MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Working Paper No. 327 June 1989

A Program Design Assistant

by

Yang Meng Tan

Abstract: The DA will be a design assistant which can assist the programmer in low-level design. The input language of the DA is a cliché-based program description language that allows the specification and high-level design of commonly-written programs to be described concisely. The DA language is high-level in the sense that programmers need not bother with detailed design. The DA will provide automatic low-level design assistance to the programmer in selecting appropriate algorithms and data structures. It will also help detect inconsistencies and incompleteness in program descriptions.

A key related issue in this research is the representation of programming knowledge in a design assistant. The knowledge needed to automate low-level design and the knowledge in specific programming clichés have to be represented explicitly to facilitate reuse.

Copyright © Massachusetts Institute of Technology, 1989

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be considered papers to which reference can be made in the literature.

Contents

1 Introduction **1**

2 The Design Assistant **3**

2.1 DA: Successor of KBEmacs 3

2.2 The Approach 4

2.3 What the DA Will Do 5

2.4 Building the DA 6

2.5 Architecture of the System 7

3 Target Scenario **7**

3.1 What the DA Knows 8

3.2 Background Description of Text-Justification 10

3.3 Scenario 11

3.4 Scene 1: Initial Program Description for Justify 12

3.5 Scene 2: Elaborating the Program Description of Justify 15

3.6 Scene 3: Preventing Errors 25

3.7 Scene 4: Explaining Design Rationale 28

3.8 Scene 5: Changing An Implementation Guideline 30

3.9 Scene 6: Correcting The Program Description 32

3.10 Scene 7: The Complete Program Descriptions 37

3.11 The Rest of the Scenario 39

4 Related Work **42**

1 Introduction

The software industry has been experiencing software productivity problems for some years: software is difficult to construct and maintain reliably and cost-effectively. The increasing cost of reliable software needs to be countered by increases in programmer productivity. In this proposed thesis, we explore one novel approach toward increasing the productivity of programmers. We propose a new programming paradigm that is knowledge-based, specification-oriented, and assistant-based.

Our approach is influenced by our view on the nature of programming. Hence, before delving into the depths of our approach, we briefly characterize our view of programming.

- Programming is Knowledge-Intensive: Different sources of knowledge are required. Knowledge of data structures and algorithms are key components of programming, so are program structuring techniques, program specification, and knowledge about the application domain. We believe that much of the knowledge needed in programming can be codified so that a computer program can make use of it mechanically.
- Requirements Constantly Change: Many changes are constantly being made to programs due to constantly changing needs. This is not just evident in program maintenance but also during the development process. Program maintenance, especially for large systems, is difficult, because much of the design rationale is not written down; and where it is written down, it is usually not kept up-to-date.

We propose codifying commonly-used programming concepts, called *clichés*, for reuse[23]. A cliché embodies a computational idea or a method of computation. Clichés connote frequent reuse. In software engineering it is important to reuse specifications and code, because they are more likely to be relatively free of bugs and because we are more familiar with them than with newly written algorithms.

Specifications can be viewed as abstractions of the actual implementations needed to achieve the desired input-output behavior. A good specification captures the essence of an input-output behavior, and serves as a useful interface among modules. We believe that specifications can play a key role in software engineering. However, specifications, as currently construed, are too difficult to construct and to understand.

We extend the concept of clichés beyond code clichés, to *specification clichés*: they capture the essence of typical problem specifications, and they can be re-used just like code clichés. We propose a program description language which is based on specification clichés and in which the high-level design of a program can be given. It is designed so that commonly used specifications can be concisely written in a readable and comprehensible manner. This should make validation of the specifications easier. The specification clichés provide contexts for the users to specify input-output behaviors and high-level design decisions in a concise manner. We expect that both specification clichés and code clichés should be designed in tandem, and we believe that low-level design knowledge can be used to relate the two kinds of clichés. This project is about building a computer program,

called the Design Assistant, or DA for short, which will help illustrate these ideas. Given a specification that includes a high-level design, the DA will automatically select the correct implementations using the low-level design knowledge that are codified in clichés.

Specifications, like programs, may be incomplete and inconsistent. Inconsistencies in a program are manifested through errors in the input-output behavior of the program. However, inconsistencies in specifications are not easily detectable. The specification writer needs tool support for the specification task. We adopt an assistant approach: we recognize that a lot of programming is routine and mundane, and can be automated to some extent. We aim to provide a synergistic division of labor between the programmer and the DA so as to provide the programmer with as much support as possible. The DA can provide significant help to the programmer by maintaining the design rationale of a program automatically. This will provide support for the evolutionary nature of programming: programmers may change the specifications and design criteria at will, and the relevant book-keeping will be done automatically for them.

The scope of the proposed approach is very wide. In order to make it manageable for this thesis, the scope has been narrowed down: the target users of the DA will be expert programmers, the programming tasks will be programming-in-the-small, and automatic support is focused on the low-level design of programs. Though the approach itself is inherently independent of the target source language, the DA will only produce Common Lisp code. Furthermore, the scenario is not based on any specific domain but rather on general programming clichés.

It is useful to state the key research goals of this project here:

- Explore a Cliché-Based Program Description Language: This language will allow the specifications and high-level design of commonly-written programs to be concisely described. The division of labor between the DA and its user is influenced by the nature of the description language. We would like to study issues related to cliché-based specification languages: How can they help increase the productivity of programmers? What are the desirable properties of such a language? What kind of specifications can realistically be captured by such a language? How limited is this approach?

- Provide Automatic Low-Level Design Assistance: Given some formal specifications, how can the DA make many of the low-level design choices automatically, or aid the user in making them?

- Provide Support to Detect Errors: How can the DA help catch some errors programmers are prone to make? Specifically, we expect the DA to be useful in detecting contradictions and inconsistencies in formal specifications.

- What Constitutes the DA? As currently conceived, the knowledge available in the DA resides mainly in the clichés themselves. An interesting question for this research to answer is: how much of the knowledge being used is independent of the specific clichés used? How best to represent such knowledge? If the library of clichés were removed from the DA, what kind of knowledge is left?

Once we have experience in using such a cliché-based specification language for programming-in-the-small, we intend to explore how similar approaches can be utilized in larger programming tasks. In particular, we are also interested in studying the implications of our approach to program maintenance.

2 The Design Assistant

In this section we first describe how the DA is different from its predecessor, KBEmacs, then we outline the basic approach we are taking in this thesis and the capabilities we want the DA to demonstrate. Next we describe some of the proposed mechanisms for achieving the capabilities of the DA and a brief sketch of the architecture of the DA.

2.1 DA: Successor of KBEmacs

This thesis is inspired by KBEmacs [23]. KBEmacs demonstrated the use of clichés in program construction. The DA will go beyond KBEmacs by automatically selecting the clichés which can be used to construct a program, based on the given specification clichés and design criteria [17].

As a successor to KBEmacs, the DA extends KBEmacs in the following respects:

1. Editing in terms of specification clichés instead of code clichés: The programmer need not specify specific code clichés to use in the DA. The programmer uses specification clichés to describe the specification. The DA will choose the appropriate code clichés to implement the given specification, according to the implementation guidelines the programmer sets. The programmer interacts with the DA at a higher level: at the specification level, the programmer worries about how the problem should be tackled, i.e., the high-level design, and the criteria to be used for low-level design. Some design criteria a programmer may be interested in specifying includes: time efficiency, space efficiency, inclusion of error-checking code, clarity of code, etc.

2. General-Purpose Automated Deduction: In the DA, the relationships between clichés and design concepts are explicitly represented as constraints in a general way. The DA propagates such constraints throughout the specification and reasons via such constraints.

3. Detection of Contradictions: The DA is able to detect contradictions in the specifications and design criteria given.

Conceptually, the DA will be built on top of KBEmacs. We assume the existence of a KBEmacs-like system which will help the DA output the correct code. That is, the DA will output the necessary commands for such a KBEmacs-like system to construct the program meeting the given specifications.

2.2 The Approach

Our approach consists of three basic ideas which are quite closely related. For convenience, they are discussed separately below:

1. Specification: A specification of a program can be viewed as an abstraction of the valid implementations of the program. A good specification captures most of the valid implementations. A key theme in computer science research in the past decades has been the use of abstraction as a technique to control the complexity of programming.

   Most past research on specification has concentrated on designing a wide-spectrum language based on logic and set theory. Such specification tends to be difficult to write and to understand. We propose to specify programs using a more formalized version of the familiar language that is used by programmers themselves and by computer science textbook writers. Our program description language is based on the concept of *specification clichés*: clichés that capture the typical components of a specification in some domain. Through the use of specification clichés, we hope to shorten the length of the description a specifier has to communicate to the DA. The primary advantage of this style of specification is understandability: it would be easier to write and to understand. This should enhance the specifier's ability to validate the descriptions. We strive for *specifications which are self-evidently correct.*

   We view specifications as encompassing both the abstract input-output behavior function as well as the constraints a program implementing the specification must satisfy. These constraints help programmers select appropriate data structures and algorithms for the program. An abstract specification is frequently used as a design tool: it serves as a contract between the specifier and the coder. The specifications we have in mind are also intended to be executable in the sense that if sufficient information is contained in the specifications, the DA can implement them in some executable target language.

   For the rest of this document, when we use the term *program description*, we refer to a combination of input-output behavior specification and the high-level design of a program implementing the specification. Specifically, the program description will be the the specifications accepted the DA accepts. This DA language is high-level in the sense that programmers need not bother with the detailed design. The DA will provide automatic low-level design assistance to the programmer. It will help select appropriate algorithms and data structures.

2. Clichés For Reuse: We believe that a lot of programming knowledge can be codified for reuse in terms of clichés [23]. Clichés are used to represent the prototypical concepts useful in computation. They are inspired by the analogous usage in ordinary language: the WEBSTER dictionary defines a cliché to be *a hackneyed theme or situation.* Though the ordinary meaning of cliché has a negative connotation of being overused, programming clichés are intended to be commonly used computational ideas and methods. Through their frequent use, programming clichés are familiar to

the programmer, and through this experience of use, they are more understandable and reliable.

In KBEmacs [23] clichés are conceived to be code templates with logical constraints between some of the holes (roles). Here, we extend the concept of cliché beyond code; we consider clichés that represent knowledge involved in other phases of programming: specification and design, in addition to code. In particular we are exploring the role specification clichés can play in programming.

For example, if the specification involves the *Graph* cliché, it conjures up a context in which many other ideas are meaningful because they are closely related to the graph cliché: concepts like the vertex set of a graph, the edge set of a graph, the properties of a graph such as its cyclicity, its directedness, and its rootedness. Furthermore clichés codifying operations on graphs such as finding a shortest path through a graph, finding a minimum spanning tree of a graph, and searching a graph are closely related.

We also propose to codify some implementation knowledge as relationships between specification clichés and code clichés. We postulate that simple deductions can serve as an effective *glue* for linking up the two kinds of programming knowledge.

3. Assistant: We realize that it is too difficult to fully automate the entire programming process. Instead of trying to automate the entire programming process, we intend the DA to be an assistant that can help take care of most of the mundane details of programming, leaving the expert programmer with more time for other more challenging tasks in the programming process. In this work, some of the mundane tasks include selection of data structures and algorithms, and program structuring.

   If the programmer had to explain to the assistant what needs to be done starting from first principles, it would be a very tedious task. We propose a specification language based on specification clichés that will allow for effective communication between the programmer and the DA. The shared knowledge between the programmer and the DA, as captured in a library of clichés allows commonly used concepts to be stated in a concise manner.

Through the combination of the above three ideas, we believe that the DA can achieve most of the capabilities outlined in the next subsection.

2.3 What the DA Will Do

Our primary motivation in the project is to study how a computer program can support an expert programmer in the programming process. The key capabilities the DA should possess are:

1. Support for Automatic Low-Level Design: The programmer need not specify the program to the fullest details. The DA is able to make some low-level design decisions

based on the constraints in the clichés used and explicit dataflow constraints. Based on the specifications, the DA will be able to select appropriate data structures and algorithms to implement the given specifications. The programmer may provide the DA with efficiency constraints or guidelines in addition to the specification. The DA shall choose data structures and algorithms respecting these constraints and guidelines.

2. Provide Explanation: The DA will keep track of the reasons why a particular design decision was taken so that it can provide some form of explanation to the programmer. To be user-friendly, the DA needs to structure the explanation in an understandable manner to the user.

3. Support for Retraction of Design Decisions: Specifications in the real-world evolve over time. To support program evolution, the DA will allow the programmer to retract design decisions and make changes to the specifications.

4. Handle Inconsistent and Incomplete Specifications: Specifications are inevitably incomplete and typically inconsistent. The DA will be able to deal with some of this incompleteness in specifications. It will also be able to help the programmer detect inconsistencies and where appropriate, offer fixes.

5. Order-Independent: The same resulting program will be obtained independent of the order in which the specifications are given.

The above five capabilities and the design of a program description language which allows cliché-based specifications and high-level design to be specified will be the focus of this project. Besides these, there are many other useful functions the DA should provide the programmer. In particular, support for informal specifications is a very useful function. By informal specifications, we refer to some high-level but not necessarily unambiguous language for conveying what the user wants to the system. Predicate calculus is an example of a formal language with an unambiguous semantics. A DA which supports only a formal language is difficult to use because the rigors it imposes on programmers make them "user-unfriendly". Support for informal specifications will make a formal language less formidable and more usable. However, support for informal specifications is orthogonal to the concerns of this project. It will be studied if time permits. We envision our specification language to be a formal language with unambiguous semantics.

2.4 Building the DA

A key component of the DA will be CAKE [16], a truth maintenance system (TMS). CAKE will provide the infrastructure on which the needed programming knowledge is represented and reasoned about.

CAKE can be used to propagate constraints and to maintain the resulting dependency structures. These dependency structures are useful in explanation generation. It also

supports automatic retraction of assertions and their consequents. The constraints are encoded in propositional calculus. As complete propositional reasoning is computationally intractable, CAKE sacrifices completeness for efficiency. Deductions in CAKE are designed to terminate quickly, i.e., CAKE supports quick but shallow deductions.

CAKE also has builtin support for reasoning about algebraic properties of operators, equality, types and frames. It also supports a pattern-directed demon invocation mechanism which can be used to implement some controlled reasoning of quantified terms.

Some sources of programming knowledge to be codified in the DA are expected to include: specification, program structuring, selection of algorithms and data structures, explanation-generation, and program optimization. The specific knowledge will be indicated in the scenario sketch in a later section where appropriate.

Most of the knowledge involved in the scenario will be codified as clichés. Clichés will be represented in CAKE as frames. The slots in a frame corresponds to the roles of a cliché which can be filled by the programmer. The type system in CAKE will be useful for representing the relationships between the different clichés. Some constraints in the domain may be represented as constraints between the parts of the clichés and other clichés. The completeness of a specification can be checked based on whether all mandatory roles of a cliché are filled. With CAKE's support for algebraic reasoning, some useful algebraic properties of programs and specifications can be conveniently codified and reasoned about in CAKE.

2.5 Architecture of the System

The DA has two main modules: the first module is called DM (design module) and the second SM, the Synthesis Module. The SM will accept KBEmacs-like commands from the DM and output code which corresponds to the commands invoked. Some of the commands to be supported by the SM are: instantiating a named cliché, filling a named role with a cliché instance, and combining clichés appropriately when needed. The DM will be the "brain" of the system. The SM may be thought of as a sophisticated pretty printer: it prints out the program fragments in clichés appropriately. It does not do any interesting reasoning beyond what is explicitly given in the cliché definitions.

It is beyond the scope of this project to build the SM. We assume such an SM exists, and concentrate our efforts on building the DM. It is also beyond the scope of this project to build a complete prototype. The scenario sketch is meant to illustrate some of the capabilities the DA will have.

3 Target Scenario

The following scenario shows how a programmer can use the DA to develop a program for text-justification. The objective of a text-justification program is to break up paragraphs

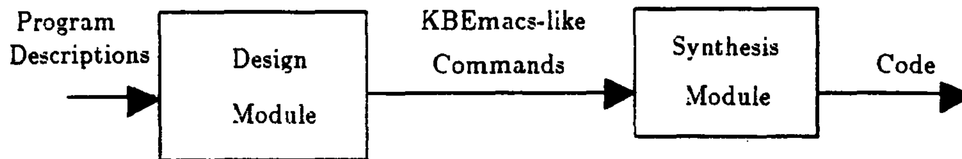

Figure 1: The Architecture of The Design Assistant

into lines as evenly as possible: to avoid narrow, cramped lines as well as to avoid widely-spaced, loosely-set lines. In this scenario, it is assumed that the DA does not know about text-justification. It knows about some standard programming clichés and some graph algorithms. The example program is chosen because we think it is a realistic and nontrivial program. There is sufficient complexity in the specification of the problem to challenge both the program description language as well as the synthesis of the program. Also, the clichés that are used to motivate this example are commonly used concepts, familiar to many people with a programming background.

Below, we first provide a brief description of the clichés and the kinds of knowledge we expect the DA to know about. Then we describe the text-justification problem and its solution. Next, we sketch the scenario in which a description of the solution to the text-justification problem is given to the DA.

3.1 What the DA Knows

The DA is expected to know various pieces of knowledge about programming. Specifically, the following clichés and functions are available to the DA: Tokenize, Segment, Build-Graph, Single-Source-Shortest-Path, Prorate, Ascii-File, Output. These general clichés are described briefly below. They will be described in more details where they are used.

- Tokenize is a cliché that converts a sequence of characters into a sequence of tokens. The various types of tokens to be produced are specified using regular expressions. A few auxiliary concepts and functions are also present in the Tokenize cliché: the Content of a token (which is a sequence of characters), the Type of a token, the set of all tokens of a specified pattern type, the Preceding-Token of a token in the sequence, etc.
- The Segment cliché takes a sequence of items and breaks it up into a sequence of sequences of items. This cliché handles the beginning token and the end token of

the sequence as boundary tokens when needed. This is a useful cliché in segmenting sequences, because it encapsulates some details about how to handle the endpoints appropriately.

- The Build-Graph cliché captures the concept of building a graph from some given inputs. There is a family of clichés that embodies some standard ways of constructing graphs. In general the programmer has to fill in the roles in the top Build-Graph cliché which specify how nodes and arcs are computed from a set of potential nodes. In more specific cases, less information is required because the cliché contains sufficient knowledge to complete the rest of the implied program descriptions. Among the many roles Build-Graph has, there is the Input-To-Node role which expects a function that maps the input items to the nodes of the output graph. Another role is the Arc-Test role, it is to be filled in by a predicate which decides whether two given input items should be connected in the output graph. The Node-Test role is analogous to the Arc-Test role but works for nodes instead. There are many different roles in the Build-Graph cliché because there are many ways of building a graph. Different graphs motivate different graph construction procedures which are captured in different build-graph clichés. Not every role is shared by the members of the family of Build-Graph clichés. For a given graph, not every role need be specified.

- The Single-Source-Shortest-Path cliché takes in a graph and runs a single source shortest path algorithm on the graph. The output of algorithm may be the length of the shortest path or the shortest path itself, or both. There are a few different shortest path algorithms, including those by Dijkstra, Bellman-Ford, and one for directed acyclic graphs [5]. For some of these algorithms, there are different data structures which can be used and they may yield different running times. The output of the algorithm may also be represented in different ways. This cliché captures some of the knowledge involved in selecting the appropriate algorithm based on the characteristics of the input graph.

- The Prorate cliché codifies the concept of proration. A family of related Prorate clichés embodies the different ways of prorating an amount among a set. In the simplest case, Prorate prorates an amount among the members of a set. Its default is to prorate the amount evenly among all members of the set. However, the Prorate cliché also handles the case when the proration should be done proportional to some function of the members, and when the prorated amount is to be incorporated into some given function of the members.

  In fact, there are some subtleties involved in the seemingly simple proration cliché, e.g., when the total shares of the members does not divide the amount to be prorated: where should the "remainder" go? It could be distributed at random, or in the case of a given ordered set, it could be distributed so as to favor earlier elements or later elements, or to favor elements from both ends. The cliché abstracts from the detailed proration computation and allows the programmer to think in terms of the abovementioned intermediate-level concepts. Prorate also obeys the *Non-Shareholders get nothing* principle: it takes care to prorate only among the members

of the given set which have non-zero shares in the proration process. In addition, some proration constraints are explicitly kept: there can be no negative shares. Here, we restrict the variations of Prorate to those that run in linear time. More sophisticated schemes of proration that involve sorting may take longer.

- Ascii-File: If a file is an Ascii file, it contains only characters belonging to the Ascii character set. The cliché encodes knowledge about the Ascii characters; for example, that there are 128 characters in the Ascii character set, and that the space character and the return character belong to the blank-printable character subset. Familiar concepts like the non-blank printable character set, the printable character set, and the control character set are codified in this cliché.
- The Output cliché puts out characters and strings to the given output file or stream depending on the kind of input it receives.

3.2 Background Description of Text-Justification

Our text-justification example is inspired by the way the TEX program breaks a paragraph into lines [11]. As a document typesetting program, TEX performs substantially more functions than text-justification. This example adopts TEX's model of paragraph typesetting but omits much of its complexity.

A paragraph is viewed as a sequence of word tokens and gap tokens. Word tokens have fixed widths, whereas the size of gap tokens may be stretched to fill up any slack. Each gap token has a width property and a stretchability property. Each gap token in the sequence is a potential point to break the current line. The word tokens and gap tokens between two adjacent breakpoints form a line.

A graph is constructed from the sequence of tokens that represents the paragraph as follows: a gap token is added to the beginning and the end of the sequence (assuming that the sequence does not start with a gap token). These two gap tokens are breakpoints in the final typeset paragraph. We want to construct the arcs of the graph in such a way that paths from the start token to the end token represent potential line breaks of the paragraph.

We can define a metric on any two input gap token in the token sequence. For efficiency, we make arcs between the nodes corresponding to the gap tokens only if the metric on the two gap tokens is within some thresholds. Now, suppose we have chosen the metric such that it is high when the lines are loosely set, and that it is low when the lines are close to the normal (optimal) settings. Then, we can view the problem of typesetting the paragraph as an instance of the shortest path problem. The path from the start token to the end token that has the smallest metric corresponds to the best way to break the paragraph into lines.

Specifically, we can define the metric to be the number of extra spaces to be distributed among the avialable gaps. The ideal situation is when the metric is 0 for all the

lines. Since we cannot shrink the spaces in our model, this becomes the lower bound of the threshold on the metric. The upper bound represents the maximum number of extra blank spaces we will allow per gap in a typeset line. To compute this metric, we need to compute the total number of gap tokens and the leftover spaces after the line length has been substracted from the mandatory word-separator spaces and the words themselves in a line.

Shortest path problems have standard efficient algorithms. In particular, the graph which arises from the text justification model turns out to be a rooted, directed, acyclic, and forward-wrt-input graph. (By a forward-wrt-input graph, we mean a graph constructed from some input sequence whose arcs have nodes which respect the order of the nodes in the input sequence, i.e., if edge (u,v) is the graph, then u came before v in the input sequence.)

After the shortest path has been computed, each arc in the path represents a line. For each line, we can proprate the extra spaces among the gap tokens and output the words and gaps in order.

In the scenario, we assume fixed width fonts. Hence gap tokens cannot be shrunk and they can only be stretched by fixed quanta.

3.3 Scenario

To aid in the viewing of the scenario, the following typographical conventions are used: keywords of the program description language are in uppercase, cliché names are in initial capitals, and changes in the input description and the output code between the steps of the scenario are in uppercase.

The semantics of the various operators in the program description language are given below:

- (`LET` ({(Variable Value)}*) {Form}*): This creates a local scope in which local variables can be named and manipulated.
- (`DO` ({Variable} {Sequence}) {Forms}*): This is an explicit iteration construct. The {Forms} are ordered.
- (`DESIGN` {Name} ({Variable}*) {Forms}*): This form defines the *Name* logical function, which takes the given argument list, to be the forms. {Forms} is a sequence of statements and/or assertions.
- (`REDESIGN` {Name} ({Variable}*) {Forms}*): This is identical to `DESIGN` except that it indicates to the DA that *Name* has been defined before, and that the current definition supersedes the previous one.
- (`WITHIN` {Name} ({Variable}*) {Forms}*): This form is used to provide additional information about a cliché used elsewhere in the program description. The optional

argument list {Variable}* allows the arguments passed to the use of the *Name* cliché to be used locally within the current description. {Forms} is a sequence of assertions.

- (**MODIFY-WITHIN** {Name} ({Variable}*) {Forms}*): This form is identical to **WITHIN** except that it indicates to the DA that the instance of *Name* used earlier is being modified.

- (Predicate &rest Variables): This is a logical assertion that says that the proposition (Predicate Variables) is true. When the *Predicate* is a cliché known to the DA, this form instantiates an instance of the cliché with the given *Variables* as arguments. The output of this form is the output of the cliché instance, if any.

- (IS Role Form): This asserts that the Role should be filled in with Form. This form occurs inside the **WITHIN** form, the role is interpreted with respect to the cliché described by the **WITHIN** form.

- (== Form-1 Form-2): This asserts that Form-1 is logically equivalent to Form-2. However, the operator == may also used to define specific relationships in some clichés; such uses will be pointed out later.

- (**INSTANCE** {type} {Keyword Form}*): This creates an object of the given type. The functions which can be applied to the object are given by the keywords and the values of the respective applications are given by the forms corresponding to the keywords.

We would like to support a language for program description that is close to the English language. However, the scope of this thesis does not allow for such exploration. Hence, the above Lisp-like description language is used. The exact form of the language is not important to the tasks at hand. Some means for the programmer to specify the clichés to use, some means to fill in the roles of the clichés, and some means to specify some logical constraints should suffice here.

3.4 Scene 1: Initial Program Description for Justify

The programmer provides the DA with the program description in Figure Description.1.

The (incomplete) program description in Figure Description.1 mirrors some of the English description an expert programmer could have provided to a junior programmer. The first description tells the DA that the Justify function takes an input file and an output file, both of which are Ascii files. Justify first tokenizes the input file, then segments the token sequence into subsequences separated by para-breaks. For each of these subsequences, called a paragraph, it builds a graph, and runs the single source shortest path on the graph. Each arc on the resulting optimal path is output to the output file in turn.

```
(DESIGN Justify (infile outfile)
  (Ascii-File infile) (Ascii-File outfile)
  (DO (paragraph (Segment (Tokenize infile) 'para-break))
    (DO (line (Single-Source-Shortest-Path (Build-Graph paragraph)))
      (lineout outfile line))
    (Output outfile #\newline)))

(DESIGN Lineout (outfile line)
  (Prorate (extras line) (gap-set line))
  (DO (token line)
    (Output outfile (Content token)))
  (Output outfile #\newline))

(IMPLEMENTATION-GUIDELINES
  (PREFER 'time-efficiency)
  (IGNORE 'error-checking))
```

Scene Description.1:: Initial Program Description of Justify.

In the use of Tokenize in the description, `infile` is viewed as a sequence of characters. The mention of the Tokenize cliché indicates to the DA that the output from `(Tokenize infile)` is a sequence of tokens. The above design specifies that this sequence is to be segmented at para-breaks (tokens). At this point, para-break is not yet defined. It will become defined when the detailed program description of the Tokenize cliché is given. The Build-Graph cliché, as used above, is given a sequence of tokens as input from which an output graph is computed. The design says that a single source shortest path algorithm should be run on the graph. The output of the single source shortest path algorithm may be the distance of the shortest path or the shortest path itself, or both. Since the output is used in an iteration construct in the given design, it clearly cannot be the cost of the optimal path. Hence the DA assumes that the result to be returned is an optimal path in the graph.

The `lineout` design says that the Extras of the given line are first to be prorated among the `gap-set` of the given line. At this point, the functions `extras` and `gap-set` are not yet defined. After the proration, the tokens on the line are output to `outfile` in order, and each line is ended by a newline character.

The programmer also gave the DA some implementation guidelines: the program should be efficient in its running time and error-checking should be ignored.

The DA is unable to generate any code based on the above description because it is incomplete. When the programmer requested the DA to produce code, the DA informs the programmer accordingly as shown in Figure Dialog.1.

The model of programming we have in mind is guided by the interaction between human programmers, a model we call program description elaboration. An expert programmer typically describes, using some intermediate level vocabulary, to the junior programmer what is needed. Such descriptions may include a high-level design of a program intended to meet the specification. The junior partner may ask the senior programmer questions to clarify doubts (manifested as inconsistencies and incompleteness in the case

```
>>> (Write-Code 'Justify)
DA> Insufficient Information given.  Unable to select implementations
for the following cliches: Tokenize, Segment, Build-Graph,
Single-Source-Shortest-Path, Prorate.
WARNING: The following are undefined: para-break, gap-set, extras.
```

Scene Dialog.1:: Dialog between the DA and the Programmer.

of the DA). They collaborate in the programming task in a co-operative manner. The descriptions need not be strict refinement of program descriptions already said. They could be looser in the sense that earlier program descriptions may be retracted, modified, superseded, redefined, and re-stated. Viewed as a structured natural language discourse, our model of program description elaboration encompasses the specification refinement approach and goes beyond it.

3.5 Scene 2: Elaborating the Program Description of Justify

Now, the programmer provides more detailed information to the DA. The additional descriptions are shown in Figure Description.2.

```
(REDESIGN Lineout (outfile line)
  (Prorate (Extras line) (Gap-Set line))
  (DO (token line)
    (IF (GAP-TOKEN-P TOKEN)                              ; NEW
        (OUTPUT OUTFILE #\SPACE :NUMBER (WIDTH TOKEN))   ; NEW
        (Output outfile (Content token))))
  (Output outfile #\newline))

(WITHIN Tokenize
  (== para-break
      (regular-expression
        "BOF (#\space | #\newline)* | (#\space | #\newline)* EOF |
        (#\space)* #\newline (#\space)* #\newline (#\space | #\newline)*"))
  (== word (regular-expression "(Non-Blank-Printable-Characters)+"))
  (== gap (regular-expression "(#\space | #\newline)+")))

(WITHIN Build-Graph (paragraph)
  (LET ((new-paragraph (Concatenate Head paragraph Tail))
        (M Input-To-Node-Mapping))
    (IS (Domain M) (Gap-Set new-paragraph))
    (Directed graph) (Forward-Wrt-Input graph) (IS Root (M Head))
    (IS Arc-Test
        (Lambda ((xn token) (yn token))
          (And (Node (M xn))
               (>= (ratio (Subsequence new-paragraph xn yn)) 0)
               (<= (ratio (Subsequence new-paragraph xn yn))
                   *Threshold*))))))

(DESIGN Ratio (seq) (/ (extras seq) (total-stretch seq)))
(DESIGN Extras (seq) (- *Line-Length* (total-length seq)))

(DESIGN Total-Stretch (seq) (Sum (Map #'stretch seq)))
(DESIGN Total-Length (seq) (Sum (Map #'width seq)))

(DESIGN Stretch (token) (if (Gap-Token-P token) 1))
(DESIGN Width (token)
  (cond ((Word-Token-P token) (Number-Of-Characters token))
        ((Gap-Token-P token)
         (If (Member (Last-Character (Preceding-Token token))
                     '(#\! #\? #\.))
             2 1))))

(== Head (INSTANCE 'gap :width *Para-Indent* :stretch 0))
(== Tail (INSTANCE 'gap :width 0 :stretch 10000))
```

Scene Description.2: Additional Program Descriptions of Justify.

In Lineout, the programmer distinguishes between two kinds of tokens: gap tokens and non-gap tokens. For gap tokens, the number of space characters to be output is equal to the width of the token. This is intended to be used after the width of the token has been processed. In Tokenize, the programmer defines the patterns to be scanned with the help of the == operator. BOF and EOF are markers for the beginning and the end of

file. Para-breaks are blanks with at least two newlines, and the beginning and the end of a file are also para-breaks. (Blanks are either spaces or newlines.) Words are any non-empty sequence of non-blank printable characters. Any non-empty sequence of blanks is a gap. Once these patterns are defined, a number of auxiliary functions are automatically constructed from the tokenize cliché: the **Gap-Set** is now understood to be the set of gap tokens, and **Para-Break-Set** and **Word-Set** are similarly defined.

The programmer specified, in **Build-Graph**, that a new paragraph is to be formed from the old one by prefixing it with a leading gap, **Head**, and suffixing it with a trailing gap, **Tail**. **Head** serves as the root of the graph to be constructed from the new paragraph. By making the width of **Head** equal to the desired paragraph indentation, kept in the global variable, ***para-indent***, it can also handle paragraph indentation. To ensure that the last line of a paragraph is broken properly the **stretch** of **Tail** is set to infinite (10000 here simulates infinity). This will force the algorithm to break the paragraph at **Tail**. For convenience, the programmer calls the **Input-To-Node-Mapping** role expects **M**; it should be noted that the function that is used to fill this role is by default partial. This is so because in general we do not expect all inputs to be mapped onto nodes. From the text-justification domain, it is known that the resulting graph is a forward graph with respect to the input: all arcs point forward with respect to the order of the input sequence. The programmer also specifies that the domain of **M** is the gap set in the new paragraph. Finally the programmer gave the DA an explicit test for arcs given two input tokens. The test says that if the first argument token, **xn** has a corresponding node **(M xn)** and if the **ratio** of the subsequence between the the two argument tokens is between 0 and ***threshold***, then there exists an arc from **(M xn)** to the node corresponding to the second argument token. Note that this implies that the second argument token, under **M**, is mapped to a node.

The programmer also specifies how the various functions should be computed. The total length of a sequence is simply the sum of the widths of the sequence. Similarly, the total stretch of a sequence is the sum of the stretches of the sequence. The stretch of a gap token is defined to be 1. The width of a word token is just the number of characters in the token, and the width of a gap token is 2 if the previous token in the sequence ends the sentence, and 1 otherwise.

With the additional information, the DA proceeds to analyze the descriptions. At this point, Para-Break has been defined in the Tokenize cliché as a particular type of token, and the Extras function has been explicitly defined. The interaction that follows is given in Figure Dialog.2.1.

The DA noted that the newline character found in the regular expressions of Tokenize is not an Ascii character. The newline character is a Common Lisp character which is frequently used to denote the sequence of Ascii characters: Linefeed Return or the Ascii Return character. These methods of recovering from the constraint violation are codified in the Ascii-File cliché. The programmer chooses to define the Newline character to be synonymous with the Return character.

Next the DA notes that the given description of Tokenize is incomplete because it

```
DA> Constraint Violation: The #\newline character is not an Ascii
character.  Note: The following are standard ways of defining the
#\newline character in the Ascii character set:
  (1) Define #\newline to be #\linefeed followed by #\return.
  (2) Define #\newline to be #\return.
>>> Use(2).

DA> Constraint violation: The characters possible in the input to
Tokenize must be spanned by the characters in the definitions of token
types.  How should the remaining Ascii characters be handled?
Note: The following are standard ways of handling the remaining Ascii
characters:
  (1) Ignore them.
  (2) Treat them as #\Space.
  (3) Flag them as errors.
>>> Explain.
DA> The characters possible in the input to Tokenize must be spanned by
the characters in the definitions of token types because
  (a) The characters possible in the input to Tokenize is the
Ascii character set.
  (b) The characters in the definitions of token types in Tokenize consists
of the Non-Blank-Printable-Characters plus #\Newline plus #\Space.
  (c) "Infile" is an Ascii file.
  (d) An Ascii file contains the Ascii character set.
  (e) The Ascii character set is not spanned by the
Non-Blank-Printable-Characters plus #\Newline plus #\Space.
>>> Use(2) ; Treat them as #\space.
DA> In (Sum (Map #'Stretch seq)), the function Stretch is not defined
for word tokens.  Please define Stretch for word tokens:
>>> 0.
DA> Constraint Violation: Non-Unique Identifier used when Unique
Identifier is needed: (Index Head) = (Index Tail).
>>> Explain.
DA> Non-Unique Identifier used when Unique Identifier is needed because:
  (a) (Index Head) = (Index Tail).
  (b) The chosen algorithm for Build-Graph requires that the tokens be
identified by some unique identifier.
  (c) The Index of a token in the token sequence has been chosen to
serve as the unique identifier.
>>> (And (= (INDEX Head) 'Head) (= (INDEX Tail) 'Tail)).
DA> In Single-Source-Shortest-Path, what should the source node, the
destination node and the distance function be?
>>> (WITHIN Single-Source-Shortest-Path
      (IS Source Head) (IS Destination Tail)
      (IS Distance-Function #'Ratio))
DA> WARNING: Ratio is undefined when Total-Stretch = 0.
>>> (REDESIGN Ratio (seq)
      (if (= (Extras seq) 0) 0
          (if (= (Total-Stretch seq) 0) 10000
              (/ (Extras seq) (Total-Stretch seq)))))
DA> WARNING: The output of (Prorate (Extras line) (Gap-Set line)) is
not used.  By default, the output of Prorate is a list of pairs (element
prorated-amount).
>>> Postpone.
```

Scene Dialog.2: Dialog between the DA and the Programmer.

does not handle all Ascii characters. This violation arose from the interaction of con-

straints which came from two sources. First, the Ascii-File cliché has a constraint that defines the Ascii character set. Second, the Tokenize cliché imposes a constraint that the character set has to be fully accounted for, and it contains knowledge about how the remaining characters may be handled. The programmer then asked the DA to explain the question posed. Here, the DA showed its explanation capability: it is able to pull out the dependencies supporting the different assertions that led to the constraint violation at hand.

Next the DA notes that the Stretch function was not defined for word tokens but was used in the Sum function. The programmer told the DA to define it to be the zero function.

In analyzing the Build-Graph cliché, the DA chooses an implementation that suits the given characteristics of the graph to be output. It found that one of the assumptions needed in the chosen implementation was that there must be a way to assign a unique identifier to tokens. The DA chose the position of the token in the token sequence, which the **Get-Token** function guarantees to be unique. However, the **index** of **Head** and **Tail**, explicitly created by the programmer were given the value nil by default, because their **Index** fields were not given in their definitions. The DA warned the programmer accordingly about what it found. The programmer changed the descriptions so that **Head** and **Tail** have different indices.

The DA also noticed that the Ratio description as currently given is undefined if the total-stretch of the sequence is 0. If it could prove to itself that Total-Stretch of the sequences passed to it is never 0, then it would not have bothered the programmer with such a warning. Here, since it is unable to do so, it reminded the programmer of a commonly-known potential pitfall. The programmer resolved the warning by defining the function to have a value of 0 when Extras = 0, and to have a large value of 10000 when Extras is not 0 and total-stretch is 0. The first case allows for the situation when a long word takes up the entire line, and the second case prevents the build-graph algorithm from building arcs with non-zero Extra spaces and no gaps to which to distribute the spaces.

The DA noted that the Single-Source-Shortest-Path description is incomplete since the source node, the destination node and the distance function to be minimized were not given. The programmer then provided the DA with the needed information.

The DA also issued a warning when it detected that the output of Prorate was not used in the description. The DA is following a general heuristic: an unused output is indicative of an omission in the program description. The programmer decided to defer the warning at this point. The DA will remember the warning so the programmer can get back to it. The DA is designed so that the programmer can steer the dialog in whatever direction the programmer wishes.

Based on the additional descriptions given, the DA generated the code shown in Figure Code.2.1 to Figure Code.2.2. For the sake of clarity, the code for **Get-Token** is not shown. Note that this could be built by a popular Unix utility, Lex. The scenario

will rely on the **Get-Token** abstraction to return the next token from the given stream. When the last token has been extracted, **Get-Token** returns Nil. **Initialize-Reading** is needed to get the stream ready for tokenization. A subtle point in the high-level design of the program should be noted here: the last token on a segment is given a Stretch of 10000 so as to force the algorithm to make it a breakpoint. Prorate will give such a gap token all the slack.

Given the program descriptions, there are many implementation choices left to the DA. In the following, some of the choices available to the DA are described and the reasons why the DA picked a particular implementation are explained.

1. Tokenize Cliché: The specification cliché for Tokenize allows the programmer to define the patterns to be looked for in the input sequence of characters. It has knowledge about how to implement a tokenize procedure. For simplicity, we are not showing the various ways tokenization can be achieved in the current scenario. We assume a non-deterministic finite state automata implementation of tokenization.

   Independent of the tokenization algorithm used, there are lower-level design choices that need to be made before a program can be constructed. For example, the physical representation of a token may vary. The sequence of tokens can also be represented in many ways: it could be implemented as a list, a vector, a doubly-linked list, a pointer chain, or kept implicit in some iteration structure.

   In the scenario, the DA chose to represent a token, by default, as a Lisp structure. A pointer chain is used to represent the sequence of tokens since such a data structure is good when there are concatenations of tokens from both the front and the back of the sequence which is present in the scenario.

2. Functions on Tokens: From the program descriptions, the DA has the signatures of the various functions given: Width and Stretch. They may be represented explicitly either locally, as a field on the structure of the token, or computed procedurally on demand.

   Here, the DA implemented functions on tokens by making them be fields of the token structure.

3. Sets: Sets can be represented in many ways, among the more popular ones include: lists, vectors, and hash tables. Some sets involved in the program descriptions are the set `'(#\! #\? #\.)`, the set of tokens associated with each line, the set of arcs of the graph, and other sets used in local computations.

   Most of the sets used in the computation were implemented as simple linked lists, by default.

4. Functions on Arcs: These include: Source, Destination, Extras, and Ratio. Like the functions on tokens, these may be computed on demand or represented explicitly.

   The DA chose to represent an arc as a Lisp structure and the functions on arcs as fields of the structure, similar to the implementation of the token. Note that since

Total-Length and Total-Stretch of an arc is not needed in later computation, they are computed on demand.

5. Build-Graph Cliché: The Build-Graph specification cliché allows the programmer to describe the abstract properties of the graph desired and of the input, in addition to the specific roles needed to construct such a graph. Based on the property of the given Arc-Test, and the special properties of the graph to be built, the Build-Forward-Wrt-Input algorithm was picked. This algorithm however leaves open the implementation of the graph. The DA then used some codified knowledge about graph construction: the implementation of graphs is determined/influenced by the specific processing to be performed on them. In the description, the DA is told that a single-source shortest path algorithm is to be run on the resultant graph from Build-Graph. Shortest path algorithms typically relaxes the edges of the graph in some order or uses the adjacency list information of nodes. At this point, the DA left the implementation choice of the graph open: it could be an arc set or an adjacency list since they are equally efficient.

6. Single-Source-Shortest-Path Cliché: As described in an earlier subsection, there are many different shortest path algorithms applicable here. In the scenario, the DA selected an algorithm that is suitable for directed acyclic graphs (dags), which is the fastest applicable algorithm. This specific algorithm requires that the arcs of the graph be topologically sorted first before the relaxation step. Here the best choice the DA found is one where the graph is implemented as an ordered arc set. In such a case, the arc set is already topologically sorted by the Build-Graph algorithm chosen, and hence the topological sorting step in the shortest path algorithm for dags can be omitted. The DA can achieve this because in the shortest path cliché, the sub-components of the shortest path cliché are identified and their relationships explicitly given.

7. Output Cliché: The implementation of the Output cliché depends on what is to be output. In the scenario, it is a simple call to the `Write-Char` Lisp function since the thing to be output is a newline character.

8. Program Structuring Choices: After the key code clichés have been chosen, there are many ways in which they can be presented to the programmer. Appropriate modules may be abstracted and made into subroutine calls, while others may be inline-substituted for efficiency.

```
(defstruct (Token (:conc-name nil)) index type content width stretch next)

(defstruct (Bkpt (:conc-name nil)) index1 token link)

(defstruct (Stuff (:constructor make-stuff (ratio extras))) ratio extras)

(defstruct (Arc (:constructor make-arc (source destination ratio extras)))
  source destination ratio extras)

(defun Justify (infile outfile)
  (with-open-file (instream infile :direction :input)
    (with-open-file (outstream outfile :direction :output)
      (let ((current-token nil) (previous-token nil) (next-token nil)
            (head (make-token :type 'gap :width *para-indent* :stretch 0
                              :index 'head)))
        (initialize-reading)
        (setq previous-token head)
        (setq current-token (remove-para-breaks instream))
        (setf (next previous-token) current-token)
        (loop
          (when (null current-token)
            (if (eq previous-token head) (return)
                (let ((tail (make-token :type 'gap :width 0 :stretch 10000
                                        :index 'tail)))
                  (setf (next previous-token) tail)
                  (process-paragraph head outstream) (return))))
          (setq next-token (get-token instream))
          (setf (next current-token) next-token)
          (cond ((word-token-p current-token)
                 (setf (width current-token) (length (content current-token)))
                 (setf (stretch current-token) 0))
                ((gap-token-p current-token)
                 (setf (stretch current-token) 1)
                 (if previous-token
                     (if (member (last-char (content previous-token))
                                 (list #\! #\? #\.))
                         (setf (width current-token) 2)
                         (setf (width current-token) 1))))
                ((para-break-token-p current-token)
                 (if (and next-token (para-break-token-p next-token))
                     (setq next-token (remove-para-breaks instream)))
                 (let ((new-head (make-token :type 'gap :width *Para-Indent*
                                             :stretch 0 :index 'head))
                       (tail (make-token :type 'gap :width 0 :stretch
                                         10000 :index 'tail)))
                   (setf (next previous-token) tail)
                   (process-paragraph head outstream)
                   (setq head new-head)
                   (setf (next head) next-token)
                   (setq current-token new-head))))
          (setq previous-token current-token)
          (setq current-token next-token))))))

(defun Word-Token-P (token) (equal (type token) 'word))

(defun Gap-Token-P (token) (equal (type token) 'gap))

(defun Para-Break-Token-P (token) (equal (type token) 'para-break))
```

Scene Code.2.1: Initial code created by the DA (part 1).

```
(defun Remove-Para-Breaks (instream)
  (let ((token (get-token instream)))
    (loop
      (if token
          (if (not (para-break-token-p token)) (return token))
          (return))
      (setq token (get-token instream)))))

(defun Last-Char (string) (car (last (coerce string 'list))))

(defun Process-Paragraph (s outstream)
  (let* ((opt-path (multiple-value-call #'compute-sssp (build-graph S)))
         (token-set nil))
    (dolist (arc opt-path)
      (setq token-set (token-set arc))
      (prorate (arc-extras arc) (gap-set token-set))
      (dolist (token token-set)
        (if (gap-token-p token)
            (print-n-spaces outstream (width token))
            (write-string (content token) outstream)))
      (write-char #\newline outstream))
    (write-char #\newline outstream)))

(defun Compute-Sssp (root-and-sink node-set-size arc-set)
  (when arc-set
    (let* ((root (first root-and-sink)) (sink (second root-and-sink))
           (distance-array (make-array node-set-size))
           (misc-array (make-array node-set-size))
           (best-previous-array (make-array node-set-size))
           (source-node nil) (destination-node nil) (destination-index 0)
           (distance-via-node 0))
      (dotimes (i node-set-size) (setf (aref distance-array i) *infinity*))
      (setf (aref distance-array (index1 root)) 0)
      (dolist (arc arc-set)
        (setq source-node (arc-source arc))
        (setq destination-node (arc-destination arc))
        (setq distance-via-node (+ (aref distance-array (index1 source-node))
                                   (arc-ratio arc)))
        (setq destination-index (index1 destination-node))
        (if (> (aref distance-array destination-index) distance-via-node)
            (setf (aref distance-array destination-index) distance-via-node
                  (aref misc-array destination-index)
                  (make-stuff (arc-ratio arc) (arc-extras arc))
                  (aref best-previous-array destination-index) source-node)))
      (let* ((previous sink) (optimal-arc-set nil) (stuff nil))
        (loop
          (if (eq previous root) (return optimal-arc-set))
          (setq destination-index (index1 previous))
          (setq destination-node previous)
          (setq previous (aref best-previous-array destination-index))
          (setq stuff (aref misc-array destination-index))
          (push (make-arc previous destination-node (stuff-ratio stuff)
                          (stuff-extras stuff))
                optimal-arc-set))))))

(defun Print-N-Spaces (stream n)
  (do ((i N (- i 1)))
      ((< i 1) (return))
    (write-char #\space stream)))
```

Scene Code.2.2: Initial code created by the DA (part 2).

```
(defun Build-Graph (v)
  (let ((root nil) (position 0) (queue nil) (end-of-queue nil)
        (node-table (make-hash-table)) (current-token v) (node-number 0)
        (total-length 0) (total-stretch 0) (queue-token nil) (arc-set nil)
        (queue-node nil) (extras 0) (ratio 0) (current-node nil))
    (setq root (make-bkpt :token current-token :index1 node-number))
    (incf node-number)
    (setq queue root end-of-queue root)
    (setf (gethash 0 node-table) root)
    (setq current-token (next current-token))
    (loop
      (if (null current-token)
          (return (values (list root end-of-queue)
                          (+ node-number 1) (reverse arc-set))))
      (setq position (index current-token))
      (when (gap-token-p current-token)
        (setq queue-node queue)
        (loop
          (if (null queue-node) (return))
          (setq queue-token (token queue-node))
          (if (eq queue-token current-token) (return))
          (setq total-length (sum queue-token current-token :field #'width))
          (setq total-stretch (sum queue-token current-token :field #'stretch))
          (setq extras (- *Line-Length* total-length))
          (setq ratio (/ extras total-stretch))
          (when (and (>= ratio 0) (<= ratio *Threshold*))
            (setq current-node (gethash position node-table))
            (when (null current-node)
              (setq current-node (make-bkpt :token current-token
                                            :index1 node-number))
              (setf (gethash position node-table) current-node)
              (incf node-number)
              (setf (link end-of-queue) current-node)
              (setq end-of-queue current-node))
            (push (make-arc queue-node current-node ratio extras) arc-set))
          (setq queue-node (link queue-node))))
      (setq current-token (next current-token)))))

(defun Prorate (amount set &key from-right)
  (when (and set (not (= amount 0)))
    (let ((total-shares 0))
      (dolist (node set) (incf total-shares (stretch node)))
      (if from-right (setq set (reverse set)))
      (let* ((amount-per-share (/ amount total-shares)) (unallocated amount))
        (mapcar
          #'(lambda (node)
              (let ((share (stretch node)))
                (if (<= share 0) (list node 0)
                    (let ((allocation
                            (if (= total-shares share) unallocated
                                (min (ceiling (* amount-per-share share))
                                     unallocated))))
                      (decf unallocated allocation)
                      (decf total-shares share)
                      (list node allocation)))))
          set)))))
```

Scene Code.2.3: Initial code created by the DA (part 3).

```
(defun Token-Set (arc)
  (let ((set nil) (begin-token (token (arc-source arc)))
        (end-token (token (arc-destination arc))))
    (loop
      (when (eq end-token begin-token)
        (push begin-token set) (return (reverse set)))
      (push begin-token set)
      (setq begin-token (next begin-token)))))

(defun Gap-Set (token-set)
  (let ((set nil))
    (dolist (token token-set (reverse set))
      (if (gap-token-p token) (push token set)))))

(defun Sum (head tail &key field (begin t) (end t))
  (let ((total 0) (head-weight (funcall field head)) (end-weight 0))
    (cond ((eq head tail) (if (or begin end) head-weight 0))
          (t (setq head (next head))
             (loop
               (when (eq head tail)
                 (setq end-weight (funcall field head))
                 (if begin (incf total head-weight))
                 (if end (incf total end-weight))
                 (return total))
               (incf total (funcall field head))
               (setq head (next head)))))))
```

Scene Code.2.4: Initial code created by the DA (part 4).

3.6 Scene 3: Preventing Errors

In this scene, the programmer resolves the warning that was deferred earlier. In Figure 3.S1 the programmer changed the descriptions so that the prorated amount is added to the width of the gap tokens, and that the proration is done proportional to the stretch of the tokens. Note that the programmer resolved the warning indirectly when Prorate was changed so that it side-effected the width of tokens; the DA deduced from the dataflow implicit in the descriptions that the widths of tokens are used in the output. The programmer also instructed the DA to favor the end of the line when prorating the line. This means that the DA should first try to allocate "extras" to the end tokens of the given line. This is the desired behavior since we want `Tail` (the last token on the last line) to soak up all the excess spaces on the last line. By default, the Prorate cliché favors the beginning elements of the input sequence.

With the change in the descriptions, Prorate now needs to compute the Total-Stretch of the given sequence of tokens. The DA is able to deduce that the Total-Stretch required by Prorate is the same as the one that is computed in Build-Graph. There is knowledge in the Prorate cliché that states that the Total-Shares computation being carried out in the cliché is really the Sum of the given Proportional-To function applied to all the elements in the set being prorated. Since the DA is able to prove that the `Arc-Total-Stretch` function on an arc (viewed as a set in the Prorate cliché) is the same as Total-Shares, an extra argument is passed into the Prorate function so that Total-Stretch need not be computed twice. The DA favors storing explicit intermediate results if they are needed more than once, because the guideline provided by the programmer in Scene 1 tells it to `(PREFER 'time-efficiency)`. This indicates to the DA that it should assume that space is cheap relative to time. Changes in the code produced by the DA are shown in Figure Code.3.1 and Figure Code.3.2.

The change requested by the programmer helps to demonstrate two points. First, given separate pieces of general knowledge in different clichés, the DA is able to prove that some intermediate results computed earlier can be saved and reused in later computations. Second, the DA can automatically handle a simple store-vs-recompute tradeoff under some user guidance.

```
>>> Return-To-Warning.
DA> WARNING:  The output of (Prorate (Extras line) (Gap-Set line)) is
not used.  By default, the output of Prorate is a list of pairs (element
prorated-amount).
>>> (WITHIN Prorate
      (IS Incf-Field 'width)
      (IS Proportional-To 'stretch)
      (IS Favor 'End))
```

Scene Dialog.3: Dialog between the DA and the Programmer.

```
(defstruct (Stuff (:constructor make-stuff (ratio extras TOTAL-STRETCH)))
   ratio extras TOTAL-STRETCH)                                    ; NEW

(defstruct (Arc (:constructor make-arc
                 (source destination ratio extras TOTAL-STRETCH)))
  source destination ratio extras TOTAL-STRETCH)                  ; NEW

(defun Process-Paragraph (s outstream)
  (let* ((opt-path (multiple-value-call #'compute-sssp (build-graph S)))
         (token-set nil))
    (dolist (arc opt-path)
      (setq token-set (token-set arc))
      (prorate (arc-extras arc) (gap-set token-set)
               (ARC-TOTAL-STRETCH ARC) :FROM-RIGHT T)    ; NEW
      (dolist (token token-set)
        (if (gap-token-p token)
            (print-n-spaces outstream (width token))
            (write-string (content token) outstream)))
      (write-char #\newline outstream))
    (write-char #\newline outstream)))

(defun Compute-Sssp (root-and-sink node-set-size arc-set)
  (when arc-set
    (let* ((root (first root-and-sink)) (sink (second root-and-sink))
           (distance-array (make-array node-set-size))
           (misc-array (make-array node-set-size))
           (best-previous-array (make-array node-set-size))
           (source-node nil) (destination-node nil) (destination-index 0)
           (distance-via-node 0))
      (dotimes (i node-set-size) (setf (aref distance-array i) *infinity*))
      (setf (aref distance-array (index1 root)) 0)
      (dolist (arc arc-set)
        (setq source-node (arc-source arc))
        (setq destination-node (arc-destination arc))
        (setq distance-via-node (+ (aref distance-array (index1 source-node))
                                   (arc-ratio arc)))
        (setq destination-index (index1 destination-node))
        (if (> (aref distance-array destination-index) distance-via-node)
            (setf (aref distance-array destination-index) distance-via-node
                  (aref misc-array destination-index)
                  (make-stuff (arc-ratio arc) (arc-extras arc)
                              (ARC-TOTAL-STRETCH ARC))          ; NEW
                  (aref best-previous-array destination-index) source-node)))
      (let* ((previous sink) (optimal-arc-set nil) (stuff nil))
        (loop
          (if (eq previous root) (return optimal-arc-set))
          (setq destination-index (index1 previous))
          (setq destination-node previous)
          (setq previous (aref best-previous-array destination-index))
          (setq stuff (aref misc-array destination-index))
          (push (make-arc previous destination-node (stuff-ratio stuff)
                          (stuff-extras stuff) (STUFF-TOTAL-STRETCH STUFF)) ; NEW
                optimal-arc-set))))))
```

Scene Code.3.1: Code with New Proration (part 1).

```
(defun Build-Graph (v)
  (let ((root nil) (position 0) (queue nil) (end-of-queue nil)
        (node-table (make-hash-table)) (current-token v) (node-number 0)
        (total-length 0) (total-stretch 0) (queue-token nil)
        (queue-node nil) (extras 0) (ratio 0) (current-node nil)
        (arc-set nil))
    (setq root (make-bkpt :token current-token :index1 node-number))
    (incf node-number)
    (setq queue root end-of-queue root)
    (setf (gethash 0 node-table) root)
    (setq current-token (next current-token))
    (loop
      (if (null current-token)
          (return (values (list root end-of-queue) (+ node-number 1)
                          (reverse arc-set))))
      (setq position (index current-token))
      (when (gap-token-p current-token)
        (setq queue-node queue)
        (loop
          (if (null queue-node) (return))
          (setq queue-token (token queue-node))
          (if (eq queue-token current-token) (return))
          (setq total-length (sum queue-token current-token :field #'width))
          (setq total-stretch (sum queue-token current-token :field #'stretch))
          (setq extras (- *line-length* total-length))
          (setq ratio (if (= extras 0) 0
                          (if (= total-stretch 0) 10000 (/ extras total-stretch))))
          (when (and (>= ratio 0) (<= ratio *threshold*))
            (setq current-node (gethash position node-table))
            (when (null current-node)
              (setq current-node (make-bkpt :token current-token
                                            :index1 node-number))
              (setf (gethash position node-table) current-node)
              (incf node-number)
              (setf (link end-of-queue) current-node)
              (setq end-of-queue current-node))
            (push (make-arc queue-node current-node ratio extras TOTAL-STRETCH) ; NEW
                  arc-set))
          (setq queue-node (link queue-node))))
      (setq current-token (next current-token)))))

(defun Prorate (amount set TOTAL-SHARES &key from-right)   ; NEW
  (when (and set (not (= amount 0)))
                                     ; NEW: computation of total-shares is gone
    (if from-right (setq set (reverse set)))
    (let* ((amount-per-share (/ amount total-shares)) (unallocated amount))
      (mapcar
        #'(lambda (node)
            (let ((share (stretch node)))
              (if (<= share 0) (list node 0)
                  (let ((allocation
                          (if (= total-shares share) unallocated
                              (min (ceiling (* amount-per-share share))
                                   unallocated))))
                    (decf unallocated allocation)
                    (decf total-shares share)
                    (INCF (WIDTH node) allocation)))))   ; NEW
        set))))
```

Scene Code.3.2: Code with New Proration (part 2).

3.7 Scene 4: Explaining Design Rationale

In this scene, the programmer queries the DA about the low-level design of the program. The tags on the description of the DA facilitates the interaction between the DA and the programmer; it allows the programmer to refer to the keys for further explanation. This scene serves to illustrate the design rationale used by the DA to arrive at the chosen implementations (see Figure Dialog.4). Since most of the interaction is self-explanatory, we shall not elaborate here.

```
>>> Explain-Design.
DA> The following subsidiary design choices have been made:
  1. A structure named Token is used to represent a Token in Tokenize.
  2. A Pointer-Chain is used to represent the token sequence.
  3. The Build-Rooted-Forward-Graph cliche has been chosen to implement
Build-Graph.
  4. A structure named Bkpt is used to represent a Node in the Graph.
  5. An ordered arc set has been chosen to represent Graph.
  6. The Simple-Segment cliche has been chosen to implement Segment.
  7. The Rooted-Dag-Sssp cliche has been chosen to implement
Single-Source-Shortest-Path.
  8. An ordered arc set is used to represent the optimal path.
  9. The Incf-Prorate cliche has been chosen to implement Prorate.
>>> Why(3)
DA> The Build-Rooted-Forward-Graph cliche is chosen because
  (3.1) Build-Rooted-Forward-Graph is an instance of the build-graph cliche.
  (3.2) Build-Rooted-Forward-Graph is rooted.
  (3.3) Build-Rooted-Forward-Graph is forward-wrt-input.
  (3.4) Favor time efficiency.
>>> Why(5)
DA> Graph is represented as an ordered arc set because:
  (5.1) An ordered arc set can be used to represent the graph.
  (5.2) The graph is to be used by a Single Source Shortest Path
algorithm in which arcs are enumerated.
  (5.3) The topological sorting step in the Rooted-Dag-Sssp algorithm
can be eliminated.
>>> Why(5.3)
DA> The topological sorting step in the Rooted-Dag-Sssp algorithm
can be eliminated because:
  (5.3.1) The first step in Rooted-Dag-Sssp enumerates the input arc
set of Graph in topologically-sorted order.
  (5.3.2) The output arc set of Build-Rooted-Forward-Graph is
topologically-sorted.
  (5.3.3) The output arc set of Build-Rooted-Forward-Graph is the input
arc set of the Rooted-Dag-Sssp algorithm.
>>> Why(7)
DA> The Rooted-Dag-Sssp cliche was chosen because:
  (7.1) Favor time efficiency.
  (7.2) Rooted-Dag-Sssp can be used to implement the
Single-Source-Shortest-Path algorithm.
  (7.3) Most-Efficient(Possible-Implementations(Sssp)) = Rooted-Dag-Sssp.
>>> Value-Of(Time-Efficiency(Rooted-Dag-Sssp))
DA> ROOTED-DAG-SSSP has time complexity O(m+n) where m is the size of
the arc set, and n is the size of the node set.
>>> Why(2)
DA> A Pointer-Chain is used to represent the token sequence because
  (2.1) The token sequence has to be extendable.
  (2.2) Pointer-Chains are useful where sequences need to be extendable.
>>> (Why 2.1)
DA> The Concatenate cliche requires the token sequence to be extendable.
```

Scene Dialog.4: Dialog between the DA and the Programmer.

3.8 Scene 5: Changing An Implementation Guideline

The programmer observes that intermediate storage is used to store auxiliary graph information. The DA is asked to store auxiliary information on the graph itself using the form:

```
(IMPLEMENTATION-GUIDELINES (STORE-AUXILIARY-INFORMATION 'Graph)
```

Scene Description.5.1: Adding a New Implementation Guildeline.

Note that in Figure Code.5.1 and Figure Code.5.2, the **index1** field of **bkpt** is no longer used. The only reason why the **index1** field was needed in the earlier scene was so that the **min-distance-array** could be accessed when given a node. All the relevant information are kept on the **bkpt** node structure itself, and so the **index1** is no longer needed. This scene shows how the implementation guidelines given can influence the choices made by the DA.

```
(defun Compute-Sssp (root-and-sink NODE-SET arc-set) ; was Node-Set-Size
  (when arc-set
    (let* ((root (first root-and-sink)) (sink (second root-and-sink))
           (source-node nil) (destination-node nil) (distance-via-node 0))
      ; All intermediate arrays are gone
      (DOLIST (NODE NODE-SET) (setf (MIN-DISTANCE node) *infinity*)) ; NEW
      (setf (MIN-DISTANCE root) 0)
      (dolist (arc arc-set)
        (setq source-node (arc-source arc))
        (setq destination-node (arc-destination arc))
        (setq distance-via-node (+ (MIN-DISTANCE source-node) (arc-ratio arc)))
        (if (> (MIN-DISTANCE destination-node) distance-via-node)    ; NEW
            (setf (MIN-DISTANCE destination-node) distance-via-node
                  (BEST-PREVIOUS destination-node) source-node
                  (RATIO destination-node) (arc-ratio arc)
                  (EXTRAS destination-node) (arc-extras arc)
                  (TOTAL-STRETCH destination-node) (arc-total-stretch arc))))
      (let* ((previous nil) (optimal-arc-set nil))
        (setq destination-node sink)
        (setq previous (BEST-PREVIOUS destination-node))             ; NEW
        (loop
          (push (make-arc previous destination-node (RATIO destination-node)
                          (EXTRAS destination-node) (TOTAL-STRETCH destination-node))
                optimal-arc-set)
          (if (eq previous root) (return optimal-arc-set))
          (setq destination-node previous)
          (setq previous (BEST-PREVIOUS destination-node))))))))
```

Scene Code.5.1: Modified Code: Part 1.

```
(defstruct (Bkpt (:conc-name nil))
  token link BEST-PREVIOUS MIN-DISTANCE EXTRAS RATIO TOTAL-STRETCH)

(defun Build-Graph (v)
  (let ((root nil) (position 0) (queue nil) (end-of-queue nil)
        (node-table (make-hash-table)) ; Node-Number is gone
        (current-token v) (total-length 0) (total-stretch 0)
        (queue-token nil) (NODE-SET nil) (queue-node nil)         ; NEW
        (extras 0) (ratio 0) (current-node nil) (arc-set nil))
    (setq root (make-bkpt :token current-token))
    (PUSH ROOT NODE-SET)
    (setq queue root end-of-queue root)
    (setf (gethash 0 node-table) root)
    (setq current-token (next current-token))
    (loop
      (if (null current-token)
          (return (values (list root end-of-queue)
                          (REVERSE NODE-SET) ; was Node-Number
                          (reverse arc-set))))
      (setq position (index current-token))
      (when (gap-token-p current-token)
        (setq queue-node queue)
        (loop
          (if (null queue-node) (return))
          (setq queue-token (token queue-node))
          (if (eq queue-token current-token) (return))
          (setq total-length (sum queue-token current-token :field #'width))
          (setq total-stretch (sum queue-token current-token :field #'stretch))
          (setq extras (- *line-length* total-length))
          (setq ratio (if (= extras 0) 0
                          (if (= total-stretch 0) 10000 (/ extras total-stretch))))
          (when (and (>= ratio 0) (<= ratio *threshold*))
            (setq current-node (gethash position node-table))
            (when (null current-node)
              (setq current-node (make-bkpt :token current-token))
              (PUSH CURRENT-NODE NODE-SET)                          ; NEW
              (setf (gethash position node-table) current-node)
              (setf (link end-of-queue) current-node)
              (setq end-of-queue current-node))
            (push (make-arc queue-node current-node ratio extras total-stretch)
                  arc-set))
         (setq queue-node (link queue-node))))
     (setq current-token (next current-token)))))
```

Scene Code.5.2: Modified Code: Part 2.

3.9 Scene 6: Correcting The Program Description

Now that the programmer is satisfied with the description, the program was executed on a test file. The programmer noticed that there are extra blank spaces on both sides of the text boundary in the output file in Figure Test.6.1 (The spaces on the first line are rendered as underscores to distinguishthem from the rest of the page.) Upon reflection, the programmer discovered a bug in the program description: the end gap tokens of internal lines were output twice: once as the end token of the previous line and the second time as the begin token of the current line. This is clearly wrong. The programmer considered specifying that the sum computations should be done ignoring the end gap tokens. However, in such a case, the first line and the last line will not be formatted correctly because Head and Tail would be removed from the computations. Head and Tail are important in ensuring that paragraph indentation are taken care of and that the paragraph ends properly. What is desired is to specify that all lines are to be computed without taking into consideration the end gap tokens unless they happen to be the Head token or the Tail token. However, to do so would be rather expensive as the check for Head and Tail will have to take place in the summations in Build-Graph. There can be potentially many such computations, most of which are useless because the threshold test will fail on them. After some thinking, the programmer came up with an idea: four tokens will be appended instead of just Head and Tail. The programmer renamed Head and Tail to Para-Head and Para-Tail respectively. A new Head is placed before Para-Head and a new Tail is placed after Para-Tail. Both Head and Tail will have 0 width and 0 stretch. Now the programmer simply specifies that the computations should be done excluding all end tokens. The `Without-Endpoints` function takes a sequence and returns the sequence without the first and the last elements. The resulting modified code are given in Figure Code.6.1 and Figure Code.6.2. The programmer runs the same test file with the modified program; and this time, the expected result is obtained. The new result file is shown in Figure Test.6.2.

In the procedure `Prorate`, the DA is again able to deduce that Total-Stretch is being used twice here. The sum of the `Without-Endpoints` function applied to elements in an arc is still the same as the Total-Stretch computed in `Build-Graph`.

```
-----Our approach is influenced by our view on the  nature--
  of programming.  Hence, before delving  into  the  depths
  of our approach, we  briefly  characterize  our  view  of
  programming.

     Programming is Knowledge-Intensive: Different sources
  of knowledge are required.  Knowledge of data  structures
  and algorithms are key components of programming, so  are
  program structuring techniques, program specification, and
 knowledge about the application domain.  We believe  that
  much of  the  knowledge  needed  in  programming  can  be
  codified so that a computer program can make  use  of  it
  mechanically.

     Requirements  Constantly  Change:  Many  changes  are
  constantly being  made  to  programs  due  to  constantly
  changing needs.  This is  not  just  evident  in  program
  maintenance but also  during  the  development  process.
   Program maintenance, especially for  large  systems,  is
  difficult, because much of the design  rationale  is  not
  written down; and where it is written down, it is usually
  not kept up-to-date.
```

Scene Test.6.1: The Result File from Justify.

```
-----Our approach is influenced by our view on the nature of
programming.  Hence, before delving into the depths  of  our
approach, we briefly characterize our view of programming.

     Programming is Knowledge-Intensive:  Different  sources
of knowledge are required.  Knowledge of data structures and
algorithms are key components of programming, so are program
structuring techniques, program specification, and knowledge
about the application domain.  We believe that much  of  the
knowledge needed in programming can be codified  so  that  a
computer program can make use of it mechanically.

     Requirements Constantly  Change:   Many   changes   are
constantly being made to programs due to constantly changing
needs.  This is not just evident in program maintenance  but
also during the development process.   Program  maintenance,
especially for large systems, is difficult, because much  of
the design rationale is not written down; and  where  it  is
written down, it is usually not kept up-to-date.
```

Scene Test.6.2: New Result File from Modified Justify.

```
(REDESIGN lineout (outfile line)
  (LET ((token-set (WITHOUT-ENDPOINTS (Token-Set line))))
    (Prorate (Extras line) (Gap-Set token-set))
    (DO (token token-set)
     (If (Gap-Token-P token)
         (output outfile #\space :number (Width token))
         (output outfile (Content token))))
    (Output outfile #\newline)))

(REDESIGN Total-Length (seq) (Sum (Map #'Width seq) :BEGIN NIL :END NIL))

(REDESIGN Total-Stretch (seq) (Sum (Map #'Stretch seq) :BEGIN NIL :END NIL))

(MODIFY-WITHIN Build-Graph (paragraph)
  (LET ((new-paragraph (Concatenate HEAD PARA-HEAD paragraph PARA-TAIL TAIL))
        (M input-to-node-mapping))
    (IS (Domain M) (Gap-Set new-paragraph))
    (Directed graph) (Forward-Wrt-Input graph) (IS Root (M Head))
    (IS Arc-Test
        (Lambda ((xn token) (yn token))
          (And (Node (M xn))
               (>= (ratio (Subsequence new-paragraph xn yn)) 0)
               (<= (ratio (Subsequence new-paragraph xn yn))
                   *Threshold*))))))

(== Para-Head (INSTANCE 'gap :width *Para-Indent* :stretch 0 :index 'para-head))
(== Para-Tail (INSTANCE 'gap :width 0 :stretch 10000 :index 'para-tail))
(== Head (INSTANCE 'gap :width 0 :stretch 0 :index 'head))
(== Tail (INSTANCE 'gap :width 0 :stretch 0 :index 'tail))
```

Scene Description.6: Adding Additional New Gap Tokens.

```
(defun Justify (input-file output-file)
  (with-open-file (instream input-file :direction :input)
    (with-open-file (outstream output-file :direction :output)
      (let* ((current-token nil) (previous-token nil) (next-token nil)
             (PARA-HEAD (MAKE-ENDTOKEN *PARA-INDENT* 0 'PARA-HEAD))   ; NEW
             (HEAD (MAKE-ENDTOKEN 0 0 'HEAD PARA-HEAD)))
        (initialize-reading)
        (setq previous-token head)
        (setq current-token (remove-parabreaks instream))
        (setf (next PARA-HEAD) current-token)
        (loop
          (when (null current-token)
            (if (eq previous-token head) (return)
                (let* ((TAIL (MAKE-ENDTOKEN 0 0 'TAIL))                ; NEW
                       (PARA-TAIL (MAKE-ENDTOKEN 0 10000 'PARA-TAIL TAIL)))
                  (setf (next previous-token) PARA-TAIL)
                  (process-paragraph head outstream) (return))))
          (setq next-token (get-token instream))
          (setf (next current-token) next-token)
          (cond ((word-token-p current-token)
                 (setf (width current-token) (length (content current-token)))
                 (setf (stretch current-token) 0))
                ((gap-token-p current-token)
                 (setf (stretch current-token) 1)
                 (if previous-token
                     (if (member (last-char (content previous-token))
                                 (list #\! #\? #\.))
                         (setf (width current-token) 2)
                         (setf (width current-token) 1))))
                ((para-break-token-p current-token)
                 (if (and next-token (para-break-token-p next-token))
                     (setq next-token (remove-parabreaks instream)))   ; NEW
                 (LET* ((NEW-PARA-HEAD (MAKE-ENDTOKEN *PARA-INDENT* 0 'PARA-HEAD))
                        (NEW-HEAD (MAKE-ENDTOKEN 0 0 'HEAD NEW-PARA-HEAD))
                        (TAIL (MAKE-ENDTOKEN 0 0 'TAIL))
                        (PARA-TAIL (MAKE-ENDTOKEN 0 10000 'PARA-TAIL TAIL)))
                   (setf (next previous-token) PARA-TAIL)
                   (process-paragraph head outstream)
                   (setq head NEW-HEAD)                                ; NEW
                   (setf (next NEW-PARA-HEAD) next-token)
                   (setq current-token NEW-HEAD))))
          (setq previous-token current-token)
          (setq current-token next-token))))))

(defun Process-Paragraph (s outstream)
  (let* ((opt-path (multiple-value-call #'compute-sssp (build-graph S)))
         (token-set nil))
    (dolist (arc opt-path)
      (setq token-set (WITHOUT-ENDPOINTS (token-set arc)))             ; NEW
      (prorate (arc-extras arc) (gap-set token-set)
               (arc-total-stretch arc) :from-right t)
      (dolist (token token-set)
        (if (gap-token-p token)
            (print-n-spaces outstream (width token))
            (write-string (content token) outstream)))
      (write-char #\newline outstream))
    (write-char #\newline outstream)))
```

Scene Code.6.1: Modified Code: Part 1.

```
(defun Build-Graph (v)
  (let ((root nil) (position 0) (queue nil) (end-of-queue nil)
        (node-table (make-hash-table)) (current-token v) (node-set nil)
        (total-length 0) (total-stretch 0) (queue-token nil)
        (queue-node nil) (extras 0) (ratio 0) (current-node nil)
        (arc-set nil))
    (setq root (make-bkpt :token current-token))
    (push root node-set)
    (setq queue root end-of-queue root)
    (setf (gethash 0 node-table) root)
    (setq current-token (next current-token))
    (loop
      (when (null current-token)
        (return (values (list root end-of-queue) (reverse node-set)
                        (reverse arc-set))))
      (setq position (index current-token))
      (when (gap-token-p current-token)
        (setq queue-node queue)
        (loop
          (if (null queue-node) (return))
          (setq queue-token (token queue-node))
          (if (eq queue-token current-token) (return))
          (setq total-length (sum queue-token current-token :field #'width
                                  :BEGIN NIL :END NIL))                    ; NEW
          (setq total-stretch (sum queue-token current-token :field #'stretch
                                   :BEGIN NIL :END NIL))                   ; NEW
          (setq extras (- *line-length* total-length))
          (setq ratio (if (= extras 0) 0
                          (if (= total-stretch 0) 10000 (/ extras total-stretch))))
          (when (and (>= ratio 0) (<= ratio *threshold*))
            (setq current-node (gethash position node-table))
            (when (null current-node)
              (setq current-node (make-bkpt :token current-token))
              (push current-node node-set)
              (setf (gethash position node-table) current-node)
              (setf (link end-of-queue) current-node)
              (setq end-of-queue current-node))
            (push (make-arc queue-node current-node ratio extras total-stretch)
                  arc-set))
          (setq queue-node (link queue-node))))
      (setq current-token (next current-token)))))

(defun Without-Endpoints (set)
  (when (listp set)
    (let ((newset nil))
      (dolist (elt (cdr set) (if newset (reverse (cdr newset))))
        (push elt newset)))))

(defun Make-Endtoken (width stretch position &optional next) ; new
  (make-token :type 'gap :width width :stretch stretch :index position
              :next next))
```

Scene Code.6.2: Modified Code: Part 2.

3.10 Scene 7: The Complete Program Descriptions

The DA performed its tasks based on its interpretation of the given program descriptions. The net cumulative description is maintained by the DA and is given in Figure Description.7.1 and Figure Description.7.2. If the programmer had given the DA the final program descriptions, it would have output the same code shown in the scene.

```
(DESIGN Justify (infile outfile)
  (Ascii-File infile) (Ascii-File outfile)
  (DO (paragraph (Segment (Tokenize infile) 'para-break))
    (DO (line (Single-Source-Shortest-Path (Build-Graph paragraph)))
      (lineout outfile line))
    (Output outfile #\newline)))

(WRITE-CODE 'Justify)

(IMPLEMENTATION-GUIDELINES
  (PREFER 'time-efficiency)
  (IGNORE 'error-checking)
  (STORE-AUXILIARY-INFORMATION 'Node))

(DESIGN Lineout (outfile line)
  (LET ((token-set (Without-Endpoints (Token-Set line))))
    (Prorate (Extras line) (Gap-Set token-set))
    (DO (token token-set)
     (If (Gap-Token-P token)
         (output outfile #\space :number (Width token))
         (output outfile (Content token))))
    (Output outfile #\newline)))

(WITHIN Tokenize
  (== para-break
      (regular-expression
        "BOF (#\space | #\newline)* | (#\space | #\newline)* EOF |
        (#\space)* #\newline (#\space)* #\newline (#\space | #\newline)*"))
  (== word (regular-expression "(Non-Blank-Printable-Characters)+"))
  (== gap (regular-expression "(#\space | #\newline)+"))
  (Treat others #\space))

(WITHIN Build-Graph (paragraph)
  (LET ((new-paragraph (Concatenate Head Para-Head paragraph Para-Tail Tail))
        (M input-to-node-mapping))
    (IS (Domain M) (Gap-Set new-paragraph))
    (Directed graph) (Forward-Wrt-Input graph) (IS Root (M Head))
    (IS Arc-Test
        (Lambda ((xn token) (yn token))
          (And (Node (M xn))
               (>= (ratio (Subsequence new-paragraph xn yn)) 0)
               (<= (ratio (Subsequence new-paragraph xn yn))
                   *Threshold*))))))
```

Scene Description.7.1: Program Description of Justify (Part 1).

```
(WITHIN Prorate
  (IS Favor 'End)
  (IS Proportional-To #'stretch)
  (IS Incf-Field #'width))

(WITHIN Single-Source-Shortest-Path
  (IS source-node Head)
  (IS destination-node Tail)
  (IS distance-function #'ratio))

(DESIGN Extras (seq) (- *Line-Length* (total-length seq)))

(DESIGN Ratio (seq)
  (if (= (Extras seq) 0) 0
      (if (= (Total-Stretch seq) 0) 10000
          (/ (Extras seq) (Total-Stretch seq)))))

(DESIGN Total-Length (seq) (Sum (Map #'Width seq) :begin nil :end nil))

(DESIGN Total-Stretch (seq) (Sum (Map #'Stretch seq) :begin nil :end nil))

(== Para-Head (INSTANCE 'gap :width *para-indent* :stretch 0 :Index 'Para-Head))
(== Para-Tail (INSTANCE 'gap :width 0 :stretch 10000 :Index 'Para-Tail))
(== Head (INSTANCE 'gap :width 0 :stretch 0 :Index 'Head))
(== Tail (INSTANCE 'gap :width 0 :stretch 0 :Index 'Tail))

(DESIGN Stretch (token) (if (Gap-Token-P token) 1 0))

(DESIGN Width (token)
  (cond ((Word-Token-P token) (Number-Of-Characters token))
        ((Gap-Token-P token)
         (If (Member (Last-Character (Preceding-Token token))
                     '(#\! #\? #\.))
             2 1))))
```

Scene Description.7.2: Program Description of Justify (Part 2).

3.11 The Rest of the Scenario

The rest of the scenarios are not as well-developed as the earlier ones. Hence we shall only provide brief descriptions of the ideas we are trying to convey but no specific output code. We hope to build as many of the scenes as possible.

Scene 8: Changing the Program Descriptions

Suppose line numbers are to be printed before each line. We could write this description with the help of the `Count` cliché. The cliché introduces a count on a given variable.

The program descriptions as currently given do not address the case when the Build-Graph algorithm fails to find a line break. This can happen when a potential line is both too short to pass the threshold test and on adding another word to the line, it becomes too long for the threshold test. Typically, this happens when words are too long compared to the line length. In such cases, the programmer may like to increase the `*threshold*` steadily so that all lines can be broken. One way is to double the threshold parameter every time the algorithm fails to find a line break. Such a program can be described concisely by the `Repeat-Until-Success` cliché.

The `Repeat-Until-Success` cliché takes several roles. The *action* role contains forms to be repeatedly executed until the predicate in the *end-test* role becomes true. Whenever the predicate fails, a function in the *fix-action* role is then run.

Scene 9: Optimizations

The programmer may notice that it is rather inefficient to be computing the `total-stretch` and `total-length` of sequences repeatedly across different iterations in `Justify`. A standard optimization trick is to keep cumulative sums across the iterations so that the needed value can be obtained via a single subtraction. Suppose the sum of all the lengths of the tokens previous to a token T is kept in its Cum-Length field, i.e., `(Cum-Length T) = (Sum (Map #'length S))` where S is the sequence starting from the beginning root token to T. Then the `total-length` between 2 tokens T1 and T2, where T2 comes after T1 in the token sequence, is simply: `(- (Cum-Length T2) (Cum-Length T1))`.

The DA may not know enough to be able to carry out this optimization in general. However, it should be able to carry out the optimization with the guidance of the programmer.

Another piece of optimization capitalizes on the nature of the `ratio` function and the `arc-test` role of the Build-Graph cliché. The ratio function and the arc-test predicate are both defined on intervals in the token sequence. `Arc-test` passes on an interval only if the ratio of the interval lies between the specified range. From the basic properties of the `total-stretch` and the `total-length` functions, the DA should be able to prove that

they are monotonically increasing wrt intervals in the token sequence.[1] With this, the DA should be able to deduce that `ratio` is monotonically decreasing wrt intervals in the token sequences (analogously defined). This piece of information, with the help of some algebraic knowledge resident on the Build-Graph cliché (or more generally, elsewhere) can help the DA prove that many tests are redundant and hence can be pruned. Specifically, it suffices to keep the intermediate `queue` in Figure Code.6.3 short (that is, to a constant size) so that the resultant procedure runs in linear time instead of quadratic time.

The interesting part of this scene is in illustrating how some generic, separate and shallow knowledge can be used to carry out some significant optimization with minimal logical inferences. Another interesting aspect of this scene is in exploring to what extent can optimization knowledge be separated from the rest of programming knowledge.

Scene 10: Making an Orthogonal Addition

In this scene, the programmer requests the DA to introduce tracing and instrumentation code by providing implementation guidelines. The form can look like

```
(ImplementationGuidelines (ADD-TRACING 'Node) (ADD-INSTRUMENTATION 'Graph)).
```

Some specific code or instructions can be attached to the clichés to enable the DA to generate tracing and instrumentation code. For tracing procedures, printing code can be automatically introduced where data are created or modified so that their modifications can be monitored. For instrumentation, the sizes of some important intermediate data structures can be printed and their size statistics kept. What constitutes *important* intermediate data structures may depend on the specific clichés used. We expect that this useful function can be automated with the help of some helpful hints provided in the clichés.

Scene 11: Adding Error Checking

The code produced by the DA is expected to be correct by construction wrt the given program description if error-checking is turned on. Despite this, the descriptions could be incomplete, hence the addition of error checking code is still important.

The error checking code may be generated to check for violated constraints. Specifically, the preconditions of every Lisp function should be provably satisfied by the input data and situations. If the DA is unable to establish such a proof, then the DA can introduce some error-checking code to check for violations.

Beyond the specific language-dependent constructs, certain general constraints given explicitly in the program descriptions or indirectly via the clichés used, can be used to generate checking code to test for violations. For example, in the scene, if we have the following constraints specified by the programmer: (1). The length of the first word

[1] An interval function F is monotonically increasing on an interval [a,b] iff (1) $(a \leq x \leq y \leq z \leq b)$ iff (F [x,y]) $\leq$ (F [x,z]); and (2) $(a \leq x \leq y \leq z \leq b)$ iff (F [y,z]) $\leq$ (F [x,z]).

token in the paragraph should be $\leq$ (*Line-Length* - *Para-Indent*). (2) The length of any word token should be $\leq$ *Line-Length*. The DA can potentially generate code to check explicitly that all tokens received by `Justify` (from `Get-Token`) obey the given constraints. However there may be some consequences arising from such constraints which are unknown to the DA. In those cases, the DA will not be able to provide much help.

The objectives of introducing such error-checking code may be summarized as: to inform the programmer of the problem before the code runs into an external debugger and to help catch violations of explicit constraints imposed on the program.

4 Related Work

Our work on the DA spans several different dimensions of automatic programming research. The key dimensions are:

- Program Synthesis: Program synthesis systems accept some kind of user specifications, and output some *executable code*.
- Support for Low-Level Design: These systems support low-level program design by automatically making some implementation choices and maintaining the design rationale behind those choices. These include systems that select data structures and algorithms, and they may or may not produce executable code.
- Support for Specification: Systems that support specification can *criticize* the given specification by detecting inconsistencies and incompleteness, and they may provide corrections to the detected problems.

As some of the reviewed work spans more than one of the above dimensions, the following account will discuss each work individually in their broad categories and indicate how they are related to the DA along the above dimensions.

Deductive Synthesis

Works on deductive synthesis are based on the idea that a program can be extracted from a constructive proof of the theorem about the desired input-output behavior. This approach is very general but it reduces the problem of program synthesis to that of automatic theorem proving, another very difficult problem. Manna and Waldinger' work [13] and Smith's CYPRESS [20] are examples in this category. The key advantage of this approach is that a verification proof comes with any program synthesized. However, this approach is not likely to scale up for larger programs and it is not clear how to generate efficient code based on such an approach.

This approach is related to the DA only in its goal of program synthesis. The approach taken by the DA explicitly shuns deep deductions and emphasizes the use of detailed knowledge.

Program Transformation

This approach emphasizes the use of correctness-preserving transformations to either implement programs from given specifications or to improve the efficiency of given programs. The former is commonly known as transformational implementation or vertical transformation, and the latter, lateral transformation. Transformational implementation approach typically takes a high level description of a program and through successive

transformations, turn it into an executable program. As it is closely related to the very high level language approach, some systems which belong to both approaches will be discussed later under very high level languages.

An example of lateral transformations is Burstall and Darlington's transformation system [4]. It uses a small set of transformation rules to improve programs which are more understandable to humans but are inefficient. Research in lateral transformation is complementary to the DA research proposed here. Some of the results can potentially be used to optimize the output of the DA.

One of the earliest system to have incorporated vertical program transformation ideas is the PSI system [7]. It consists of two modules, the PECOS module [2] which generates the program based on a library of synthesis refinement rules and the LIBRA [8] system which controls the search in the generation of the program. Researchers at the Kestrel Institute and the Information Sciences Institute, University of Southern California are actively pursuing this approach to implement high level specification languages.

The vertical transformation approach is similar our approach; the difference lies mainly in emphasis. Most program transformation systems start with more declarative and more abstract specifications, and through a long series of transformations, arrive at an implementation. The DA produces executable code and maintains the explicit design rationale for the implementations chosen for program descriptions based on specification clichés.

Very High Level Languages

This approach aims to provide a high-level language in which abstract specifications and concrete implementations can be written side by side. Typically, a transformation system is built to transform the abstract specifications into executable constructs.

The SETL language [19] is a set-based, tuple-based language specifically designed to allow the programmer to ignore the detailed design of data structures. Kestrel Institute's REFINE language is another example of a very high level language [1]. REFINE is a general-purpose, set-based, logic-based, wide-spectrum language with facilities for manipulating a knowledge base of transformation rules. The REFINE compiler is a set of transformation rules that turns a REFINE program into executable Common Lisp code. The GIST specification language [6] is designed to retain the expressiveness of natural language. Some transformation rules have been studied in this context but no compiler has been implemented.

The DA can be viewed as supporting a specialized, very high level specification language. Like all systems which follow this approach, the DA aims to provide the programmer with a language which is higher level than that provided by conventional programming languages. SETL is similar to our work here in that we both strive to free the programmer from detailed design of data structures. However, a key difference that makes the DA stand out among these systems is the nature of the specification language.

The DA is an interactive system specifically designed to capture some of the intermediate vocabulary used by programmers in order to make the specifications more understandable to human programmers.

Program Generator

A program generator uses a high level specification language typically designed for a specific domain, and turns a description in such a language into a program in some conventional language. Program generators are similar to the very high level languages except that they typically use more traditional compilation techniques, and their focus is typically very narrow.

The Draco approach to automatic programming [15] can be viewed as advocating program generators for specific domains. Suitable domains are domains in which many systems need to be built. A domain analyst studies such a domain and creates the objects and operations of interest to the domain, possibly in terms of other domain objects and operations known to the Draco system. A domain designer creates alternative implementations for each object and operation. The resultant Draco system is then used by system analysts and system designers. A system analyst specifies a new system in the domain using the objects and operations provided by the Draco system; a system designer takes this specification and creates an executable system by selecting the appropriate implementations for each object and operation in the specification using the Draco system.

At some level of description, our approach is similar to that of Draco's. The clichés we are trying to codify are similar to the software components the Draco approach calls for. The domain analysis and design is similar to our analysis of some programming domains in order to come up with appropriate clichés. One way of viewing our work is an attempt to automate the process of low-level design in Draco: Draco expects a system designer to select implementations for objects and operations in the specifications; the DA is expected to choose implementations automatically. Besides automatic selection of implementations, the DA also maintains the design rationale for the chosen implementations and critiques the given specification.

Another domain-specific program generator is the Φnix system [3]. It applies an extensive body of oil well logging knowledge to synthesize analysis programs. The Φnix approach and our approach shares a common emphasis on domain-specific knowledge if we view programs manipulating graphs as a domain. Taking such an emphasis allows the users of our systems to specify their needs concisely and in terms natural to the domain.

Algorithm Design

Another dimension to automatic programming is algorithm design. Kant [9] defined algorithm design as *the process of coming up with a sketch, in a very high level language,*

of a computationally feasible technique for accomplishing a specified behavior. Kant and Steier at CMU [21] included observing human algorithm designers at work. Steier and Newell's work on DESIGNER-SOAR [22] investigated using production system architecture to design and discover algorithms. DESIGNER-SOAR explores algorithm design by successive refinement using symbolic execution and test-case execution as primary control mechanisms. It integrated 9 sources of knowledge: weak methods, design strategies, program transformation, efficiency, symbolic execution, target language, domain definition, domain equations and learning.

The scope of these works included original algorithm design and discovery. Our current proposed work aims at re-using commonly known data structures and algorithms.

Selection of Data Structures and Algorithms

Low's system [12] automatically selects data structures based on how the data are used, the domain and size of data, and the operations performed on them. Techniques used included static flow analysis, monitoring execution of sample runs, and user interaction. The system concentrated on selection of data structures for sets and lists. Our work will build on the experience of Low's system, extending the techniques to select algorithms.

Rowe and Tonge [18] described a class of abstract data structures, called modeling structures, in terms of properties involving their component elements, relations between elements, and operations upon them. Each modeling structure may have several implementation structures which implement it. They described a system that accepts specifications which contain modeling structures and turns the modeling structures into implementation structures with the help of a library of mappings of modeling structures to implementation structures. They also described an algorithm which can implement the remaining modeling structures that failed to match any of the mappings available in the library. They acknowledged that their system is difficult to use partly because the descriptions of modeling structures were complicated. The question of how to combine several modeling structures into one representation is left. Our work also makes use of descriptions of familiar modeling structures to help select implementations of data structures. However our descriptions are not necessarily fixed by the system since they are general propositions which are maintained by a truth maintenance system. New clichés may introduce new descriptions which can be specified by the user. Furthermore, the descriptions in the DA are designed to be easily understandable to programmers.

Katz and Zimmerman [10] built an interactive advisory system for choosing data structures. It was able to provide new combinations of known structures which satisfy complex requirements not anticipated in advance. The key idea embodied in the system is that it provides a linguistic base of known vocabulary about data structures in which the selection of implementation structures was carried out. This is also the same idea being exploited by our approach in providing a familiar vocabulary for describing more abstract data structures and algorithms, and for describing the selection criteria. The vocabulary of this language are also terms in a truth maintenance system in which deductions can be

carried out.

McCartney's MEDUSA system [14] synthesizes a number of functional geometric algorithms based on the given input-output specifications and performance constraints. MEDUSA achieves this using a library of templates. This is similar to the our cliché-based approach. Since almost all objects of interest are sets, MEDUSA does not handle different set implementations explicitly, i.e. the data structures used are factored into the templates used. In contrast, the proposed DA research will get to examine the interaction of data structure selection and algorithm selection. We expect that some of the techniques developed in MEDUSA may be useful in our research.

References

[1] L. M. Abráido-Fandiño. An overview of *refine*^{TM} 2.0. A revised edition of the paper published in Proc. 2nd Int. Symposium on Knowledge Engineering - Soft. Eng., Madrid, Spain, April, 1987.

[2] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12(1 & 2):73–119, 1979. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[3] D. R. Barstow. A perspective on automatic programming. *AI Magazine*, 5(1):5–27, Spring 1984. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[4] R. M. Burstall and J. L. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), January 1977.

[5] C.E. Cormen, T.H. Leiserson and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1989.

[6] M. S. Feather and P. E. London. Implementing specification freedoms. *Science of Computer Programming*, 2:91–131, 1982.

[7] C. Green. A summary of the PSI program synthesis system. In *Proc. 5th Int. Joint Conf. Artificial Intelligence*, pages 380–381, Cambridge, MA, August 1977.

[8] E. Kant. A knowledge-based approach to using efficiency estimation in program synthesis. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pages 457–462, Tokyo, Japan, August 1979. Vol. 1.

[9] E. Kant. Understanding and automating algorithm design. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 1243–1253, 1985.

[10] S. Katz and R. Zimmerman. An advisory system for developing data representations. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1030–1036, Vancouver, British Columbia, Canada, August 1981. Vol. 2.

[11] D. E. Knuth and M. P. Plass. Breaking paragraphs into lines. *Software - Practice and Experience*, 11:1119–1184, 1981.

[12] J. R. Low. Automatic data structure selection: An example and overview. *Comm. of the ACM*, 21(5):376–384, May 1978.

[13] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. on Programming Languages and Systems*, 2(1):90–121, January 1980. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[14] R. McCartney. Synthesizing algorithms with performance techniques. In *Proc. 6th National Conf. on Artificial Intelligence*, pages 149–154, Seattle, WN, July 1987.

[15] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Trans. on Software Engineering*, 10(5):564–574, September 1984. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[16] 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.

[17] C. Rich and R. C. Waters. The Programmer's Apprentice: A program design scenario. Memo 933A, MIT Artificial Intelligence Lab., November 1987.

[18] L. A. Rowe and F. M. Tonge. Automating the selection of implementation structures. *IEEE Trans. on Software Engineering*, 4(6):494–506, November 1978. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[19] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representation in SETL programs. *ACM Trans. on Programming Languages and Systems*, 3(2):126–143, April 1981. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[20] D. R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[21] D. M. Steier and E. Kant. Symbolic execution in algorithm design. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 225–231, 1985.

[22] D. M. Steier and A. Newell. Integrating multiple sources of knowledge into designer-soar, an automatic algorithm designer. In *Proc. 7th National Conf. on Artificial Intelligence*, pages 8–13, 1988.

[23] R. C. Waters. The Programmer's Apprentice: A session with KBEmacs. *IEEE Trans. on Software Engineering*, 11(11):1296–1320, November 1985. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986 and in T. Ichekawa, editor, *Language Architectures and Programming Environments*, MIT Press, in preparation.