :RHS-Node-Types
  ((EXTRACT-FROM-STACK . STACK-POP))
  :Input-Embedding
  (((QUEUE-EXTRACT 1) (EXTRACT-FROM-STACK 1)
    STACK>QUEUE))
  :Output-Embedding
  (((QUEUE-EXTRACT 2) (EXTRACT-FROM-STACK 2))
   ((QUEUE-EXTRACT 3) (EXTRACT-FROM-STACK 3)
    STACK>QUEUE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("extracts an element from the queue ~A, which is implemented as a ~
    Stack."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-EXTRACT 1)))))

(Defrule QUEUE-INSERT
  "Queue Insert"
  :RHS-Node-Types
  ((ADD-TO-Q3 . PQ-INSERT))
  :Input-Embedding
  (((QUEUE-INSERT 1) (ADD-TO-Q3 1))
   ((QUEUE-INSERT 2) (ADD-TO-Q3 2)
    PRIORITY-QUEUE>QUEUE))
  :Output-Embedding
  (((QUEUE-INSERT 3) (ADD-TO-Q3 3)
    PRIORITY-QUEUE>QUEUE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("enqueues ~A on the Queue ~A, which is implemented as a ~
    Priority-Queue."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-INSERT 2)))))

(Defrule QUEUE-INSERT
  "Queue Insert"
  :RHS-Node-Types
  ((ADD-TO-Q2 . FIFO-ENQUEUE))
  :Input-Embedding
  (((QUEUE-INSERT 1) (ADD-TO-Q2 1))
   ((QUEUE-INSERT 2) (ADD-TO-Q2 2)
    FIFO>QUEUE))
  :Output-Embedding
  (((QUEUE-INSERT 3) (ADD-TO-Q2 3)
    FIFO>QUEUE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("enqueues ~A on the Queue ~A, which is implemented as a FIFO."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-INSERT 2)))))

(Defrule QUEUE-INSERT
  "Queue Insert"
  :RHS-Node-Types
  ((ADD-TO-Q1 . STACK-PUSH))
  :Input-Embedding
  (((QUEUE-INSERT 1) (ADD-TO-Q1 1))
   ((QUEUE-INSERT 2) (ADD-TO-Q1 2)
    STACK>QUEUE))
  :Output-Embedding
  (((QUEUE-INSERT 3) (ADD-TO-Q1 3)
    STACK>QUEUE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("enqueues ~A on the Queue ~A, which is implemented as a Stack."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-INSERT 2)))))

(Defrule QUEUE-EMPTY?
  "Queue Empty?"
  :RHS-Node-Types
  ((EMPTY3? . PQ-EMPTY))
  :Input-Embedding
  (((QUEUE-EMPTY? 1) (EMPTY3? 1)
    PRIORITY-QUEUE>QUEUE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("tests whether the Queue ~A is empty.~%~
    The Queue is implemented as a Priority-Queue."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-EMPTY? 1)))))

(Defrule QUEUE-EMPTY?
  "Queue Empty?"
  :RHS-Node-Types
  ((EMPTY2? . FIFO-EMPTY?))
  :Input-Embedding
  (((QUEUE-EMPTY? 1) (EMPTY2? 1)
    FIFO>QUEUE))
  :L-R-Link IMPLEMENTATION
```

```
  :Doc
  ("tests whether the Queue ~A is empty.~%~
    The Queue is implemented as a FIFO."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-EMPTY? 1)))))

(Defrule QUEUE-EMPTY?
  "Queue Empty?"
  :RHS-Node-Types
  ((EMPTY1? . STACK-EMPTY?))
  :Input-Embedding
  (((QUEUE-EMPTY? 1) (EMPTY1? 1)
    STACK>QUEUE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("tests whether the Queue ~A is empty.~%~
    The Queue is implemented as a Stack."
   (INPUT-PORT-NAME> (DOC-BP> (QUEUE-EMPTY? 1)))))

(Defrule STACK-EMPTY?
  "Stack Empty?"
  :RHS-Node-Types
  ((LL-EMPTY? . LIST-EMPTY))
  :Input-Embedding
  (((STACK-EMPTY? 1) (LL-EMPTY? 1)
    LINKED-LIST>STACK))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("tests whether the Stack ~A is empty.~%~
    The Stack is implemented as a Linked List."
   (INPUT-PORT-NAME> (DOC-BP> (STACK-EMPTY? 1)))))

(Defrule STACK-EMPTY?
  "Stack Empty?"
  :RHS-Node-Types
  ((IS-EMPTY? . INDEXED-SEQUENCE-EMPTY))
  :Input-Embedding
  (((STACK-EMPTY? 1) (IS-EMPTY? 1)
    INDEXED-SEQUENCE>STACK))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("tests whether the Stack ~A is empty.~%~
    The Stack is implemented as an Indexed Sequence."
   (INPUT-PORT-NAME> (DOC-BP> (STACK-EMPTY? 1)))))

(Defrule STACK-PUSH
  "Stack Push"
  :RHS-Node-Types
  ((ADD-TO-LL . LIST-PUSH))
  :Input-Embedding
  (((STACK-PUSH 1) (ADD-TO-LL 1))
   ((STACK-PUSH 2) (ADD-TO-LL 2)
    LINKED-LIST>STACK))
  :Output-Embedding
  (((STACK-PUSH 3) (ADD-TO-LL 3)
    LINKED-LIST>STACK))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("pushes ~A onto the stack ~A, which is implemented as a ~
    Linked List."
   (INPUT-PORT-NAME> (DOC-BP> (STACK-PUSH 1)))
   (INPUT-PORT-NAME> (DOC-BP> (STACK-PUSH 2)))))

(Defrule STACK-PUSH
  "Stack Push"
  :RHS-Node-Types
  ((ADD-TO-IS . INDEXED-SEQUENCE-INSERT))
  :Input-Embedding
  (((STACK-PUSH 1) (ADD-TO-IS 1))
   ((STACK-PUSH 2) (ADD-TO-IS 2)
    INDEXED-SEQUENCE>STACK))
  :Output-Embedding
  (((STACK-PUSH 3) (ADD-TO-IS 3)
    INDEXED-SEQUENCE>STACK))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("pushes ~A onto the stack ~A, which is implemented as an ~
    Indexed Sequence."
   (INPUT-PORT-NAME> (DOC-BP> (STACK-PUSH 1)))
   (INPUT-PORT-NAME> (DOC-BP> (STACK-PUSH 2)))))

(Defrule STACK-POP
  "Stack-Pop"
  :RHS-Node-Types
  ((EXTRACT-FROM-LL . LIST-POP))
  :Input-Embedding
  (((STACK-POP 1) (EXTRACT-FROM-LL 1)
    LINKED-LIST>STACK))
  :Output-Embedding
  (((STACK-POP 2) (EXTRACT-FROM-LL 2))
   ((STACK-POP 3) (EXTRACT-FROM-LL 3)
    LINKED-LIST>STACK))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("pops the stack ~A, which is implemented as a Linked List."
   (INPUT-PORT-NAME> (DOC-BP> (STACK-POP 1)))))

(Defrule STACK-POP
  "Stack-Pop"
  :RHS-Node-Types
  ((EXTRACT-FROM-IS . INDEXED-SEQUENCE-EXTRACT))
  :Input-Embedding
  (((STACK-POP 1) (EXTRACT-FROM-IS 1)
    INDEXED-SEQUENCE>STACK))
  :Output-Embedding
  (((STACK-POP 2) (EXTRACT-FROM-IS 2))
   ((STACK-POP 3) (EXTRACT-FROM-IS 3)
    INDEXED-SEQUENCE>STACK))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("pops the stack ~A, which is implemented as an indexed-sequence."
   (INPUT-PORT-NAME> (DOC-BP> (STACK-POP 1)))))

(Defrule CIS-DESTRUCTIVE-ENUMERATION
  "Circular-Indexed-Sequence Destructive Enumeration"
  :RHS-Node-Types
  ((ENUM-FINISHED? . CIS-EMPTY)
   (EXTRACT-NEXT . CIS-EXTRACT))
  :Input-Embedding
  (((CIS-DESTRUCTIVE-ENUMERATION 1) (EXTRACT-NEXT 1))
   ((CIS-DESTRUCTIVE-ENUMERATION 1) (ENUM-FINISHED? 1)))
  :Output-Embedding
  (((CIS-DESTRUCTIVE-ENUMERATION 2) (EXTRACT-NEXT 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates all of the elements in the Circular-Indexed-Sequence ~A, ~
    by destructively extracting them from the sequence.  The sequence ~
    is filled in ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CIS-DESTRUCTIVE-ENUMERATION 1)))
   (GROWTH-DIRECTION (N> CIS-DESTRUCTIVE-ENUMERATION))))

(Defrule FIFO-DESTRUCTIVE-ENUMERATION
  "FIFO Destructive Enumeration"
  :RHS-Node-Types
  ((ENUM-CIS-DESTRUCTIVELY . CIS-DESTRUCTIVE-ENUMERATION))
  :Input-Embedding
  (((FIFO-DESTRUCTIVE-ENUMERATION 1) (ENUM-CIS-DESTRUCTIVELY 1)
    CIRCULAR-INDEXED-SEQUENCE>FIFO))
  :Output-Embedding
  (((FIFO-DESTRUCTIVE-ENUMERATION 2) (ENUM-CIS-DESTRUCTIVELY 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("destructively enumerates the FIFO queue ~A, which is implemented ~
    as a Circular Indexed Sequence."
   (INPUT-PORT-NAME> (DOC-BP> (FIFO-DESTRUCTIVE-ENUMERATION 1)))))

(Defrule CIS-EMPTY
  "CIS Empty"
  :RHS-Node-Types
  ((ZERO-FILL-COUNT? . COMMUTATIVE-BINARY-FUNCTION)
   (TEST-EQUALITY . NULL-TEST))
  :Edge-List
  (((ZERO-FILL-COUNT? 3) . (TEST-EQUALITY 1)))
  :Input-Embedding
  (((CIS-EMPTY 1) (ZERO-FILL-COUNT? 1)
    FILL-COUNT))
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether the Circular-Indexed-Sequence ~A is empty."
   (INPUT-PORT-NAME> (DOC-BP> (CIS-EMPTY 1)))))

(Defrule FIFO-EMPTY?
  "FIFO Empty"
  :RHS-Node-Types
  ((CIS-EMPTY? . CIS-EMPTY))
  :Input-Embedding
  (((FIFO-EMPTY? 1) (CIS-EMPTY? 1)
    CIRCULAR-INDEXED-SEQUENCE>FIFO))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("tests whether the FIFO queue ~A is empty.  The FIFO is implemented ~
    as a Circular Indexed Sequence."
   (INPUT-PORT-NAME> (DOC-BP> (FIFO-EMPTY? 1)))))

(Defrule CIS-FULL
  "CIS Full"
  :RHS-Node-Types
  ((ONE-LESS . DECREMENT)
   (MAX-FILL-COUNT? . LT)
   (TEST-COMPARISON . NULL-TEST))
  :Edge-List
  (((ONE-LESS 2) . (MAX-FILL-COUNT? 2))
   ((MAX-FILL-COUNT? 3) . (TEST-COMPARISON 1)))
  :Input-Embedding
  (((CIS-FULL 1) (ONE-LESS 1)
    SIZE)
   ((CIS-FULL 1) (MAX-FILL-COUNT? 1) FILL-COUNT))
```

```
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether the Circular-Indexed-Sequence ~A is full."
   (INPUT-PORT-NAME> (DOC-BP> (CIS-FULL 1)))))

(Defrule GROW-CIS
  "Grow Circular-Indexed-Sequence"
  :RHS-Node-Types
  ((THE-GROWER . INTERMEDIATE-GROW-CIS))
  :Input-Embedding
  (((GROW-CIS 1) (THE-GROWER 1)))
  :Output-Embedding
  (((GROW-CIS 2) (THE-GROWER 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("makes a new Circular Indexed Sequence that is double the ~
    size of the Circular Indexed Sequence ~A  and then ~
    transfers all of the elements of ~A to the new CIS.  The ~
    new CIS's First is at index 0 and its Last is at index = ~
    the number of elements in the sequence.~%~
    The new sequence grows ~A."
   (INPUT-PORT-NAME> (DOC-BP> (THE-GROWER 1)))
   (INPUT-PORT-NAME> (DOC-BP> (THE-GROWER 1)))
   (GROWTH-DIRECTION (N> THE-GROWER))))

(Defrule INTERMEDIATE-GROW-CIS
  "Grow Circular-Indexed-Sequence (Intermediate)"
  :RHS-Node-Types
  ((ENUMERATE-WHOLE-CIS . BOUNDED-CIS-ENUMERATION)
   (DOUBLE-SIZE . DOUBLE)
   (MAKE-NEW-BASE . NEW-SEQUENCE)
   (SUCCESSIVE-INDICES . COUNT)
   (ACCUMULATE-NEW-BASE . SEQUENCE-ACCUMULATE))
  :Edge-List
  (((ENUMERATE-WHOLE-CIS 5) . (ACCUMULATE-NEW-BASE 1))
   ((DOUBLE-SIZE 2) . (MAKE-NEW-BASE 1))
   ((MAKE-NEW-BASE 2) . (ACCUMULATE-NEW-BASE 3))
   ((SUCCESSIVE-INDICES 2) . (ACCUMULATE-NEW-BASE 2)))
  :Input-Embedding
  (((INTERMEDIATE-GROW-CIS 1) (ENUMERATE-WHOLE-CIS 1)
    BASE)
   ((INTERMEDIATE-GROW-CIS 1) (ENUMERATE-WHOLE-CIS 2)
    FIRST)
   ((INTERMEDIATE-GROW-CIS 1) (ENUMERATE-WHOLE-CIS 3)
    FILL-COUNT)
   ((INTERMEDIATE-GROW-CIS 1) (DOUBLE-SIZE 1) SIZE)
   ((INTERMEDIATE-GROW-CIS 1) (ENUMERATE-WHOLE-CIS 4)
    SIZE)
   ((INTERMEDIATE-GROW-CIS 2) (SUCCESSIVE-INDICES 1)))
  :Output-Embedding
  (((INTERMEDIATE-GROW-CIS 3) (ACCUMULATE-NEW-BASE 4)
    BASE)
   ((INTERMEDIATE-GROW-CIS 3) (DOUBLE-SIZE 2)
    SIZE))
  :St-Thrus
  (((INTERMEDIATE-GROW-CIS 2) (INTERMEDIATE-GROW-CIS 3))
   ((INTERMEDIATE-GROW-CIS 1) (INTERMEDIATE-GROW-CIS 3)
    FILL-COUNT)
   ((INTERMEDIATE-GROW-CIS 1) (INTERMEDIATE-GROW-CIS 3)
    FILL-COUNT))
  :L-R-Link COMPOSITION
  :Doc
  ("intermediate non-terminal: Grow-CIS."))

(Defrule COMBINATION-FUNCTION
  "Combination Function"
  :RHS-Node-Types
  ((SUBTRACT-THEM . MINUS))
  :Input-Embedding
  (((COMBINATION-FUNCTION 1) (SUBTRACT-THEM 1))
   ((COMBINATION-FUNCTION 2) (SUBTRACT-THEM 2)))
  :Output-Embedding
  (((COMBINATION-FUNCTION 3) (SUBTRACT-THEM 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("subtracts ~A from ~A."
   (INPUT-PORT-NAME> (DOC-BP> (COMBINATION-FUNCTION 2)))
   (INPUT-PORT-NAME> (DOC-BP> (COMBINATION-FUNCTION 1)))))

(Defrule COMBINATION-FUNCTION
  "Combination Function"
  :RHS-Node-Types
  ((SUM-THEM . COMMUTATIVE-BINARY-FUNCTION))
  :Input-Embedding
  (((COMBINATION-FUNCTION 1) (SUM-THEM 1))
   ((COMBINATION-FUNCTION 2) (SUM-THEM 2)))
  :Output-Embedding
  (((COMBINATION-FUNCTION 3) (SUM-THEM 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("combines ~A and ~A by adding them to each other."
   (INPUT-PORT-NAME> (DOC-BP> (COMBINATION-FUNCTION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (COMBINATION-FUNCTION 2)))))

(Defrule BOUNDED-CIS-ENUMERATION
  "Bounded Circular-Indexed-Sequence Enumeration"
  :RHS-Node-Types
  ((COUNT-N-TIMES . BOUNDED-COUNT)
   (COMBINE-COUNT-FIRST . COMBINATION-FUNCTION)
   (WRAP-INDEX . MOD)
   (MAP-ACCESS-CIS . SELECT-TERM))
  :Edge-List
  (((COUNT-N-TIMES 3) . (COMBINE-COUNT-FIRST 2))
   ((COMBINE-COUNT-FIRST 3) . (WRAP-INDEX 1))
   ((WRAP-INDEX 3) . (MAP-ACCESS-CIS 2)))
  :Input-Embedding
  (((BOUNDED-CIS-ENUMERATION 1) (MAP-ACCESS-CIS 1))
   ((BOUNDED-CIS-ENUMERATION 2) (COMBINE-COUNT-FIRST 1))
   ((BOUNDED-CIS-ENUMERATION 3) (COUNT-N-TIMES 2))
   ((BOUNDED-CIS-ENUMERATION 4) (WRAP-INDEX 2)))
  :Output-Embedding
  (((BOUNDED-CIS-ENUMERATION 5) (MAP-ACCESS-CIS 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates N elements of the Circular-Indexed-Sequence ~A starting ~
    from ~A, where N = ~A.  The sequence is filled in ~A."
   (INPUT-PORT-NAME> (DOC-BP> (BOUNDED-CIS-ENUMERATION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (BOUNDED-CIS-ENUMERATION 2)))
   (INPUT-PORT-NAME> (DOC-BP> (BOUNDED-CIS-ENUMERATION 3)))
   (GROWTH-DIRECTION (N> BOUNDED-CIS-ENUMERATION))))

(Defrule CIRCULAR-INDEXED-SEQUENCE-ENUMERATION
  "Circular-Indexed-Sequence Enumeration"
  :RHS-Node-Types
  ((ENUMERATE-ENTIRE-CIS . BOUNDED-CIS-ENUMERATION))
  :Input-Embedding
  (((CIRCULAR-INDEXED-SEQUENCE-ENUMERATION 1) (ENUMERATE-ENTIRE-CIS 1)
    BASE)
   ((CIRCULAR-INDEXED-SEQUENCE-ENUMERATION 1) (ENUMERATE-ENTIRE-CIS 2)
    FIRST)
   ((CIRCULAR-INDEXED-SEQUENCE-ENUMERATION 1) (ENUMERATE-ENTIRE-CIS 3)
    FILL-COUNT)
   ((CIRCULAR-INDEXED-SEQUENCE-ENUMERATION 1) (ENUMERATE-ENTIRE-CIS 4)
    SIZE))
  :Output-Embedding
  (((CIRCULAR-INDEXED-SEQUENCE-ENUMERATION 2) (ENUMERATE-ENTIRE-CIS 5)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("enumerates all of the elements in the Circular-Indexed-Sequence ~A. ~
    The sequence is filled in ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CIRCULAR-INDEXED-SEQUENCE-ENUMERATION 1)))
   (GROWTH-DIRECTION (N> CIRCULAR-INDEXED-SEQUENCE-ENUMERATION))))

(Defrule FIFO-ENUMERATION
  "FIFO Enumeration"
  :RHS-Node-Types
  ((ENUMERATE-CIS . CIRCULAR-INDEXED-SEQUENCE-ENUMERATION))
  :Input-Embedding
  (((FIFO-ENUMERATION 1) (ENUMERATE-CIS 1)
    CIRCULAR-INDEXED-SEQUENCE>FIFO))
  :Output-Embedding
  (((FIFO-ENUMERATION 2) (ENUMERATE-CIS 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("enumerates the FIFO queue ~A, which is implemented as a Circular ~
    Indexed Sequence.  The queue is not changed.  The queue grows ~A."
   (INPUT-PORT-NAME> (DOC-BP> (FIFO-ENUMERATION 1)))
   (GROWTH-DIRECTION (N> FIFO-ENUMERATION))))

(Defrule CIS-ADD
  "Circular-Indexed-Sequence Add"
  :RHS-Node-Types
  ((FULL? . CIS-FULL)
   (ROOMY-ADD . ROOMY-CIS-ADD)
   (MAKE-ROOM . GROW-CIS))
  :Edge-List
  (((MAKE-ROOM 2) . (ROOMY-ADD 2)))
  :Input-Embedding
  (((CIS-ADD 1) (ROOMY-ADD 1))
   ((CIS-ADD 2) (MAKE-ROOM 1))
   ((CIS-ADD 2) (ROOMY-ADD 2))
   ((CIS-ADD 2) (FULL? 1)))
  :Output-Embedding
  (((CIS-ADD 3) (ROOMY-ADD 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("adds the element ~A to the Circular-Indexed-Sequence ~A,~%~
    making room for it if the Circular-Indexed-Sequence is full.~%~
      The sequence is filled in ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CIS-ADD 1)))
   (INPUT-PORT-NAME> (DOC-BP> (CIS-ADD 2)))
   (GROWTH-DIRECTION (N> CIS-ADD))))

(Defrule ROOMY-CIS-ADD
  "Roomy Circular-Indexed-Sequence Add"
  :RHS-Node-Types
  ((ADD-TO-DATA . NEW-TERM)
```

```
    (BUMP-LAST . INCREMENT-OR-DECREMENT)
    (WRAP-INDEX-AROUND . MOD)
    (INCREMENT-FILL-COUNT . INCREMENT))
   :Edge-List
   (((BUMP-LAST 2) . (WRAP-INDEX-AROUND 1)))
   :Input-Embedding
   (((ROOMY-CIS-ADD 1) (ADD-TO-DATA 1))
    ((ROOMY-CIS-ADD 2) (ADD-TO-DATA 3)
     BASE)
    ((ROOMY-CIS-ADD 2) (WRAP-INDEX-AROUND 2)
     SIZE)
    ((ROOMY-CIS-ADD 2) (INCREMENT-FILL-COUNT 1)
     FILL-COUNT)
    ((ROOMY-CIS-ADD 2) (BUMP-LAST 1)
     LAST)
    ((ROOMY-CIS-ADD 2) (ADD-TO-DATA 2)
     LAST))
   :Output-Embedding
   (((ROOMY-CIS-ADD 3) (WRAP-INDEX-AROUND 3)
     LAST)
    ((ROOMY-CIS-ADD 3) (INCREMENT-FILL-COUNT 2)
     FILL-COUNT)
    ((ROOMY-CIS-ADD 3) (ADD-TO-DATA 4)
     BASE))
   :St-Thrus
   (((ROOMY-CIS-ADD 2) (ROOMY-CIS-ADD 3)
     SIZE)
    ((ROOMY-CIS-ADD 2) (ROOMY-CIS-ADD 3)
     FIRST))
   :L-R-Link COMPOSITION
   :Doc
   ("adds the element ~A to the Circular-Indexed-Sequence ~A, ~
     (which has room for it).~%~
       The sequence is filled in ~A."
    (INPUT-PORT-NAME> (DOC-BP> (ROOMY-CIS-ADD 1)))
    (INPUT-PORT-NAME> (DOC-BP> (ROOMY-CIS-ADD 2)))
    (GROWTH-DIRECTION (N> ROOMY-CIS-ADD))))

(Defrule FIFO-ENQUEUE
  "FIFO Enqueue"
  :RHS-Node-Types
  ((ADD-TO-CIS-LAST . CIS-ADD))
  :Input-Embedding
  (((FIFO-ENQUEUE 1) (ADD-TO-CIS-LAST 1))
   ((FIFO-ENQUEUE 2) (ADD-TO-CIS-LAST 2)
    CIRCULAR-INDEXED-SEQUENCE>FIFO))
  :Output-Embedding
  (((FIFO-ENQUEUE 3) (ADD-TO-CIS-LAST 3)
    CIRCULAR-INDEXED-SEQUENCE>FIFO))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("enqueues ~A on the FIFO queue ~A, which is implemented as ~
    a Circular Indexed Sequence.~%~
      The queue grows ~A."
   (INPUT-PORT-NAME> (DOC-BP> (FIFO-ENQUEUE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (FIFO-ENQUEUE 2)))
   (GROWTH-DIRECTION (N> FIFO-ENQUEUE))))

;;; Figures 3-24, 4-11.

(Defrule CIS-EXTRACT
  "Circular-Indexed-Sequence Extract"
  :RHS-Node-Types
  ((ACCESS-BASE . SELECT-TERM)
   (BUMP-FIRST . INCREMENT-OR-DECREMENT)
   (WRAP-AROUND-INDEX . MOD)
   (DECREMENT-FILL-COUNT . DECREMENT))
  :Edge-List
  (((BUMP-FIRST 2) . (WRAP-AROUND-INDEX 1)))
  :Input-Embedding
  (((CIS-EXTRACT 1) (BUMP-FIRST 1)
    FIRST)
   ((CIS-EXTRACT 1) (ACCESS-BASE 2)
    FIRST)
   ((CIS-EXTRACT 1) (ACCESS-BASE 1)
    BASE)
   ((CIS-EXTRACT 1) (WRAP-AROUND-INDEX 2)
    SIZE)
   ((CIS-EXTRACT 1) (DECREMENT-FILL-COUNT 1)
    FILL-COUNT))
  :Output-Embedding
  (((CIS-EXTRACT 2) (ACCESS-BASE 3))
   ((CIS-EXTRACT 3) (WRAP-AROUND-INDEX 3)
    FIRST)
   ((CIS-EXTRACT 3) (DECREMENT-FILL-COUNT 2)
    FILL-COUNT))
  :St-Thrus
  (((CIS-EXTRACT 1) (CIS-EXTRACT 3)
    LAST)
   ((CIS-EXTRACT 1) (CIS-EXTRACT 3)
    SIZE)
   ((CIS-EXTRACT 1) (CIS-EXTRACT 3)
    BASE))
  :L-R-Link COMPOSITION
  :Doc
  ("extracts the First element from the Circular Indexed-Sequence ~A.~%~
      The sequence is filled in ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CIS-EXTRACT 1)))
   (GROWTH-DIRECTION (N> CIS-EXTRACT))))

;;; Figure 4-12.

(Defrule FIFO-DEQUEUE
  "FIFO Dequeue"
  :RHS-Node-Types
  ((EXTRACT-CIS-FIRST . CIS-EXTRACT))
  :Input-Embedding
  (((FIFO-DEQUEUE 1) (EXTRACT-CIS-FIRST 1)
    CIRCULAR-INDEXED-SEQUENCE>FIFO))
  :Output-Embedding
  (((FIFO-DEQUEUE 2) (EXTRACT-CIS-FIRST 2))
   ((FIFO-DEQUEUE 3) (EXTRACT-CIS-FIRST 3)
    CIRCULAR-INDEXED-SEQUENCE>FIFO))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("dequeues the FIFO queue ~A, which is implemented as a Circular ~
    Indexed-Sequence.~%~
      The queue grows ~A."
   (INPUT-PORT-NAME> (DOC-BP> (FIFO-DEQUEUE 1)))
   (GROWTH-DIRECTION (N> FIFO-DEQUEUE))))

(Defrule EVALUATE-ARGUMENTS
  "Evaluate-Arguments"
  :RHS-Node-Types
  ((EVAL-EXPS . ENUM-EVAL-COLLECT))
  :Input-Embedding
  (((EVALUATE-ARGUMENTS 1) (EVAL-EXPS 1))
   ((EVALUATE-ARGUMENTS 2) (EVAL-EXPS 2))
   ((EVALUATE-ARGUMENTS 3) (EVAL-EXPS 3))
   ((EVALUATE-ARGUMENTS 4) (EVAL-EXPS 4)))
  :Output-Embedding
  (((EVALUATE-ARGUMENTS 5) (EVAL-EXPS 5))
   ((EVALUATE-ARGUMENTS 6) (EVAL-EXPS 6))
   ((EVALUATE-ARGUMENTS 7) (EVAL-EXPS 7))
   ((EVALUATE-ARGUMENTS 8) (EVAL-EXPS 8)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("evaluates the arguments ~A."
   (INPUT-PORT-NAME> (DOC-BP> (EVAL-EXPS 1)))))

(Defrule ENUM-EVAL-COLLECT
  "Enumerate, Evaluate, and Collect"
  :RHS-Node-Types
  ((ENUMERATE-ARGS . LE)
   (EVALUATE-THEM . EVALUATE-MAP)
   (COLLECT-RESULTS . CONS-ACCUMULATE-UP))
  :Edge-List
  (((ENUMERATE-ARGS 2) . (EVALUATE-MAP 1)))
  :Input-Embedding
  (((ENUM-EVAL-COLLECT 1) (ENUMERATE-ARGS 1))
   ((ENUM-EVAL-COLLECT 2) (EVALUATE-MAP 2))
   ((ENUM-EVAL-COLLECT 3) (EVALUATE-MAP 3))
   ((ENUM-EVAL-COLLECT 4) (EVALUATE-MAP 4)))
  :Output-Embedding
  (((ENUM-EVAL-COLLECT 5) (COLLECT-RESULTS 2))
   ((ENUM-EVAL-COLLECT 6) (EVALUATE-MAP 6))
   ((ENUM-EVAL-COLLECT 7) (EVALUATE-MAP 7))
   ((ENUM-EVAL-COLLECT 8) (EVALUATE-MAP 8)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the arguments ~A, evaluates each one, and collects~%~
    the evaluated arguments in a list, which it returns."
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-ARGS 1)))))

(Defrule EVALUATE-MAP
  "Evaluate Map"
  :RHS-Node-Types
  ((ITER-EVAL . ITERATIVE-EVALUATION))
  :Input-Embedding
  (((EVALUATE-MAP 1) (ITER-EVAL 1))
   ((EVALUATE-MAP 2) (ITER-EVAL 2))
   ((EVALUATE-MAP 3) (ITER-EVAL 3))
   ((EVALUATE-MAP 4) (ITER-EVAL 4)))
  :Output-Embedding
  (((EVALUATE-MAP 5) (ITER-EVAL 5))
   ((EVALUATE-MAP 6) (ITER-EVAL 6))
   ((EVALUATE-MAP 7) (ITER-EVAL 7))
   ((EVALUATE-MAP 8) (ITER-EVAL 8)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("applies the function EVALUATE to each expression in the input ~
    series of expressions."))

(Defrule ITERATIVE-EVALUATION
  "Iterative Evaluation"
  :RHS-Node-Types
```

```
   ((MAP-EVAL . EVALUATE))
   :Input-Embedding
   (((ITERATIVE-EVALUATION 1) (MAP-EVAL 1))
    ((ITERATIVE-EVALUATION 2) (MAP-EVAL 2))
    ((ITERATIVE-EVALUATION 3) (MAP-EVAL 3))
    ((ITERATIVE-EVALUATION 4) (MAP-EVAL 4)))
   :Output-Embedding
   (((ITERATIVE-EVALUATION 5) (MAP-EVAL 5)))
   :St-Thrus
   (((ITERATIVE-EVALUATION 4) (ITERATIVE-EVALUATION 8))
    ((ITERATIVE-EVALUATION 3) (ITERATIVE-EVALUATION 7))
    ((ITERATIVE-EVALUATION 2) (ITERATIVE-EVALUATION 6)))
   :L-R-Link COMPOSITION
   :Doc
   ("iteratively applies the function Evaluate."))

(Defrule RUNNING-STATUS?
  "Execution Still Running Predicate"
  :RHS-Node-Types
  ((STATUS-RUNNING? . RUNNING-TEST))
  :Input-Embedding
  (((RUNNING-STATUS? 1) (STATUS-RUNNING? 1)
    STATUS))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("checks whether the execution context ~A is still running ~
    by looking at its STATUS part."
   (INPUT-PORT-NAME> (DOC-BP> (STATUS-RUNNING? 1)))))

(Defrule RUNNING-TEST
  "Running Test"
  :RHS-Node-Types
  ((RUNNING? . COMMUTATIVE-BINARY-FUNCTION)
   (RUN-SPLIT . NULL-TEST))
  :Edge-List
  (((RUNNING? 3) . (RUN-SPLIT 1)))
  :Input-Embedding
  (((RUNNING-TEST 1) (RUNNING? 1)))
  :L-R-Link COMPOSITION
  :Doc
  ("checks whether ~A ~A ~A."
   (INPUT-PORT-NAME> (DOC-BP> (RUNNING? 1)))
   (FUNCTION-TYPE (FUNCTION-INFO (N> RUNNING?)))
   (SOURCE-TYPE (DOC-BP> (RUNNING? 2)))))

(Defrule HANDLE-MESSAGE
  "Handle Message"
  :RHS-Node-Types
  ((PROCESS . LOOKUP-AND-EXECUTE-HANDLER))
  :Input-Embedding
  (((HANDLE-MESSAGE 1) (PROCESS 1))
   ((HANDLE-MESSAGE 2) (PROCESS 2))
   ((HANDLE-MESSAGE 3) (PROCESS 3)))
  :Output-Embedding
  (((HANDLE-MESSAGE 4) (PROCESS 6))
   ((HANDLE-MESSAGE 5) (PROCESS 7)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("handles the message ~A by looking up its handler code and ~
    executing it."
   (INPUT-PORT-NAME> (DOC-BP> (HANDLE-MESSAGE 1)))))

(Defrule LOOKUP-HANDLER-FOR-MESSAGE
  "Lookup Message Handler"
  :RHS-Node-Types
  ((LOOKUP-HANDLER-OF-TYPE . LOOKUP-HANDLER))
  :Input-Embedding
  (((LOOKUP-HANDLER-FOR-MESSAGE 1) (LOOKUP-HANDLER-OF-TYPE 1)
    TYPE))
  :Output-Embedding
  (((LOOKUP-HANDLER-FOR-MESSAGE 2) (LOOKUP-HANDLER-OF-TYPE 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up the handler for message ~A's type ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-HANDLER-FOR-MESSAGE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-HANDLER-FOR-MESSAGE 1)
                              TYPE))))

(Defrule LOOKUP-HANDLER
  "Lookup Handler"
  :RHS-Node-Types
  ((ASSOCIATE-HANDLER-NAME . ASSOCIATIVE-SET-LOOKUP))
  :Input-Embedding
  (((LOOKUP-HANDLER 1) (ASSOCIATE-HANDLER-NAME 1)))
  :Output-Embedding
  (((LOOKUP-HANDLER 2) (ASSOCIATE-HANDLER-NAME 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up the handler named ~A.~%~
    The global associative set of operators is ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-HANDLER 1)))
   (SOURCE-TYPE (P> (ASSOCIATE-HANDLER-NAME 2)))))

(Defrule LOOKUP-HANDLER
  "Lookup Handler"
  :RHS-Node-Types
  ((LOOKUP-HANDLER-PROPERTY . PROPERTY-LIST-LOOKUP))
  :Input-Embedding
  (((LOOKUP-HANDLER 1) (LOOKUP-HANDLER-PROPERTY 1)))
  :Output-Embedding
  (((LOOKUP-HANDLER 2) (LOOKUP-HANDLER-PROPERTY 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up the handler named ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-HANDLER 1)))))

(Defrule FETCH-OP
  "Fetch Operator"
  :RHS-Node-Types
  ((LOOKUP-OP . ASSOCIATIVE-SET-LOOKUP))
  :Input-Embedding
  (((FETCH-OP 1) (LOOKUP-OP 1)))
  :Output-Embedding
  (((FETCH-OP 2) (LOOKUP-OP 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up the operator named ~A.~%~
    The global associative set of operators is ~A."
   (INPUT-PORT-NAME> (DOC-BP> (FETCH-OP 1)))
   (SOURCE-TYPE (P> (LOOKUP-OP 2)))))

(Defrule FETCH-OP
  "Fetch Operator"
  :RHS-Node-Types
  ((THE-PLIST-LOOKUP . PROPERTY-LIST-LOOKUP))
  :Input-Embedding
  (((FETCH-OP 1) (THE-PLIST-LOOKUP 1)))
  :Output-Embedding
  (((FETCH-OP 2) (THE-PLIST-LOOKUP 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up the operator named ~A."
   (INPUT-PORT-NAME> (DOC-BP> (FETCH-OP 1)))))

(Defrule FETCH-AND-APPLY-OPERATOR
  "Fetch and Apply Operator"
  :RHS-Node-Types
  ((GET-OPERATOR . FETCH-OP)
   (APPLY-OPERATOR . APPLY))
  :Edge-List
  (((GET-OPERATOR 2) . (APPLY-OPERATOR 1)))
  :Input-Embedding
  (((FETCH-AND-APPLY-OPERATOR 1) (GET-OPERATOR 1))
   ((FETCH-AND-APPLY-OPERATOR 2) (APPLY-OPERATOR 2))
   ((FETCH-AND-APPLY-OPERATOR 3) (APPLY-OPERATOR 3))
   ((FETCH-AND-APPLY-OPERATOR 4) (APPLY-OPERATOR 4))
   ((FETCH-AND-APPLY-OPERATOR 5) (APPLY-OPERATOR 5)))
  :Output-Embedding
  (((FETCH-AND-APPLY-OPERATOR 6) (APPLY-OPERATOR 6))
   ((FETCH-AND-APPLY-OPERATOR 7) (APPLY-OPERATOR 7))
   ((FETCH-AND-APPLY-OPERATOR 8) (APPLY-OPERATOR 8))
   ((FETCH-AND-APPLY-OPERATOR 9) (APPLY-OPERATOR 9)))
  :L-R-Link COMPOSITION
  :Doc
  ("fetches the operator associated w/ ~A and applies it to the~%~
    evaluated arguments ~A."
   (INPUT-PORT-NAME> (DOC-BP> (FETCH-AND-APPLY-OPERATOR 1)))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH-AND-APPLY-OPERATOR 2)))))

(Defrule EVALUATE-AND-APPLY
  "Evaluate Arguments and Apply Operator"
  :RHS-Node-Types
  ((EVAL-ARGS . EVALUATE-ARGUMENTS)
   (APPLY-OP . FETCH-AND-APPLY-OPERATOR))
  :Edge-List
  (((EVAL-ARGS 8) . (APPLY-OP 5))
   ((EVAL-ARGS 7) . (APPLY-OP 4))
   ((EVAL-ARGS 6) . (APPLY-OP 3))
   ((EVAL-ARGS 5) . (APPLY-OP 2)))
  :Input-Embedding
  (((EVALUATE-AND-APPLY 1) (APPLY-OP 1))
   ((EVALUATE-AND-APPLY 2) (EVAL-ARGS 1))
   ((EVALUATE-AND-APPLY 3) (EVAL-ARGS 2))
   ((EVALUATE-AND-APPLY 4) (EVAL-ARGS 3))
   ((EVALUATE-AND-APPLY 5) (EVAL-ARGS 4)))
  :Output-Embedding
  (((EVALUATE-AND-APPLY 6) (APPLY-OP 6))
   ((EVALUATE-AND-APPLY 7) (APPLY-OP 7))
   ((EVALUATE-AND-APPLY 8) (APPLY-OP 8))
   ((EVALUATE-AND-APPLY 9) (APPLY-OP 9)))
  :L-R-Link COMPOSITION
  :Doc
  ("evaluates the arguments ~A, fetches the operation ~A and applies~%~
    it to the evaluated arguments."
   (INPUT-PORT-NAME> (DOC-BP> (EVALUATE-AND-APPLY 2)))
   (INPUT-PORT-NAME> (DOC-BP> (EVALUATE-AND-APPLY 1)))))
```

```
(Defrule INTERPRET-INSTRUCTION
  "Interpret Instruction"
  :RHS-Node-Types
  ((EVAL-APPLY . EVALUATE-AND-APPLY))
  :Input-Embedding
  (((INTERPRET-INSTRUCTION 1) (EVAL-APPLY 1)
    OP)
   ((INTERPRET-INSTRUCTION 1) (EVAL-APPLY 2)
    ARGS)
   ((INTERPRET-INSTRUCTION 2) (EVAL-APPLY 3))
   ((INTERPRET-INSTRUCTION 3) (EVAL-APPLY 4))
   ((INTERPRET-INSTRUCTION 4) (EVAL-APPLY 5)))
  :Output-Embedding
  (((INTERPRET-INSTRUCTION 5) (EVAL-APPLY 7))
   ((INTERPRET-INSTRUCTION 6) (EVAL-APPLY 8))
   ((INTERPRET-INSTRUCTION 7) (EVAL-APPLY 9)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("interprets the instruction ~A by evaluating its arguments ~
    ~A and applying its operator ~A to them."
   (INPUT-PORT-NAME> (DOC-BP> (INTERPRET-INSTRUCTION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (INTERPRET-INSTRUCTION 1)
                              INST-ARGS))
   (INPUT-PORT-NAME> (DOC-BP> (INTERPRET-INSTRUCTION 1)
                              INST-OP))))

(Defrule LOOKUP-AND-EXECUTE-HANDLER
  "Lookup and Execute Message Handler"
  :RHS-Node-Types
  ((GET-DESTINATION-NODE . LOOKUP-DESTINATION)
   (LOAD-ARGS . LOAD-ARGUMENTS)
   (RECORD-NEW-NODE . RECORD-AT-DESTINATION)
   (GET-HANDLER-CODE . LOOKUP-HANDLER-FOR-MESSAGE)
   (GET-NEXT-INSTRUCTION . FETCH-INSTRUCTION)
   (INTERPRET . INTERPRET-INSTRUCTION)
   (STILL-RUNNING? . RUNNING-STATUS?))
  :Edge-List
  (((GET-DESTINATION-NODE 3) . (LOAD-ARGS 2))
   ((LOAD-ARGS 3) . (INTERPRET 3))
   ((LOAD-ARGS 3) . (RECORD-NEW-NODE 1))
   ((RECORD-NEW-NODE 4) . (INTERPRET 2))
   ((GET-HANDLER-CODE 2) . (INTERPRET 3))
   ((GET-HANDLER-CODE 2) . (GET-NEXT-INSTRUCTION 2))
   ((GET-NEXT-INSTRUCTION 4) . (INTERPRET 3))
   ((GET-NEXT-INSTRUCTION 3) . (INTERPRET 1))
   ((INTERPRET 6) . (STILL-RUNNING? 1)))
  :Input-Embedding
  (((LOOKUP-AND-EXECUTE-HANDLER 1) (RECORD-NEW-NODE 2))
   ((LOOKUP-AND-EXECUTE-HANDLER 1) (LOAD-ARGS 1))
   ((LOOKUP-AND-EXECUTE-HANDLER 1) (GET-DESTINATION-NODE 2))
   ((LOOKUP-AND-EXECUTE-HANDLER 1) (GET-HANDLER-CODE 1))
   ((LOOKUP-AND-EXECUTE-HANDLER 2) (RECORD-NEW-NODE 3))
   ((LOOKUP-AND-EXECUTE-HANDLER 2) (GET-DESTINATION-NODE 1))
   ((LOOKUP-AND-EXECUTE-HANDLER 3) (INTERPRET 4))
   ((LOOKUP-AND-EXECUTE-HANDLER 4) (GET-NEXT-INSTRUCTION 1))
   ((LOOKUP-AND-EXECUTE-HANDLER 5) (INTERPRET 3)))
  :Output-Embedding
  (((LOOKUP-AND-EXECUTE-HANDLER 6) (INTERPRET 5))
   ((LOOKUP-AND-EXECUTE-HANDLER 7) (INTERPRET 7)))
  :L-R-Link COMPOSITION
  :Doc
  ("looks up the handler for the message ~A, loads the ~
    arguments of the message into the message's destination ~
    node, and then executes the handler instructions, starting ~
    with the one pointed to by ~A.  As long as the execution ~
    context's status is ~A, the next instruction (pointed to ~
     by ~A) is executed."
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-AND-EXECUTE-HANDLER 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-AND-EXECUTE-HANDLER 4)))
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-AND-EXECUTE-HANDLER 5)))
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-AND-EXECUTE-HANDLER 4)))))

(Defrule FETCH-INSTRUCTION
  "Fetch Next Instruction"
  :RHS-Node-Types
  ((FETCH-I1 . INDEXED-SEQUENCE-EXTRACT))
  :Output-Embedding
  (((FETCH-INSTRUCTION 3) (FETCH-I1 2))
   ((FETCH-INSTRUCTION 4) (FETCH-I1 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("fetches the next instruction (pointed to by ~A) in the ~
    sequence ~A"
   (INPUT-PORT-NAME> (DOC-BP> (FETCH-INSTRUCTION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH-INSTRUCTION 2)))))

(Defrule LOAD-ARGUMENTS-INTO-MEMORY
  "Load Arguments into Memory"
  :RHS-Node-Types
  ((TRANSFER-ARG-LIST . LIST-TO-SEQUENCE)
   (ADD-TO-MEMORY . ASSOCIATIVE-SET-ADD))
  :Edge-List
  (((TRANSFER-ARG-LIST 3) . (ADD-TO-MEMORY 1)))
  :Input-Embedding
  (((LOAD-ARGUMENTS-INTO-MEMORY 1) (TRANSFER-ARG-LIST 1)
    ARGUMENTS)
   ((LOAD-ARGUMENTS-INTO-MEMORY 1) (TRANSFER-ARG-LIST 2)
    STORAGE-REQUIREMENTS)
   ((LOAD-ARGUMENTS-INTO-MEMORY 2) (ADD-TO-MEMORY 3)))
  :Output-Embedding
  (((LOAD-ARGUMENTS-INTO-MEMORY 3) (ADD-TO-MEMORY 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("takes the list of arguments in the message ~A and converts it to ~
    an indexed-sequence of size ~A, which it then stores in the memory ~
    ~A, at key ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS-INTO-MEMORY 1)
                              ARGUMENTS))
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS-INTO-MEMORY 1)
                              STORAGE-REQUIREMENTS))
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS-INTO-MEMORY 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ADD-TO-MEMORY 2)))))

(Defrule LOAD-ARGUMENTS-INTO-SN
  "Load Arguments into Synch-Node"
  :RHS-Node-Types
  ((BASE-LOAD-ARGUMENTS . LOAD-ARGUMENTS-INTO-MEMORY))
  :Input-Embedding
  (((LOAD-ARGUMENTS-INTO-SN 1) (BASE-LOAD-ARGUMENTS 1))
   ((LOAD-ARGUMENTS-INTO-SN 2) (BASE-LOAD-ARGUMENTS 2)
    MEMORY))
  :Output-Embedding
  (((LOAD-ARGUMENTS-INTO-SN 3) (BASE-LOAD-ARGUMENTS 3)
    MEMORY))
  :St-Thrus
  (((LOAD-ARGUMENTS-INTO-SN 2) (LOAD-ARGUMENTS-INTO-SN 3)
    LOCAL-BUFFER))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("loads the arguments of the Message ~A into the Memory part of the ~
    Node ~A~ which is implemented as a Synch-Node."
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS-INTO-SN 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS-INTO-SN 2)))))

(Defrule LOAD-ARGUMENTS-INTO-AN
  "Load Arguments into Asynch-Node"
  :RHS-Node-Types
  ((BASE-LOAD-ARGUMENTS . LOAD-ARGUMENTS-INTO-MEMORY))
  :Input-Embedding
  (((LOAD-ARGUMENTS-INTO-AN 1) (BASE-LOAD-ARGUMENTS 1))
   ((LOAD-ARGUMENTS-INTO-AN 2) (BASE-LOAD-ARGUMENTS 2) MEMORY))
  :Output-Embedding
  (((LOAD-ARGUMENTS-INTO-AN 3) (BASE-LOAD-ARGUMENTS 3) MEMORY))
  :St-Thrus
  (((LOAD-ARGUMENTS-INTO-AN 2) (LOAD-ARGUMENTS-INTO-AN 3)
    TIME))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("loads the arguments of the Message ~A into the Memory part of the ~
    Node ~A which is implemented as an Asynch-Node."
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS-INTO-AN 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS-INTO-AN 2)))))

(Defrule LOAD-ARGUMENTS
  "Load Arguments"
  :RHS-Node-Types
  ((LOAD-AN . LOAD-ARGUMENTS-INTO-AN))
  :Input-Embedding
  (((LOAD-ARGUMENTS 1) (LOAD-AN 1))
   ((LOAD-ARGUMENTS 2) (LOAD-AN 2)
    ASYNCH-NODE>NODE))
  :Output-Embedding
  (((LOAD-ARGUMENTS 3) (LOAD-AN 3)
    ASYNCH-NODE>NODE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("loads the arguments of Message ~A into the memory of node ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS 2)))))

(Defrule LOAD-ARGUMENTS
  "Load Arguments"
  :RHS-Node-Types
  ((LOAD-SN . LOAD-ARGUMENTS-INTO-SN))
  :Input-Embedding
  (((LOAD-ARGUMENTS 1) (LOAD-SN 1))
   ((LOAD-ARGUMENTS 2) (LOAD-SN 2)
    SYNCH-NODE>NODE))
  :Output-Embedding
  (((LOAD-ARGUMENTS 3) (LOAD-SN 3)
    SYNCH-NODE>NODE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("loads the arguments of Message ~A into the memory of node ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LOAD-ARGUMENTS 2)))))
```

```
(Defrule FETCH+UPDATE
  "Fetch and Update"
  :RHS-Node-Types
  ((FETCH-FROM-BASE . SELECT-TERM)
   (BACKUP-INDEX . INCREMENT-OR-DECREMENT))
  :Input-Embedding
  (((FETCH+UPDATE 1) (FETCH-FROM-BASE 2) INDEX)
   ((FETCH+UPDATE 1) (BACKUP-INDEX 1) INDEX)
   ((FETCH+UPDATE 1) (FETCH-FROM-BASE 1) BASE))
  :Output-Embedding
  (((FETCH+UPDATE 2) (FETCH-FROM-BASE 3))
   ((FETCH+UPDATE 3) (BACKUP-INDEX 2) INDEX))
  :St-Thrus
  (((FETCH+UPDATE 1) (FETCH+UPDATE 3) BASE))
  :L-R-Link COMPOSITION
  :Doc
  ("extracts an element from an Indexed-Sequence, which has
    parts:~%~
      Base (an sequence) ~A,~%~
      and an Index ~A into the sequence.~%~
    The sequence is filled in ~A.  The Index is updated after ~
    the output is fetched from the Base."
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+UPDATE 1) BASE))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+UPDATE 1) INDEX))
   (GROWTH-DIRECTION (N> FETCH+UPDATE))))

(Defrule UPDATE+FETCH
  "Update and Fetch"
  :RHS-Node-Types
  ((FETCH-FROM-BASE2 . SELECT-TERM)
   (BACKUP-INDEX2 . INCREMENT-OR-DECREMENT))
  :Edge-List
  (((BACKUP-INDEX2 2) . (FETCH-FROM-BASE2 2)))
  :Input-Embedding
  (((UPDATE+FETCH 1) (BACKUP-INDEX2 1) INDEX)
   ((UPDATE+FETCH 1) (FETCH-FROM-BASE2 1) BASE))
  :Output-Embedding
  (((UPDATE+FETCH 2) (FETCH-FROM-BASE2 3))
   ((UPDATE+FETCH 3) (BACKUP-INDEX2 2) INDEX))
  :St-Thrus
  (((UPDATE+FETCH 1) (UPDATE+FETCH 3) BASE))
  :L-R-Link COMPOSITION
  :Doc
  ("extracts an element from an Indexed-Sequence, which has ~
    parts:~%~
      Base (an sequence) ~A,~%~
      and an Index ~A into the sequence.~%~
    The sequence is filled in ~A.  The Index is updated before ~
    the output is fetched from the Base."
   (INPUT-PORT-NAME> (DOC-BP> (UPDATE+FETCH 1) BASE))
   (INPUT-PORT-NAME> (DOC-BP> (UPDATE+FETCH 1) INDEX))
   (GROWTH-DIRECTION (N> UPDATE+FETCH))))

(Defrule UPDATE+BUMP
  "Update and Bump"
  :RHS-Node-Types
  ((BUMP-INDEX . INCREMENT-OR-DECREMENT)
   (ADD-TO-BASE . NEW-TERM))
  :Edge-List
  (((BUMP-INDEX 2) . (ADD-TO-BASE 2)))
  :Input-Embedding
  (((UPDATE+BUMP 2) (BUMP-INDEX 1) INDEX)
   ((UPDATE+BUMP 2) (ADD-TO-BASE 3) BASE)
   ((UPDATE+BUMP 1) (ADD-TO-BASE 1)))
  :Output-Embedding
  (((UPDATE+BUMP 3) (BUMP-INDEX 2) INDEX)
   ((UPDATE+BUMP 3) (ADD-TO-BASE 4) BASE))
  :L-R-Link COMPOSITION
  :Doc
  ("adds ~A to an Indexed-Sequence, which has parts:~%~
      Base (an sequence) ~A,~%~
      and an Index ~A into the sequence.~%~
    The sequence is filled in ~A.~%~
    The Index is updated before the input is added to the Base."
   (INPUT-PORT-NAME> (DOC-BP> (UPDATE+BUMP 1)))
   (INPUT-PORT-NAME> (DOC-BP> (UPDATE+BUMP 2) BASE))
   (INPUT-PORT-NAME> (DOC-BP> (UPDATE+BUMP 2) INDEX))
   (GROWTH-DIRECTION (N> UPDATE+BUMP))))

(Defrule BUMP+UPDATE
  "Bump and Update"
  :RHS-Node-Types
  ((BUMP-INDEX2 . INCREMENT-OR-DECREMENT)
   (ADD-TO-BASE2 . NEW-TERM))
  :Input-Embedding
  (((BUMP+UPDATE 2) (ADD-TO-BASE2 2) INDEX)
   ((BUMP+UPDATE 2) (BUMP-INDEX2 1) INDEX)
   ((BUMP+UPDATE 2) (ADD-TO-BASE2 3) BASE)
   ((BUMP+UPDATE 1) (ADD-TO-BASE2 1)))
  :Output-Embedding
  (((BUMP+UPDATE 3) (BUMP-INDEX2 2) INDEX)
   ((BUMP+UPDATE 3) (ADD-TO-BASE2 4) BASE))
  :L-R-Link COMPOSITION
  :Doc
  ("adds ~A to an Indexed-Sequence, which has parts:~%~
      Base (an sequence) ~A,~%~
      and an Index ~A into the sequence.~%~
    The sequence is filled in ~A.~%~
    The Index is updated after the input is added to the Base."
   (INPUT-PORT-NAME> (DOC-BP> (BUMP+UPDATE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (BUMP+UPDATE 2) BASE))
   (INPUT-PORT-NAME> (DOC-BP> (BUMP+UPDATE 2) INDEX))
   (GROWTH-DIRECTION (N> BUMP+UPDATE))))

(Defrule INDEXED-SEQUENCE-INSERT
  "Indexed-Sequence Insert"
  :RHS-Node-Types
  ((I-S-INSERT2 . UPDATE+BUMP))
  :Input-Embedding
  (((INDEXED-SEQUENCE-INSERT 1) (I-S-INSERT2 1))
   ((INDEXED-SEQUENCE-INSERT 2) (I-S-INSERT2 2)))
  :Output-Embedding
  (((INDEXED-SEQUENCE-INSERT 3) (I-S-INSERT2 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~a into the Indexed Sequence ~A."
   (INPUT-PORT-NAME> (DOC-BP> (INDEXED-SEQUENCE-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (INDEXED-SEQUENCE-INSERT 2)))))

(Defrule INDEXED-SEQUENCE-INSERT
  "Indexed-Sequence Insert"
  :RHS-Node-Types
  ((I-S-INSERT1 . BUMP+UPDATE))
  :Input-Embedding
  (((INDEXED-SEQUENCE-INSERT 1) (I-S-INSERT1 1))
   ((INDEXED-SEQUENCE-INSERT 2) (I-S-INSERT1 2)))
  :Output-Embedding
  (((INDEXED-SEQUENCE-INSERT 3) (I-S-INSERT1 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~a into the Indexed Sequence ~A."
   (INPUT-PORT-NAME> (DOC-BP> (INDEXED-SEQUENCE-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (INDEXED-SEQUENCE-INSERT 2)))))

(Defrule INDEXED-SEQUENCE-EXTRACT
  "Indexed-Sequence Extract"
  :RHS-Node-Types
  ((I-S-EXTRACT2 . UPDATE+FETCH))
  :Input-Embedding
  (((INDEXED-SEQUENCE-EXTRACT 1) (I-S-EXTRACT2 1)))
  :Output-Embedding
  (((INDEXED-SEQUENCE-EXTRACT 2) (I-S-EXTRACT2 2))
   ((INDEXED-SEQUENCE-EXTRACT 3) (I-S-EXTRACT2 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("extracts the current element from the Indexed Sequence ~A."
   (INPUT-PORT-NAME> (DOC-BP> (INDEXED-SEQUENCE-EXTRACT 1)))))

(Defrule INDEXED-SEQUENCE-EXTRACT
  "Indexed-Sequence Extract"
  :RHS-Node-Types
  ((I-S-EXTRACT1 . FETCH+UPDATE))
  :Input-Embedding
  (((INDEXED-SEQUENCE-EXTRACT 1) (I-S-EXTRACT1 1)))
  :Output-Embedding
  (((INDEXED-SEQUENCE-EXTRACT 2) (I-S-EXTRACT1 2))
   ((INDEXED-SEQUENCE-EXTRACT 3) (I-S-EXTRACT1 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("extracts the current element from the Indexed Sequence ~A."
   (INPUT-PORT-NAME> (DOC-BP> (INDEXED-SEQUENCE-EXTRACT 1)))))

(Defrule INDEXED-SEQUENCE-ACCUMULATION
  "Indexed-Sequence Accumulation"
  :RHS-Node-Types
  ((INSERT-INTO-I-S . INDEXED-SEQUENCE-INSERT))
  :Input-Embedding
  (((INDEXED-SEQUENCE-ACCUMULATION 1) (INSERT-INTO-I-S 1))
   ((INDEXED-SEQUENCE-ACCUMULATION 2) (INSERT-INTO-I-S 2)))
  :St-Thrus
  (((INDEXED-SEQUENCE-ACCUMULATION 2) (INDEXED-SEQUENCE-ACCUMULATION 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("accumulates the elements in the series into a new indexed-sequence."
   (INPUT-PORT-NAME> (DOC-BP> (INDEXED-SEQUENCE-ACCUMULATION 1)))))

(Defrule ASSOCIATIVE-SET-ADD
  "Associative Set Add"
  :RHS-Node-Types
  ((THE-ALIST-INSERT . ASSOCIATIVE-LIST-INSERT))
  :Input-Embedding
  (((ASSOCIATIVE-SET-ADD 1) (THE-ALIST-INSERT 1))
   ((ASSOCIATIVE-SET-ADD 2) (THE-ALIST-INSERT 2))
   ((ASSOCIATIVE-SET-ADD 3) (THE-ALIST-INSERT 3)))
  :Output-Embedding
  (((ASSOCIATIVE-SET-ADD 4) (THE-ALIST-INSERT 4)))
```

```
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~A (associated w/ key ~A) in the associative set ~A.~
    An element X occurs before another Y if X's key ~A Y's key. ~
    An element X replaces another Y if X's key ~A Y's key."
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-ADD 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-ADD 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-ADD 3)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-COMPARATOR-INFO (N> THE-ALIST-INSERT))))
   (FUNCTION-TYPE (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> THE-ALIST-INSERT))))))

(Defrule ASSOCIATIVE-SET-ADD
  "Associative Set Add"
  :RHS-Node-Types
  ((THE-HT-INSERT . HASH-INSERT))
  :Input-Embedding
  (((ASSOCIATIVE-SET-ADD 1) (THE-HT-INSERT 1))
   ((ASSOCIATIVE-SET-ADD 2) (THE-HT-INSERT 2))
   ((ASSOCIATIVE-SET-ADD 3) (THE-HT-INSERT 3)))
  :Output-Embedding
  (((ASSOCIATIVE-SET-ADD 4) (THE-HT-INSERT 4)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~A (associated w/ key ~A) in the associative set ~A.~
    An element X occurs before another Y if X's key ~A Y's key.~
    An element X replaces another Y if X's key ~A Y's key."
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-ADD 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-ADD 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-ADD 3)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-COMPARATOR-INFO (N> THE-HT-INSERT))))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> THE-HT-INSERT))))))

(Defrule ASSOCIATIVE-SET-REMOVE
  "Associative Set Remove"
  :RHS-Node-Types
  ((THE-ALIST-DELETE . ASSOCIATIVE-LIST-DELETE))
  :Input-Embedding
  (((ASSOCIATIVE-SET-REMOVE 1) (THE-ALIST-DELETE 1))
   ((ASSOCIATIVE-SET-REMOVE 2) (THE-ALIST-DELETE 2)))
  :Output-Embedding
  (((ASSOCIATIVE-SET-REMOVE 3) (THE-ALIST-DELETE 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("deletes an element associated w/ key ~A in the associative ~
    set ~A.  An element X occurs before another Y if X's key ~A ~
    Y's key.  Keys are compared using ~A."
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-REMOVE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-REMOVE 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-COMPARATOR-INFO (N> THE-ALIST-DELETE))))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> THE-ALIST-DELETE))))))

(Defrule ASSOCIATIVE-SET-REMOVE
  "Associative Set Remove"
  :RHS-Node-Types
  ((THE-HT-DELETE . HASH-DELETE))
  :Input-Embedding
  (((ASSOCIATIVE-SET-REMOVE 1) (THE-HT-DELETE 1))
   ((ASSOCIATIVE-SET-REMOVE 2) (THE-HT-DELETE 2)))
  :Output-Embedding
  (((ASSOCIATIVE-SET-REMOVE 3) (THE-HT-DELETE 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("deletes an element associated w/ key ~A in the associative ~
    set ~A.  An element X occurs before another Y if X's key ~A ~
    Y's key.  Keys are compared for equality using ~A."
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-REMOVE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-REMOVE 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-COMPARATOR-INFO (N> THE-HT-DELETE))))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> THE-HT-DELETE))))))

(Defrule ASSOCIATIVE-SET-LOOKUP
  "Associative Set Lookup"
  :RHS-Node-Types
  ((THE-ALIST-LOOKUP . ASSOCIATIVE-LIST-LOOKUP))
  :Input-Embedding
  (((ASSOCIATIVE-SET-LOOKUP 1) (THE-ALIST-LOOKUP 1))
   ((ASSOCIATIVE-SET-LOOKUP 2) (THE-ALIST-LOOKUP 2)))
  :Output-Embedding
  (((ASSOCIATIVE-SET-LOOKUP 3) (THE-ALIST-LOOKUP 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up an element associated w/ key ~A in the associative ~
    set ~A.  An element X occurs before another Y if X's key ~A ~
    Y's key.  Keys are compared using ~A."
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-LOOKUP 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-LOOKUP 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
                   (KEY-COMPARATOR-INFO (N> THE-ALIST-LOOKUP))))
   (FUNCTION-NAME (FUNCTION-TYPE
                   (KEY-EQUALITY-INFO (N> THE-ALIST-LOOKUP))))))

(Defrule ASSOCIATIVE-SET-LOOKUP
  "Associative Set Lookup"
  :RHS-Node-Types
  ((THE-HT-LOOKUP . HASH-LOOKUP))
  :Input-Embedding
  (((ASSOCIATIVE-SET-LOOKUP 1) (THE-HT-LOOKUP 1))
   ((ASSOCIATIVE-SET-LOOKUP 2) (THE-HT-LOOKUP 2)))
  :Output-Embedding
  (((ASSOCIATIVE-SET-LOOKUP 3) (THE-HT-LOOKUP 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up an element associated w/ key ~A in the associative set ~A. ~
    An element X occurs before another Y if X's key ~A Y's key. ~
    An element X is retrieved if X's key ~A ~A."
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-LOOKUP 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-LOOKUP 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
                   (KEY-COMPARATOR-INFO (N> THE-HT-LOOKUP))))
   (FUNCTION-NAME (FUNCTION-TYPE
                   (KEY-EQUALITY-INFO (N> THE-HT-LOOKUP))))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-SET-LOOKUP 1)))))

(Defrule PROPERTY-LIST-LOOKUP
  "Property List Lookup"
  :RHS-Node-Types
  ((GET-AT-INDICATOR . GET))
  :Input-Embedding
  (((PROPERTY-LIST-LOOKUP 1) (GET-AT-INDICATOR 1))
   ((PROPERTY-LIST-LOOKUP 2) (GET-AT-INDICATOR 2)))
  :Output-Embedding
  (((PROPERTY-LIST-LOOKUP 3) (GET-AT-INDICATOR 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up the value associated w/ the indicator ~A in the ~
    property-list of the symbol ~A."
   (INPUT-PORT-NAME> (DOC-BP> (PROPERTY-LIST-LOOKUP 2)))
   (INPUT-PORT-NAME> (DOC-BP> (PROPERTY-LIST-LOOKUP 1)))))

(Defrule HASH-LOOKUP
  "Hash Table Lookup"
  :RHS-Node-Types
  ((CHT-LOOKUP . CHAINING-HT-LOOKUP))
  :Input-Embedding
  (((HASH-LOOKUP 1) (CHT-LOOKUP 1))
   ((HASH-LOOKUP 2) (CHT-LOOKUP 2)))
  :Output-Embedding
  (((HASH-LOOKUP 3) (CHT-LOOKUP 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up an element with key ~A from the Hash-Table ~A."
   (INPUT-PORT-NAME> (DOC-BP> (HASH-LOOKUP 1)))
   (INPUT-PORT-NAME> (ALL-BP> (HASH-LOOKUP 2)))))

(Defrule HASH-DELETE
  "Hash Table Delete"
  :RHS-Node-Types
  ((CHT-DELETE . CHAINING-HT-DELETE))
  :Input-Embedding
  (((HASH-DELETE 1) (CHT-DELETE 1))
   ((HASH-DELETE 2) (CHT-DELETE 2)))
  :Output-Embedding
  (((HASH-DELETE 3) (CHT-DELETE 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("deletes an element with key ~A from the Hash-Table ~A."
   (INPUT-PORT-NAME> (DOC-BP> (HASH-DELETE 1)))
   (INPUT-PORT-NAME> (ALL-BP> (HASH-DELETE 2)))))

(Defrule HASH-INSERT
  "Hash Table Insert"
  :RHS-Node-Types
  ((CHT-INSERT . CHAINING-HT-INSERT))
  :Input-Embedding
  (((HASH-INSERT 1) (CHT-INSERT 1))
   ((HASH-INSERT 2) (CHT-INSERT 2))
   ((HASH-INSERT 3) (CHT-INSERT 3)))
  :Output-Embedding
  (((HASH-INSERT 4) (CHT-INSERT 4)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~A with key ~A into the Hash-Table ~A."
   (INPUT-PORT-NAME> (DOC-BP> (HASH-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (HASH-INSERT 2)))
   (INPUT-PORT-NAME> (ALL-BP> (HASH-INSERT 3)))))

(Defrule CHAINING-HT-LOOKUP
  "Chaining Hash Table Lookup"
```

```
  :RHS-Node-Types
  ((RETRIEVE-AND-SEARCH . FETCH+LOOKUP))
  :Input-Embedding
  (((CHAINING-HT-LOOKUP 1) (RETRIEVE-AND-SEARCH 1))
   ((CHAINING-HT-LOOKUP 2) (RETRIEVE-AND-SEARCH 2)))
  :Output-Embedding
  (((CHAINING-HT-LOOKUP 3) (RETRIEVE-AND-SEARCH 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up an element with key ~A from the chaining ~
    hash-table ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-LOOKUP 1)))
   (INPUT-PORT-NAME> (ALL-BP> (CHAINING-HT-LOOKUP 2)))))

(Defrule CHAINING-HT-DELETE
  "Chaining Hash Table Delete"
  :RHS-Node-Types
  ((RETRIEVE-AND-DELETE . CHAINING-HT-FILL-COUNT-DELETE))
  :Input-Embedding
  (((CHAINING-HT-DELETE 1) (RETRIEVE-AND-DELETE 1))
   ((CHAINING-HT-DELETE 2) (RETRIEVE-AND-DELETE 2)))
  :Output-Embedding
  (((CHAINING-HT-DELETE 3) (RETRIEVE-AND-DELETE 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("deletes an element with key ~A from the chaining ~
    hash-table ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-DELETE 1)))
   (INPUT-PORT-NAME> (ALL-BP> (CHAINING-HT-DELETE 2)))))

(Defrule CHAINING-HT-INSERT
  "Chaining Hash Table Insert"
  :RHS-Node-Types
  ((RETRIEVE-AND-INSERT . CHAINING-HT-FILL-COUNT-INSERT))
  :Input-Embedding
  (((CHAINING-HT-INSERT 1) (RETRIEVE-AND-INSERT 1))
   ((CHAINING-HT-INSERT 2) (RETRIEVE-AND-INSERT 2))
   ((CHAINING-HT-INSERT 3) (RETRIEVE-AND-INSERT 3)))
  :Output-Embedding
  (((CHAINING-HT-INSERT 4) (RETRIEVE-AND-INSERT 4)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~A with key ~A into the chaining Hash-Table ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-INSERT 2)))
   (INPUT-PORT-NAME> (ALL-BP> (CHAINING-HT-INSERT 3)))))

(Defrule FETCH+LOOKUP
  "Fetch Bucket and Lookup Element"
  :RHS-Node-Types
  ((HASH-KEY-AND-SIZE . HASH-FUNCTION)
   (GET-BUCKET . SELECT-TERM)
   (LOOKUP . ASSOCIATIVE-LIST-LOOKUP))
  :Edge-List
  (((HASH-KEY-AND-SIZE 3) . (GET-BUCKET 2))
   ((GET-BUCKET 3) . (LOOKUP 2)))
  :Input-Embedding
  (((FETCH+LOOKUP 1) (LOOKUP 1))
   ((FETCH+LOOKUP 1) (HASH-KEY-AND-SIZE 1))
   ((FETCH+LOOKUP 2) (HASH-KEY-AND-SIZE 2)
    NUMBER-BUCKETS)
   ((FETCH+LOOKUP 2) (GET-BUCKET 1)
    BUCKETS))
  :Output-Embedding
  (((FETCH+LOOKUP 3) (LOOKUP 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("looks up an element with key ~A from the hash-table ~A, ~
    which is implemented as an sequence ~A of buckets.  The ~
    bucket is fetched indexing into the sequence using an ~
    index computed by applying a hash function to the key ~
    ~A and the number of buckets in the hash table ~A.~%~
    Each bucket is implemented as an associative list.~%~
    Collision resolution is performed using a chaining strategy."
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+LOOKUP 1)))
   (INPUT-PORT-NAME> (ALL-BP> (FETCH+LOOKUP 2)))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+LOOKUP 2) BUCKETS))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+LOOKUP 1)))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+LOOKUP 2) NUMBER-BUCKETS))))

(Defrule FETCH+DELETE
  "Fetch Bucket and Delete Element"
  :RHS-Node-Types
  ((HASH-THE-KEY . HASH-FUNCTION)
   (FETCH-BUCKET . SELECT-TERM)
   (REMOVE . ASSOCIATIVE-LIST-DELETE)
   (UPDATE-BUCKETS . NEW-TERM))
  :Edge-List
  (((HASH-THE-KEY 3) . (UPDATE-BUCKETS 2))
   ((HASH-THE-KEY 3) . (FETCH-BUCKET 2))
   ((FETCH-BUCKET 3) . (REMOVE 2))
   ((REMOVE 3) . (UPDATE-BUCKETS 1)))
  :Input-Embedding
  (((FETCH+DELETE 1) (REMOVE 1))
   ((FETCH+DELETE 1) (HASH-THE-KEY 1))
   ((FETCH+DELETE 2) (HASH-THE-KEY 2)
    NUMBER-BUCKETS)
   ((FETCH+DELETE 2) (UPDATE-BUCKETS 3)
    BUCKETS)
   ((FETCH+DELETE 2) (FETCH-BUCKET 1)
    BUCKETS))
  :Output-Embedding
  (((FETCH+DELETE 3) (UPDATE-BUCKETS 4)
    BUCKETS))
  :St-Thrus
  (((FETCH+DELETE 2) (FETCH+DELETE 3)
    NUMBER-BUCKETS))
  :L-R-Link COMPOSITION
  :Doc
  ("deletes an element with key ~A from the hash-table ~A, which is ~
    implemented as a sequence ~A of buckets.  The bucket is fetched by ~
    indexing into the sequence using an index computed by applying a ~
    hash function to the key ~A and the number of buckets in the hash ~
    table ~A.~%~
    Each bucket is implemented as an associative list.~%~
    Collision resolution is performed using a chaining strategy."
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+DELETE 1)))
   (INPUT-PORT-NAME> (ALL-BP> (FETCH+DELETE 2)))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+DELETE 2) BUCKETS))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+DELETE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+DELETE 2) NUMBER-BUCKETS))))

(Defrule FETCH+INSERT
  "Fetch Bucket and Insert Element"
  :RHS-Node-Types
  ((COMPUTE-HASH . HASH-FUNCTION)
   (FETCH . SELECT-TERM)
   (INSERT . ASSOCIATIVE-LIST-INSERT)
   (UPDATE . NEW-TERM))
  :Edge-List
  (((COMPUTE-HASH 3) . (UPDATE 2))
   ((COMPUTE-HASH 3) . (FETCH 2))
   ((FETCH 3) . (INSERT 3))
   ((INSERT 4) . (UPDATE 1)))
  :Input-Embedding
  (((FETCH+INSERT 1) (INSERT 1))
   ((FETCH+INSERT 2) (INSERT 2))
   ((FETCH+INSERT 2) (COMPUTE-HASH 1))
   ((FETCH+INSERT 3) (COMPUTE-HASH 2)
    NUMBER-BUCKETS)
   ((FETCH+INSERT 3) (UPDATE 3)
    BUCKETS)
   ((FETCH+INSERT 3) (FETCH 1)
    BUCKETS))
  :Output-Embedding
  (((FETCH+INSERT 4) (UPDATE 4)
    BUCKETS))
  :St-Thrus
  (((FETCH+INSERT 3) (FETCH+INSERT 4)
    NUMBER-BUCKETS))
  :L-R-Link COMPOSITION
  :Doc
  ("inserts ~A into the hash-table ~A, which is implemented as a ~
    sequence ~A of buckets.  The bucket is fetched by indexing into ~
    the sequence using an index computed by applying a hash function ~
    to the key ~A and the number of buckets in the hash table ~A.~%~
    Each bucket is implemented as an associative list.~%~
    Collision resolution is performed using a chaining strategy."
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+INSERT 1)))
   (INPUT-PORT-NAME> (ALL-BP> (FETCH+INSERT 3)))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+INSERT 3) BUCKETS))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+INSERT 2)))
   (INPUT-PORT-NAME> (DOC-BP> (FETCH+INSERT 3) NUMBER-BUCKETS))))

(Defrule CHAINING-HT-FILL-COUNT-DELETE
  "Hash Table with Fill Count Delete"
  :RHS-Node-Types
  ((DELETE-ELEMENT . FETCH+DELETE)
   (DECREMENT-ELT-COUNT . DECREMENT))
  :Input-Embedding
  (((CHAINING-HT-FILL-COUNT-DELETE 1) (DELETE-ELEMENT 1))
   ((CHAINING-HT-FILL-COUNT-DELETE 2) (DELETE-ELEMENT 2)
    HASH-TABLE)
   ((CHAINING-HT-FILL-COUNT-DELETE 2) (DECREMENT-ELT-COUNT 1)
    FILL-COUNT))
  :Output-Embedding
  (((CHAINING-HT-FILL-COUNT-DELETE 3) (DELETE-ELEMENT 3)
    HASH-TABLE)
   ((CHAINING-HT-FILL-COUNT-DELETE 3) (DECREMENT-ELT-COUNT 2)
    FILL-COUNT))
  :St-Thrus
  (((CHAINING-HT-FILL-COUNT-DELETE 2) (CHAINING-HT-FILL-COUNT-DELETE 3)
    FILL-COUNT))
  :L-R-Link COMPOSITION
```

```
  :Doc
  ("deletes an element with key ~A from the chaining ~
    Hash-Table+Fill-Count ~A.  This is a hash-table which ~
    contains a fill count ~A, keeping track of the number of ~
    elements in the hash table."
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-FILL-COUNT-DELETE 1)))
   (INPUT-PORT-NAME> (ALL-BP> (CHAINING-HT-FILL-COUNT-DELETE 2)))
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-FILL-COUNT-DELETE 2)
                              FILL-COUNT))))

(Defrule CHAINING-HT-FILL-COUNT-INSERT
  "Hash Table with Fill Count Insert"
  :RHS-Node-Types
  ((ADD-ELEMENT . FETCH+INSERT)
   (INCREMENT-ELT-COUNT . INCREMENT))
  :Input-Embedding
  (((CHAINING-HT-FILL-COUNT-INSERT 1) (ADD-ELEMENT 1))
   ((CHAINING-HT-FILL-COUNT-INSERT 2) (ADD-ELEMENT 2))
   ((CHAINING-HT-FILL-COUNT-INSERT 3) (ADD-ELEMENT 3)
    HASH-TABLE)
   ((CHAINING-HT-FILL-COUNT-INSERT 3) (INCREMENT-ELT-COUNT 1)
    FILL-COUNT))
  :Output-Embedding
  (((CHAINING-HT-FILL-COUNT-INSERT 4) (ADD-ELEMENT 4)
    HASH-TABLE)
   ((CHAINING-HT-FILL-COUNT-INSERT 4) (INCREMENT-ELT-COUNT 2)
    FILL-COUNT))
  :St-Thrus
  (((CHAINING-HT-FILL-COUNT-INSERT 3)
    (CHAINING-HT-FILL-COUNT-INSERT 4)
    FILL-COUNT))
  :L-R-Link COMPOSITION
  :Doc
  ("inserts ~A with key ~A into the chaining ~
    Hash-Table+Fill-Count ~A.  This is a hash-table which ~
    contains a fill count ~A, keeping track of the number of ~
    elements in the hash table."
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-FILL-COUNT-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-FILL-COUNT-INSERT 2)))
   (INPUT-PORT-NAME> (ALL-BP> (CHAINING-HT-FILL-COUNT-INSERT 3)))
   (INPUT-PORT-NAME> (DOC-BP> (CHAINING-HT-FILL-COUNT-INSERT 3)
                              FILL-COUNT))))

;;; Figure 4-24.

(Defrule LOOKUP-DESTINATION
  "Lookup Destination Node"
  :RHS-Node-Types
  ((COMPUTE-DEST . SELECT-TERM))
  :Input-Embedding
  (((LOOKUP-DESTINATION 1) (COMPUTE-DEST 1))
   ((LOOKUP-DESTINATION 2) (COMPUTE-DEST 2)
    DEST-ADDR))
  :Output-Embedding
  (((LOOKUP-DESTINATION 3) (COMPUTE-DEST 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("looks up the node whose address is in the Dest-Addr part of ~
    message ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LOOKUP-DESTINATION 2)))))

;;; Figure 4-24.

(Defrule RECORD-AT-DESTINATION
  "Record Node at Message Destination"
  :RHS-Node-Types
  ((RECORD . NEW-TERM))
  :Input-Embedding
  (((RECORD-AT-DESTINATION 1) (RECORD 1))
   ((RECORD-AT-DESTINATION 2) (RECORD 2)
    DEST-ADDR)
   ((RECORD-AT-DESTINATION 3) (RECORD 3)))
  :Output-Embedding
  (((RECORD-AT-DESTINATION 4) (RECORD 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("records node ~A at the address in the Dest-Addr part of ~
    message ~A in the address map ~A."
   (INPUT-PORT-NAME> (DOC-BP> (RECORD-AT-DESTINATION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (RECORD-AT-DESTINATION 2)))
   (INPUT-PORT-NAME> (DOC-BP> (RECORD-AT-DESTINATION 3)))))

(Defrule ASSOCIATIVE-LIST-LOOKUP
  "Associative Linked List Lookup"
  :RHS-Node-Types
  ((THE-UOAL-LOOKUP . UNORDERED-ASSOC-LIST-LOOKUP))
  :Input-Embedding
  (((ASSOCIATIVE-LIST-LOOKUP 1) (THE-UOAL-LOOKUP 1))
   ((ASSOCIATIVE-LIST-LOOKUP 2) (THE-UOAL-LOOKUP 2)))
  :Output-Embedding
  (((ASSOCIATIVE-LIST-LOOKUP 3) (THE-UOAL-LOOKUP 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up the element associated w/ key ~A ~A in the associative ~
    list ~A."
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> ASSOCIATIVE-LIST-LOOKUP))))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-LOOKUP 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-LOOKUP 2)))))

(Defrule ASSOCIATIVE-LIST-LOOKUP
  "Associative Linked List Lookup"
  :RHS-Node-Types
  ((THE-OAL-LOOKUP . ORDERED-ASSOC-LIST-LOOKUP))
  :Input-Embedding
  (((ASSOCIATIVE-LIST-LOOKUP 1) (THE-OAL-LOOKUP 1))
   ((ASSOCIATIVE-LIST-LOOKUP 2) (THE-OAL-LOOKUP 2)))
  :Output-Embedding
  (((ASSOCIATIVE-LIST-LOOKUP 3) (THE-OAL-LOOKUP 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("looks up the element associated w/ key ~A ~A in the associative ~
    list ~A."
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> ASSOCIATIVE-LIST-LOOKUP))))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-LOOKUP 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-LOOKUP 2)))))

(Defrule ASSOCIATIVE-LIST-DELETE
  "Associative Linked List Delete"
  :RHS-Node-Types
  ((THE-UOAL-DELETE . UNORDERED-ASSOC-LIST-DELETE))
  :Input-Embedding
  (((ASSOCIATIVE-LIST-DELETE 1) (THE-UOAL-DELETE 1))
   ((ASSOCIATIVE-LIST-DELETE 2) (THE-UOAL-DELETE 2)))
  :Output-Embedding
  (((ASSOCIATIVE-LIST-DELETE 3) (THE-UOAL-DELETE 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("deletes the element associated w/ key ~A ~A in the associative ~
    list ~A."
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> ASSOCIATIVE-LIST-DELETE))))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-DELETE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-DELETE 2)))))

(Defrule ASSOCIATIVE-LIST-DELETE
  "Associative Linked List Delete"
  :RHS-Node-Types
  ((THE-OAL-DELETE . ORDERED-ASSOC-LIST-DELETE))
  :Input-Embedding
  (((ASSOCIATIVE-LIST-DELETE 1) (THE-OAL-DELETE 1))
   ((ASSOCIATIVE-LIST-DELETE 2) (THE-OAL-DELETE 2)))
  :Output-Embedding
  (((ASSOCIATIVE-LIST-DELETE 3) (THE-OAL-DELETE 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("deletes the element associated w/ key ~A ~A in the associative ~
    list ~A."
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> ASSOCIATIVE-LIST-DELETE))))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-DELETE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-DELETE 2)))))

(Defrule ASSOCIATIVE-LIST-INSERT
  "Associative Linked List Insert"
  :RHS-Node-Types
  ((THE-UNORDERED-AL-INSERT . UNORDERED-ASSOC-LIST-INSERT))
  :Input-Embedding
  (((ASSOCIATIVE-LIST-INSERT 1) (THE-UNORDERED-AL-INSERT 1))
   ((ASSOCIATIVE-LIST-INSERT 2) (THE-UNORDERED-AL-INSERT 2))
   ((ASSOCIATIVE-LIST-INSERT 3) (THE-UNORDERED-AL-INSERT 3)))
  :Output-Embedding
  (((ASSOCIATIVE-LIST-INSERT 4) (THE-UNORDERED-AL-INSERT 4)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~A (associated w/ key ~A) in the associative list ~A.~%~
    An element X replaces another Y if X's key ~A Y's key."
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-INSERT 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-INSERT 3)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> THE-UNORDERED-AL-INSERT))))))

(Defrule ASSOCIATIVE-LIST-INSERT
  "Associative Linked List Insert"
  :RHS-Node-Types
  ((THE-OAL-INSERT . ORDERED-ASSOC-LIST-INSERT))
  :Input-Embedding
  (((ASSOCIATIVE-LIST-INSERT 1) (THE-OAL-INSERT 1))
   ((ASSOCIATIVE-LIST-INSERT 2) (THE-OAL-INSERT 2))
   ((ASSOCIATIVE-LIST-INSERT 3) (THE-OAL-INSERT 3)))
  :Output-Embedding
  (((ASSOCIATIVE-LIST-INSERT 4) (THE-OAL-INSERT 4)))
  :L-R-Link IMPLEMENTATION
```

```
   :Doc
   ("inserts ~A (associated w/ key ~A) in the associative
     list ~A.~%~
     An element X replaces another Y if X's key ~A Y's key."
    (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-INSERT 1)))
    (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-INSERT 2)))
    (INPUT-PORT-NAME> (DOC-BP> (ASSOCIATIVE-LIST-INSERT 3)))
    (FUNCTION-NAME (FUNCTION-TYPE
      (KEY-EQUALITY-INFO (N> ASSOCIATIVE-LIST-INSERT))))))

(Defrule UNORDERED-ASSOC-LIST-LOOKUP
  "Unordered Associative Linked List Lookup"
  :RHS-Node-Types
  ((UOAL-ENUM . LE)
   (FIND-ELT . EARLIEST-EQUAL-PRIORITY))
  :Edge-List
  (((UOAL-ENUM 2) . (FIND-ELT 1)))
  :Input-Embedding
  (((UNORDERED-ASSOC-LIST-LOOKUP 1) (FIND-ELT 2))
   ((UNORDERED-ASSOC-LIST-LOOKUP 2) (UOAL-ENUM 1)))
  :Output-Embedding
  (((UNORDERED-ASSOC-LIST-LOOKUP 3) (FIND-ELT 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("searches the elements of the unordered associative list ~A ~
    for an element with key ~A ~A.  If no such element is ~
    found, NIL is returned."
   (INPUT-PORT-NAME> (DOC-BP> (UNORDERED-ASSOC-LIST-LOOKUP 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> UNORDERED-ASSOC-LIST-LOOKUP))))
   (INPUT-PORT-NAME> (DOC-BP> (UNORDERED-ASSOC-LIST-LOOKUP 1)))))

(Defrule UNORDERED-ASSOC-LIST-INSERT
  "Unordered Associative Linked List Insert"
  :RHS-Node-Types
  ((UOAL-PUSH . LIST-PUSH))
  :Input-Embedding
  (((UNORDERED-ASSOC-LIST-INSERT 1) (UOAL-PUSH 1))
   ((UNORDERED-ASSOC-LIST-INSERT 2) (UOAL-PUSH 2)))
  :Output-Embedding
  (((UNORDERED-ASSOC-LIST-INSERT 3) (UOAL-PUSH 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~A into the unordered associative list ~A."
   (INPUT-PORT-NAME> (DOC-BP> (UNORDERED-ASSOC-LIST-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (UNORDERED-ASSOC-LIST-INSERT 2)))))

(Defrule UNORDERED-ASSOC-LIST-EMPTY?
  "Unordered Associative List Empty"
  :RHS-Node-Types
  ((UOAL-EMPTY? . LIST-EMPTY))
  :Input-Embedding
  (((UNORDERED-ASSOC-LIST-EMPTY? 1) (UOAL-EMPTY? 1)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("tests whether the unordered associative list ~A is empty."
   (INPUT-PORT-NAME> (DOC-BP> (UNORDERED-ASSOC-LIST-EMPTY? 1)))))

(Defrule INTERMEDIATE-UOAL-DELETE
  "Unordered Associative Linked List Delete (Intermediate)"
  :RHS-Node-Types
  ((GENERATE-CURRENT+NEXT-SUBLIST . TRAILING-GENERATE)
   (LIST-EXHAUSTED . TRUNCATE)
   (ELTS-BEFORE-P . TRUNCATE-EQUAL-PRIORITY-HEAD)
   (COLLECT-REMAINING . CONS-ACCUMULATE-UP-FROM-SUBLIST))
  :Edge-List
  (((GENERATE-CURRENT+NEXT-SUBLIST 3) . (COLLECT-REMAINING 2))
   ((GENERATE-CURRENT+NEXT-SUBLIST 2) . (LIST-EXHAUSTED 1))
   ((LIST-EXHAUSTED 2) . (ELTS-BEFORE-P 1))
   ((ELTS-BEFORE-P 3) . (COLLECT-REMAINING 1)))
  :Input-Embedding
  (((INTERMEDIATE-UOAL-DELETE 1) (ELTS-BEFORE-P 2))
   ((INTERMEDIATE-UOAL-DELETE 2)
    (GENERATE-CURRENT+NEXT-SUBLIST 1))
   ((INTERMEDIATE-UOAL-DELETE 3) (COLLECT-REMAINING 2)))
  :Output-Embedding
  (((INTERMEDIATE-UOAL-DELETE 4) (COLLECT-REMAINING 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("intermediate nonterminal: Unordered-Assoc-List-Delete."))

(Defrule UNORDERED-ASSOC-LIST-DELETE
  "Unordered Associative Linked List Delete"
  :RHS-Node-Types
  ((SPLICE-OUT-ELT . INTERMEDIATE-UOAL-DELETE))
  :Input-Embedding
  (((UNORDERED-ASSOC-LIST-DELETE 1) (SPLICE-OUT-ELT 1))
   ((UNORDERED-ASSOC-LIST-DELETE 2) (SPLICE-OUT-ELT 2)))
  :Output-Embedding
  (((UNORDERED-ASSOC-LIST-DELETE 3) (SPLICE-OUT-ELT 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("splices out the element of the unordered associative list ~
    ~A whose key is ~A ~A."
   (INPUT-PORT-NAME> (DOC-BP> (UNORDERED-ASSOC-LIST-DELETE 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (KEY-EQUALITY-INFO (N> UNORDERED-ASSOC-LIST-DELETE))))
   (INPUT-PORT-NAME> (DOC-BP> (UNORDERED-ASSOC-LIST-DELETE 1)))))

(Defrule PQ-ENUMERATION
  "Priority Queue Enumeration"
  :RHS-Node-Types
  ((PQ-ENUM-FINISHED? . PQ-EMPTY)
   (PQ-EXTRACT-NEXT . PQ-EXTRACT))
  :Input-Embedding
  (((PQ-ENUMERATION 1) (PQ-EXTRACT-NEXT 1))
   ((PQ-ENUMERATION 1) (PQ-ENUM-FINISHED? 1)))
  :Output-Embedding
  (((PQ-ENUMERATION 2) (PQ-EXTRACT-NEXT 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates all of the elements in the Priority-Queue ~A,~%~
    by destructively extracting them from the queue."
   (INPUT-PORT-NAME> (DOC-BP> (PQ-ENUMERATION 1)))))

(Defrule PQ-EMPTY
  "Priority Queue Empty"
  :RHS-Node-Types
  ((EMPTY-LIST? . TEST-PREDICATE))
  :Input-Embedding
  (((PQ-EMPTY 1) (EMPTY-LIST? 1)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("tests whether the Priority Queue ~A is empty."
   (INPUT-PORT-NAME> (DOC-BP> (PQ-EMPTY 1)))))

(Defrule PQ-EXTRACT
  "Priority Queue Extract"
  :RHS-Node-Types
  ((EXTRACT-FROM-OAL . ORDERED-ASSOC-LIST-EXTRACT))
  :Input-Embedding
  (((PQ-EXTRACT 1) (EXTRACT-FROM-OAL 1)))
  :Output-Embedding
  (((PQ-EXTRACT 2) (EXTRACT-FROM-OAL 2))
   ((PQ-EXTRACT 3) (EXTRACT-FROM-OAL 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("extracts the highest priority element in the Priority Queue ~A.~%~
    The priority queue is implemented as an ordered associative list."
   (INPUT-PORT-NAME> (DOC-BP> (EXTRACT-FROM-OAL 1)))))

(Defrule PQ-INSERT
  "Priority Queue Insert"
  :RHS-Node-Types
  ((ORDERED-SPLICE-IN . ORDERED-ASSOC-LIST-INSERT))
  :Input-Embedding
  (((PQ-INSERT 1) (ORDERED-SPLICE-IN 1))
   ((PQ-INSERT 2) (ORDERED-SPLICE-IN 2))
   ((PQ-INSERT 3) (ORDERED-SPLICE-IN 3)))
  :Output-Embedding
  (((PQ-INSERT 4) (ORDERED-SPLICE-IN 4)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~A in the priority queue ~A.~%~
    An element's priority P is higher than another's Q, if P ~A Q.~%~
    If an element already exists in the priority queue with the same ~
    priority, then the new element is inserted into the queue after ~
    the existing element."
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-SPLICE-IN 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-SPLICE-IN 3)))
   (FUNCTION-NAME (FUNCTION-TYPE
    (PRIORITY-COMPARATOR-INFO (N> ORDERED-SPLICE-IN))))))

(Defrule ORDERED-ASSOC-LIST-INSERT
  "Ordered Associative List Insert"
  :RHS-Node-Types
  ((THE-UNSAFE-INSERT . ORDERED-ASSOC-LIST-INSERT-UNSAFE))
  :Input-Embedding
  (((ORDERED-ASSOC-LIST-INSERT 1) (THE-UNSAFE-INSERT 1))
   ((ORDERED-ASSOC-LIST-INSERT 2) (THE-UNSAFE-INSERT 2))
   ((ORDERED-ASSOC-LIST-INSERT 3) (THE-UNSAFE-INSERT 3)))
  :Output-Embedding
  (((ORDERED-ASSOC-LIST-INSERT 4) (THE-UNSAFE-INSERT 4)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("inserts ~A in the ordered associative list ~A, associated with ~
    priority ~A.  An element X occurs before another Y if X's priority ~
    ~A Y's priority."
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-INSERT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-INSERT 3)))
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-INSERT 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-COMPARATOR-INFO (N> THE-UNSAFE-INSERT))))))

(Defrule ORDERED-ASSOC-LIST-INSERT
  "Ordered Associative List Insert"
```

```
   :RHS-Node-Types
   ((THE-SAFE-INSERT . ORDERED-ASSOC-LIST-INSERT-SAFE))
   :Input-Embedding
   (((ORDERED-ASSOC-LIST-INSERT 1) (THE-SAFE-INSERT 1))
    ((ORDERED-ASSOC-LIST-INSERT 2) (THE-SAFE-INSERT 2))
    ((ORDERED-ASSOC-LIST-INSERT 3) (THE-SAFE-INSERT 3)))
   :Output-Embedding
   (((ORDERED-ASSOC-LIST-INSERT 4) (THE-SAFE-INSERT 4)))
   :L-R-Link IMPLEMENTATION
   :Doc
   ("inserts ~A in the ordered associative list ~A, associated ~
     with priority ~A.  An element X occurs before another Y if ~
     X's priority ~A Y's priority."
    (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-INSERT 1)))
    (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-INSERT 3)))
    (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-INSERT 2)))
    (FUNCTION-NAME (FUNCTION-TYPE
      (PRIORITY-COMPARATOR-INFO (N> THE-SAFE-INSERT))))))

(Defrule ORDERED-ASSOC-LIST-INSERT-SAFE
  "Ordered Associative List Insert Safe"
  :RHS-Node-Types
  ((ENUMERATE-FRONT . ENUM-OAL-FRONT)
   (FIND-TAIL . FIND-OAL-TAIL)
   (DO-INSERT . OAL-SPLICE-IN))
  :Edge-List
  (((ENUMERATE-FRONT 3) . (DO-INSERT 1))
   ((FIND-TAIL 3) . (DO-INSERT 3)))
  :Input-Embedding
  (((ORDERED-ASSOC-LIST-INSERT-SAFE 1) (DO-INSERT 2))
   ((ORDERED-ASSOC-LIST-INSERT-SAFE 2) (FIND-TAIL 2))
   ((ORDERED-ASSOC-LIST-INSERT-SAFE 2) (ENUMERATE-FRONT 2))
   ((ORDERED-ASSOC-LIST-INSERT-SAFE 3) (FIND-TAIL 1))
   ((ORDERED-ASSOC-LIST-INSERT-SAFE 3) (ENUMERATE-FRONT 1)))
  :Output-Embedding
  (((ORDERED-ASSOC-LIST-INSERT-SAFE 4) (DO-INSERT 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("inserts ~A (associated w/ priority ~A) in the ordered ~
    associative list ~A.  An element X occurs before another Y ~
    if X's priority ~A Y's priority.~%~
    If an element already exists in the list with priority ~A, ~
    then the new element is inserted into the list after the ~
    existing element."
   (INPUT-PORT-NAME> (DOC-BP> (DO-INSERT 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-FRONT 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-FRONT 1)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-COMPARATOR-INFO
      (N> ORDERED-ASSOC-LIST-INSERT-SAFE))))
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-FRONT 2)))))

(Defrule ENUM-OAL-FRONT
  "Enumerate Ordered Associative List Front"
  :RHS-Node-Types
  ((CDR-DOWN . GENERATE)
   (HEAD-IN-FRONT? . TRUNCATE-OAL-POSITION)
   (THE-HEAD-MAP . CAR-MAP))
  :Edge-List
  (((CDR-DOWN 2) . (HEAD-IN-FRONT? 1))
   ((HEAD-IN-FRONT? 3) . (THE-HEAD-MAP 1)))
  :Input-Embedding
  (((ENUM-OAL-FRONT 1) (CDR-DOWN 1))
   ((ENUM-OAL-FRONT 2) (HEAD-IN-FRONT? 2)))
  :Output-Embedding
  (((ENUM-OAL-FRONT 3) (THE-HEAD-MAP 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the elements of the Ordered Associative list ~A ~
    up to, but not including, the element (if any) that has ~
    lower priority than ~A.  If there is no such element, all ~
    elements of the list are enumerated."
   (INPUT-PORT-NAME> (DOC-BP> (CDR-DOWN 1)))
   (INPUT-PORT-NAME> (DOC-BP> (HEAD-IN-FRONT? 2)))))

(Defrule FIND-OAL-TAIL
  "Find Ordered Associative List Tail"
  :RHS-Node-Types
  ((CDR-DOWN2 . GENERATE)
   (HEAD-OF-TAIL? . EARLIEST-OAL-POSITION))
  :Edge-List
  (((CDR-DOWN2 2) . (HEAD-OF-TAIL? 1)))
  :Input-Embedding
  (((FIND-OAL-TAIL 1) (CDR-DOWN2 1))
   ((FIND-OAL-TAIL 2) (HEAD-OF-TAIL? 2)))
  :Output-Embedding
  (((FIND-OAL-TAIL 3) (HEAD-OF-TAIL? 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("finds the tail of ~A (if any) whose head has lower priority ~
    than ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CDR-DOWN2 1)))
   (INPUT-PORT-NAME> (DOC-BP> (HEAD-OF-TAIL? 2)))))

(Defrule ENUM-OAL-FRONT-UNSAFE
  "Unsafe Enumerate Ordered Associative List Front"
  :RHS-Node-Types
  ((CDR-DOWN-FRONT . GENERATE)
   (HEAD-BELONG-IN-FRONT? . TRUNCATE-OAL-POSITION-UNSAFE)
   (EXTRACT-HEAD . CAR-MAP))
  :Edge-List
  (((CDR-DOWN-FRONT 2) . (EXTRACT-HEAD 1))
   ((CDR-DOWN-FRONT 2) . (HEAD-BELONG-IN-FRONT? 1)))
  :Input-Embedding
  (((ENUM-OAL-FRONT-UNSAFE 1) (CDR-DOWN-FRONT 1))
   ((ENUM-OAL-FRONT-UNSAFE 2) (HEAD-BELONG-IN-FRONT? 2)))
  :Output-Embedding
  (((ENUM-OAL-FRONT-UNSAFE 3) (EXTRACT-HEAD 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the elements of the Ordered Associative list ~A up to,~%~
    but not including, the element (if any) that has equal or lower ~
    priority than ~A.  If there is no such element, all elements of the ~
    list are enumerated.  Priority equality is tested using ~A and the ~
    priorities are ordered by ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CDR-DOWN-FRONT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (HEAD-BELONG-IN-FRONT? 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-EQUALITY-INFO (N> ENUM-OAL-FRONT-UNSAFE))))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-COMPARATOR-INFO (N> ENUM-OAL-FRONT-UNSAFE))))))

(Defrule FIND-OAL-TAIL-UNSAFE
  "Unsafe Find Ordered Associative List Tail"
  :RHS-Node-Types
  ((PREV-CURRENT-SUBLISTS . TRAILING-GENERATE)
   (THE-SAFE-EARLIEST . EARLIEST-OAL-POSITION)
   (THE-UNSAFE-EARLIEST . EARLIEST-EQUAL-PRIORITY-HEAD))
  :Edge-List
  (((PREV-CURRENT-SUBLISTS 2) . (THE-UNSAFE-EARLIEST 1))
   ((PREV-CURRENT-SUBLISTS 2) . (THE-SAFE-EARLIEST 1)))
  :Input-Embedding
  (((FIND-OAL-TAIL-UNSAFE 1) (PREV-CURRENT-SUBLISTS 1))
   ((FIND-OAL-TAIL-UNSAFE 2) (THE-UNSAFE-EARLIEST 2))
   ((FIND-OAL-TAIL-UNSAFE 2) (THE-SAFE-EARLIEST 2)))
  :Output-Embedding
  (((FIND-OAL-TAIL-UNSAFE 3) (PREV-CURRENT-SUBLISTS 3))
   ((FIND-OAL-TAIL-UNSAFE 3) (THE-SAFE-EARLIEST 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("finds the tail of ~A (if any) whose head has equal or lower priority ~
    than ~A.  Priority equality is tested using ~A and the priorities ~
    are ordered by ~A."
   (INPUT-PORT-NAME> (DOC-BP> (PREV-CURRENT-SUBLISTS 1)))
   (INPUT-PORT-NAME> (DOC-BP> (THE-SAFE-EARLIEST 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-EQUALITY-INFO (N> FIND-OAL-TAIL-UNSAFE))))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-COMPARATOR-INFO (N> FIND-OAL-TAIL-UNSAFE))))))

(Defrule ORDERED-ASSOC-LIST-DELETE
  "Ordered Associative List Delete"
  :RHS-Node-Types
  ((UNSAFE-FRONT-ENUMERATION . ENUM-OAL-FRONT-UNSAFE)
   (UNSAFE-TAIL-SEARCH . FIND-OAL-TAIL-UNSAFE)
   (CONS-UP-REMAINING . CONS-ACCUMULATE-UP-FROM-SUBLIST))
  :Edge-List
  (((UNSAFE-FRONT-ENUMERATION 3) . (CONS-UP-REMAINING 1))
   ((UNSAFE-TAIL-SEARCH 3) . (CONS-UP-REMAINING 2)))
  :Input-Embedding
  (((ORDERED-ASSOC-LIST-DELETE 2) (UNSAFE-TAIL-SEARCH 1))
   ((ORDERED-ASSOC-LIST-DELETE 2) (UNSAFE-FRONT-ENUMERATION 1))
   ((ORDERED-ASSOC-LIST-DELETE 1) (UNSAFE-TAIL-SEARCH 2))
   ((ORDERED-ASSOC-LIST-DELETE 1) (UNSAFE-FRONT-ENUMERATION 2)))
  :Output-Embedding
  (((ORDERED-ASSOC-LIST-DELETE 3) (CONS-UP-REMAINING 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("deletes the element associated w/ priority ~A from the ordered ~
    associative list ~A.~%~
    The predicate used to test for priority equality is ~A.~%~
    If there is more than 1 element with this priority, only the first ~
    is removed.  An element X occurs before another Y if X's priority ~
    ~A Y's priority."
   (INPUT-PORT-NAME> (DOC-BP> (UNSAFE-FRONT-ENUMERATION 2)))
   (INPUT-PORT-NAME> (DOC-BP> (UNSAFE-FRONT-ENUMERATION 1)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-EQUALITY-INFO (N> ORDERED-ASSOC-LIST-DELETE))))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-COMPARATOR-INFO (N> ORDERED-ASSOC-LIST-DELETE))))))

(Defrule ORDERED-ASSOC-LIST-INSERT-UNSAFE
  "Unsafe Ordered Associative List Insert"
  :RHS-Node-Types
  ((ENUMERATE-FRONT-UNSAFELY . ENUM-OAL-FRONT-UNSAFE)
   (FIND-TAIL-UNSAFELY . FIND-OAL-TAIL-UNSAFE)
   (THE-INSERTION . OAL-SPLICE-IN))
```

```
  :Edge-List
  (((ENUMERATE-FRONT-UNSAFELY 3) . (THE-INSERTION 1))
   ((FIND-TAIL-UNSAFELY 3) . (THE-INSERTION 3)))
  :Input-Embedding
  (((ORDERED-ASSOC-LIST-INSERT-UNSAFE 1) (THE-INSERTION 2))
   ((ORDERED-ASSOC-LIST-INSERT-UNSAFE 2) (FIND-TAIL-UNSAFELY 2))
   ((ORDERED-ASSOC-LIST-INSERT-UNSAFE 2)
    (ENUMERATE-FRONT-UNSAFELY 2))
   ((ORDERED-ASSOC-LIST-INSERT-UNSAFE 3)
    (FIND-TAIL-UNSAFELY 1))
   ((ORDERED-ASSOC-LIST-INSERT-UNSAFE 3)
    (ENUMERATE-FRONT-UNSAFELY 1)))
  :Output-Embedding
  (((ORDERED-ASSOC-LIST-INSERT-UNSAFE 4) (THE-INSERTION 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("inserts ~A (associated w/ priority ~A) in the ordered ~
    associative list ~A.  The insertion is unsafe in that if ~
    there is an existing element in the list that has priority ~
    ~A ~A, then that existing element is replaced by ~A.~%~
    An element X occurs before another Y if X's priority ~A Y's ~
    priority."
   (INPUT-PORT-NAME> (DOC-BP> (THE-INSERTION 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-FRONT-UNSAFELY 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-FRONT-UNSAFELY 1)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-EQUALITY-INFO
      (N> ORDERED-ASSOC-LIST-INSERT-UNSAFE))))
   (INPUT-PORT-NAME> (DOC-BP> (ENUMERATE-FRONT-UNSAFELY 2)))
   (INPUT-PORT-NAME> (DOC-BP> (THE-INSERTION 2)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-COMPARATOR-INFO
      (N> ORDERED-ASSOC-LIST-INSERT-UNSAFE))))))

(Defrule OAL-RETRIEVE-IF-EXISTS
  "Ordered-Associative List Retrieve (If Exists)"
  :RHS-Node-Types
  ((ENUM-OAL . ORDERED-ASSOC-LE)
   (EARLIEST-ELEMENT . EARLIEST-EQUAL-PRIORITY))
  :Edge-List
  (((ENUM-OAL 3) . (EARLIEST-ELEMENT 1)))
  :Input-Embedding
  (((OAL-RETRIEVE-IF-EXISTS 1) (EARLIEST-ELEMENT 2))
   ((OAL-RETRIEVE-IF-EXISTS 1) (ENUM-OAL 2))
   ((OAL-RETRIEVE-IF-EXISTS 2) (ENUM-OAL 1)))
  :Output-Embedding
  (((OAL-RETRIEVE-IF-EXISTS 4) (EARLIEST-ELEMENT 3)))
  :St-Thrus
  (((OAL-RETRIEVE-IF-EXISTS 3) (OAL-RETRIEVE-IF-EXISTS 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("intermediate non-terminal: Ordered-Assoc-List-Lookup."))

(Defrule ORDERED-ASSOC-LIST-LOOKUP
  "Ordered Associative List Lookup"
  :RHS-Node-Types
  ((THE-RETRIEVAL . OAL-RETRIEVE-IF-EXISTS))
  :Input-Embedding
  (((ORDERED-ASSOC-LIST-LOOKUP 1) (THE-RETRIEVAL 1))
   ((ORDERED-ASSOC-LIST-LOOKUP 2) (THE-RETRIEVAL 2)))
  :Output-Embedding
  (((ORDERED-ASSOC-LIST-LOOKUP 3) (THE-RETRIEVAL 4)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("finds and returns the element associated w/ priority ~A in ~
    the ordered associative list ~A.~%~
    If no element with priority ~A is found, NIL is returned.~%~
    The predicate used to test for priority equality is ~A.~%~
    If there is more than 1 element with this priority, only ~
    the first is retrieved.  An element X occurs before another ~
    Y if X's priority ~A Y's priority."
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-LOOKUP 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-LOOKUP 2)))
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LIST-LOOKUP 1)))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-EQUALITY-INFO (N> ORDERED-ASSOC-LIST-LOOKUP))))
   (FUNCTION-NAME (FUNCTION-TYPE
     (PRIORITY-COMPARATOR-INFO
      (N> ORDERED-ASSOC-LIST-LOOKUP))))))

(Defrule ORDERED-ASSOC-LE
  "Ordered Associative List Enumeration"
  :RHS-Node-Types
  ((THE-ORDERED-ASSOC-SLE . ORDERED-ASSOC-SLE)
   (EACH-ELEMENT . CAR-MAP))
  :Edge-List
  (((THE-ORDERED-ASSOC-SLE 3) . (EACH-ELEMENT 1)))
  :Input-Embedding
  (((ORDERED-ASSOC-LE 1) (THE-ORDERED-ASSOC-SLE 1))
   ((ORDERED-ASSOC-LE 2) (THE-ORDERED-ASSOC-SLE 2)))
  :Output-Embedding
  (((ORDERED-ASSOC-LE 3) (EACH-ELEMENT 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the elements of ~A, up to, but not including,~%~
      the element that has lower priority than ~A."
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-LE 2)))))

(Defrule ORDERED-ASSOC-SLE
  "Ordered Associative Sublist Enumeration"
  :RHS-Node-Types
  ((OAL-GENERATE . GENERATE)
   (OAL-TRUNCATE . TRUNCATE-OAL-POSITION))
  :Edge-List
  (((OAL-GENERATE 2) . (OAL-TRUNCATE 1)))
  :Input-Embedding
  (((ORDERED-ASSOC-SLE 1) (OAL-GENERATE 1))
   ((ORDERED-ASSOC-SLE 2) (OAL-TRUNCATE 2)))
  :Output-Embedding
  (((ORDERED-ASSOC-SLE 3) (OAL-TRUNCATE 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the successive sublists of ~A, up to, but not including,~%~
      the sublist with a head that has lower priority than ~A."
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-SLE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ORDERED-ASSOC-SLE 2)))))

(Defrule LIST-PUSH
  "List Push"
  :RHS-Node-Types
  ((THE-CONS . CONS))
  :Input-Embedding
  (((LIST-PUSH 1) (THE-CONS 1))
   ((LIST-PUSH 2) (THE-CONS 2)))
  :Output-Embedding
  (((LIST-PUSH 3) (THE-CONS 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("pushes ~A onto the list ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LIST-PUSH 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LIST-PUSH 2)))))

(Defrule OAL-SPLICE-OUT
  "Splice out of Ordered Associative List"
  :RHS-Node-Types
  ((POP-TAIL . CDR)
   (ADD-FRONT . CONS-ACCUMULATE-UP-FROM-SUBLIST))
  :Edge-List
  (((POP-TAIL 2) . (ADD-FRONT 2)))
  :Input-Embedding
  (((OAL-SPLICE-OUT 1) (ADD-FRONT 1))
   ((OAL-SPLICE-OUT 2) (POP-TAIL 1)))
  :Output-Embedding
  (((OAL-SPLICE-OUT 3) (ADD-FRONT 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("splices the head of the ~A out of the ordered associative list~%~
    that contains it as a tail."
   (INPUT-PORT-NAME> (DOC-BP> (POP-TAIL 1)))))

(Defrule OAL-SPLICE-IN
  "Ordered Associative List Splice In"
  :RHS-Node-Types
  ((PUSH-ONTO-TAIL . LIST-PUSH)
   (CONS-UP-FRONT . CONS-ACCUMULATE-UP-FROM-SUBLIST))
  :Edge-List
  (((PUSH-ONTO-TAIL 3) . (CONS-UP-FRONT 2)))
  :Input-Embedding
  (((OAL-SPLICE-IN 1) (CONS-UP-FRONT 1))
   ((OAL-SPLICE-IN 2) (PUSH-ONTO-TAIL 1))
   ((OAL-SPLICE-IN 3) (PUSH-ONTO-TAIL 2)))
  :Output-Embedding
  (((OAL-SPLICE-IN 4) (CONS-UP-FRONT 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("splices ~A in between the front of the list ~A and the tail ~A."
   (INPUT-PORT-NAME> (DOC-BP> (PUSH-ONTO-TAIL 1)))
   (INPUT-PORT-NAME> (DOC-BP> (CONS-UP-FRONT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (PUSH-ONTO-TAIL 2)))))

(Defrule TRUNCATE-OAL-POSITION-UNSAFE
  "Unsafe Truncate at Priority Position"
  :RHS-Node-Types
  ((THE-SAFE-TRUNCATE . TRUNCATE-OAL-POSITION)
   (THE-UNSAFE-TRUNCATE . TRUNCATE-EQUAL-PRIORITY-HEAD))
  :Edge-List
  (((THE-SAFE-TRUNCATE 3) . (THE-UNSAFE-TRUNCATE 1)))
  :Input-Embedding
  (((TRUNCATE-OAL-POSITION-UNSAFE 1) (THE-SAFE-TRUNCATE 1))
   ((TRUNCATE-OAL-POSITION-UNSAFE 2) (THE-UNSAFE-TRUNCATE 2))
   ((TRUNCATE-OAL-POSITION-UNSAFE 2) (THE-SAFE-TRUNCATE 2)))
  :Output-Embedding
  (((TRUNCATE-OAL-POSITION-UNSAFE 3) (THE-UNSAFE-TRUNCATE 3)))
  :L-R-Link COMPOSITION
  :Doc
```

```
   ("outputs the elements of the input series (each elt. is an ~
     ordered associative list), ~
     up to but not including the one that is empty or has a head ~
     with priority less than or equal to ~A.~
     A priority P is less than another Q if P ~A Q.~
     A priority P is equal to another Q if P ~A Q."
    (INPUT-PORT-NAME> (DOC-BP> (THE-SAFE-TRUNCATE 2)))
    (FUNCTION-NAME (FUNCTION-TYPE
      (PRIORITY-COMPARATOR-INFO (N> THE-SAFE-TRUNCATE))))
    (FUNCTION-NAME (FUNCTION-TYPE
      (PRIORITY-EQUALITY-INFO (N> THE-UNSAFE-TRUNCATE))))))

(Defrule TRUNCATE-EQUAL-PRIORITY-HEAD
  "Truncate Equal Priority Head"
  :RHS-Node-Types
  ((PH-EQUALITY-TEST . EQUAL-PRIORITY-HEAD))
  :Input-Embedding
  (((TRUNCATE-EQUAL-PRIORITY-HEAD 1) (PH-EQUALITY-TEST 1))
   ((TRUNCATE-EQUAL-PRIORITY-HEAD 2) (PH-EQUALITY-TEST 2)))
  :St-Thrus
  (((TRUNCATE-EQUAL-PRIORITY-HEAD 1)
    (TRUNCATE-EQUAL-PRIORITY-HEAD 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the elements of the input series (each elt. is an ~
    associative list), up to but not including the one that is ~
    empty or has a head with lower priority than ~A."
    (INPUT-PORT-NAME> (DOC-BP> (PH-EQUALITY-TEST 2)))))

(Defrule EARLIEST-EQUAL-PRIORITY-HEAD
  "Earliest Equal Priority Head"
  :RHS-Node-Types
  ((EQUAL-PH-SEARCH . EQUAL-PRIORITY-HEAD))
  :Input-Embedding
  (((EARLIEST-EQUAL-PRIORITY-HEAD 1) (EQUAL-PH-SEARCH 1))
   ((EARLIEST-EQUAL-PRIORITY-HEAD 2) (EQUAL-PH-SEARCH 2)))
  :St-Thrus
  (((EARLIEST-EQUAL-PRIORITY-HEAD 1)
    (EARLIEST-EQUAL-PRIORITY-HEAD 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the first element of the input series (each elt. is ~
    an ordered associative list), that has a head with ~
    priority ~A."
  (INPUT-PORT-NAME> (DOC-BP> (EARLIEST-EQUAL-PRIORITY-HEAD 2)))))

(Defrule EQUAL-PRIORITY-HEAD
  "Equal Priority Head"
  :RHS-Node-Types
  ((ACCESS-HEAD . CAR)
   (CHECK-PRIORITIES . EQUAL-PRIORITY-TEST))
  :Edge-List
  (((ACCESS-HEAD 2) . (CHECK-PRIORITIES 2)))
  :Input-Embedding
  (((EQUAL-PRIORITY-HEAD 1) (ACCESS-HEAD 1))
   ((EQUAL-PRIORITY-HEAD 2) (CHECK-PRIORITIES 1)))
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether the head of the input associative list ~A has ~
    priority ~A."
   (INPUT-PORT-NAME> (DOC-BP> (ACCESS-HEAD 1)))
   (INPUT-PORT-NAME> (DOC-BP> (CHECK-PRIORITIES 1)))))

(Defrule TRUNCATE-EQUAL-PRIORITY
  "Truncate Equal Priority"
  :RHS-Node-Types
  ((PRIORITY-EQUALITY-TEST . EQUAL-PRIORITY-TEST))
  :Input-Embedding
  (((TRUNCATE-EQUAL-PRIORITY 1) (PRIORITY-EQUALITY-TEST 2))
   ((TRUNCATE-EQUAL-PRIORITY 2) (PRIORITY-EQUALITY-TEST 1)))
  :St-Thrus
  (((TRUNCATE-EQUAL-PRIORITY 1) (TRUNCATE-EQUAL-PRIORITY 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the elements of the input series,~
    up to but not including the one that has lower priority ~
    than ~A."
   (INPUT-PORT-NAME> (DOC-BP> (PRIORITY-EQUALITY-TEST 1)))))

(Defrule TRUNCATE-EQUAL-PRIORITY
  "Truncate Equal Priority"
  :RHS-Node-Types
  ((PRIORITY-EQUALITY-TEST . EQUAL-PRIORITY-TEST))
  :Input-Embedding
  (((TRUNCATE-EQUAL-PRIORITY 1) (PRIORITY-EQUALITY-TEST 1))
   ((TRUNCATE-EQUAL-PRIORITY 2) (PRIORITY-EQUALITY-TEST 2)))
  :St-Thrus
  (((TRUNCATE-EQUAL-PRIORITY 1) (TRUNCATE-EQUAL-PRIORITY 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the elements of the input series, up to but not ~
    including the one that has lower priority than ~A."
   (INPUT-PORT-NAME> (DOC-BP> (PRIORITY-EQUALITY-TEST 2)))))

(Defrule EARLIEST-EQUAL-PRIORITY
  "Earliest Equal Priority"
  :RHS-Node-Types
  ((EQUAL-P-SEARCH . EQUAL-PRIORITY-TEST))
  :Input-Embedding
  (((EARLIEST-EQUAL-PRIORITY 1) (EQUAL-P-SEARCH 2))
   ((EARLIEST-EQUAL-PRIORITY 2) (EQUAL-P-SEARCH 1)))
  :St-Thrus
  (((EARLIEST-EQUAL-PRIORITY 1) (EARLIEST-EQUAL-PRIORITY 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the first element of the input series~
    ~&that has priority ~A."
   (INPUT-PORT-NAME> (DOC-BP> (EQUAL-P-SEARCH 1)))))

(Defrule EARLIEST-EQUAL-PRIORITY
  "Earliest Equal Priority"
  :RHS-Node-Types
  ((EQUAL-P-SEARCH . EQUAL-PRIORITY-TEST))
  :Input-Embedding
  (((EARLIEST-EQUAL-PRIORITY 1) (EQUAL-P-SEARCH 1))
   ((EARLIEST-EQUAL-PRIORITY 2) (EQUAL-P-SEARCH 2)))
  :St-Thrus
  (((EARLIEST-EQUAL-PRIORITY 1) (EARLIEST-EQUAL-PRIORITY 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the first element of the input series~
    ~&that has priority ~A."
   (INPUT-PORT-NAME> (DOC-BP> (EQUAL-P-SEARCH 2)))))

(Defrule EQUAL-PRIORITY-TEST
  "Equal Priority Test"
  :RHS-Node-Types
  ((EQUAL-PRIORITIES . COMMUTATIVE-BINARY-FUNCTION)
   (THE-TEST . NULL-TEST))
  :Edge-List
  (((EQUAL-PRIORITIES 3) . (THE-TEST 1)))
  :Input-Embedding
  (((EQUAL-PRIORITY-TEST 1) (EQUAL-PRIORITIES 1))
   ((EQUAL-PRIORITY-TEST 2) (EQUAL-PRIORITIES 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether ~A and ~A have ~A priorities."
   (INPUT-PORT-NAME> (DOC-BP> (EQUAL-PRIORITY-TEST 1)))
   (INPUT-PORT-NAME> (DOC-BP> (EQUAL-PRIORITY-TEST 2)))
   (EQUALITY-PREDICATE? (N> EQUAL-PRIORITY-TEST))))

(Defrule TRUNCATE-OAL-POSITION
  "Truncate at Priority Position"
  :RHS-Node-Types
  ((POSITION-TEST . EMPTY-OR-LOW-PRIORITY-HEAD))
  :Input-Embedding
  (((TRUNCATE-OAL-POSITION 1) (POSITION-TEST 1))
   ((TRUNCATE-OAL-POSITION 2) (POSITION-TEST 2)))
  :St-Thrus
  (((TRUNCATE-OAL-POSITION 1) (TRUNCATE-OAL-POSITION 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the elements of the input series (each elt. is an ~
    ordered associative list), ~
    ~&up to but not including the one that is empty or has a head ~
    ~&with lower priority than ~A."
   (INPUT-PORT-NAME> (DOC-BP> (POSITION-TEST 2)))))

(Defrule EARLIEST-OAL-POSITION
  "Earliest Priority Position"
  :RHS-Node-Types
  ((OAL-POSITION-SEARCH . EMPTY-OR-LOW-PRIORITY-HEAD))
  :Input-Embedding
  (((EARLIEST-OAL-POSITION 1) (OAL-POSITION-SEARCH 1))
   ((EARLIEST-OAL-POSITION 2) (OAL-POSITION-SEARCH 2)))
  :St-Thrus
  (((EARLIEST-OAL-POSITION 1) (EARLIEST-OAL-POSITION 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the first element of the input series (each elt. is an ~
    ordered associative list),~
    ~&that is either empty or has a head with lower priority than ~A."
   (INPUT-PORT-NAME> (DOC-BP> (EARLIEST-OAL-POSITION 2)))))

(Defrule EMPTY-OR-LOW-PRIORITY-HEAD
  "Empty or Low Priority Head"
  :RHS-Node-Types
  ((EMPTY? . NULL)
   (CONTROL-COMPARISON . NULL-TEST)
   (GET-HEAD . CAR)
   (COMPARE-PRIORITIES . ANY-COMPARATOR)
   (OR-TEST . NULL-TEST))
  :Edge-List
  (((EMPTY? 2) . (OR-TEST 1))
   ((EMPTY? 2) . (CONTROL-COMPARISON 1))
   ((GET-HEAD 2) . (COMPARE-PRIORITIES 2))
   ((COMPARE-PRIORITIES 3) . (OR-TEST 1)))
```

```
  :Input-Embedding
  (((EMPTY-OR-LOW-PRIORITY-HEAD 1)
    (GET-HEAD 1))
   ((EMPTY-OR-LOW-PRIORITY-HEAD 1) (EMPTY? 1))
   ((EMPTY-OR-LOW-PRIORITY-HEAD 2) (COMPARE-PRIORITIES 1)))
  :L-R-Link COMPOSITION
  :Doc
  ("tests whether the list ~A is either empty or has a first ~
    element that has a lower priority than ~A."
   (INPUT-PORT-NAME> (DOC-BP> (EMPTY-OR-LOW-PRIORITY-HEAD 1)))
   (INPUT-PORT-NAME> (DOC-BP> (EMPTY-OR-LOW-PRIORITY-HEAD 2)))))

(Defrule ORDERED-ASSOC-LIST-EXTRACT
  "Ordered Associative List Extract"
  :RHS-Node-Types
  ((THE-POP . LIST-POP))
  :Input-Embedding
  (((ORDERED-ASSOC-LIST-EXTRACT 1) (THE-POP 1)))
  :Output-Embedding
  (((ORDERED-ASSOC-LIST-EXTRACT 2) (THE-POP 2))
   ((ORDERED-ASSOC-LIST-EXTRACT 3) (THE-POP 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("extracts the highest priority element from the ordered ~
    associative list ~A by popping the first element."
   (INPUT-PORT-NAME> (DOC-BP> (THE-POP 1)))))

(Defrule LIST-POP
  "List Pop"
  :RHS-Node-Types
  ((PULL-OFF-HEAD . CAR)
   (GET-TAIL . CDR))
  :Input-Embedding
  (((LIST-POP 1) (GET-TAIL 1))
   ((LIST-POP 1) (PULL-OFF-HEAD 1)))
  :Output-Embedding
  (((LIST-POP 2) (PULL-OFF-HEAD 2))
   ((LIST-POP 3) (GET-TAIL 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("pops the first element off of the list ~A."
   (INPUT-PORT-NAME> (DOC-BP> (GET-TAIL 1)))))

(Defrule ACCUMULATION-UP
  "Accumulation Up"
  :RHS-Node-Types
  ((ACCUM-FUNCTION . ANY-BIN-F))
  :Input-Embedding
  (((ACCUMULATION-UP 2) (ACCUM-FUNCTION 1)))
  :Output-Embedding
  (((ACCUMULATION-UP 3) (ACCUM-FUNCTION 3)))
  :St-Thrus
  (((ACCUMULATION-UP 1) (ACCUMULATION-UP 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("iteratively applies the function ~A to the result of the ~
    recursive call and a new value.  The result of the application
~
    is returned as the result of the recursive call."
   (FUNCTION-TYPE (FUNCTION-INFO (N> ACCUM-FUNCTION)))))

(Defrule ACCUMULATE-UP
  "Accumulate on the way up"
  :RHS-Node-Types
  ((ITER-ACCUM-UP . ACCUMULATION-UP))
  :Input-Embedding
  (((ACCUMULATE-UP 1) (ITER-ACCUM-UP 1))
   ((ACCUMULATE-UP 2) (ITER-ACCUM-UP 2)))
  :Output-Embedding
  (((ACCUMULATE-UP 3) (ITER-ACCUM-UP 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("accumulates the values of the input series 'on the way up' ~
    using the function ~A.  The initial value of the accumulation ~
    is ~A."
   (FUNCTION-TYPE (FUNCTION-INFO (N> ITER-ACCUM-UP)))
   (INIT-VALUE (N> ITER-ACCUM-UP))))

(Defrule CONS-ACCUMULATE-UP
  "Cons Accumulate on the way up"
  :RHS-Node-Types
  ((THE-UP-ACCUM . ACCUMULATE-UP))
  :Input-Embedding
  (((CONS-ACCUMULATE-UP 1) (THE-UP-ACCUM 2)))
  :Output-Embedding
  (((CONS-ACCUMULATE-UP 2) (THE-UP-ACCUM 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("accumulates the elements of ~A into a list using cons."
   (INPUT-PORT-NAME> (DOC-BP> (CONS-ACCUMULATE-UP 1)))))

(Defrule CONS-ACCUMULATE-UP-FROM-SUBLIST
  "Cons Accumulate on the way up from Sublist"
  :RHS-Node-Types
  ((THE-UP-ACCUM . ACCUMULATE-UP))
  :Input-Embedding
  (((CONS-ACCUMULATE-UP-FROM-SUBLIST 1) (THE-UP-ACCUM 2))
   ((CONS-ACCUMULATE-UP-FROM-SUBLIST 2) (THE-UP-ACCUM 1)))
  :Output-Embedding
  (((CONS-ACCUMULATE-UP-FROM-SUBLIST 3) (THE-UP-ACCUM 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("accumulates the elements of ~A into a list whose tail is ~A."
   (INPUT-PORT-NAME> (DOC-BP> (CONS-ACCUMULATE-UP-FROM-SUBLIST 1)))
   (INPUT-PORT-NAME> (DOC-BP> (CONS-ACCUMULATE-UP-FROM-SUBLIST 2)))))

(Defrule LIST-EMPTY
  "List Empty"
  :RHS-Node-Types
  ((THE-NULL . TEST-PREDICATE))
  :Input-Embedding
  (((LIST-EMPTY 1) (THE-NULL 1)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("checks whether the list ~A is empty."
   (INPUT-PORT-NAME> (DOC-BP> (LIST-EMPTY 1)))))

;;; Figure 4-14.

(Defrule GENERATION
  "Generation"
  :RHS-Node-Types
  ((GEN-FUNCTION . ANY-GEN-F))
  :Input-Embedding
  (((GENERATION 1) (GEN-FUNCTION 1)))
  :St-Thrus
  (((GENERATION 1) (GENERATION 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("generates the successive elements of ~A by repeatedly applying the ~
    function ~A to the result of its preceding application."
   (INPUT-PORT-NAME> (DOC-BP> (GENERATION 1)))
   (FUNCTION-TYPE (FUNCTION-INFO (N> GEN-FUNCTION)))))

(Defrule GENERATE
  "Generate"
  :RHS-Node-Types
  ((THE-COUNT . COUNT))
  :Input-Embedding
  (((GENERATE 1) (THE-COUNT 1)))
  :Output-Embedding
  (((GENERATE 2) (THE-COUNT 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("generates the elements of ~A by counting them."
   (INPUT-PORT-NAME> (DOC-BP> (GENERATE 1)))))

(Defrule GENERATE
  "Generate"
  :RHS-Node-Types
  ((ITER-GEN . GENERATION))
  :Input-Embedding
  (((GENERATE 1) (ITER-GEN 1)))
  :Output-Embedding
  (((GENERATE 2) (ITER-GEN 2)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("generates a series of elements of ~A by repeatedly applying the ~
    function ~A."
   (INPUT-PORT-NAME> (DOC-BP> (GENERATE 1)))
   (FUNCTION-TYPE (FUNCTION-INFO (N> ITER-GEN)))))

(Defrule COMMUTATIVE-BINARY-FUNCTION
  "Commutative Binary Function"
  :RHS-Node-Types
  ((COMM-BIN-FUNCTION . ANY-COMM-BIN-F))
  :Input-Embedding
  (((COMMUTATIVE-BINARY-FUNCTION 1) (COMM-BIN-FUNCTION 2))
   ((COMMUTATIVE-BINARY-FUNCTION 2) (COMM-BIN-FUNCTION 1)))
  :Output-Embedding
  (((COMMUTATIVE-BINARY-FUNCTION 3) (COMM-BIN-FUNCTION 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("applies the commutative binary function ~A."
   (FUNCTION-TYPE (FUNCTION-INFO (N> COMM-BIN-FUNCTION)))))

(Defrule COMMUTATIVE-BINARY-FUNCTION
  "Commutative Binary Function"
  :RHS-Node-Types
  ((COMM-BIN-FUNCTION . ANY-COMM-BIN-F))
  :Input-Embedding
  (((COMMUTATIVE-BINARY-FUNCTION 1) (COMM-BIN-FUNCTION 1))
   ((COMMUTATIVE-BINARY-FUNCTION 2) (COMM-BIN-FUNCTION 2)))
  :Output-Embedding
  (((COMMUTATIVE-BINARY-FUNCTION 3) (COMM-BIN-FUNCTION 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
```

```
  ("applies the commutative binary function ~A."
   (FUNCTION-TYPE (FUNCTION-INFO (N> COMM-BIN-FUNCTION)))))

(Defrule INCREMENT
  "Increment"
  :RHS-Node-Types
  ((COMM-INC . COMMUTATIVE-BINARY-FUNCTION))
  :Input-Embedding
  (((INCREMENT 1) (COMM-INC 1)))
  :Output-Embedding
  (((INCREMENT 2) (COMM-INC 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("increments ~A by 1."
   (INPUT-PORT-NAME> (DOC-BP> (INCREMENT 1)))))

;;; Figure 4-5.

(Defrule COUNTING-UP
  "Counting Up"
  :RHS-Node-Types
  ((COUNTER . INCREMENT))
  :Input-Embedding
  (((COUNTING-UP 1) (COUNTER 1)))
  :St-Thrus
  (((COUNTING-UP 1) (COUNTING 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("repeatedly increments ~A by 1."
   (INPUT-PORT-NAME> (DOC-BP> (COUNTING-UP 1)))))

(Defrule COUNT
  "Count"
  :RHS-Node-Types
  ((ITER-COUNTING . COUNTING-UP))
  :Input-Embedding
  (((COUNT 1) (ITER-COUNTING 1)))
  :Output-Embedding
  (((COUNT 2) (ITER-COUNTING 2)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("generates a series of successive integers starting with ~A."
   (INPUT-PORT-NAME> (DOC-BP> (COUNT 1)))))

(Defrule BOUNDED-COUNT
  "Bounded Count"
  :RHS-Node-Types
  ((THE-COUNTER . COUNT)
   (STOP-AT-LIMIT . BINARY-TRUNCATE))
  :Edge-List
  (((THE-COUNTER 2) . (STOP-AT-LIMIT 1)))
  :Input-Embedding
  (((BOUNDED-COUNT 1) (THE-COUNTER 1))
   ((BOUNDED-COUNT 2) (STOP-AT-LIMIT 2)))
  :Output-Embedding
  (((BOUNDED-COUNT 3) (STOP-AT-LIMIT 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("generates a series of successive integers from ~A up to, but ~
    not including ~A."
   (INPUT-PORT-NAME> (DOC-BP> (BOUNDED-COUNT 1)))
   (INPUT-PORT-NAME> (DOC-BP> (BOUNDED-COUNT 2)))))

(Defrule DECREMENT
  "Decrement"
  :RHS-Node-Types
  ((SUBTRACT . MINUS))
  :Input-Embedding
  (((DECREMENT 1) (SUBTRACT 1)))
  :Output-Embedding
  (((DECREMENT 2) (SUBTRACT 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("decrements ~A by 1."
   (INPUT-PORT-NAME> (DOC-BP> (DECREMENT 1)))))

(Defrule INCREMENT-OR-DECREMENT
  "Increment or Decrement"
  :RHS-Node-Types
  ((DECREMENTER . DECREMENT))
  :Input-Embedding
  (((INCREMENT-OR-DECREMENT 1) (DECREMENTER 1)))
  :Output-Embedding
  (((INCREMENT-OR-DECREMENT 2) (DECREMENTER 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("Increments or decrements ~A."
   (INPUT-PORT-NAME (DOC-BP> (DECREMENTER 1)))))

(Defrule INCREMENT-OR-DECREMENT
  "Increment or Decrement"
  :RHS-Node-Types
  ((COUNTER . INCREMENT))
  :Input-Embedding
  (((INCREMENT-OR-DECREMENT 1) (COUNTER 1)))
  :Output-Embedding
  (((INCREMENT-OR-DECREMENT 2) (COUNTER 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("increments or decrements ~A."
   (INPUT-PORT-NAME (DOC-BP> (COUNTER 1)))))

(Defrule DOUBLE
  "Double"
  :RHS-Node-Types
  ((COMM-TIMES . COMMUTATIVE-BINARY-FUNCTION))
  :Input-Embedding
  (((DOUBLE 1) (COMM-TIMES 1)))
  :Output-Embedding
  (((DOUBLE 2) (COMM-TIMES 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("multiplies ~A by 2."
   (INPUT-PORT-NAME> (DOC-BP> (DOUBLE 1)))))

(Defrule CAR-MAP
  "Car Map"
  :RHS-Node-Types
  ((MAP-HEAD . CAR))
  :Input-Embedding
  (((CAR-MAP 1) (MAP-HEAD 1)))
  :Output-Embedding
  (((CAR-MAP 2) (MAP-HEAD 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("applies the function CAR to each element of the input series."))

(Defrule SELECT-TERM
  "Select Term"
  :RHS-Node-Types
  ((ACCESS-ARRAY . AREF))
  :Input-Embedding
  (((SELECT-TERM 1) (ACCESS-ARRAY 1)
    ARRAY>SEQUENCE)
   ((SELECT-TERM 2) (ACCESS-ARRAY 2)))
  :Output-Embedding
  (((SELECT-TERM 3) (ACCESS-ARRAY 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("selects the element at index ~A from the sequence ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SELECT-TERM 2)))
   (INPUT-PORT-NAME> (DOC-BP> (SELECT-TERM 1)))))

(Defrule SELECT-TERM-MAP
  "Select-Term Map"
  :RHS-Node-Types
  ((MAP-SEQUENCE-REF . SELECT-TERM))
  :Input-Embedding
  (((SELECT-TERM-MAP 1) (MAP-SEQUENCE-REF 1))
   ((SELECT-TERM-MAP 2) (MAP-SEQUENCE-REF 2)))
  :Output-Embedding
  (((SELECT-TERM-MAP 3) (MAP-SEQUENCE-REF 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("references the sequence ~A at each index in the input series ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SELECT-TERM-MAP 1)))
   (INPUT-PORT-NAME> (DOC-BP> (SELECT-TERM-MAP 2)))))

(Defrule FILTERING
  "Filtering"
  :RHS-Node-Types
  ((FILTER-PREDICATE . TEST-PREDICATE))
  :Input-Embedding
  (((FILTERING 1) (FILTER-PREDICATE 1)))
  :St-Thrus
  (((FILTERING 1) (FILTERING 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("repeatedly applies the predicate ~A to ~A."
   (FUNCTION-TYPE (PREDICATE-INFO (N> FILTER-PREDICATE)))
   (INPUT-PORT-NAME> (DOC-BP> (FILTER-PREDICATE 1)))))

(Defrule FILTER
  "Filter"
  :RHS-Node-Types
  ((FILTER-ELTS . FILTERING))
  :Input-Embedding
  (((FILTER 1) (FILTER-ELTS 1)))
  :Output-Embedding
  (((FILTER 2) (FILTER-ELTS 2)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("filters the elements of the input series using the predicate ~A."
   (FUNCTION-TYPE (PREDICATE-INFO (N> FILTER-ELTS)))))
```

```
(Defrule ACCUMULATION-DOWN
  "Accumulation Down"
  :RHS-Node-Types
  ((ACCUM-F . ANY-BIN-F))
  :Input-Embedding
  (((ACCUMULATION-DOWN 1) (ACCUM-F 1))
   ((ACCUMULATION-DOWN 2) (ACCUM-F 2)))
  :St-Thrus
  (((ACCUMULATION-DOWN 2) (ACCUMULATION-DOWN 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("repeatedly applies the function ~A to the result of its ~
    previous application and a new value.  When the iteration ~
    terminates, the result of the last application is returned."
   (FUNCTION-TYPE (FUNCTION-INFO (N> ACCUM-F)))))

(Defrule ACCUMULATE-DOWN
  "Accumulate Down"
  :RHS-Node-Types
  ((ITER-ACCUM . ACCUMULATION-DOWN))
  :Input-Embedding
  (((ACCUMULATE-DOWN 1) (ITER-ACCUM 1))
   ((ACCUMULATE-DOWN 2) (ITER-ACCUM 2)))
  :Output-Embedding
  (((ACCUMULATE-DOWN 3) (ITER-ACCUM 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("accumulates the values of the input series 'on the way down' ~
    using the function ~A."
   (FUNCTION-TYPE (FUNCTION-INFO (N> ITER-ACCUM)))))

(Defrule TRUNCATION
  "Truncation"
  :RHS-Node-Types
  ((STOP? . TEST-PREDICATE))
  :Input-Embedding
  (((TRUNCATION 1) (STOP? 1)))
  :St-Thrus
  (((TRUNCATION 1) (TRUNCATION 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("repeatedly applies the exit test ~A to a value, terminating ~
    the iteration if the test succeeds."
   (FUNCTION-TYPE (PREDICATE-INFO (N> STOP?)))))

(Defrule TRUNCATE
  "Truncate"
  :RHS-Node-Types
  ((ITER-TRUNCATION . TRUNCATION))
  :Input-Embedding
  (((TRUNCATE 1) (ITER-TRUNCATION 1)))
  :Output-Embedding
  (((TRUNCATE 2) (ITER-TRUNCATION 2)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the elements of the input series up to but not ~
    including the one that passes the predicate ~A."
   (FUNCTION-TYPE (PREDICATE-INFO (N> ITER-TRUNCATION)))))

(Defrule BINARY-TRUNCATION
  "Binary Truncation"
  :RHS-Node-Types
  ((BINARY-STOP? . BINARY-TEST-PREDICATE))
  :Input-Embedding
  (((BINARY-TRUNCATION 1) (BINARY-STOP? 1))
   ((BINARY-TRUNCATION 2) (BINARY-STOP? 2)))
  :St-Thrus
  (((BINARY-TRUNCATION 1) (BINARY-TRUNCATION 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("repeatedly applies the binary exit test ~A to a value, ~
    terminating the iteration if the test succeeds."
   (FUNCTION-TYPE (PREDICATE-INFO (N> BINARY-TRUNCATION)))))

(Defrule BINARY-TRUNCATE
  "Binary Truncate"
  :RHS-Node-Types
  ((ITER-BIN-TRUNCATION . BINARY-TRUNCATION))
  :Input-Embedding
  (((BINARY-TRUNCATE 1) (ITER-BIN-TRUNCATION 1))
   ((BINARY-TRUNCATE 2) (ITER-BIN-TRUNCATION 2)))
  :Output-Embedding
  (((BINARY-TRUNCATE 3) (ITER-BIN-TRUNCATION 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the elements of the input series up to but not ~
    including the one that passes the binary predicate ~A."
   (FUNCTION-TYPE (PREDICATE-INFO (N> BINARY-TRUNCATE)))))

(Defrule SLE
  "Sublist Enumeration"
  :RHS-Node-Types
  ((THE-GENERATE . GENERATE)
   (THE-TRUNCATE . TRUNCATE))
  :Edge-List
  (((THE-GENERATE 2) . (THE-TRUNCATE 1)))
  :Input-Embedding
  (((SLE 1) (THE-GENERATE 1)))
  :Output-Embedding
  (((SLE 2) (THE-TRUNCATE 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the successive sublists of ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SLE 1)))))

(Defrule LE
  "List Enumeration"
  :RHS-Node-Types
  ((THE-SLE . SLE)
   (THE-CAR-MAP . CAR-MAP))
  :Edge-List
  (((THE-SLE 2) . (THE-CAR-MAP 1)))
  :Input-Embedding
  (((LE 1) (THE-SLE 1)))
  :Output-Embedding
  (((LE 2) (THE-CAR-MAP 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the elements of ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LE 1)))))

;;; Figure 4-16.

(Defrule ITERATIVE-SEARCH
  "Iterative Search"
  :RHS-Node-Types
  ((SEARCH-P . TEST-PREDICATE))
  :Input-Embedding
  (((ITERATIVE-SEARCH 1) (SEARCH-P 1)))
  :St-Thrus
  (((ITERATIVE-SEARCH 1) (ITERATIVE-SEARCH 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("repeatedly applies the search predicate ~A to a value, ~
    terminating if an element is found that satisfies it."
   (FUNCTION-TYPE (PREDICATE-INFO (N> SEARCH-P)))))

;;; Figure 4-17.

(Defrule EARLIEST
  "Earliest"
  :RHS-Node-Types
  ((EARLIEST? . ITERATIVE-SEARCH))
  :Input-Embedding
  (((EARLIEST 1) (EARLIEST? 1)))
  :Output-Embedding
  (((EARLIEST 2) (EARLIEST? 2)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("outputs the first element of the input series which passes the ~
    predicate ~A."
   (FUNCTION-TYPE (PREDICATE-INFO (N> EARLIEST?)))))

(Defrule SEQUENTIAL-SEARCH
  "Sequential Search"
  :RHS-Node-Types
  ((EXIT . TEST-PREDICATE)
   (SEARCH . EARLIEST))
  :Input-Embedding
  (((SEQUENTIAL-SEARCH 1) (SEARCH 1)))
  :Output-Embedding
  (((SEQUENTIAL-SEARCH 2) (SEARCH 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("finds the first element of ~A satisfying the predicate ~A,~
    unless ~A is satisfied first."
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENTIAL-SEARCH 1)))
   (FUNCTION-TYPE (PREDICATE-INFO (N> SEARCH)))
   (FUNCTION-TYPE (PREDICATE-INFO (N> EXIT)))))

(Defrule SEQ-LIST-SEARCH
  "Sequential List Search"
  :RHS-Node-Types
  ((LIST-ENUM . LE)
   (SEQ-SEARCH . SEQUENTIAL-SEARCH))
  :Edge-List
  (((LIST-ENUM 2) . (SEQ-SEARCH 1)))
  :Input-Embedding
  (((SEQ-LIST-SEARCH 1) (LIST-ENUM 1)))
  :Output-Embedding
  (((SEQ-LIST-SEARCH 2) (SEQ-SEARCH 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("sequentially searches the elements of the list ~A until either the ~
    list is exhausted or an element is found that satisfies the test ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SEQ-LIST-SEARCH 1)))
```

```
   (FUNCTION-TYPE (PREDICATE-INFO (N> SEQ-SEARCH)))))

(Defrule CONS-ACCUMULATE-DOWN
  "Cons Accumulate on the way down"
  :RHS-Node-Types
  ((THE-ACCUM . ACCUMULATE-DOWN))
  :Input-Embedding
  (((CONS-ACCUMULATE-DOWN 1) (THE-ACCUM 1)))
  :Output-Embedding
  (((CONS-ACCUMULATE-DOWN 2) (THE-ACCUM 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("accumulates the elements of the input series ~A into a list ~
    using cons."
   (INPUT-PORT-NAME> (DOC-BP> (CONS-ACCUMULATE-DOWN 1)))))

(Defrule REVERSE-LIST
  "Reverse List"
  :RHS-Node-Types
  ((ENUMERATE-LIST . LE)
   (ACCUM-LIST . CONS-ACCUMULATE-DOWN))
  :Edge-List
  (((ENUMERATE-LIST 2) . (ACCUM-LIST 1)))
  :Input-Embedding
  (((REVERSE-LIST 1) (ENUMERATE-LIST 1)))
  :Output-Embedding
  (((REVERSE-LIST 2) (ACCUM-LIST 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("constructs a list containing the elements of ~A in reverse."
   (INPUT-PORT-NAME> (DOC-BP> (REVERSE-LIST 1)))))

(Defrule TRAILING-GENERATION
  "Trailing Generation"
  :RHS-Node-Types
  ((TR-GEN-FUNCTION . ANY-GEN-F))
  :Input-Embedding
  (((TRAILING-GENERATION 1) (TR-GEN-FUNCTION 1)))
  :Output-Embedding
  (((TRAILING-GENERATION 3) (TR-GEN-FUNCTION 2)))
  :St-Thrus
  (((TRAILING-GENERATION 1) (TRAILING-GENERATION 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("generates the successive previous and current elements of ~A ~
    by repeatedly applying the function ~A to the result of ~
    the preceding application of that function."
   (INPUT-PORT-NAME> (DOC-BP> (TRAILING-GENERATION 1)))
   (FUNCTION-TYPE (FUNCTION-INFO (N> TR-GEN-FUNCTION)))))

(Defrule TRAILING-GENERATE
  "Trailing Generate"
  :RHS-Node-Types
  ((ITER-TRAILING-GEN . TRAILING-GENERATION))
  :Input-Embedding
  (((TRAILING-GENERATE 1) (ITER-TRAILING-GEN 1)))
  :Output-Embedding
  (((TRAILING-GENERATE 2) (ITER-TRAILING-GEN 2))
   ((TRAILING-GENERATE 3) (ITER-TRAILING-GEN 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("generates a series of the elements of ~A and a series of the ~
    elements immediately preceding each of the elements in that ~
    series."
   (INPUT-PORT-NAME> (DOC-BP> (TRAILING-GENERATE 1)))))

(Defrule TRAILING-PTR-LE
  "Trailing Pointer List Enumeration"
  :RHS-Node-Types
  ((TR-GEN . TRAILING-GENERATE)
   (PREVIOUS-CAR-MAP . CAR-MAP)
   (CURRENT-CAR-MAP . CAR-MAP)
   (NULL-TRUNC . TRUNCATE))
  :Edge-List
  (((TR-GEN 3) . (CURRENT-CAR-MAP 1))
   ((TR-GEN 3) . (NULL-TRUNC 1))
   ((TR-GEN 2) . (PREVIOUS-CAR-MAP 1)))
  :Input-Embedding
  (((TRAILING-PTR-LE 1) (TR-GEN 1)))
  :Output-Embedding
  (((TRAILING-PTR-LE 2) (PREVIOUS-CAR-MAP 2))
   ((TRAILING-PTR-LE 3) (CURRENT-CAR-MAP 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the elements of the list ~A, along with their ~
    immediately preceding elements."
   (INPUT-PORT-NAME> (DOC-BP> (TRAILING-PTR-LE 1)))))

(Defrule NEW-SEQUENCE
  "New Sequence"
  :RHS-Node-Types
  ((MAKE-SEQ . MAKE-ARRAY))
  :Input-Embedding
  (((NEW-SEQUENCE 1) (MAKE-SEQ 1)))
  :Output-Embedding
  (((NEW-SEQUENCE 2) (MAKE-SEQ 2)
    ARRAY>SEQUENCE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("creates a new sequence of size ~A."
   (INPUT-PORT-NAME> (DOC-BP> (NEW-SEQUENCE 1)))))

(Defrule SEQUENCE-SIZE
  "Sequence Size"
  :RHS-Node-Types
  ((MEASURE-SEQUENCE . ARRAY-TOTAL-SIZE))
  :Input-Embedding
  (((SEQUENCE-SIZE 1) (MEASURE-SEQUENCE 1)
    ARRAY>SEQUENCE))
  :Output-Embedding
  (((SEQUENCE-SIZE 2) (MEASURE-SEQUENCE 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("computes the size of the sequence ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENCE-SIZE 1)))))

(Defrule NEW-TERM
  "New Term"
  :RHS-Node-Types
  ((THE-CR . COPY-REPLACE-ELT))
  :Input-Embedding
  (((NEW-TERM 1) (THE-CR 3)
    ARRAY>SEQUENCE)
   ((NEW-TERM 2) (THE-CR 2))
   ((NEW-TERM 3) (THE-CR 1)))
  :Output-Embedding
  (((NEW-TERM 4) (THE-CR 4)
    ARRAY>SEQUENCE))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("creates a new sequence with the same elements as the input sequence ~
    ~A at the same locations, except that the element ~A is at the ~
    index ~A."
   (INPUT-PORT-NAME> (DOC-BP> (NEW-TERM 1)))
   (INPUT-PORT-NAME> (DOC-BP> (NEW-TERM 3)))
   (INPUT-PORT-NAME> (DOC-BP> (NEW-TERM 2)))))

(Defrule SEQUENCE-ACCUMULATION
  "Sequence Accumulation"
  :RHS-Node-Types
  ((THE-NT . NEW-TERM))
  :Input-Embedding
  (((SEQUENCE-ACCUMULATION 1) (THE-NT 3))
   ((SEQUENCE-ACCUMULATION 2) (THE-NT 2))
   ((SEQUENCE-ACCUMULATION 3) (THE-NT 1)))
  :St-Thrus
  (((SEQUENCE-ACCUMULATION 3) (SEQUENCE-ACCUMULATION 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("repeatedly inserts an element ~A (a new element on each iteration) ~
    in the sequence ~A at the location ~A (which is a different index on ~
    each iteration).  When the iteration terminates, the sequence ~
    resulting from the last insertion is returned."
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENCE-ACCUMULATION 1)))
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENCE-ACCUMULATION 3)))
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENCE-ACCUMULATION 2)))))

(Defrule SEQUENCE-ACCUMULATE
  "Sequence Accumulate"
  :RHS-Node-Types
  ((ARRAY-ACCUM . SEQUENCE-ACCUMULATION))
  :Input-Embedding
  (((SEQUENCE-ACCUMULATE 1) (ARRAY-ACCUM 1))
   ((SEQUENCE-ACCUMULATE 2) (ARRAY-ACCUM 2))
   ((SEQUENCE-ACCUMULATE 3) (ARRAY-ACCUM 3)))
  :Output-Embedding
  (((SEQUENCE-ACCUMULATE 4) (ARRAY-ACCUM 4)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("accumulates the values of the input series ~A into a sequence ~A at the ~
    series of indices ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENCE-ACCUMULATE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENCE-ACCUMULATE 3)))
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENCE-ACCUMULATE 2)))))

(Defrule SEQUENCE-ENUMERATION
  "Sequence Enumeration"
  :RHS-Node-Types
  ((GENERATE-INDICES . BOUNDED-COUNT)
   (COMPUTE-INDEX-LIMIT . SEQUENCE-SIZE)
   (ACCESS-SEQUENCE . SELECT-TERM-MAP))
  :Edge-List
  (((GENERATE-INDICES 3) . (ACCESS-SEQUENCE 2))
   ((COMPUTE-INDEX-LIMIT 2) . (GENERATE-INDICES 2)))
  :Input-Embedding
  (((SEQUENCE-ENUMERATION 1) (ACCESS-SEQUENCE 1))
   ((SEQUENCE-ENUMERATION 1) (COMPUTE-INDEX-LIMIT 1)))
```

```
  :Output-Embedding
  (((SEQUENCE-ENUMERATION 2) (ACCESS-SEQUENCE 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the elements of the sequence ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SEQUENCE-ENUMERATION 1)))))

(Defrule SEQUENCE-AND-INDEX-ENUMERATION
  "Sequence and Index Enumeration"
  :RHS-Node-Types
  ((GENERATE-INDICES . BOUNDED-COUNT)
   (COMPUTE-INDEX-LIMIT . SEQUENCE-SIZE)
   (ACCESS-SEQUENCE . SELECT-TERM-MAP))
  :Edge-List
  (((GENERATE-INDICES 3) . (ACCESS-SEQUENCE 2))
   ((COMPUTE-INDEX-LIMIT 2) . (GENERATE-INDICES 2)))
  :Input-Embedding
  (((SEQUENCE-AND-INDEX-ENUMERATION 1) (ACCESS-SEQUENCE 1))
   ((SEQUENCE-AND-INDEX-ENUMERATION 1) (COMPUTE-INDEX-LIMIT 1)))
  :Output-Embedding
  (((SEQUENCE-AND-INDEX-ENUMERATION 2) (ACCESS-SEQUENCE 3))
   ((SEQUENCE-AND-INDEX-ENUMERATION 3) (GENERATE-INDICES 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("enumerates the elements of the sequence ~A and their indices."
   (INPUT-PORT-NAME>
     (DOC-BP> (SEQUENCE-AND-INDEX-ENUMERATION 1)))))

(Defrule LIST-TO-SEQUENCE
  "Transfer List to Sequence"
  :RHS-Node-Types
  ((ENUMERATE-LIST-ELTS . LE)
   (NEW-BASE . NEW-SEQUENCE)
   (COUNT-INDICES . COUNT)
   (ACCUMULATE-SEQUENCE . SEQUENCE-ACCUMULATE))
  :Edge-List
  (((ENUMERATE-LIST-ELTS 2) . (ACCUMULATE-SEQUENCE 1))
   ((NEW-BASE 2) . (ACCUMULATE-SEQUENCE 3))
   ((COUNT-INDICES 2) . (ACCUMULATE-SEQUENCE 2)))
  :Input-Embedding
  (((LIST-TO-SEQUENCE 1) (ENUMERATE-LIST-ELTS 1))
   ((LIST-TO-SEQUENCE 2) (NEW-BASE 1)))
  :Output-Embedding
  (((LIST-TO-SEQUENCE 3) (ACCUMULATE-SEQUENCE 4)))
  :L-R-Link COMPOSITION
  :Doc
  ("transfers the elements in the list ~A into a sequence~%~
    of size ~A, by enumerating the elements of the list ~%~
    and accumulating them in the sequence at successive indices,~%~
    starting with index ~A."
   (INPUT-PORT-NAME> (DOC-BP> (LIST-TO-SEQUENCE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (LIST-TO-SEQUENCE 2)))
   (INPUT-PORT-NAME> (DOC-BP> (COUNT-INDICES 1)))))

(Defrule UNARY-PREDICATE
  "Unary Predicate"
  :RHS-Node-Types
  ((ANY-PRED . ANY-P))
  :Input-Embedding
  (((UNARY-PREDICATE 1) (ANY-PRED 1)))
  :Output-Embedding
  (((UNARY-PREDICATE 2) (ANY-PRED 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("applies the unary predicate ~A to ~A."
   (FUNCTION-TYPE (FUNCTION-INFO (N> ANY-PRED)))
   (INPUT-PORT-NAME> (DOC-BP> (ANY-PRED 1)))))

(Defrule TEST-PREDICATE
  "Test Predicate"
  :RHS-Node-Types
  ((TP-UNARY-P . UNARY-PREDICATE)
   (CHECK-IT . NULL-TEST))
  :Edge-List
  (((TP-UNARY-P 2) . (CHECK-IT 1)))
  :Input-Embedding
  (((TEST-PREDICATE 1) (TP-UNARY-P 1)))
  :L-R-Link COMPOSITION
  :Doc
  ("tests ~A using the unary predicate ~A."
   (INPUT-PORT-NAME> (DOC-BP> (TEST-PREDICATE 1)))
   (FUNCTION-TYPE (FUNCTION-INFO (N> CHECK-IT)))))

(Defrule BINARY-PREDICATE
  "Binary Predicate"
  :RHS-Node-Types
  ((ANY-BIN-PRED . ANY-BINARY-P))
  :Input-Embedding
  (((BINARY-PREDICATE 1) (ANY-BIN-PRED 1))
   ((BINARY-PREDICATE 2) (ANY-BIN-PRED 2)))
  :Output-Embedding
  (((BINARY-PREDICATE 3) (ANY-BIN-PRED 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("applies the binary predicate ~A to ~A and ~A."
   (FUNCTION-TYPE (FUNCTION-INFO (N> ANY-BIN-PRED)))
   (INPUT-PORT-NAME> (DOC-BP> (ANY-BIN-PRED 1)))
   (INPUT-PORT-NAME> (DOC-BP> (ANY-BIN-PRED 2)))))

(Defrule BINARY-TEST-PREDICATE
  "Binary Test Predicate"
  :RHS-Node-Types
  ((TP-BINARY-P . BINARY-PREDICATE)
   (NULL-CHECK . NULL-TEST))
  :Edge-List
  (((TP-BINARY-P 3) . (NULL-CHECK 1)))
  :Input-Embedding
  (((BINARY-TEST-PREDICATE 1) (TP-BINARY-P 1))
   ((BINARY-TEST-PREDICATE 2) (TP-BINARY-P 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("tests ~A and ~A using the binary predicate ~A."
   (INPUT-PORT-NAME> (DOC-BP> (BINARY-TEST-PREDICATE 1)))
   (INPUT-PORT-NAME> (DOC-BP> (BINARY-TEST-PREDICATE 2)))
   (FUNCTION-TYPE (FUNCTION-INFO (N> NULL-CHECK)))))

(Defrule SUMMING
  "Summing"
  :RHS-Node-Types
  ((THE-TALLY . COMMUTATIVE-BINARY-FUNCTION))
  :Input-Embedding
  (((SUMMING 1) (THE-TALLY 1))
   ((SUMMING 2) (THE-TALLY 2)))
  :St-Thrus
  (((SUMMING 2) (SUMMING 3)))
  :L-R-Link COMPOSITION
  :Doc
  ("keeps a running total of the numbers ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SUMMING 1)))))

(Defrule SUM
  "Sum"
  :RHS-Node-Types
  ((TALLYING . SUMMING))
  :Input-Embedding
  (((SUM 1) (TALLYING 1)))
  :Output-Embedding
  (((SUM 2) (TALLYING 3)))
  :L-R-Link TEMPORAL-ABSTRACTION
  :Doc
  ("returns the sum of the numbers in the input series ~A."
   (INPUT-PORT-NAME> (DOC-BP> (SUM 1)))))

(Defrule MAX
  "Maximum"
  :RHS-Node-Types
  ((COMPUTE-MAX . BINARY-TEST-PREDICATE))
  :Input-Embedding
  (((MAX 1) (COMPUTE-MAX 1))
   ((MAX 2) (COMPUTE-MAX 2)))
  :St-Thrus
  (((MAX 2) (MAX 3))
   ((MAX 1) (MAX 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("computes the maximum of ~A and ~A."
   (INPUT-PORT-NAME> (DOC-BP> (MAX 1)))
   (INPUT-PORT-NAME> (DOC-BP> (MAX 2)))))

(Defrule MIN
  "Minimum"
  :RHS-Node-Types
  ((COMPUTE-MIN . BINARY-TEST-PREDICATE))
  :Input-Embedding
  (((MIN 1) (COMPUTE-MIN 1))
   ((MIN 2) (COMPUTE-MIN 2)))
  :St-Thrus
  (((MIN 2) (MIN 3))
   ((MIN 1) (MIN 3)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("computes the minimum of ~A and ~A."
   (INPUT-PORT-NAME> (DOC-BP> (MAX 1)))
   (INPUT-PORT-NAME> (DOC-BP> (MAX 2)))))

;;; Figure 3-9.

(Defrule SQUARE-ROOT-OF-SQUARE
  "Square-Root of Square"
  :RHS-Node-Types
  ((SQ . SQUARE)
   (TAKE-ROOT . SQRT))
  :Edge-List
  (((SQ 2) . (TAKE-ROOT 1)))
  :Input-Embedding
  (((SQUARE-ROOT-OF-SQUARE 1) (SQ 1)))
  :Output-Embedding
```

```
  (((SQUARE-ROOT-OF-SQUARE 2) (TAKE-ROOT 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("computes the square root of the square of ~A"
   (INPUT-PORT-NAME> (DOC-BP> (SQUARE-ROOT-OF-SQUARE 1)))))

;;; Figures 3-9, 4-4.

(Defrule NEGATE-IF-NEGATIVE
  "Negate if Negative"
  :RHS-Node-Types
  ((NEGATIVE? . LT)
   (CONTROL-NEGATION . NULL-TEST)
   (THE-NEGATE . NEGATE))
  :Edge-List
  (((NEGATIVE? 3) . (CONTROL-NEGATION 1)))
  :Input-Embedding
  (((NEGATE-IF-NEGATIVE 1) (THE-NEGATE 1))
   ((NEGATE-IF-NEGATIVE 1) (NEGATIVE? 1)))
  :Output-Embedding
  (((NEGATE-IF-NEGATIVE 2) (THE-NEGATE 2)))
  :St-Thrus
  (((NEGATE-IF-NEGATIVE 1) (NEGATE-IF-NEGATIVE 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("negates ~A if its negative."
   (INPUT-PORT-NAME> (DOC-BP> (NEGATE-IF-NEGATIVE 1)))))

;;; Figure 3-9.

(Defrule ABSOLUTE-VALUE
  "Absolute Value"
  :RHS-Node-Types
  ((SQRT-OF-SQ . SQUARE-ROOT-OF-SQUARE))
  :Input-Embedding
  (((ABSOLUTE-VALUE 1) (SQRT-OF-SQ 1)))
  :Output-Embedding
  (((ABSOLUTE-VALUE 2) (SQRT-OF-SQ 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("computes the absolute value of ~A by taking the square root of
~
    its square."
   (INPUT-PORT-NAME> (DOC-BP> (ABSOLUTE-VALUE 1)))))

;;; Figure 3-9.

(Defrule ABSOLUTE-VALUE
  "Absolute Value"
  :RHS-Node-Types
  ((NIN . NEGATE-IF-NEGATIVE))
  :Input-Embedding
  (((ABSOLUTE-VALUE 1) (NIN 1)))
  :Output-Embedding
  (((ABSOLUTE-VALUE 2) (NIN 2)))
  :L-R-Link IMPLEMENTATION
  :Doc
  ("computes the absolute value of ~A by negating it if it is ~
    negative."
   (INPUT-PORT-NAME> (DOC-BP> (ABSOLUTE-VALUE 1)))))

;;; Figure 3-9.

(Defrule EQUALITY-WITHIN-EPSILON
  "Equality Within an Epsilon"
  :RHS-Node-Types
  ((DIFF . MINUS)
   (TAKE-ABS . ABSOLUTE-VALUE)
   (WITHIN-EPSILON . LTE)
   (TEST-EWE . NULL-TEST))
  :Edge-List
  (((DIFF 3) . (ABSOLUTE-VALUE 1))
   ((WITHIN-EPSILON 3) . (TEST-EWE 1)))
  :Input-Embedding
  (((EQUALITY-WITHIN-EPSILON 1) (DIFF 1))
   ((EQUALITY-WITHIN-EPSILON 2) (DIFF 2)))
  :L-R-Link COMPOSITION
  :Doc
  ("determines whether ~A and ~A are within an epsilon ~A of each ~
    other."
   (INPUT-PORT-NAME> (DOC-BP> (EQUALITY-WITHIN-EPSILON 1)))
   (INPUT-PORT-NAME> (DOC-BP> (EQUALITY-WITHIN-EPSILON 2)))
   (INPUT-PORT-NAME> (DOC-BP> (EQUALITY-WITHIN-EPSILON 3)))))
```

Index of Non-Terminal Node Types

314 (ABSOLUTE-VALUE 1:INTEGER 2:INTEGER)
311 (ACCUMULATE-DOWN 1:SERIES 2:ANY 3:ANY)
309 (ACCUMULATE-UP 1:SERIES 2:ANY 3:ANY)
311 (ACCUMULATION-DOWN 1:ANY 2:ANY 3:ANY)
309 (ACCUMULATION-UP 1:ANY 2:ANY 3:ANY)
293 (ADVANCE-NODES 1:SEQUENCE 2:SEQUENCE 3:QUEUE)
304 (ASSOCIATIVE-LIST-DELETE 1:ANY 2:ASSOCIATIVE-LIST
3:ASSOCIATIVE-LIST)
304 (ASSOCIATIVE-LIST-INSERT 1:ANY 2:ANY 3:ASSOCIATIVE-LIST
4:ASSOCIATIVE-LIST)
304 (ASSOCIATIVE-LIST-LOOKUP 1:ANY 2:ASSOCIATIVE-LIST 3:ANY)
301 (ASSOCIATIVE-SET-ADD 1:ANY 2:ANY 3:ASSOCIATIVE-SET
4:ASSOCIATIVE-SET)
302 (ASSOCIATIVE-SET-LOOKUP 1:ANY 2:ASSOCIATIVE-SET 3:ANY)
302 (ASSOCIATIVE-SET-REMOVE 1:ANY 2:ASSOCIATIVE-SET
3:ASSOCIATIVE-SET)
294 (AVERAGE-LOCAL-BUFFER-SIZE 1:SEQUENCE 2:INTEGER)
313 (BINARY-PREDICATE 1:ANY 2:ANY 3:ANY)
313 (BINARY-TEST-PREDICATE 1:ANY 2:ANY)
311 (BINARY-TRUNCATE 1:SERIES 2:ANY 3:SERIES)
311 (BINARY-TRUNCATION 1:ANY 2:ANY 3:ANY)
297 (BOUNDED-CIS-ENUMERATION 1:CIRCULAR-INDEXED-SEQUENCE
2:INTEGER 3:INTEGER 4:INTEGER
5:SERIES)
310 (BOUNDED-COUNT 1:INTEGER 2:INTEGER 3:SERIES)
301 (BUMP+UPDATE 1:ANY 2:INDEXED-SEQUENCE 3:INDEXED-SEQUENCE)
310 (CAR-MAP 1:SERIES 2:SERIES)
303 (CHAINING-HT-DELETE 1:ANY 2:HASH-TABLE 3:HASH-TABLE)
303 (CHAINING-HT-FILL-COUNT-DELETE 1:ANY 2:HASH-TABLE
3:HASH-TABLE)
304 (CHAINING-HT-FILL-COUNT-INSERT 1:ANY 2:ANY 3:HASH-TABLE
4:HASH-TABLE)
303 (CHAINING-HT-INSERT 1:ANY 2:ANY 3:HASH-TABLE 4:HASH-TABLE)
302 (CHAINING-HT-LOOKUP 1:ANY 2:HASH-TABLE 3:ANY)
297 (CIRCULAR-INDEXED-SEQUENCE-ENUMERATION
1:CIRCULAR-INDEXED-SEQUENCE 2:SERIES)
297 (CIS-ADD 1:ANY 2:CIRCULAR-INDEXED-SEQUENCE
3:CIRCULAR-INDEXED-SEQUENCE)
296 (CIS-DESTRUCTIVE-ENUMERATION 1:CIRCULAR-INDEXED-SEQUENCE
2:SERIES)
296 (CIS-EMPTY 1:CIRCULAR-INDEXED-SEQUENCE)
298 (CIS-EXTRACT 1:CIRCULAR-INDEXED-SEQUENCE 2:ANY
3:CIRCULAR-INDEXED-SEQUENCE)
296 (CIS-FULL 1:CIRCULAR-INDEXED-SEQUENCE)
291 (CO-EARLIEST-EDS-FINISHED 1:SERIES 2:SERIES 3:SEQUENCE)
291 (CO-ITERATIVE-EDS-FINISHED 1:PRIORITY-QUEUE 2:SEQUENCE 3:ANY)
297 (COMBINATION-FUNCTION 1:INTEGER 2:INTEGER 3:INTEGER)
309 (COMMUTATIVE-BINARY-FUNCTION 1:ANY 2:ANY 3:ANY)
312 (CONS-ACCUMULATE-DOWN 1:SERIES 2:LINKED-LIST)
309 (CONS-ACCUMULATE-UP 1:SERIES 2:LINKED-LIST)
309 (CONS-ACCUMULATE-UP-FROM-SUBLIST 1:SERIES 2:LINKED-LIST
3:LINKED-LIST)
310 (COUNT 1:INTEGER 2:SERIES)
310 (COUNTING-UP 1:INTEGER 2:INTEGER)
310 (DECREMENT 1:INTEGER 2:INTEGER)
292 (DELIVER-MESSAGE 1:MESSAGE 2:SEQUENCE 3:SEQUENCE)
292 (DELIVER-MESSAGE-ACCUMULATE 1:SERIES 2:SEQUENCE 3:SEQUENCE)
292 (DELIVER-MESSAGES 1:QUEUE 2:SEQUENCE 3:SEQUENCE)
294 (DELIVER-MESSAGES-AND-STEP-NODES 1:SEQUENCE 2:QUEUE
3:SEQUENCE 4:QUEUE)
291 (DEQUEUE-AND-PROCESS-GENERATION 1:PRIORITY-QUEUE 2:SEQUENCE
3:PRIORITY-QUEUE 4:SEQUENCE)
294 (DESTRUCTIVE-QUEUE-ENUMERATION 1:QUEUE 2:SERIES)
293 (DO-WORK-ACCUMULATE 1:SERIES 2:INTEGER 3:SEQUENCE 4:QUEUE
5:SEQUENCE 6:QUEUE)
293 (DO-WORK-ACCUMULATION 1:SYNCH-NODE 2:INTEGER 3:SEQUENCE
4:QUEUE 5:SEQUENCE 6:QUEUE)
310 (DOUBLE 1:INTEGER 2:INTEGER)
311 (EARLIEST 1:SERIES 2:ANY)
308 (EARLIEST-EQUAL-PRIORITY 1:SERIES 2:ANY 3:ANY)
308 (EARLIEST-EQUAL-PRIORITY-HEAD 1:SERIES 2:ANY
3:ORDERED-ASSOCIATIVE-LIST)
308 (EARLIEST-OAL-POSITION 1:SERIES 2:ANY
3:ORDERED-ASSOCIATIVE-LIST)
294 (EARLIEST-SIMULATION-FINISHED 1:SEQUENCE 2:QUEUE 3:SEQUENCE)
308 (EMPTY-OR-LOW-PRIORITY-HEAD 1:ORDERED-ASSOCIATIVE-LIST 2:ANY)
298 (ENUM-EVAL-COLLECT 1:LINKED-LIST 2:SEQUENCE
3:EXECUTION-CONTEXT 4:QUEUE 5:LINKED-LIST
6:SEQUENCE 7:EXECUTION-CONTEXT 8:QUEUE)
293 (ENUM-NODES+CHECK-BUFFERS 1:SEQUENCE)
306 (ENUM-OAL-FRONT 1:ORDERED-ASSOCIATIVE-LIST 2:ANY 3:SERIES)
306 (ENUM-OAL-FRONT-UNSAFE 1:ORDERED-ASSOCIATIVE-LIST 2:ANY
3:SERIES)
292 (ENUMERATE-AND-DELIVER-MESSAGES 1:QUEUE 2:SEQUENCE
3:SEQUENCE)
294 (ENUMERATE-NODES+COMPUTE-AVERAGE 1:SEQUENCE 2:INTEGER)
308 (EQUAL-PRIORITY-HEAD 1:ORDERED-ASSOCIATIVE-LIST 2:ANY)
308 (EQUAL-PRIORITY-TEST 1:ANY 2:ANY)
314 (EQUALITY-WITHIN-EPSILON 1:INTEGER 2:INTEGER)
299 (EVALUATE-AND-APPLY 1:SYMBOL 2:LINKED-LIST 3:SEQUENCE
4:EXECUTION-CONTEXT 5:QUEUE 6:ANY
7:SEQUENCE 8:EXECUTION-CONTEXT 9:QUEUE)
298 (EVALUATE-ARGUMENTS 1:LINKED-LIST 2:SEQUENCE 3:EXECUTION-CONTEXT
4:QUEUE 5:LINKED-LIST 6:SEQUENCE
7:EXECUTION-CONTEXT 8:QUEUE)
298 (EVALUATE-MAP 1:SERIES 2:SEQUENCE 3:EXECUTION-CONTEXT 4
:QUEUE 5:SERIES 6:SEQUENCE 7:EXECUTION-CONTEXT
8:QUEUE)
291 (EVENT-DRIVEN-SIMULATION 1:EVENT 2:PRIORITY-QUEUE 3:SEQUENCE
4:SEQUENCE)
293 (EXTRACT-AND-HANDLE-FIRST-MESSAGE 1:SYNCH-NODE 2:INTEGER 3:SEQUENCE
4:QUEUE 5:SEQUENCE 6:QUEUE)
303 (FETCH+DELETE 1:ANY 2:HASH-TABLE 3:HASH-TABLE)
303 (FETCH+INSERT 1:ANY 2:ANY 3:HASH-TABLE 4:HASH-TABLE)
303 (FETCH+LOOKUP 1:ANY 2:HASH-TABLE 3:ANY)
301 (FETCH+UPDATE 1:INDEXED-SEQUENCE 2:ANY 3:INDEXED-SEQUENCE)
299 (FETCH-AND-APPLY-OPERATOR 1:SYMBOL 2:LINKED-LIST 3:SEQUENCE
4:EXECUTION-CONTEXT 5:QUEUE 6:ANY
7:SEQUENCE 8:EXECUTION-CONTEXT 9:QUEUE)
300 (FETCH-INSTRUCTION 1:INTEGER 2:SEQUENCE 3:INSTRUCTION
4:INDEXED-SEQUENCE)
299 (FETCH-OP 1:SYMBOL 2:OPERATOR)
298 (FIFO-DEQUEUE 1:FIFO 2:ANY 3:FIFO)
296 (FIFO-DESTRUCTIVE-ENUMERATION 1:FIFO 2:SERIES)
296 (FIFO-EMPTY? 1:FIFO)
298 (FIFO-ENQUEUE 1:ANY 2:FIFO 3:FIFO)
297 (FIFO-ENUMERATION 1:FIFO 2:SERIES)
310 (FILTER 1:SERIES 2:SERIES)
310 (FILTERING 1:ANY 2:ANY)
306 (FIND-OAL-TAIL 1:ORDERED-ASSOCIATIVE-LIST 2:ANY
3:ORDERED-ASSOCIATIVE-LIST)
306 (FIND-OAL-TAIL-UNSAFE 1:ORDERED-ASSOCIATIVE-LIST 2:ANY
3:ORDERED-ASSOCIATIVE-LIST)
309 (GENERATE 1:ANY 2:SERIES)
291 (GENERATE-EVENT-QUEUES-AND-NODES 1:PRIORITY-QUEUE 2:SEQUENCE
3:SERIES 4:SERIES)
294 (GENERATE-GLOBAL-BUFFERS-AND-NODES 1:SEQUENCE 2:QUEUE 3:SERIES
4:SERIES)
309 (GENERATION 1:ANY 2:ANY)
293 (GLOBAL-AND-LOCAL-BUFFERS-EMPTY? 1:SEQUENCE 2:QUEUE)
297 (GROW-CIS 1:CIRCULAR-INDEXED-SEQUENCE 2:CIRCULAR-INDEXED-SEQUENCE)
299 (HANDLE-MESSAGE 1:MESSAGE 2:SEQUENCE 3:QUEUE 4:SEQUENCE 5:QUEUE)
302 (HASH-DELETE 1:ANY 2:HASH-TABLE 3:HASH-TABLE)
302 (HASH-INSERT 1:ANY 2:ANY 3:HASH-TABLE 4:HASH-TABLE)
302 (HASH-LOOKUP 1:ANY 2:HASH-TABLE 3:ANY)
310 (INCREMENT 1:INTEGER 2:INTEGER)
310 (INCREMENT-OR-DECREMENT 1:INTEGER 2:INTEGER)
301 (INDEXED-SEQUENCE-ACCUMULATION 1:SERIES 2:INDEXED-SEQUENCE
3:INDEXED-SEQUENCE)
301 (INDEXED-SEQUENCE-EXTRACT 1:INDEXED-SEQUENCE 2:ANY
3:INDEXED-SEQUENCE)
301 (INDEXED-SEQUENCE-INSERT 1:ANY 2:INDEXED-SEQUENCE
3:INDEXED-SEQUENCE)
297 (INTERMEDIATE-GROW-CIS 1:CIRCULAR-INDEXED-SEQUENCE 2:INTEGER
3:CIRCULAR-INDEXED-SEQUENCE)
305 (INTERMEDIATE-UOAL-DELETE 1:ANY 2:UNORDERED-ASSOCIATIVE-LIST
3:LINKED-LIST 4:UNORDERED-ASSOCIATIVE-LIST)
300 (INTERPRET-INSTRUCTION 1:INSTRUCTION 2:SEQUENCE 3:EXECUTION-CONTEXT
4:QUEUE 5:SEQUENCE 6:EXECUTION-CONTEXT 7:QUEUE)
298 (ITERATIVE-EVALUATION 1:ANY 2:SEQUENCE 3:EXECUTION-CONTEXT 4:QUEUE
5:ANY 6:SEQUENCE 7:EXECUTION-CONTEXT 8:QUEUE)
311 (ITERATIVE-SEARCH 1:ANY 2:ANY)
311 (LE 1:LINKED-LIST 2:SERIES)
309 (LIST-EMPTY 1:LINKED-LIST)
309 (LIST-POP 1:LINKED-LIST 2:ANY 3:LINKED-LIST)
307 (LIST-PUSH 1:ANY 2:LINKED-LIST 3:LINKED-LIST)
313 (LIST-TO-SEQUENCE 1:LINKED-LIST 2:INTEGER 3:SEQUENCE)
300 (LOAD-ARGUMENTS 1:MESSAGE 2:NODE 3:NODE)
300 (LOAD-ARGUMENTS-INTO-AN 1:MESSAGE 2:ASYNCH-NODE 3:ASYNCH-NODE)
300 (LOAD-ARGUMENTS-INTO-MEMORY 1:MESSAGE 2:ASSOCIATIVE-SET
3:ASSOCIATIVE-SET)
300 (LOAD-ARGUMENTS-INTO-SN 1:MESSAGE 2:SYNCH-NODE 3:SYNCH-NODE)
292 (LOCAL-BUFFER-DQ 1:SYNCH-NODE 2:MESSAGE 3:SYNCH-NODE)
292 (LOCAL-BUFFER-EMPTY? 1:SYNCH-NODE)
292 (LOCAL-BUFFER-NONEMPTY? 1:SYNCH-NODE)
292 (LOCAL-BUFFER-NQ 1:MESSAGE 2:SYNCH-NODE 3:SYNCH-NODE)
293 (LOCAL-BUFFERS-ALWAYS-EMPTY? 1:SERIES)
293 (LOCAL-BUFFERS-EMPTY? 1:SEQUENCE)
300 (LOOKUP-AND-EXECUTE-HANDLER 1:MESSAGE 2:SEQUENCE 3:QUEUE 4:INTEGER
5:SYMBOL 6:SEQUENCE 7:QUEUE)
304 (LOOKUP-DESTINATION 1:SEQUENCE 2:MESSAGE 3:ANY)
299 (LOOKUP-HANDLER 1:SYMBOL 2:HANDLER)
299 (LOOKUP-HANDLER-FOR-MESSAGE 1:MESSAGE 2:HANDLER)
292 (LOOKUP-NODE+NQ+UPDATE 1:MESSAGE 2:SEQUENCE 3:SEQUENCE)
313 (MAX 1:INTEGER 2:INTEGER 3:INTEGER)
313 (MIN 1:INTEGER 2:INTEGER 3:INTEGER)
314 (NEGATE-IF-NEGATIVE 1:INTEGER 2:INTEGER)
312 (NEW-SEQUENCE 1:INTEGER 2:SEQUENCE)
312 (NEW-TERM 1:SEQUENCE 2:INTEGER 3:ANY 4:SEQUENCE)
307 (OAL-RETRIEVE-IF-EXISTS 1:ANY 2:ORDERED-ASSOCIATIVE-LIST 3:ANY 4:ANY)
307 (OAL-SPLICE-IN 1:SERIES 2:ANY 3:ORDERED-ASSOCIATIVE-LIST
4:ORDERED-ASSOCIATIVE-LIST)

```
307 (OAL-SPLICE-OUT 1:SERIES 2:ORDERED-ASSOCIATIVE-LIST
                    3:ORDERED-ASSOCIATIVE-LIST)
307 (ORDERED-ASSOC-LE 1:ORDERED-ASSOCIATIVE-LIST 2:ANY 3:SERIES)
306 (ORDERED-ASSOC-LIST-DELETE 1:ANY 2:ORDERED-ASSOCIATIVE-LIST
                               3:ORDERED-ASSOCIATIVE-LIST)
309 (ORDERED-ASSOC-LIST-EXTRACT 1:ORDERED-ASSOCIATIVE-LIST 2:ANY
                                3:ORDERED-ASSOCIATIVE-LIST)
305 (ORDERED-ASSOC-LIST-INSERT 1:ANY 2:ANY
                    3:ORDERED-ASSOCIATIVE-LIST
                    4:ORDERED-ASSOCIATIVE-LIST)
306 (ORDERED-ASSOC-LIST-INSERT-SAFE 1:ANY 2:ANY
                                    3:ORDERED-ASSOCIATIVE-LIST
                                    4:ORDERED-ASSOCIATIVE-LIST)
306 (ORDERED-ASSOC-LIST-INSERT-UNSAFE 1:ANY 2:ANY
                                      3:ORDERED-ASSOCIATIVE-LIST
                                      4:ORDERED-ASSOCIATIVE-LIST)
307 (ORDERED-ASSOC-LIST-LOOKUP 1:ANY 2:ORDERED-ASSOCIATIVE-LIST
                               3:ANY)
307 (ORDERED-ASSOC-SLE 1:ORDERED-ASSOCIATIVE-LIST 2:ANY
                       3:SERIES)
293 (POLL-NODES-AND-DO-WORK 1:SEQUENCE 2:SEQUENCE 3:QUEUE)
305 (PQ-EMPTY 1:PRIORITY-QUEUE)
305 (PQ-ENUMERATION 1:PRIORITY-QUEUE 2:ANY)
305 (PQ-EXTRACT 1:PRIORITY-QUEUE 2:ANY 3:PRIORITY-QUEUE)
305 (PQ-INSERT 1:ANY 2:ANY 3:PRIORITY-QUEUE 4:PRIORITY-QUEUE)
291 (PROCESS-EVENT 1:EVENT 2:PRIORITY-QUEUE 3:SEQUENCE
                   4:PRIORITY-QUEUE 5:SEQUENCE)
302 (PROPERTY-LIST-LOOKUP 1:SYMBOL 2:SYMBOL 3:ANY)
295 (QUEUE-EMPTY? 1:QUEUE)
295 (QUEUE-EXTRACT 1:QUEUE 2:ANY 3:QUEUE)
295 (QUEUE-INSERT 1:ANY 2:QUEUE 3:QUEUE)
304 (RECORD-AT-DESTINATION 1:ANY 2:MESSAGE 3:SEQUENCE 4:SEQUENCE)
312 (REVERSE-LIST 1:LINKED-LIST 2:LINKED-LIST)
297 (ROOMY-CIS-ADD 1:ANY 2:CIRCULAR-INDEXED-SEQUENCE
                   3:CIRCULAR-INDEXED-SEQUENCE)
299 (RUNNING-STATUS? 1:EXECUTION-CONTEXT)
299 (RUNNING-TEST 1:SYMBOL)
310 (SELECT-TERM 1:SEQUENCE 2:INTEGER 3:ANY)
310 (SELECT-TERM-MAP 1:SEQUENCE 2:SERIES 3:SERIES)
311 (SEQ-LIST-SEARCH 1:LINKED-LIST 2:ANY)
312 (SEQUENCE-ACCUMULATE 1:SERIES 2:SERIES 3:SEQUENCE 4:SEQUENCE)
312 (SEQUENCE-ACCUMULATION 1:ANY 2:INTEGER 3:SEQUENCE 4:SEQUENCE)
313 (SEQUENCE-AND-INDEX-ENUMERATION 1:SEQUENCE 2:SERIES 3:SERIES)
312 (SEQUENCE-ENUMERATION 1:SEQUENCE 2:SERIES)
312 (SEQUENCE-SIZE 1:SEQUENCE 2:INTEGER)
311 (SEQUENTIAL-SEARCH 1:SERIES 2:ANY)
291 (SEQUENTIAL-SIMULATION-OF-MESSAGE-PASSING-SYSTEM
       1:SEQUENCE 2:ANY 3:SEQUENCE)
311 (SLE 1:LINKED-LIST 2:SERIES)
313 (SQUARE-ROOT-OF-SQUARE 1:INTEGER 2:INTEGER)
296 (STACK-EMPTY? 1:STACK)
295 (STACK-ENUMERATION 1:STACK 2:SERIES)
296 (STACK-POP 1:STACK 2:ANY 3:STACK)
296 (STACK-PUSH 1:ANY 2:STACK 3:STACK)
313 (SUM 1:SERIES 2:INTEGER)
313 (SUMMING 1:INTEGER 2:INTEGER 3:INTEGER)
294 (SYNCHRONOUS-SIMULATION 1:SEQUENCE 2:MESSAGE 3:SEQUENCE)
293 (SYNCHRONOUS-SIMULATION-FINISHED? 1:SEQUENCE 2:QUEUE
                                      3:SEQUENCE)
294 (SYNCHRONOUS-SIMULATION-W-GLOBAL-MESSAGE-BUFFER
       1:SEQUENCE 2:MESSAGE 3:SEQUENCE)
313 (TEST-PREDICATE 1:ANY)
312 (TRAILING-GENERATE 1:ANY 2:SERIES 3:SERIES)
312 (TRAILING-GENERATION 1:ANY 2:ANY 3:ANY)
312 (TRAILING-PTR-LE 1:LINKED-LIST 2:SERIES 3:SERIES)
311 (TRUNCATE 1:SERIES 2:SERIES)
308 (TRUNCATE-EQUAL-PRIORITY 1:SERIES 2:ANY 3:SERIES)
308 (TRUNCATE-EQUAL-PRIORITY-HEAD 1:SERIES 2:ANY 3:SERIES)
308 (TRUNCATE-OAL-POSITION 1:SERIES 2:ANY 3:SERIES)
307 (TRUNCATE-OAL-POSITION-UNSAFE 1:SERIES 2:ANY 3:SERIES)
311 (TRUNCATION 1:ANY 2:ANY)
313 (UNARY-PREDICATE 1:ANY 2:ANY)
305 (UNORDERED-ASSOC-LIST-DELETE
       1:ANY 2:UNORDERED-ASSOCIATIVE-LIST
       3:UNORDERED-ASSOCIATIVE-LIST)
305 (UNORDERED-ASSOC-LIST-EMPTY? 1:UNORDERED-ASSOCIATIVE-LIST)
305 (UNORDERED-ASSOC-LIST-INSERT 1:ANY
                                 2:UNORDERED-ASSOCIATIVE-LIST
                                 3:UNORDERED-ASSOCIATIVE-LIST)
305 (UNORDERED-ASSOC-LIST-LOOKUP
       1:ANY 2:UNORDERED-ASSOCIATIVE-LIST 3:ANY)
301 (UPDATE+BUMP 1:ANY 2:INDEXED-SEQUENCE 3:INDEXED-SEQUENCE)
301 (UPDATE+FETCH 1:INDEXED-SEQUENCE 2:ANY 3:INDEXED-SEQUENCE)
292 (UPDATE-NODE-TIME 1:ASYNCH-NODE 2:INTEGER 3:ASYNCH-NODE)
```

List of Figures

1-1 A hybrid program understanding system. 9
1-2 GRASPR's architecture. 13

2-1 Synchronous simulation clichés. 26
2-2 Aggregate data clichés. 27
2-3 Event-driven simulation clichés. 30
2-4 Node action simulation clichés. 32
2-5 General-purpose clichés. 34
2-6 A message handler for Factorial. 36
2-7 The definition of two Machine Operations. 37
2-8 Design tree for PiSim. 40
2-9 Some of the documentation generated for PiSim. 41
2-10 Top-level portion of PiSim code. 43
2-11 A syntactic variation of the portion of PiSim shown in Figure 2-10. 44
2-12 An organizational variation of the top-level portion of PiSim. 45
2-13 Top-level portion of CST. Question marks indicate unfamiliar code. 47
2-14 A portion of design tree produced in recognizing CST. 49
2-15 A portion of the documentation generated for CST. 50
2-16 Buffer queue implemented as a FIFO, which in turn is implemented as a CIS. 52
2-17 Buffer queue implemented as a stack (LIFO). 53
2-18 Design tree for implementational variation in which the buffer is a stack. . . 54
2-19 Portion of CST that averages node queue lengths. 55
2-20 Design tree for queue length averaging computation. 55
2-21 Optimization in which averaging is performed while advancing nodes. . . . 56
2-22 Design tree for optimized code, with shared sub-tree. 57
2-23 Code containing a redundant CAR computation. 58
2-24 Code in which the result of CAR is cached and reused. 58

3-1 An example attributed flow graph. 61
3-2 An example flow graph grammar. 64
3-3 An example derivation sequence. 66
3-4 An example derivation tree. 67

3-5 An example attributed flow graph grammar. 68
3-6 An attributed derivation tree. 69
3-7 Testing whether the three input sides form a right triangle. 70
3-8 Attributed flow graph for RIGHTP. 71
3-9 Flow graph grammar encoding clichés found in RIGHTP. 72
3-10 Clichés recognized in RIGHTP. 74
3-11 These flow graphs should all be seen as equivalent. 76
3-12 a) A grammar. b) Its core language. c) Some flow graphs in its expanded language. 77
3-13 a) A grammar. b) A derivation sequence. c) A derivation graph representing the derivation. 78
3-14 (a) A grammar. (b) Two derivations of same flow graph. (c) Two derivation graphs representing the derivations. 79
3-15 A grammar representing aggregation, using Spread and Make nodes. 82
3-16 F_1 is the flow graph in the language of the grammar in Figure 3-15. The rest are flow graphs aggregation-equivalent to it. 83
3-17 F_3 and F_1 can be transformed to this flow graph by flattening nested Makes and Spreads. 85
3-18 Two programs each performing two consecutive Stack Pops. 88
3-19 The flow graph for the programs POP-TWICE and POP-TWICE2. 89
3-20 Flow graph with a node whose output port is of type Any. 89
3-21 (a) A rule which aggregates port types. (b) The same rule with aggregation information moved to the embedding relation. 91
3-22 (a) An edge connects a Spread and Make. (b) This edge becomes a st-thru when aggregation information is moved to the embedding relation. 92
3-23 Circular Indexed Sequence data structure. 93
3-24 The rule for Circular Indexed Sequence Extract. 93
3-25 The grammar of Figure 3-15 with aggregation encoded in the embedding relation. 95
3-26 A reduction sequence using the grammar of Figure 3-25. 96
3-27 The reduction of a sub-flow graph using the rule for D from Figure 3-25. . . 97
3-28 (a) A flow graph only partially recognizable as the non-terminal S, whose rule is in (b). (c) Result of reduction. (d) Breaking up residual Spreads and Makes to facilitate partial recognition. 99
3-29 Flow graph parser evolution. 101
3-30 Graph chart parsing. 102
3-31 (a) Adding a complete item to the chart. (b) Adding a partial item to the chart. 104
3-32 A bottom-up rule invocation strategy affects adding a complete item to chart. 105
3-33 Search strategy as input to parser. 106

3-34 Additional monitors. 107
3-35 Sharing a sub-derivation. 109
3-36 (a) A graph grammar that maximally shares the non-terminal A. (b) An input flow graph containing two redundant instances of A. (c) An alternative view created by "zipping up" the input graph. 111
3-37 (a) A flow graph with location pointers. (b) Items created during parsing. . 112
3-38 Simulating the break up of residual Spreads and Makes. 114
3-39 Grammar containing a rule with a st-thru. 115
3-40 Constraint on combination imposed by st-thrus. 115
3-41 Constrained and unconstrained st-thrus. 117
3-42 Propagating matches of st-thrus. 118

4-1 A recursive function with multiple exits. 124
4-2 Flow graph representing `HT-Insert`. 125
4-3 Annotated partial order graph representing the relationships between the control environments of `HT-Insert`. 127
4-4 Flow graph grammar rule for Negate-if-Negative, with actual attribute conditions. 129
4-5 Grammar rule for counting-up cliché. 130
4-6 The plan diagram for a code fragment. 132
4-7 A recursively defined plan. 133
4-8 Data plan for Circular Indexed Sequence. 133
4-9 Plan for extracting an element from a Circular Indexed Sequence. 134
4-10 Implementation overlay showing how FIFO-Dequeue can be implemented by CIS-Extract. 135
4-11 Rule encoding plan for CIS-Extract. 137
4-12 Rule encoding the CIS-Extract-as-FIFO-Dequeue overlay. 138
4-13 Temporal overlay showing the view of Generation as a Generate operation. 139
4-14 Grammar rule encoding the plan for Generation. 140
4-15 Temporal overlay relating the plan for Iterative Search and the operation Earliest. 141
4-16 Grammar rule for Iterative Search cliché. 142
4-17 Grammar rule encoding the temporal overlay Iterative-Search-as-Earliest. . 143
4-18 Plan definition for Event-Driven Simulation cliché. 144
4-19 Overlay showing the temporal abstraction of the iteration cliché Dequeue-and-Process-Generation. 146
4-20 Overlay showing the temporal abstraction of the iteration cliché Co-Iterative-EDS-Finished. 147
4-21 Grammar rules for some Event-Driven Simulation clichés. 148
4-22 Grammar rules for clichés used by Event-Driven Simulation cliché. 149

4-23 Plan definition for the Process-Event cliché. 151
4-24 Rules for Process-Event cliché. 152
4-25 Plan definition for the Update-Node-Time cliché. 153
4-26 Grammar rule encoding the Update-Node-Time plan. 154
4-27 Code that side effects the mutable data structure `*Event-Queue*`. 156
4-28 Functional version of `Insert-Queue`. 157
4-29 Version of `Insert-Queue-Pure` in which recursion is folded up. 157
4-30 Flow graph representing `Insert-Queue-Pure`. 158
4-31 Partial ordering relationships between the control environments of `Insert-Queue-Pure`'s flow graph. 159
4-32 Documentation containing a clichéd-to-user-defined name mapping. 162

5-1 Flow graph representing the code in Figures 2-10, 2-11, and 2-12. 165
5-2 Attribute values for accessor and constructor attributes annotating the flow graphs representing the programs in Figures 2-10 (column a), 2-11 (column b), and 2-12 (column c). 166
5-3 Flow graph representing the `CST` code of Figure 2-13. 170
5-4 a) Average cliché. b-c) Some cases in which a program can be partially recognized. 171
5-5 Rules for Extract-Message and Local-Buffer-Dequeue cliché. 172
5-6 Code containing a partially recognized data structure. 172
5-7 Flow graph representation for `step`. 173
5-8 Some valid variations of Synchronous Simulation algorithm. 182

6-1 Two series of extensions resulting in duplicate items. 191
6-2 Partitions of the total item set. 193
6-3 Grammar and input graph leading to an illegal, cyclic reduction. 199
6-4 The plan for extracting from a Circular-Indexed Sequence. 201
6-5 Bushy item tree produced in recognizing CIS-Extract with weak match-interleaved constraints. 202
6-6 The restriction on legal instances imposed by the precedence relation constraint. 203
6-7 Skinny item tree produced in recognizing CIS-Extract with strong match-interleaved constraints. 204
6-8 Results of running `CST` example with constraints parse-interleaved versus match-interleaved. 205
6-9 Relationship of the sets of successful, killed, and extendable item sets to the sets of complete and partial items. 205
6-10 Results of running `PISIM` example with constraints parse-interleaved versus match-interleaved. 206
6-11 The shapes of item trees having maximum maximum width. 210

7-1 Four ways of implementing Stack-Push and Stack-Pop with the Stack implemented as an Indexed-Sequence. 236

A-1 Reducing fixed-UCFG recognition to flow graph recognition. 257

Bibliography

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, 1985.

[2] A. Adam and J. Laurent. LAURA, A system to debug student programs. *Artificial Intelligence*, 15:75–122, 1980.

[3] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1983.

[4] D. Allemang. Understanding programs as devices. Technical report, Ohio State University, 1990. PhD thesis.

[5] D. Allemang. Using functional models in automatic debugging. *IEEE Expert*, pages 13–18, December 1991.

[6] G. Alpern, A. Carle, B. Rosen, P. Sweeney, and K. Zadeck. Graph attribution as a specification paradigm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 121–129, Boston, MA, November 1988.

[7] J. Ambras and V. O'Day. MicroScope: A knowledge-based programming environment. *IEEE Software*, 5(3):50–58, 1988.

[8] C. Bamji. Graph-based representations and coupled verification of VLSI schematics and layouts. Technical Report 547, MIT Research Laboratory of Electronics, October 1989. PhD thesis.

[9] C. Bamji and J. Allen. GRASP: A grammar-based schematic parser. VLSI Memo 89-515, MIT Research Laboratory of Electronics, March 1989. Also in Proc. 26th Design Automation Conference, pp.448-453.

[10] E. Barton, R. Berwick, and Ristad E. *Computational Complexity and Natural Language.* The MIT Press, Cambridge, MA, 1987.

[11] K. Bertels. Qualitative reasoning in novice program analysis. Technical report, Universiteit Antwerpen, June 1991. PhD thesis.

[12] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, July 1989. Also published as MCC Technical Report STP-378-88.

[13] T. Biggerstaff, J. Hoskins, and D. Webster. DESIRE: A system for design recovery. Technical Report STP-081-89, MCC, April 1989.

[14] R. Boyer and J. Moore. The sharing of structure in theorem-proving programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 101–116. John Wiley and Sons, New York, 1972.

[15] D. Brotsky. An algorithm for parsing flow graphs. Technical Report 704, MIT Artificial Intelligence Lab., March 1984. Master's thesis.

[16] H. Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 4(6), November 1982.

[17] H. Bunke. Graph grammars as a generative tool in image understanding. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, pages 8–19. Springer-Verlag, October 1982. Lecture Notes In Computer Science Series, Vol. 153.

[18] H. Bunke and B. Haller. A parser for context free plex grammars. In M. Nagl, editor, *15th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 136–150. Springer-Verlag, June 1989. Lecture Notes In Computer Science Series, Vol. 411.

[19] S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, January 1990.

[20] L. Cleveland. An environment for understanding programs. Technical Report 12889, IBM T.J. Watson Research Center, Yorktown Hgts., NY, June 1987.

[21] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[22] D. Corneil and D. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM Journal of Computing*, 9(2):281–297, May 1980.

[23] B. Courcelle. A representation of graphs by algebraic expressions and its use for graph rewriting systems. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 112–132, 1986. Lecture Notes In Computer Science Series, Vol. 291.

[24] D. S. Cyphers. Automated program description. Working Paper 237, MIT Artificial Intelligence Lab., August 1982.

[25] W. Dally and A. Chien. Object-oriented concurrent programming in CST. In *The Third Conference on: Hypercube Concurrent Computers and Applications, Volume I - Architecture, Software, Computer Systems and General Issues.* ACM, January 1988.

[26] W. Dally, A. Chien, S. Fiske, W. Horwat, J. Keene, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler. The J-Machine: A fine-grain concurrent computer. In *Int. Fed. of Info. Processing Societies*, 1989.

[27] P. Della-Vigna and C. Ghezzi. Context-free graph grammars. *Information and Control*, 37(2):207–233, 1978.

[28] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *8th Annual ACM Symp. on Principles of Prog. Langs.*, pages 105–116, Williamsburg, VA, January 1981.

[29] G. Dueck and G. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, 1990.

[30] A. Duncan and J. Hutchison. Using attributed grammars to test designs and implementations. In *5th Int. Conf. on Software Engineering*, pages 170–178, San Diego, CA, March 1981.

[31] J. Earley. *An Efficient Context-Free Parsing Algorithm.* PhD thesis, Carnegie-Mellon Univ. Computer Science Dept., 1968.

[32] J. Earley. An efficient context-free parsing algorithm. *Comm. of the ACM*, 13(2):94–102, 1970.

[33] H. Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 3–14. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[34] H. Ehrig, M. Nagl, and G. Rozenberg, editors. *Graph-Grammars and Their Application to Computer Science.* Springer-Verlag, Haus Ohrbeck, Germany, October 1982. Lecture Notes In Computer Science Series, Vol. 153.

[35] H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors. *Graph-Grammars and Their Application to Computer Science.* Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[36] J. Engelfriet and G Rozenberg. A comparison of boundary graph grammars and context-free hypergraph grammars. *Information and Control*, 84:163–206, 1990.

[37] R. Engelmore and T. Morgan, editors. *Blackboard Systems.* Addison-Wesley, Reading, MA, 1988.

[38] G. Engels, C. Lewerentz, and W. Schafer. Graph grammar engineering: A software specification method. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 186–201. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[39] M.A. Eshera and K. Fu. An image understanding system using attributed symbolic representation and inexact graph-matching. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(5), September 1986.

[40] R. Farrow. Experience with an attribute grammar-based compiler. In *9th Annual ACM Symp. on Principles of Prog. Langs.*, pages 95–107, Albuquerque, NM, January 1982.

[41] R. Farrow, K. Kennedy, and L. Zucconi. Graph grammars and global program data flow analysis. In *Proc. 17th Annual IEEE Symposium on Foundations of Computer Science*, Houston, Texas, 1976.

[42] G. Faust. Semiautomatic translation of COBOL into HIBOL. Technical Report 256, MIT Lab. of Computer Science, March 1981. Master's thesis.

[43] S. F. Fickas and R. Brooks. Recognition in a program understanding system. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pages 266–268, Tokyo, Japan, August 1979.

[44] R. Franck. A class of linearly parsable graph grammars. *Acta Informatica*, 10:175–201, 1978.

[45] C. Frank. A step towards automatic documentation. Working Paper 213, MIT Artificial Intelligence Lab., December 1980.

[46] K. Gallagher. Using program slicing in software maintenance. Technical Report CS-90-05, Loyola College in Maryland, 1990.

[47] H. Ganzinger, R. Giegerich, M. Ulrich, and W. Reinhard. A truly generative semantics-directed compiler generator. In *SIGPLAN 82 Symposium on Compiler Construction*, pages 172–184, 1982.

[48] E. Gmur and H Bunke. 3-D object recognition base on subgraph matching in polynomial time. In R. Mohr, T. Pavlidis, and A. Sanfeliu, editors, *Structural Pattern Analysis*, pages 131–147. World Scientific, New Jersey, 1989.

[49] W.E.L. Grimson. The combinatorics of object recognition in cluttered environments using constrained search. Memo 1019, MIT Artificial Intelligence Lab., February 1988.

[50] W.E.L. Grimson. The effect of indexing on the complexity of object recognition. Memo 1226, MIT Artificial Intelligence Lab., April 1990.

[51] W. Griswold and D. Notkin. Program restructuring to aid software maintenance. Technical Report 90-08-05, Univ. of Washington, September 1990.

[52] A. Habel and H. Kreowski. On context-free graph languages generated by edge replacement. In *Graph-Grammars and Their Application to Computer Science*, pages 143–158, 1983. Lecture Notes In Computer Science Series, Vol. 153.

[53] A. Habel and H. Kreowski. May we introduce to you: Hyperedge replacement. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 15–26. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[54] M. Harandi and J. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–81, January 1990.

[55] J. Hartman. Automatic control understanding for natural programs. Technical Report AI 91-161, University of Texas at Austin, 1991. PhD thesis.

[56] P. Hausler, M. Pleszkoch, R. Linger, and A. Hevner. Using function abstraction to understand program behavior. *IEEE Software*, pages 55–63, January 1990.

[57] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[58] R. Holt, D. Boehm-Davis, and A. Schultz. Mental representations of programs for student and professional programmers. In G. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop.* Ablex Publishing Corp., Norwood, N.J., 1987.

[59] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. Technical Report 756, Univ. of Wisconsin at Madison, Computer Sciences Dept., March 1988.

[60] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980.

[61] G. Huet and D. Oppen. Equations and rewrite rules: a survey. In *Formal Languages: perspectives and open problems.* Applied Psycholinguistics, Boston, MA, 1980.

[62] D. Hutchens and V. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. on Software Engineering*, 11(8), August 1985.

[63] V. Jagannathan, R. Dodhiawala, and L.S. Baum, editors. *Blackboard Architectures and Applications.* Academic Press, Inc., Boston, MA, 1989.

[64] M. Jazayeri, F. Ogden, and W. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Comm. of the ACM*, 8(12), December 1975.

[65] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors.* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[66] G. E. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3), 1988.

[67] L. Karttunen and M. Kay. Structure sharing with binary trees. In *Proc. 23rd Annual Meeting of the ACL*, pages 133–136, Chicago, IL, 1985.

[68] U. Kastens, B. Hutt, and E. Zimmermann. GAG: A practical compiler generator. In *Lecture Notes in Computer Science Series.* Springer-Verlag, 1982.

[69] M. Kaul. Parsing of graphs in linear time. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 206–218, Haus Ohrbeck, Germany, October 1982. Springer-Verlag. Lecture Notes In Computer Science Series, Vol. 153.

[70] M. Kaul. Practical applications of precedence graph grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 326–342. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[71] M. Kay. The MIND system. In R. Rustin, editor, *Natural Language Processing.* Prentice-Hall, Englewood-Cliffs, NJ, 1973.

[72] M. Kay. Algorithm schemata and data structures in syntactic processing. In B. Grosz, K. Sparck-Jones, and B. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[73] K. Kennedy and S. Warren. Automatic generation of efficient evaluators for attribute grammars. In *3rd Annual ACM Symp. on Principles of Prog. Langs.*, pages 32–49, Atlanta, GA, 1976.

[74] K. Kennedy and L. Zucconi. Applications of a graph grammar for program control flow analysis. In *4th Annual ACM Symp. on Principles of Prog. Langs.*, pages 72–85, Santa Monica, CA, 1977.

[75] J. Klop. Term rewriting systems: A tutorial. *Bulletin of European Assoc. for Theor. Computer Science*, (32):143–182, 1987.

[76] D. Knuth. *The Art of Computer Programming.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1968,1969,1973.

[77] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[78] K. Koskimies. A specification language for one-pass semantic analysis. In *SIGPLAN 84 Symposium on Compiler Construction*, pages 179–189, Montreal, Canada, 1984.

[79] K. Koskimies, K. Raiha, and M. Sarjakoski. Compiler construction using attribute grammars. In *SIGPLAN 82 Symposium on Compiler Construction*, pages 153–159, 1982.

[80] H. Kreowski and G. Rozenberg. Note on node-rewriting graph grammars. *Information Processing Letters*, 18:21–24, 1984.

[81] J. Laubsch and M. Eisenstadt. Domain specific debugging aids for novice programmers. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 964–969, Vancouver, British Columbia, Canada, August 1981.

[82] J. Laubsch and M. Eisenstadt. Using temporal abstraction to understand recursive programs involving side effects. In *Proc. 2nd National Conf. on Artificial Intelligence*, Pittsburgh, PA, August 1982.

[83] S. Letovsky. Cognitive processes in program comprehension. In G. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop.* Ablex Publishing Corp., Norwood, N.J., 1987.

[84] S. Letovsky. Plan analysis of programs. Research Report 662, Yale University, December 1988. PhD Thesis.

[85] C.K. Looi. APROPOS2: A program analyser for a Prolog intelligent teaching system. Research paper 377, Dept. of AI, University of Edinburgh, 1988.

[86] S. Lu and A. Wong. Synthesis of attributed hypergraphs for knowledge representation of 3-D objects. In J. Kittler, editor, *Lecture Notes in Computer Science Series No. 301*, pages 546–556. Springer-Verlag, 1988.

[87] F. J. Lukey. Understanding and debugging programs. *Int. Journal of Man-Machine Studies*, 12:189–202, 1980.

[88] R. Lutz. Program debugging by near-miss recognition and symbolic evaluation. Technical Report CSRP.044, Univ. of Sussex, England, 1984.

[89] R. Lutz. Diagram parsing — A new technique for artificial intelligence. Technical Report CSRP.054, Univ. of Sussex, England, 1986.

[90] R. Lutz. Chart parsing of flowgraphs. In *Proc. 11th Int. Joint Conf. Artificial Intelligence*, pages 116–121, Detroit, Michigan, 1989.

[91] M.H. MacDougall. *Simulating Computer Systems: Techniques and Tools.* The MIT Press, Cambridge, MA, 1987.

[92] M. Main and G. Rozenberg. Edge-label controlled graph grammars. *Journal of Computation and Systems Sciences*, 40:188–228, 1990.

[93] M. Minsky. Logical versus analogical or symbolic versus connectionist or neat versus scruffy. *AI Magazine*, 12(2):34–51, Summer 1991.

[94] U. G. Montanari. Separable graphs, planar graphs, and web grammars. *Information and Control*, 16(3):243–267, March 1970.

[95] W. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[96] M. Nagl. Set theoretic approaches to graph grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 41–54. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[97] M. Nagl. A software development environment based on graph technology. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 458–478. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[98] M. Nagl, G. Engels, R. Gall, and W. Schafer. Software specification by graph grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 265–287, Haus Ohrbeck, Germany, October 1982. Springer-Verlag. Lecture Notes In Computer Science Series, Vol. 153.

[99] H.P. Nii. Blackboard systems. In A. Barr, P. Cohen, and E. A. Feigenbaum, editors, *Handbook of Artificial Intelligence*, pages 1–82. Addison-Wesley Publishing Co., 1989. Vol.IV.

[100] J.Q. Ning. A knowledge-based approach to automatic program analysis. Technical report, University of Illinois, Urbana-Champaign, 1989. PhD thesis.

[101] R. Nord and F. Pfenning. The Ergo attribute system. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 110–120, Boston, MA, November 1988.

[102] T. Pavlidis. Linear and context-free graph grammars. *Journal of the ACM*, 19(1):11–23, January 1972.

[103] K. Peng, T. Yamamoto, and Y. Aoki. A new parsing algorithm for plex grammars. *Pattern Recognition*, 23(3-4):393–402, 1990.

[104] F. Pereira. A structure-sharing representation for unification-based grammar formalisms. In *Proc. 23rd Annual Meeting of the ACL*, pages 137–144, Chicago, IL, 1985.

[105] J. L. Pfaltz and A. Rosenfeld. Web grammars. In *Proc. 1st Int. Joint Conf. Artificial Intelligence*, pages 609–619, Washington, D.C., September 1969.

[106] R. Prieto-Diaz and G. Arango, editors. *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, Los Alamitos, CA, 1991.

[107] K. Raiha. Bibliography on attribute grammars. *ACM Sigplan Notices*, 15(3):35–44, March 1980.

[108] R. Read and D. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

[109] T. Reps and A. Demers. Sublinear-space evaluation algorithms for attribute grammars. *ACM Trans. on Programming Languages and Systems*, 9(3):408–440, July 1987.

[110] C. Rich. Inspection methods in programming. Technical Report 604, MIT Artificial Intelligence Lab., June 1981. PhD thesis.

[111] C. Rich. Knowledge representation languages and predicate calculus: How to have your cake and eat it too. In *Proc. 2nd National Conf. on Artificial Intelligence*, Pittsburgh, PA, August 1982.

[112] C. Rich. Inspection methods in programming: Clichés and plans. Memo 1005, MIT Artificial Intelligence Lab., December 1987.

[113] C Rich, editor. *Implemented Knowledge Representation and Reasoning Systems.* ACM Press, New York, NY, June 1991. SIGART Bulletin: Special Issue, Volume 2, Number 3.

[114] C. Rich and H. E. Shrobe. Initial report on a lisp programmer's apprentice. Technical Report 354, MIT Artificial Intelligence Lab., December 1976. Master's thesis.

[115] C. Rich, H. E. Shrobe, and R. C. Waters. An overview of the Programmer's Apprentice. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, Tokyo, Japan, 1979.

[116] C. Rich and R. C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, 21(11):10–25, November 1988. Also published as MIT AI Memo 1004.

[117] C. Rich and R. C. Waters. *The Programmer's Apprentice.* Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD, 1990.

[118] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990. Reprinted in P. H. Winston, editor, *Artificial Intelligence at MIT: Expanding Frontiers*, MIT Press, Cambridge, MA, In press.

[119] A. Rosenfeld and D. Milgram. Web automata and web grammars. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 307–324. John Wiley and Sons, New York, 1972.

[120] G. Rozenberg. An introduction to the NLC way of rewriting graphs. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 55–66. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[121] G. Rozenberg and E. Welzl. Boundary NLC graph grammars – basic definitions, normal forms, and complexity. *Information and Control*, 69:136–167, 1986.

[122] G. R. Ruth. Analysis of algorithm implementations. Technical Report 130, MIT Project Mac, 1974. PhD thesis.

[123] R. Schwanke. An intelligent tool for re-engineering software modularity. In *IEEE Conf. on Software Maintenance - 1991*, pages 83–92, 1991.

[124] R. Schwanke, R. Altucher, and M. Platoff. Discovering, visualizing, and controlling software structure. In *Proc. 5th Int. Wrkshp on Software Specs. and Design*, pages 147–150, Pittsburgh, PA, 1989.

[125] V. Sembugamoorthy and B. Chandrasekaran. Functional representation of devices and compilation of diagnostic problem-solving systems. In J. Kolodner and C. Riesbeck, editors, *Experience, Memory, and Reasoning*, pages 47–73. Lawrence Erlbaum Assoc., Hillsdale, NJ, 1986.

[126] H. E. Shrobe. Common sense reasoning about side effects to complex data structures. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, Tokyo, Japan, August 1979.

[127] H. E. Shrobe. Dependency directed reasoning for complex program understanding. Technical Report 503, MIT Artificial Intelligence Lab., April 1979. PhD thesis.

[128] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, 10(5):595–609, September 1984. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[129] D. Soni. Maintenance of large software systems: Treating global interactions. In *Proc. of AAAI Spring Symposium*, March 1989.

[130] D. Soni. A study of data structure cliches for software design and maintenance. Working paper, Siemens Corporation, 1989. in preparation.

[131] L. Tan, Y. Shinoda, and T. Katayama. Coping with changes in an object management system based on attribute grammars. In *4th ACM SIGSOFT Symposium on Sofware Development Environments*, pages 56–65, Irvine, CA, December 1990.

[132] H. Thompson. Chart parsing and rule schemata in GPSG. In *Proc. 19th Annual Meeting of the ACL*, Stanford, CA, 1981.

[133] H. Thompson and G. Ritchie. Implementing natural language parsers. In T. O'Shea and M. Eisenstadt, editors, *Artificial Intelligence: Tools, Techniques, and Applications*, pages 245–300. Harper and Row, New York, 1984.

[134] G. Tinhofer and G. Schmidt, editors. *Graph-Theoretic Concepts in Computer Science.* Springer-Verlag, June 1986. Lecture Notes In Computer Science Series, Vol. 246.

[135] W. Tsai and K. Fu. Attributed grammars – A tool for combining syntactic and statistical approaches to pattern recognition. *IEEE Trans. on Systems, Man and Cybernetics*, 10(12), December 1980.

[136] W. Vogler. On hyperedge replacement and BNLC graph grammars. In M. Nagl, editor, *Graph-Theoretic Concepts in Computer Science*, pages 78–93. Springer-Verlag, 1989. Lecture Notes In Computer Science Series.

[137] R. C. Waters. Automatic analysis of the logical structure of programs. Technical Report 492, MIT Artificial Intelligence Lab., December 1978. PhD thesis.

[138] R. C. Waters. A method for analyzing loop programs. *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[139] R. C. Waters. KBEmacs: A step towards the Programmer's Apprentice. Technical Report 753, MIT Artificial Intelligence Lab., May 1985.

[140] M. Weiser. Program slicing. In *5th Int. Conf. on Software Engineering*, pages 439–449, San Diego, CA, 1981.

[141] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10:352–357, 1984.

[142] S. Wiedenbeck. Novice/expert differences in programming skills. *Int. Journal of Man-Machine Studies*, 23:383–390, 1985.

[143] N. Wilde, R. Huitt, and S Huitt. Dependency analysis tools: Reusable components for software maintenance. In *IEEE Conf. on Software Maintenance - 1989*, pages 126–131, Miami, Florida, 1989.

[144] L. Wills. Automated program recognition. Technical Report 904, MIT Artificial Intelligence Lab., January 1987. Master's thesis.

[145] L. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(1-2):113–172, 1990.

[146] S. Wills. Pi: A parallel architecture interface for multi-model execution. Technical Report 1245, MIT Artificial Intelligence Lab., June 1990. PhD Thesis.

[147] S. Wills and W. Dally. Pi: A parallel architecture interface. In *FRONTIERS '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[148] P. H. Winston and B. K. P. Horn. *LISP.* Addison-Wesley Publishing Company, Reading, MA, 1981.

[149] M. Wiren. Interactive incremental chart parsing. In *4th Conf. of the European Chapter of the ACL*, pages 241–248, Manchester, England, 1989.

[150] K. Wittenburg, L. Weitzman, and J. Talley. Unification-based grammars and tabular parsing for graphical languages. Technical Report ACT-OODS-208-91, MCC, June 1991.

[151] B.P. Zeigler. *Theory of Modeling and Simulation.* John Wiley and Sons, New York, 1976.