An Extensible, Object-Oriented Compiler for the Timeliner User Interface Language

by

Frank Tien-Fu Liu

B.S., Computer Science,
Massachusetts Institute of Technology, 2000

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© 2001 Frank Liu. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part.

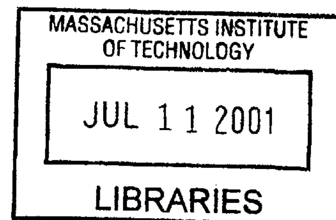
Signature of Author ______________________
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by ______________________
Dr. Robert Brown
Charles Stark Draper Laboratory
Thesis Supervisor

Certified by ______________________
Prof. Martin C. Rinard
Assistant Professor, EECS, MIT
M.I.T. Thesis Advisor

Accepted by ______________________
Arthur C. Smith
Chairman, Department Committee on Graduate Students

BARKER



This page intentionally left blank

An Extensible, Object-Oriented Compiler for the Timeliner User Interface Language

by

Frank Tien-Fu Liu

Submitted to the Department of Electrical Engineering and Computer Science

on May 23, 2001, in partial fulfillment of the

requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a compiler for the Timeliner User Interface Language. The Timeliner User Interface Language (UIL) is a specialized computer language designed for the writing of sequencing procedures to operate a complex system. The old version of the compiler is written in Ada, and is hard to maintain. By using tools such as JLex and CUP, I have created a compiler that is easier to maintain and extend. The use of an object-oriented language like Java™ allowed us to dramatically reduce the amount of code needed to create the compiler through inheritance and code reuse. Finally, the output of the compiler was switched from a set of data tables to a serialized abstract syntax tree, represented as an XML file. This change allows us to remove a layer of complexity from the Timeliner system. Comparing the results from an earlier project showed the dramatic improvement in the extensibility of the compiler.

Technical Supervisor: Dr. Robert Brown
Title: Principal Member of Technical Staff

Thesis Advisor: Professor Martin C. Rinard
Title: Assistant Professor, Department of EECS, MIT

This page intentionally left blank

ACKNOWLEDGMENT

May 23, 2001

I would like to thank Robert Brown, my Draper Supervisor, without whose support this work would not have been possible. I would also like to thank Professor Martin Rinard, for being so patient and understanding. To my family, thanks for always being there for me.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under NASA Contract Number NAS-9-01069.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

____

(author's signature)

This page intentionally left blank

Contents

1 Introduction **11**

1.1 Objective 12

1.1.1 Organization 13

1.2 Motivation 13

1.3 Scope 14

2 The Timeliner User Interface Language **15**

2.1 Background 15

2.1.1 Notation 16

2.2 Language Hierarchy 16

2.2.1 Bundles 17

2.2.2 Sequences/subsequences 18

2.2.3 Statements 18

2.2.4 Components 20

2.2.5 Comments 20

2.2.6 Sample Input Script 21

2.3 Custom Adapters 21

2.4 The Timeliner System 22

2.4.1 The Compiler 22

2.4.2 The Executor 24

3 Implementation **27**

3.1 Lexer 28

3.1.1 Directives 28

3.1.2 Regular Expressions. 30
3.2 Parser. 30
3.2.1 Parser Setup. 31
3.2.2 The Grammar. 33
3.3 Abstract Syntax. 36
3.3.1 Abstract Syntax Tree 36
3.3.1.1 Command and Install Statements 39
3.3.2 Manipulating the Tree Nodes. 40
3.3.3 Line Numbers. 41
3.3.4 PPrint 43
3.4 Semantic Analysis 44
3.4.1 Type Checking 44
3.4.1.1 Symbol Table Implementation 45
3.4.1.2 The Ground Database 46
3.4.1.3 Custom Types 46
3.4.2 Semantic Checking. 47
3.4.2.1 Arrays 47
3.5 Tree Serialization 48
3.6 Algebraic Simplification. 50

4 Compiler operations **51**
4.1 Running the Compiler 51
4.2 Extending the Compiler 52
4.2.1 Adding the Print Statement. 52
4.2.2 Adding a New Integer Type 54

5 Results **59**
5.1 Correctness. 59
5.2 Improvements. 60
5.2.1 Lines of Code. 60
5.2.2 Compiler Extensibility 61

6 Discussions **65**

6.1 Future Work 65

6.1.1 Executor. 65

6.1.2 Error Messages. 66

6.1.3 Optimizations. 66

6.1.4 Compiler Output Design. 67

6.2 Conclusion. 67

A Appendix **69**

A.1 File Listings. 69

A.2 Ada Compiler Code Breakdown (Slocs) 71

A.3 Java Compiler Code Breakdown (Slocs) 72

A.4 Makefile. 75

A.5 JLex Specifications File. 77

A.6 Timeliner Grammar. 83

A.7 CUP Specifications File. 88

A.8 XML File Output for Figure 2-2. 103

Bibliography 104

List of Figures

2-1 Hierarchy of input script files 17
2-2 Example input script for the Timeliner compiler. 21
2-3 List file for example input script 23
2-4 The executable data file format 24
2-5 The interactions between the Timeliner executor and other components 25
3-1 Phases of the Timeliner compiler. 27
3-2 Macros defined to recognize integer literals, decimal literals, and exponential literals. 29
3-3 A macro to ensure the language is case insensitive. 29
3-4 Two interpretations for the statement "10 – 2 * 3" 32
3-5 The ELSEIF statement reduction. 35
3-6 CUP productions for the ELSEIF statement 35
3-7 Java™ representation of the abstract syntax tree for the Timeliner grammar. . 38
3-8 Output of PPrint() on the sample input file in Figure 2-2 44
3-9 A graphical view of the Symbol Table data structure 45
3-10 The hierarchy of the different types. 46
3-11 Node replacement by the optimize() method. 50
4-1 The Java™ classes IntegerType and IntegerArrayType 56
5-1 396 Slocs needed to incorporate an Integer type in the Ada compiler 62

Chapter 1

Introduction

When designing a mission-critical system, such as the International Space Station, it is essential that the system be thoroughly tested to prevent potential mishaps during a mission. Any mistake, no matter how small, could devastate the entire mission, costing hundreds of millions of dollars each time.

Testing such a large system can be an expensive task itself, both in terms of human operator time and monetary resources. It would therefore be beneficial to have the ability to do automated testing, where the need for a human operator at the helm is removed by allowing machines to do the work instead. Timeliner is such a system, and has been employed on the Space Shuttle since 1982 [7].

Timeliner was created to emulate the timelines for onboard crew procedures followed by the crew of the Space Shuttle. It was used as a simulation driver in tests of the Space Shuttle system, mimicking crew actions in monitoring and controlling the spacecraft systems [7]. More recently, Timeliner was selected by NASA to be used as the procedure executor for payload development, simulation, and test environments for payloads being developed for the International Space Station (ISS) [7]. Since that time,

Timeliner has evolved into a modular, extensible system that allows scripts to be developed and executed in virtually any systems environment.

By using a system such as Timeliner, we not only obtain monetary savings and productivity gains, but also achieve significant improvements to mission success, reliability, and safety since human errors are eliminated.

1.1 Objective

The objective of this research is to develop a new compiler for the Timeliner system using newer compiler technologies and Java™ in order to augment the extensibility of the compiler and make it easier to maintain. We also introduce modifications to the compiler output in order to simplify the compiler and enhance performance.

Timeliner is a tool to automate procedural tasks. The tasks may be sequential operations that would typically be performed by a human operator, or precisely ordered sequencing tasks that allow autonomous execution of a control process. The Timeliner system consists of a specialized computer language and an execution environment. It includes elements for compiling and executing "sequences" that are predefined in the Timeliner language. The execution environment provides real-time monitoring and control based on the commands and conditions defined in the Timeliner language. The Timeliner sequence control may be pre-programmed, compiled from Timeliner "scripts," or it may consist of real-time, interactive inputs from system operators [7].

1.1.1 Organization

This thesis is presented in six sections. In the introduction, I will set the stage by explaining the background, motivations and scope of the work that has been done. I will then describe the Timeliner language and system in detail. In the third section, I will discuss the actual implementation of the compiler. In the fourth section, I will explain how to operate the compiler, as well as the steps involved in extending the compiler. In the final two sections, I will summarize the results, and discuss the future of the work.

1.2 Motivation

The initial purpose of this project was to make the compiler more manageable for engineers. The current compiler is written in Ada, a language that is not commonly used anymore. By implementing the compiler in a language like Java™, we increase the number of individuals that can contribute to the project. Java™ also allows us to reduce the number of lines of code that currently exist in the compiler source code. By utilizing object-oriented programming and inheritance, we can eliminate repetitive code that exists in the current version of the compiler. Another advantage of Java™ is that it is inherently platform independent. Developers no longer need to work on multiple ports of the program in order to allow Timeliner to run on other platforms. Finally, because there are so many development environments for Java™, this allows people to work in familiar environments, as opposed to having to learn a new programming language for the specific task of write this compiler.

The current compiler is implemented with numerous hard-coded case statements. As a result, the Timeliner grammar is represented implicitly in the code instead of

explicitly as a set of rules and actions. This makes it difficult to understand the language implementation, increasing the resources required to maintain the compiler. In addition, it makes it difficult to extend or modify the language. Newer compiler technologies such as JLex and CUP make it extremely straightforward to create custom lexers and parsers that are easily extensible. These tools require an explicit definition of a language grammar. Moreover, they are backed by large amounts of online documentation and example code, and the tools themselves are widely used in writing compilers and parsers. Rewriting the compiler with these tools will result in a more maintainable and flexible program. It will make it easier for new team members to become acquainted with the Timeliner language and compiler as well as making it easier to modify and/or extend the Timeliner language.

1.3 Scope

Timeliner's execution environment consists of both a compiler and an executor. For the scope of this project, I am only concerned with the compiler portion of the system. Currently, there are multiple implementations of Timeliner that are designed for specific applications. This project deals with the compiler for the User Interface Language that is used by the International Space Station Program Office.

Chapter 2

The Timeliner User Interface Language

2.1 Background

The Timeliner User Interface Language (UIL) is a specialized computer language designed for the writing of sequencing procedures (scripts) to operate a complex system. The language itself is based on temporal constructs such as "when," "every," etc. Tasks can be scheduled to run at specified intervals or under specific conditions. Timeliner is useful because it increases the operational flexibility of an automated system. The scripts are designed to supplement or replace flexibility otherwise provided by human operators. Since Timeliner allows for parallel, automated sequencing, multiple operations and experiments can occur simultaneously, allowing engineers to test the system without having human operators present to re-enact each specific, repetitive task.

The Timeliner system is both a specialized computer language and an execution environment. It includes elements for compiling and executing sequences that are defined in the Timeliner language. The Timeliner compiler and executor allow scripts to be defined, verified, and executed in a rapid, "roll-in" fashion, independent of target system software builds [7].

2.1.1 Notation

An important aspect of the Timeliner language is that it is a simple procedural language that is easy to learn. Each statement follows the general form

KEYWORD <argument> [<optional_args>]

This makes it very easy for new operators to learn the language and use the system effectively.

Another characteristic of the Timeliner language is that it is a time-oriented language. Whereas an ordinary computer program will say

IF *condition,* **DO** *something*

A task-scheduling program like Timeliner will say

WHEN *condition,* **DO** *something*

Timeliner makes heavy use of the concepts embodied in such time-related English words as "when", "whenever", "every", "before", "within", and "wait". A full description of the Timeliner language can be found in [8].

2.2 Language Hierarchy

Because the Timeliner language schedules tasks according to time, it allows the user to create multiple sequences to be processed in parallel. This capability makes it easy to create a procedure that reacts with different actions to multiple conditions that may occur in any order. We achieve this by dividing the input script into "sequences", each functioning as a separate task while operating in parallel with other sequences. These sequences constitute the basic unit into which a Timeliner script is organized. Besides sequences, the Timeliner language also permits the creation of "subsequences". Unlike a

sequence, a subsequence does not create its own stream of execution; it is executed when called by a sequence or another subsequence. It can be considered a helper function that can be reused by multiple sequences/subsequences. Each sequence or subsequence contains statements that are made up of keywords and components. The component is the basic building block of the Timeliner language implementation. Most components represent statements, clauses, or data items of various sorts. Each component is encoded in a set of cells in the *component data table* [8]. A graphical view of the script hierarchy for Timeliner is shown in figure 2-1.

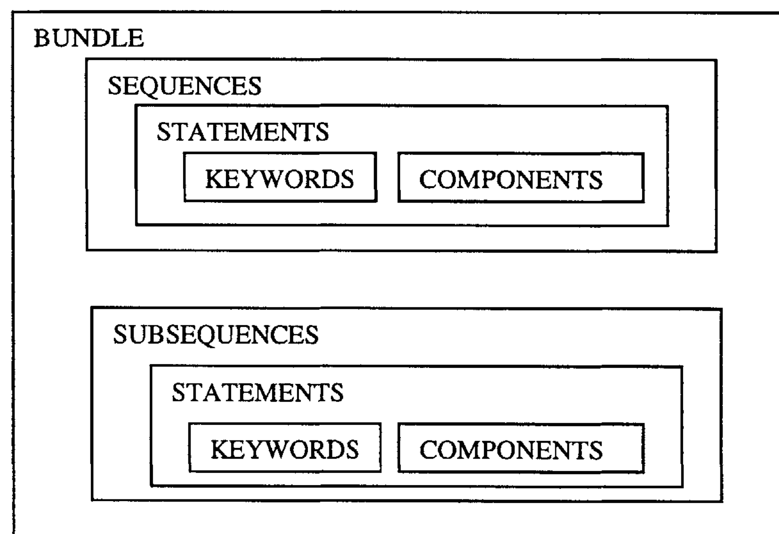


Figure 2-1. Hierarchy of input script files.

2.2.1 Bundles

As we can see from the diagram, the bundle is the uppermost hierarchical level of a UIL script. Each input script file contains one bundle, which in turn contains one or more sequences/subsequences. The maximum number of sequences and subsequences in a bundle are application-dependent. The only kind of statement that may appear inside a

bundle, but outside sequences and subsequences, are DECLARE and DEFINE statements. These statements declare and define variables used within the bundle.

2.2.2 Sequences/subsequences

Sequences/subsequences contain a set of statements that are executed serially. That is, they are executed in their order of occurrence, except as modified by the operation of conditional constructs such as WHEN, WHENEVER, EVERY, and IF. The difference between a sequence and a subsequence is that a sequence establishes an independent thread of execution, and a subsequence does not. As explained earlier, a subsequence must be called by a sequence or another subsequence in order to be executed. Therefore, a subsequence that is not called will compile, but will never execute. Timeliner does not allow for forward reference, so any subsequence that is going to be accessed by a sequence must be declared and defined before the sequence that calls it.

2.2.3 Statements

As shown in Figure 2-1, statements occur inside sequences and subsequences. Every statement starts with a distinctive reserved keyword, also known as the "statement keyword." This keyword uniquely determines the statement type, and must be the first word on a line.

There are four types of statements in Timeliner: Blocking statements, Control statements, Action statements, and Non-Executable statements.

Blocking Statements

Blocking statements are statements used to mark the beginning and end of bundles, sequences, and subsequences. These statements are initiated by the following keywords: BUNDLE, SEQUENCE, SUBSEQUENCE, CLOSE.

Control Statements

Control statements are the principal means provided by Timeliner to allow the user to specify the conditions under which actions should occur. These statements initiate multiple-statement structures, closed by the END statement. Control statements are initiated by the following keywords: WHEN, WHENEVER, EVERY, IF, BEFORE, WITHIN, OTHERWISE, ELSE, ELSEIF, END, WAIT, CALL.

Action Statements

Action statements are statements that perform specific "actions." The SET statement is used to write to an external or internal variable. The COMMAND statement is used to issue commands to the external system. The MESSAGE, WARNING, CONFIRM, QUERY, and DISREGARD statements are used to issue a text message, and INSTALL, REMOVE, HALT, START, STOP, and RESUME statements control the execution of individual bundles and sequences.

Non-Executable Statements

Non-executable statements consist of the DECLARE statement, used to create internal variables, and the DEFINE statement, used to create a name that points to a "component," which is discussed below.

2.2.4 Components

Every statement is made up of keywords and components. A component is a general term used to cover a variety of statement elements that may occur in a Timeliner statement. The components belong to one of the three primitive types in Timeliner: boolean, numeric, and character. Within these types, a component may be a literal, a constant, a combination, a list, a definition, a built-in constant, a built-in function, an internal variable, or an application-defined reference to an external variable [8].

2.2.5 Comments

Input scripts may contain comments and blank lines. Comments are only visible in the raw script. As in any computer language, comments are not retained in the executable code. Two or more adjacent hyphens (i.e. "--") mark the beginning of a comment. All material from the first double hyphen on a line to the end of the line is considered part of the comment. A comment may occupy a line by itself, or share a line with a statement. However, a comment occupying a line by itself may not be present within a multiple-line statement.

2.2.6 Sample Input Script

To give the reader a better idea of how the actual input script file would look, a sample input script is shown in figure 2-2.

```
1       BUNDLE EXAMPLE
2           DECLARE X NUMERIC
3           SEQUENCE ONE
4               WHEN X > 1
5                   SET X = 0
6               END WHEN
7           CLOSE SEQUENCE ONE
8       CLOSE BUNDLE
```

Figure 2-2. Example input script for the Timeliner compiler.

The sample script declares the bundle as "EXAMPLE", and contains one sequence that resets the variable *X* back to zero when it becomes greater than one.

2.3 Custom Adapters

The Timeliner language utilizes the Kernel/Adapter architecture. The kernel contains the necessary code to process the Timeliner language, and the custom adapters provide all the features of the Timeliner system that are dependent on host machine or target system interfaces. This is the current method for extending the Timeliner language. All custom commands and features are defined in adapters that are written for specific applications.

2.4 The Timeliner System

The Timeliner system consists of two segments: the compiler and the executor. The compiler is considered the ground segment, because it is located at operational sites instead of onboard. It converts procedures from ASCII input to executable data tables that are then passed on to the executor. The executor is located onboard the target system, and interfaces with the onboard data management system in order to execute the commands as defined in the data tables.

2.4.1 The Compiler

The design of the compiler system provides several key abstractions. The Timeliner scripts are provided to the compiler as simple ASCII text files, and therefore can be produced from any text editor. The compiler also supports independent definition of system data object and command formats, types and name definitions. In this way, the system definition database may be developed, maintained, and provided for script compilation independent of the Timeliner compiler environment software build, and allows script algorithmic development in abstraction from detailed system data formats [7].

The output of the compiler consists of two files: A listing file that is used for display purposes, and an executable data file that is passed onto the executor. The listing file that would result from the example script above is shown in Figure 2-3.

```
------  TIMELINER EXECUTABLE DATA FOR BUNDLE 'SCRIPT' IS AS FOLLOWS:
------    BLOCK/STATEMENT/COMPONENT INFORMATION AS FILED:
------
------    block     stat    comp    type                      dat1  dat2  dat3  dat4  dat5
------    ------    ----    ----    ---------------------     ----  ----  ----  ----  ----
------    bundle    1       1       BUNDLE_STATEMENT          1     8     1     6     7
------                                                        13
------                      8       NUM_INT_VAR               1     8     8     1
------              2       13      DECLARE_STATEMENT         8
------    seq 1     3       15      SEQ_STATEMENT             3     7     14    16    1
------                                                        0     1     0
------                      24      NUM_NTGR_LIT              1
------                      26      BOOLEAN_COMBO             1     8     6     24
------              4       31      WHEN_STATEMENT            6     0     26
------                      35      NUM_NTGR_LIT              2
------              5       37      SET_STATEMENT             8     35
------              6       40      END_STATEMENT             4
------              7       42      CLOSE_STATEMENT           3
------              8       44      CLOSE_STATEMENT           1
------    NUMERIC LITERALS:
------
------    1       1.00000000000000E+00    0.00000000000000E+00
------    CHARACTER LITERALS:
------
------    1       SCRIPTEXAMPLE
```

Figure 2-3. List file for example input script.

The executable data file contains information including the application-dependent header, program memory requirements, the sizes and offsets of various data, the tables accumulated by the compiler (table of sequence and subsequence information, table of statement information, table of component data, and tables of numeric and character string literals), and information on areas to be allocated at the time of installation onto the executor. Figure 2-4 shows the relationship graphically.

In order to gain information on onboard data and commands, the compiler interfaces with a ground database that contains the necessary information.

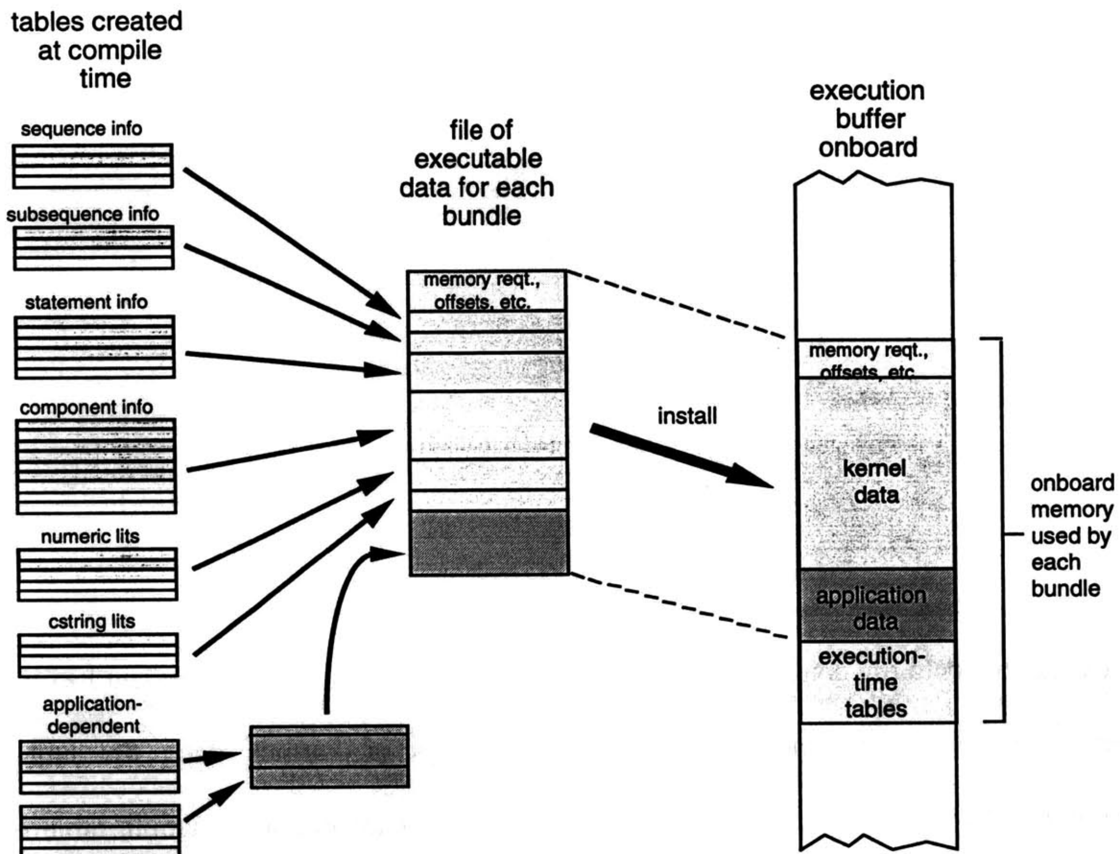


Figure 2-4. The executable data file format.

2.4.2 The Executor

The executor is an engine that works as an interpreter to process the files of executable data that have been compiled from the script. It interprets the executable data file, reads the appropriate data from the target system, uses the data to evaluate conditional and time-based statements, and then writes data back to the target system in order to control its behavior. The listing file is used to help correlate the data to the display, providing messages that are comprehensible to the human operator. The executor supports execution and independent control of multiple bundles, in parallel, which may itself

contain multiple sequences that execute in parallel (either in a synchronous or asynchronous fashion) and can also be independently controlled [7].

In addition to running executable data files, the executor can also respond to commands from a mission operator. Commands entered by the operator are treated like any other operation, and are scheduled by the task scheduler within the executor. The figure below illustrates the interactions between the executor and other parts of the system.

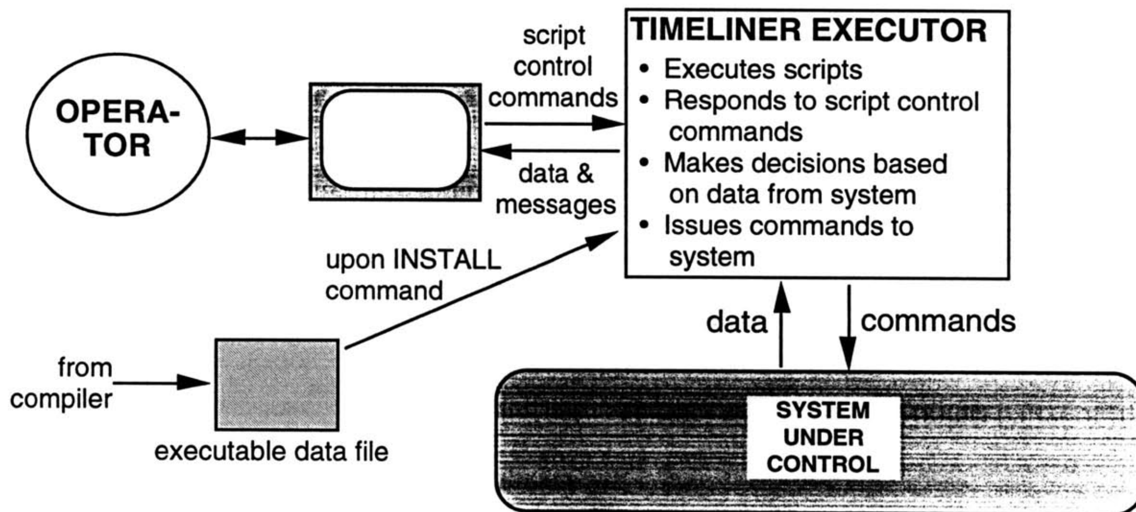


Figure 2-5. The interactions between the Timeliner executor and other components [1].

This page intentionally left blank

Chapter 3

Implementation

My implementation of the Timeliner compiler was done in Java™, to make use of the object-oriented programming methodology, as well as Java™'s platform independence. The compiler consists of five main stages. First, the source file is passed through the lexical analyzer, which breaks the input into tokens. The file then goes through a parser that groups the tokens according to the Timeliner grammar. Next, an AST (abstract syntax tree) is built to create a clean interface between the parser and the later phases of the compiler. We do the semantic analysis by walking the abstract syntax tree, and the output to the executor is generated via methods implemented in the tree nodes. Figure 3-1 shows a block diagram of the compiler phases.

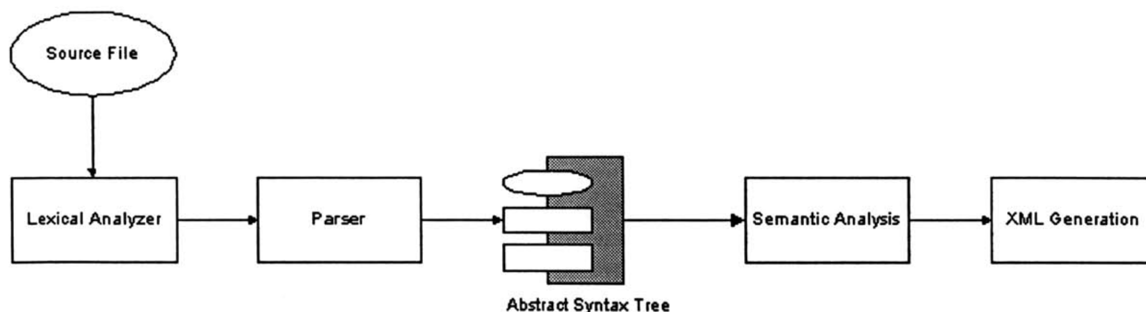


Figure 3-1. Phases of the Timeliner compiler.

3.1 Lexer

The first stage of the compiler involves breaking the input file into individual custom tokens. As defined in [2], a token is "a sequence of characters that can be treated as a unit in the grammar of a programming language." In order to break the input file into tokens, I used JLex [3], a lexical analyzer generator. JLex produces a Java™ program from a lexical specification file. For each token type in the programming language to be lexically analyzed, the specification contains a regular expression [1] and an action. The action passes along information about the token type as well as other information to the next phase of the compiler. The specification file that JLex takes in consists of two main sections: JLex directives and Regular Expression Rules. The two sections are discussed below, and the actual lexical specification file can be found in Appendix A.5.

3.1.1 Directives

In the Directives section, several private methods that are used by the compiler are defined. In order to print out useful error messages, we want to keep track of the current line number as the lexer traverses through the input file. This is accomplished by including a private field that stores the current line number. Each time we reach the end of a line, a method is called to increase the line number by one. In addition to the private methods, this section also contains macros that help define tokens. Macro definitions are valid regular expressions, which are defined in [3].

Numeric Literals

Because Timeliner is a time-oriented language, it has to be able to recognize time literals. The language also provides support for integers, decimals, and exponentials. In order to distinguish them as tokens, macros were used. The macros used to define several types of numeric literals are shown in figure 3-2.

```
digit = [0-9]
int_lit = (({digit})+)
num_lit = ({int_lit}|{dec_lit})
exp_lit = ({num_lit}(E|e)(\+|\-)?{int_lit})
dec_lit = (({int_lit})(\.)({int_lit}))
```

Figure 3-2. Macros defined to recognize integer literals, decimal literals, and exponential literals.

Keywords

As stated in section 2.1.1, Timeliner statements consist of keywords and components. Therefore, every keyword should be a token. Because the language is case insensitive, the lexer has to account for all possible lower and upper case combinations. This was accomplished by defining custom macros for each keyword. Figure 3-3 shows the macro defined for the keyword BUNDLE.

```
bundle = (B|b)(U|u)(N|n)(D|d)(L|l)(E|e)
```

Figure 3-3. A macro to ensure the language is case insensitive.

3.1.2 Regular Expressions

The regular expressions section of the lexical specification tells the lexer what to do when it recognizes something as a token. Each line consists of two parts: The regular expression corresponding to a specific type of token, and the action to take when JLex locates such a token. In most cases, an object of type `java_cup.runtime.Symbol` [5] is returned. It contains information on what kind of token it is, the token's leftmost character position (in the input file), the token's rightmost character position (in the input file), as well as the token's value, represented as a Java™ `String` object. When the token is a numeric literal, however, we want to pass the actual numeric value, not the string representation of the numeric literal. To do this, two methods (FixUpIntLit() and FixUpDecLit()) are called that convert the string token into its numeric equivalent. This value is used to create the `java_cup.runtime.Symbol` object, which is passed on to later phases of compilation. This section also contains regular expressions that tell the lexer what to do with new lines, tabs, comments, and anything else defined in Timeliner. If something within the input file does not match up with any of the regular expressions, an error is reported.

3.2 Parser

The second stage of the compiler is the parser. In this stage, tokens from the lexer are passed to the parser, and are grouped into phrases that fit within the grammar of the language. To describe the language, a context-free grammar is used. The grammar has a set of productions that are of the form:

symbol -> symbol symbol … symbol

where there are zero or more symbols on the right-hand side [2]. Each symbol is either a terminal (a token in the language) or a nonterminal (it appears on the left-hand side of some production in the grammar). More information on context-free grammars can be found in [1, 2]. The parser breaks down the input file by reducing multiple tokens into a nonterminal when the tokens match a specific production. This reducing is done at multiple levels until it reaches a top-level nonterminal.

The Timeliner parser was created with CUP, a custom parser generator that generates an LR(1) parser (implemented in Java™) from a specifications file. An LR(1) parser is a Left-to-right parse, Rightmost-derivation, 1 token look-ahead parser [2]. The CUP specs file contains two main parts: Parser setup and defining the grammar. Parser setup consists of integrating the lexer with the parser, declaring the necessary terminals and nonterminals, and correctly setting up the precedence. Detailed explanation on the specs file can be found in [5]. The actual CUP specs file is located in Appendix A.7.

3.2.1 Parser Setup

The first part of parser setup is integrating the lexer with the parser. In order to do this, we declare a constructor that takes in one parameter of type `TimelinerTokenizer`. The `TimelinerTokenizer` object is the lexer created by JLex. The constructor then calls the default parser constructor, and sets the lexer to the one that was passed in to the constructor.

The next part of parser setup is declaring the necessary terminals and nonterminals. Before we can use the terminals that are passed in by the lexer, we must declare them in CUP. The same is true for nonterminals. Before CUP can make sense of

the grammar, it must be aware of any nonterminals that are going to appear in the grammar. By default, each terminal and nonterminal are created as objects of type `java.lang.Object` when CUP generates the parser code. However, CUP gives us the capability to associate each terminal/nonterminal with a specific Java™ object type. This will come in play later on when we create Custom Java™ data types to represent the nonterminals. For now, we are only concerned with the terminals. Terminals STRLIT and ID are declared as String objects, and INTLIT and DECLIT's are declared as `Long` and `Float` objects, respectively.

The last part of parser setup is setting precedence. This is used to resolve conflicts in ambiguous grammars. An ambiguous grammar is one that can be reduced in more than one way. For example, if the grammar contains productions such as

expr → integer | decimal
expr → expr MINUS expr
expr → expr TIMES expr

Then a statement such as 10-2*3 can be interpreted two different ways, as shown in figure 3-4.

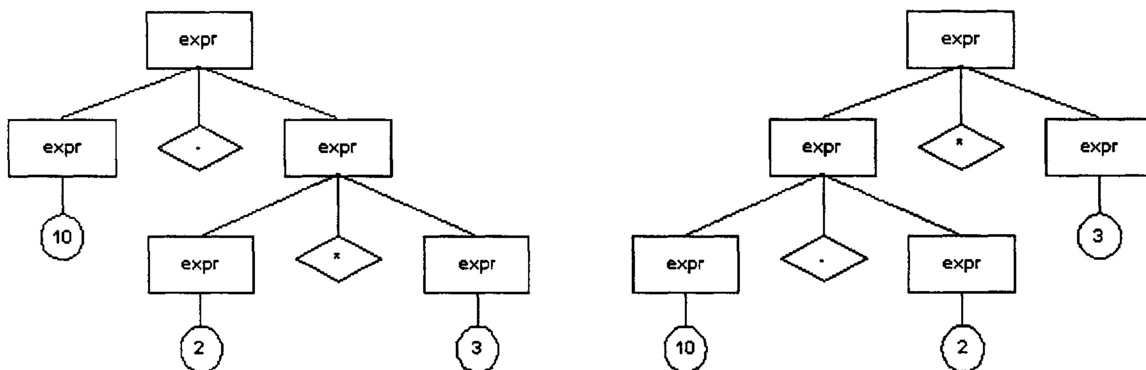


Figure 3-4. Two interpretations for the statement "10 – 2 * 3".

In order to resolve these conflicts, we make use the precedence directive. The precedence directive includes two parameters. The first is the associativity, the second is a comma-separated list of terminals that belong to a certain precedence. The order of precedence is bottom up, with the last precedence statement having the highest precedence. So to resolve the conflict from the grammar above, we include the following precedence statements:

```
Precedence left MINUS
Precedence left TIMES
```

Since `TIMES` has a higher precedence than `MINUS`, the parser will reduce the statement correctly and perform the multiplication before the addition. The associativity is used to resolve conflicts of equal precedence. It can have values *left*, *right*, and *nonassoc*. If the associativity of a terminal is *left*, then the parser will reduce them left to right, and vice versa. So a statement such as 5 – 3 – 2 will be reduced from left to right, because the associativity for the `MINUS` terminal is left. If a terminal is declared as *nonassoc*, then the terminal cannot appear consecutively without generating an error.

3.2.2 The Grammar

The grammar for Timeliner appears in Appendix A.6. Most of the productions are fairly straightforward due to the <keyword> <component> structure of the language. There are a few cases, however, that are worth discussing.

One-line WHEN Statement

Timeliner permits several forms of the WHEN statement. One of them being the one-line WHEN statement. This form of the WHEN construct consists of a single statement of the form:

WHEN <singular_boolean> **[THEN] CONTINUE**

This statement causes the sequence to pause until the condition <singular_boolean> is true. When the condition is met execution continues with the next statement. This is the only instance of a control statement where the statement is not closed with an END statement. Therefore, there was no way for the parser to know when to reduce this specific production, because it does not require an END statement. In order to compensate for this construct, we note two things about this particular statement. First, we do not need to worry about encapsulating another set of statements, because this statement consists of only one-line. Second, the keyword CONTINUE is only used in the one-line WHEN statement. It is therefore sufficient to have a production specifically for the WHEN... CONTINUE case in the Timeliner grammar. In this case, the CONTINUE keyword acts as the statement closer, as opposed to the typical END <cond_word> keywords.

Else If Statements

Timeliner allows for IF, ELSEIF, and ELSE statements. Because the syntax for IF statements and ELSEIF statements are exactly the same, we could be tempted to treat them the same way. Both require a boolean expression as the condition, and both can be

nested within other statements. There are two problems with that approach. First, an ELSEIF statement cannot exist without an IF statement. This requirement is easy to fulfill in CUP. Second, if we did treat them the same way, CUP has no way to determine whether the END statement goes with the ELSEIF or the IF preceding it. Figure 3-5 shows the problem graphically.

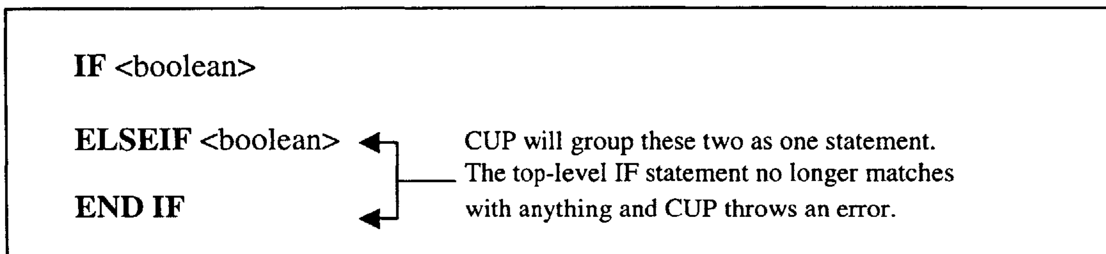
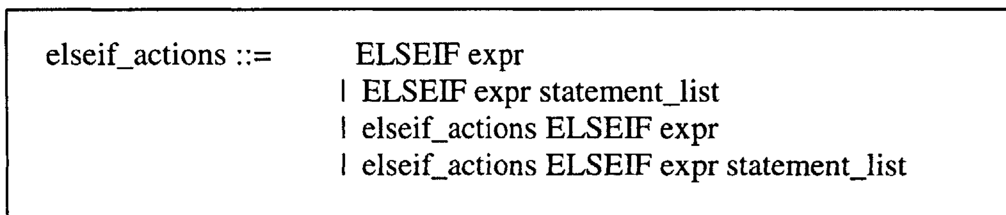


Figure 3-5. The ELSEIF statement reduction.

In order to solve this problem, we have to create new productions to deal with ELSEIF statements. Figure 3-6 shows the corresponding CUP productions for ELSEIF statements.



```
elseif_actions ::=    ELSEIF expr
                    | ELSEIF expr statement_list
                    | elseif_actions ELSEIF expr
                    | elseif_actions ELSEIF expr statement_list
```

Figure 3-6. CUP productions for the ELSEIF statement.

ELSEIF statements are treated just like ELSE statements, except there's no limit to the number of ELSEIF statements that can occur.

3.3 Abstract Syntax

To this point, the compiler can only recognize whether a statement belongs in the grammar, but it has no idea what the statement means. The semantic actions of the parser help us give meaning to the statements. CUP allows us to incorporate the semantic actions into the CUP specs file as Java™ statements. Immediately after each production, anything that falls between a set of {: :} brackets will be executed when that production is reduced [2]. One approach, therefore, is to include all the semantic actions within the parser. This is actually the approach taken by the current Timeliner compiler. It combines parsing and semantic actions all into one step, creating a large file that is difficult to read and maintain. Also, it forces the compiler to analyze the program in the order that it is parsed. We can improve the compiler's modularity and ease of maintenance by separating parsing from semantics.

One clean interface between the parser and the later phases of a compiler is an abstract syntax tree. The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation [2].

3.3.1 Abstract Syntax Tree

In building the abstract syntax tree, I followed, for the most part, the principles outlined in Section 1.3 of [2]. The guidelines are as follows:

1. A tree is described by one or more abstract classes, each corresponding to a symbol in the grammar.
2. Each abstract class is extended by one or more subclasses, one for each grammar rule.
3. For each nontrivial symbol in the right-hand side of a rule, there will be one field in the corresponding class.
4. Every class will have a constructor function that initializes all the fields.

5. Data structures are initialized when they are created (by the constructor functions), and are never modified after that.

In other words, every nonterminal was represented in Java™ with an abstract class, and when these nonterminals appeared on the right-hand side of productions, they were represented in Java™ by subclasses that extend from the abstract class. Each Java™ object is a node in the tree, and every class in the abstract syntax tree is a subclass of the class TreeNode. Figure 3-7 shows the Java™ representation of the abstract syntax tree for the Timeliner grammar. Each directed edge represents the relationship between tree nodes; each undirected edge links a base class and its subclasses.

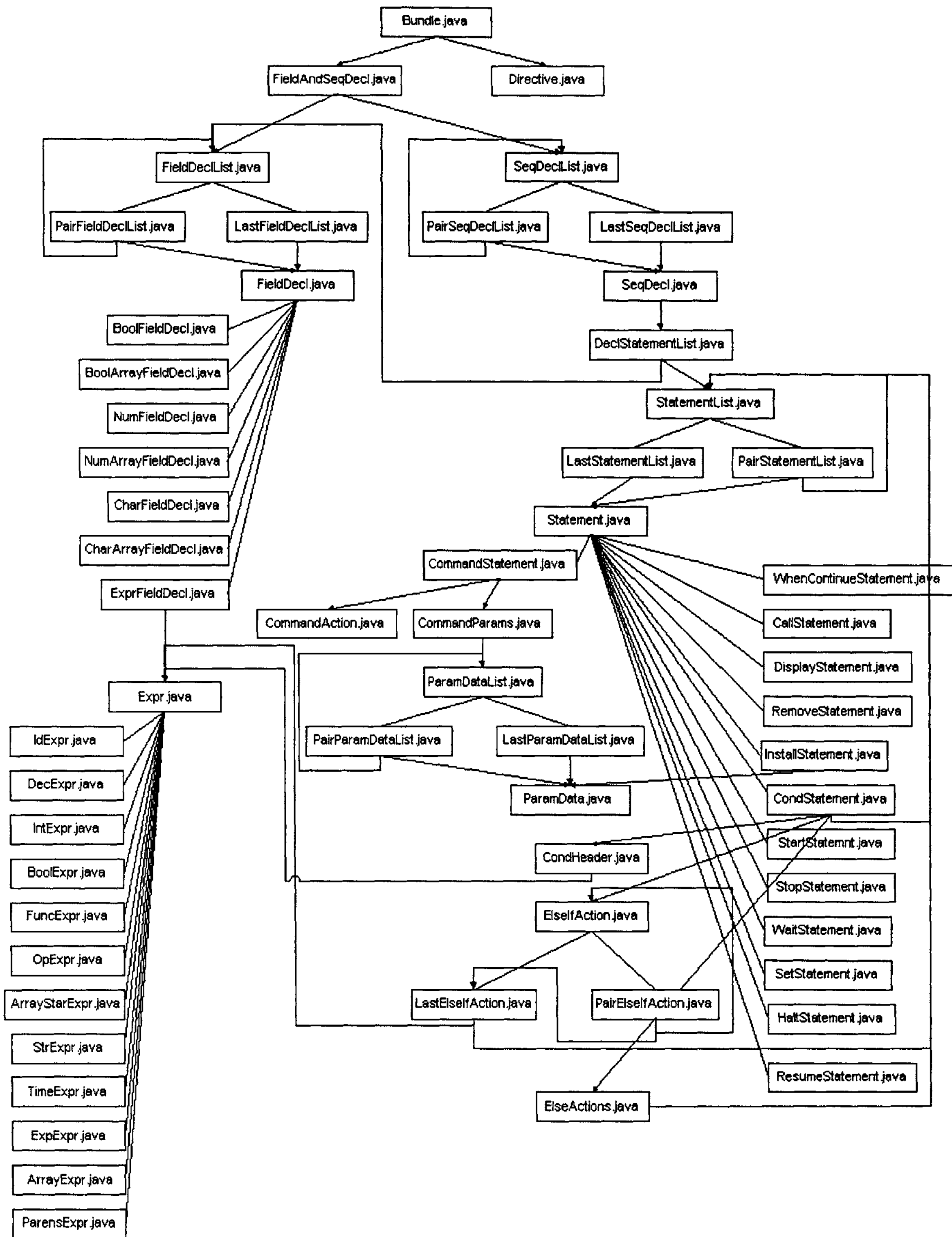


Figure 3-7. Java™ representation of the abstract syntax tree for the Timeliner grammar.

3.3.1.1 COMMAND and INSTALL Statements

One case where I had to deviate from the guidelines for building the abstract syntax tree was representing the parameters for COMMAND and INSTALL statements. The COMMAND statement is used to output an action with accompany parameters, if necessary. It is of the form:

COMMAND <action> {, <parameter_N_name> => <parameter_N_value> }

where <action> consists of one or more words specifying the specific action and its destination, <parameter_N_name> represents the name of a parameter that is part of the command, and <parameter_N_value> represents the value assigned to the parameter. The INSTALL statement commands the Executor to install a bundle. It is of the form:

INSTALL <bundle_file_name> {, <parameter_N_name> => <parameter_N_value> }

where <bundle_file_name> is the name, including the path or directory information, of a file containing the executable data of the bundle to be installed.

The syntax for these statements are identical, except for one thing: COMMAND statements have no limits on the number of parameters that can appear; INSTALL statements are limited to a maximum of two parameters. Instead of being able to handle both parameter lists with a common production, we need to have separate productions to handle each of these cases. When representing them as Java™ objects, however, there is no reason to have different implementations that hold exactly the same information. In order to solve this problem, I used Java™ interfaces. A Java™ interface defines a set of methods but does not implement them. Any class that implements the interface will

implement all the methods defined in the interface, thereby agreeing to certain behavior. By creating a class ParamData that implemented the CommandParam and InstallParam interfaces, we have one implementation that is used for both statements, and we still abide by the guidelines above of having different classes representing each right hand side of a production.

3.3.2 Manipulating the Tree Nodes

After the tree is created, we have to choose an approach that is going to be used to manipulate the tree nodes. One approach is to create data structures that have public class variables that represent subtrees, and are examined with `instanceof` statements [2]. This programming style is called *syntax separate from interpretations.* Another approach to this problem is to have no public class variables. Any use of these objects must be through public methods declared in each class. This is the *object-oriented* style of programming [2].

The two styles have an effect on the modularity of the compiler. Whereas the *syntax separate from interpretations* method makes it easy to add new *interpretations* of the objects, the *object-oriented* style makes it easy to add new *kinds* of objects. An *interpretation* can be thought of as a translation of the objects. For example, an interpretation might translate the code from one language to another, or optimize the existing code, etc. A language contains many *kinds* of objects, like assignment statements, print statements, command statements, etc. Each *kind* of object represents something in the language [2].

The *syntax separate from interpretations* method allows us to write one new function for each interpretation, with clauses for the different kinds all grouped logically together [2], making it very easy to create new *interpretations.* This method can be implemented in practice by using the Visitor pattern. In the Visitor pattern, related operations from each class are packaged into a "visitor" that is fed elements of the abstract syntax tree as it is traversed. This makes it easy to create new operations that need to traverse the tree. However, it is not easy to add a new *kind* of object, because cases must be inserted in each new operation. The *object-oriented* method is exactly the opposite: It is easy to create new *kinds* of objects, but hard to create new *interpretations.* With the *object-oriented* style, each interpretation is a method that is found in every class. Therefore, to add a new *kind,* we simply create a class and implement all of the methods that correspond to the different *interpretations.* But to add a new *interpretation,* we would have to add a new method in every class.

One of the goals for the new compiler was that it be extendable. The ability to add to the Timeliner language is very important. It is therefore advantageous to use the *object-oriented* style of programming described above. In order to manipulate the tree nodes, we have to call public methods that are implemented in each class.

3.3.3 Line Numbers

When the compiler displays error messages, it is extremely useful to be able to include the line number in the input file that the error occurred on. In a one-pass compiler, where everything (lexical analysis, parsing, semantic analysis) is done simultaneously, this is an easy task. We can simply have the lexical analyzer keep track of the current line number,

and return it whenever an error is encountered. In the case where abstract syntax trees are used, however, this strategy will not work. Since parsing and semantic analysis are not done in one pass, we will have reached the end of the input file before we even begin semantic analysis, making the current line number that the lexical analyzer is holding useless. Thus, we must keep track of the source file line number of each node in the abstract syntax tree.

To keep track of the line number of each node, a *line* field is inserted in the TreeNode class. Because every class in the abstract syntax tree extends from the TreeNode class, they will inherit the *line* field that is in TreeNode. Every constructor must then include a parameter that specifies its own line number.

In order to get the actual line number, we could use the `getLine()` function that is already implemented to display line numbers during parsing errors. Every time a production is reduced, we could immediately retrieve the line number from the scanner and use that as the line number parameter in the constructor. The problem with this approach is that productions may span multiple lines. Thus the line number that is returned could be extremely inaccurate, and would not be useful. We could get slightly better results by recognizing that because every object has a *line* field, we could reuse these values on productions that contain multiple objects. So a complex production's line number would be equivalent to the line number of the first Java™ object in its production. This turns out to work pretty well, but there are still instances where the line number is wrong because even the simplest reduction spanned multiple lines.

The problem with the approach above was that it retrieved the current line number when an entire production is matched. This gives the wrong result when a production

spans multiple lines in the source file. If, however, we could gain access to the line number of each token instead of each production, we could then use that line number to represent the entire production. In my implementation of the lexer, the object that is passed to the parser is of type `java_cup.runtime.Symbol`. This object contains information including the type id, the position of the leftmost and rightmost character of the token in the source file, and the value of the object. After inspecting the .java file that is automatically generated by CUP, I saw that every time a production is reduced, CUP automatically creates `int` variables that represent the leftmost and rightmost character of each symbol. Since I am not making use of these two variables, I changed the lexer so that instead of passing the rightmost character of the token, it passed the line number of the token in the source file. Since I have already written methods to retrieve the line numbers in the lexer, this method allowed us to return useful line numbers without having to make substantial changes/additions to the lexer/parser.

3.3.4 PPrint

In order to make sure the abstract syntax tree was being generated correctly and to demonstrate its object-oriented nature, I inserted `PPrint()` methods in every class. Each object would print itself and recursively call `PPrint()` on its child nodes. To show the different levels of the tree, I added a `String` parameter to the method. Before the node prints itself, it would print the `String` parameter first to show what level it is on. It then adds to the `String` parameter and passes that parameter on to the children's `PPrint()` methods. Figure 3-8 shows the tree returned by `PPrint()` for the sample input file in Figure 2-2.

```
+-BUNDLE:EXAMPLE
|  |
|  +-FieldAndSeqDecl
|  |  |
|  |  +-NumFieldDecl: DECLARE X NUMERIC
|  |  |
|  |  +-SEQUENCE Decl: ONE
|  |  |  -> STATUS:= INACTIVE
|  |  |  |
|  |  |  +-WHEN X > 1
|  |  |  |  +-SET X TO 0
|  |  |  --END WHEN
|  |  --END SEQUENCE: ONE
|  |
|  --END FieldAndSeqDecl
|
--END BUNDLE:EXAMPLE
```

Figure 3-8. Output of PPrint() on the sample input file in Figure 2-2.

3.4 Semantic Analysis

In the semantic analysis phase, we begin to examine the source file to make sure that not only is it correct syntactically, but also semantically. That is, the expression inside a conditional statement is boolean, both sides of an assignment statement are of the same types, an access of a particular array element is within the array's bounds, etc. The semantic analysis phase is characterized by the maintenance of symbol tables [2] that map identifiers to their types.

3.4.1 Type Checking

The symbol table is used to store the identifiers and their types so that they may be retrieved later on for type checking purposes. If we did not have to worry about variable scopes, then a simple hashtable would be sufficient. However, Timeliner does allow for

local variables to override global variables. Therefore, in order to keep track of the different scopes, a different data structure is needed to represent the symbol table.

3.4.1.1 Symbol Table Implementation

In order to keep track of the different scopes, the data structure in Figure 3-9 was implemented. Each table represents a scope, and contains a pointer to the scope directly above it. So if a variable does not exist in the current scope, the compiler would follow the pointer to the parent table and attempt to search for the variable there. Various references have suggestions for other implementations, but this implementation was the simplest. Using the `java.util.Hashtable` class eliminated the need to implement our own hashtable, reducing the amount of work.

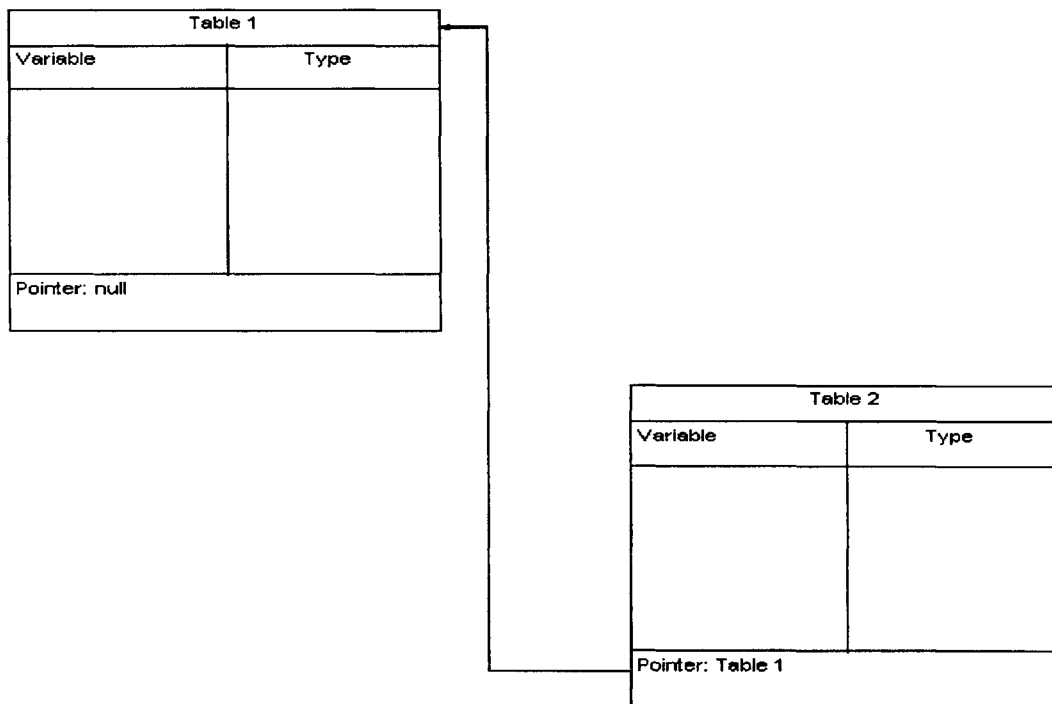


Figure 3-9. A graphical view of the Symbol Table data structure.

3.4.1.2 The Ground Database

In addition to the symbol tables, Timeliner also has a ground database that contains the commands/fields that exist on the target system. If it is found that a variable does not exist in the symbol tables, the compiler must then access the Ground database (GDB) and check for its existence there. We created an interface to interact with the GDB. The interface requires a String object representing the variable name, and returned the type associated with the external variable.

3.4.1.3 Custom Types

The Timeliner language has three basic types: Character, Numeric, and Boolean. These types are supplemented by their respective array counterparts. To store these types in the symbol table, custom data types were created to represent each of them. The hierarchy of the types is shown in figure 3-10.

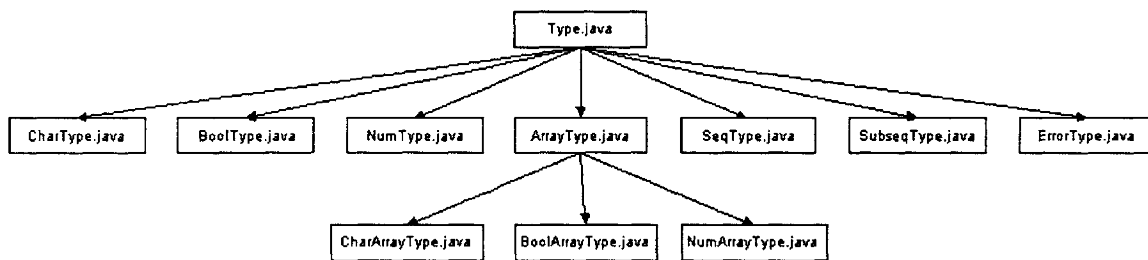


Figure 3-10. The hierarchy of the different types.

By creating custom data types to represent them, the code for semantic checking is cleaner, and we are able to hold more information about each object. For example, the `ArrayType` not only describes the variable's type, but also contains a field that indicates the size of the array. This implementation augments the ability to extend the language by creating an abstraction barrier so that future engineers in charge of extending

the language will only have to know the appropriate methods associated with each type without having to worry about the actual implementation.

3.4.2 Semantic Checking

The actual semantic checking is done by inserting the `CheckSemantics()` method in each subclass of TreeNode. Initially, an empty instance of the symbol table is passed to the root node, and semantic analysis is done recursively as each parent node calls the `CheckSemantics()` method if its children with the modified symbol table. Each time a DECLARE statement is encountered, an entry is added to the symbol table with the `put()` method. This method checks to make sure the variable does not already exist in the current scope before inserting it into the symbol table. Each time a variable is used, the `get()` method is called that returns the variable's type. The symbol table traverses through the tables to search for the variable, from the local scope all the way up to the GDB.

The `SeqDecl` statement represents the beginning of a new scope. Variables declared inside a sequence or subsequence is only valid within the sequence/subsequence, and overrides any global variables of the same name. In this statement, the `CheckSemantics()` method is called with a new symbol table object that contains a pointer to the existing symbol table.

3.4.2.1 Arrays

Timeliner is a very straightforward language for the most part, but there are a few cases involving arrays that are noteworthy. First, an assignment such as

SET ARRAY1 TO NUMVAR

where ARRAY1 represents an array and NUMVAR represents a numeric variable is allowed. This simply sets every element of ARRAY1 to the value NUMVAR. Another assignment statements that is specific to the language is shown below:

SET ARRAY1 TO (ARRAY2(1..7), ARRAY3(1..3))

The statement above sets ARRAY1 to contain the first seven elements of ARRAY2, followed by the first three elements of ARRAY3. If ARRAY1's size is not 10, an error is displayed.

Besides assignments, arrays can also be compared as part of a conditional statement. For example, the IF statement

```
IF bool_array1 = NOT bool_array2 THEN
        <statements>
```

Compares the values of each elements in the boolean arrays, and if they are all equal then the statements in the IF case are executed. These cases, and many others, are all accounted for in the `CheckSemantics()` method. If the input file contains no errors, the information is written to a file that is passed on to the executor for execution.

3.5 Tree Serialization

The existing compiler output is a set of data tables that the executor then interprets during execution. Generating the data tables is a straightforward process given the abstract syntax tree. After further consideration, however, it seems wasteful to generate the data tables in the first place. The tree is walked once to generate the data tables that are stored in a file. The executor then processes this file to execute the corresponding actions. The format of the data tables (Figure 3-9), however, is complicated and difficult the process.

A better solution would be to somehow pass the abstract syntax tree to the executor, allowing it to walk the tree during execution, instead of interpreting the set of complex data tables. This ultimately removes a layer of processing from the entire procedure, simplifying the process and improving performance.

In order to pass the tree on to the executor, the tree was written to a file using XML. There were a number of reasons for using XML. First, XML was designed to describe data. [6] defines XML as "a standard language used to structure and describe data that can be understood by different applications." Second, XML is easy to process. There are numerous XML parsers readily available that make the process of reconstructing the tree from the XML file extremely straightforward. The most important feature of XML is that it is extensible. The ability to create custom tags for custom types is important given the goal of this project. XML allows us to create and insert new types with ease.

XML Custom Types

The `GenXML()` method was implemented for each subclass of TreeNode to produce the output file. The method takes in a `java.io.FileWriter` pointer, and writes to the specified file. Each internal class variable in each Java™ class is treated as a custom type with a set of custom tags. To describe information such as the variable's actual Java™ type, the custom tags describing the variable name are followed by a set of tags that indicate the actual Java™ Object type. The necessary headers are written within the `GenXML()` method in Bundle.java. The XML file generated for the input file in Figure 2-2 is located in Appendix A.8.

3.6 Algebraic Simplification

The abstract syntax tree allows us to do algebraic simplification very easily. This is because any simplification must be done on objects of type `Expr`. So algebraic simplification can take place by making every `Expr` object check itself. This can be done by implementing an `optimize()` method in the `Expr` class, and have any object that contains `Expr's` make the `optimize()` call in the constructor. The base `optimize()` method simply returns a pointer to the object itself. This is the method that every subclass of `Expr` inherits. Only those subclasses that could potentially be optimized will have their own implementation of `optimize()`. For example, in order to simplify expressions such as "1 + 2" to "3", we can have an overriding version of `optimize()` in the `OpExpr` class. The method first checks to make sure both sides are constants. If so, a new object of type `IntExpr` whose value is three is returned. Otherwise, a pointer to the current object is returned. This node replacement is shown pictorially in figure 3-11.

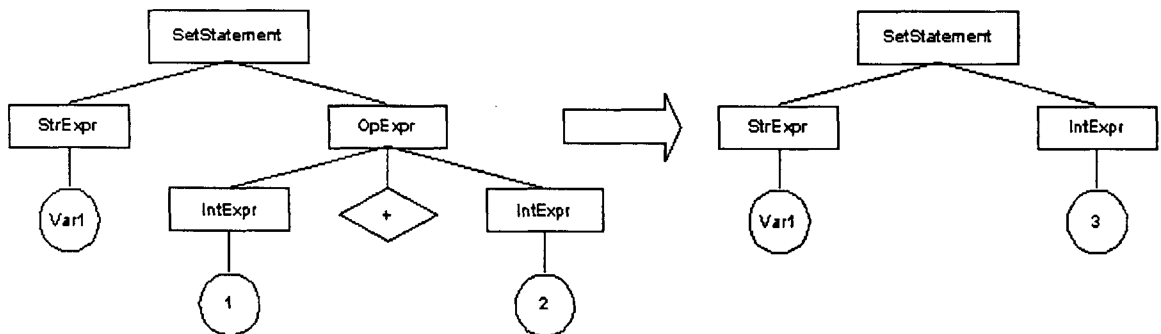


Figure 3-11. Node replacement by the optimize() method.

Chapter 4

Compiler Operations

4.1 Running the Compiler

The new compiler can be found in SSFSUN1 under the ftl2704/TL_JavaCompiler/ directory. This directory contains all of the necessary files and subdirectories needed to make and run the compiler. Users within Draper Labs can also download a local copy of the compiler via CVS by checking out the ftl2704 module. This creates the same directory structure and downloads the same files as those located in /ftl2704/TL_JavaCompiler. A makefile (Appendix A.4) is included in the set of files. The user can then create an executable version of the compiler by typing "make new" in the command prompt.

To run the compiler, the user should type "java Compiler " followed by the filename. This will create a file that can be passed to the executor for execution. Typing "java Compiler" by itself will display a list of compiler options.

4.2 Extending the Compiler

One of the main purposes of the project was to create a compiler that can be extended to include changes/additions to the language. To demonstrate the new compiler's extensibility, we will describe the process to add a new PRINT statement to the Timeliner language with the syntax

PRINT <string_literal>

and also how to incorporate a new Integer type into the compiler.

4.2.1 Adding the Print Statement

In order to add a new statement to the Timeliner language, we have to make changes to the lexer, parser, and the abstract syntax tree. The sections below illustrate the steps involved in each part of the process.

Lexical Analyzer

The first step is to incorporate the PRINT keyword into the lexical analyzer as a new token. In the JLex file, we first add the following macro in the appropriate section

```
print = (P|p)(R|r)(I|i)(N|n)(T|t)
```

Then, the following regular expression is added to the file

```
{print}        {return tok(sym.PRINT), yytext()); }
```

The PRINT keyword is now incorporated in the lexical analyzer as a custom token. Note that this will not compile correctly because sym.PRINT does not exist yet. The sym.java

file is generated by the CUP parser generator. In order for the JLex file to compile, we must make changes to the parser.

Parser

The first addition to the CUP file is in the terminal declaration section. The PRINT token must be declared a terminal in order for CUP to recognize it correctly. Since the PRINT statement is a type of statement, we do not need to introduce a new nonterminal.

To add the new statement to the existing grammar, we add a new case to the `statement` production in the grammar. The new production is shown below, with the other cases removed.

```
Statement ::= PRINT STRLIT
```

Note that the terminal STRLIT represents a string literal, which is the necessary parameter for the PRINT statement. Compiling the files now will generate the correct Java™ files, but because only the grammar has been added, no semantic actions take place.

Creating the Java™ Classes

The next step is to create the PrintStatement class that is used in creating the abstract syntax tree. This class must extend the abstract `Statement` class, which in turn extends the TreeNode class, making the PrintStatement class a subclass of TreeNode. In order to meet the requirements, the PrintStatement class must implement the `PPrint()` and `GenXML()` methods. The signatures for these and other methods are found in

TreeNode.java. TreeNode.java also contains information on the correct formatting of the XML output. Once these methods have been implemented, the PRINT statement is ready to be incorporated into the compiler. The last step is to insert the corresponding Java™ segment into the CUP file. The resulting production is given below:

```
statement ::= PRINT STRLIT:str_lit
          {: RESULT = new PrintStatement(str_lit,
                                 str_litright); :}
```

Note that the line number is captured by the `str_litright` variable (section 3.3.3). The compiler is now able to correctly handle the new PRINT statement, assuming the Java™ methods were correctly implemented.

4.2.2 Adding a New Integer Type

The current Timeliner language supports three basic types: boolean, numeric, and character. A new Integer type would essentially render the numeric type obsolete. We should note that it would make sense to add both a new Float type and a new Integer type, but since the steps involved are nearly identical, we only show the steps involved in creating a new Integer type. The type will be used in variable declarations, and will have the syntax

DECLARE <variable_name> INTEGER [(<integer_literal>)]

The steps involved in adding a new type to the Timeliner language is very similar to the steps described above for adding a new statement.

Lexical Analyzer

The first step is to incorporate the INTEGER keyword into the lexical analyzer as a new token. In the JLex file, we first add the following macro in the appropriate section

```
integer = (I|i)(N|n)(T|t)(E|e)(G|g)(E|e)(R|r)
```

Then, the following regular expression is added to the file

```
{integer}        {return tok(sym.INTEGER), yytext()); }
```

The INTEGER keyword is now incorporated in the lexical analyzer as a custom token. Once again, this will not compile until we have incorporated the changes into the parser.

Parser

The first change is to declare the INTEGER token as a terminal. Since the INTEGER type is only used in variable declarations, we do need to introduce a new nonterminal.

To allow the new type to be used in variable declarations, we add a new case to the `field_decl` production in the grammar. The new productions are shown below, with the other cases removed.

```
field_decl ::= DECLARE var_name INTEGER
             | DECLARE var_name INTEGER LPAREN INTLIT RPAREN
```

Note that the nonterminal var_name represents any acceptable variable name. The next step is to create the new Java™ classes that are associated with these productions.

Creating the Java™ Classes

The next step is to create the IntegerFieldDecl and IntegerArrayFieldDecl classes that are used in creating the abstract syntax tree. These classes must extend the abstract FieldDecl class, which in turn extends the TreeNode class. In order for these classes to compile, they must implement the `PPrint()` and `GenXML()` methods. In addition, each class should have a `CheckSemantics()` method that overrides the default version. This method is essentially identical to the ones found in the classes NumFieldDecl and NumFieldDecl.

Because we are introducing a new basic type into the language, we must also add it to our Java™ classes. To accomplish this, we have to create two more classes, IntegerType and IntegerArrayType. These classes are essentially tags that are used to check against each other during semantic checking. The actual Java™ codes for these two files are shown in Figure 4-1.

```
Public class IntegerType extends Type {
        Public IntegerType() {
                Type = Type.INTEGER;
        }
}

public class IntegerArrayType extends ArrayType {
        public IntegerArrayType(int l) {
                type = Type.INTEGER_ARRAY;
                length = l;
        }
}
```

Figure 4-1. The Java™ classes IntegerType and IntegerArrayType.

Before attempting to compile the classes, note that we must add two public static integer variables to Type.java: INTEGER and INTEGER_ARRAY.

The last step is to insert the corresponding Java™ segment into the CUP file. The resulting production is given below:

```
field_decl ::= DECLARE var_name:vn INTEGER
               {:RESULT = new IntegerFieldDecl(vn,
                                   vn.getLine()); :}
             | DECLARE var_name:vn INTEGER LPAREN
               INTLIT:length RPAREN
               {:RESULT = new IntegerArrayFieldDecl(vn,
                                   length, vn.getLine());:}
```

The line number for the new FieldDecl objects is obtained by retrieving the line number for the nonterminal `var_name`. The compiler is now able to correctly handle the new Integer type.

This page intentionally left blank

Chapter 5

Results

The goal of the project was to create a new compiler for Timeliner that was extensible and easier to maintain. Two inherent criteria for the project was that the compiler be correct, and can be deemed an improvement over the previous version.

5.1 Correctness

Draper Labs had a large suite of test files that were used to test the current compiler. The suite contains not only test files, but also the compiler outputs. These files were used to test the correctness of the new compiler. The files were compiled with the new compiler, and the results were compared with the results from the old compiler. In every case, files containing syntax or semantic errors were caught. For those files that compiled without errors, the resulting XML files were checked to make sure the tree was correctly represented in each file. Besides the test files found in the Draper directories, another suite of test files were created in order to test other unusual cases. In every case the compiler either generated the correct XML output or caught at least one error.

5.2 Improvements

Two metrics were used to see whether this project was an improvement over the previous version of the compiler. The first metric is the sheer number of lines of code it took to write the compiler. Since one of the goals of this project was to use an object-oriented language like Java™ to promote code reuse, we should be able to reduce the lines of code needed to create the compiler. The second metric is ease of extensibility. By comparing the process of extending the compiler between the two versions, we are able to show the improvement in extensibility in the new compiler.

5.2.1 Lines of Code

One metric that can be used to deem this project an improvement is the number of statements in the two versions of the compiler. A statement is defined here as a line of code that is followed by a semicolon. The original compiler contained a total of 4539 statements. A complete breakdown of the modules is found in Appendix A.2.

In counting the number of statements in the new compiler, we have a slightly different approach. Instead of counting the number of statements in the Java™ files generated by JLex and CUP, we counted the number of statements in the JLex and CUP specification files. Since the statements in the JLex and CUP specification files do not always end in a semicolon, these files were treated differently. In the JLex file, each macro definition and regular expression was treated as a statement. In the CUP file, every terminal/nonterminal line, as well as each precedence rule, was considered a

statement. In the end, the new compiler contained a total of 2104 statements. This total is broken down into the individual files in Appendix A.3.

By taking advantage of newer technologies like JLex, CUP, and Java™, we were able to dramatically reduce the amount of code needed to create the compiler.

5.2.2 Compiler Extensibility

The old compiler was written such that each Ada file contained all the information necessary for every possible statement in the Timeliner language. Therefore, adding/changing the compiler would involve changes in all of the major compiler modules. The table in Figure 5-1 shows the number of Slocs (Source Line of Code) that are needed to incorporate the change.

Module	Old Slocs	New Slocs	Delta Slocs
Common Modules (Compiler and Executor)			
TL_SEED_S	47	57	+10
TL_COMMON_S	540	553	+13
TL_XXX_ADAPT_INFO_S	47	51	+4
Compiler Modules			
TL_COMP_COM_S	120	130	+10
TL_COMP_COM_B	131	131	0
TL_COMP_SUBS_S	22	23	+1
TL_COMP_SUBS_B	1685	1950	+265
TL_COMP_UTIL_S	61	63	+2
TL_COMP_UTIL_B	452	461	+9
TL_CUSSER_S	22	25	+3
TL_CUSSER_B	257	276	+19
TL_PARSER_B	911	971	+60
TL_XXX_ADAPT_PARSING_S	12	12	0
COMPILER CHANGE (including common modules)			+396

Figure 5-1. 396 Slocs needed to incorporate an Integer type in the Ada compiler.†

In section 4.2.2, we described the steps needed to incorporate the Integer type into the new compiler. While changes were needed in every compiler phase, the changes were very simple and straightforward. Incorporating the INTEGER keyword into the lexical analyzer took two lines of code, adding the necessary productions in the parser were another two lines of code. The Java™ classes IntegerFieldDecl and IntegerArrayFieldDecl are essentially identical to the classes NumFieldDecl and NumArrayFieldDecl, respectively, and therefore contain roughly the same number of statements (around 20 each). Finally, the IntegerType and IntegerArrayType classes

† Internal Draper Memo.

contain three total Slocs, making the total number of statements needed to incorporate the Integer type in the new compiler an estimated 47 Slocs. This is a dramatic improvement over the 396 Slocs needed to make the change in the old compiler.

This page intentionally left blank

Chapter 6

Discussions

While the compiler that I have produced is not a turnkey system, it does meet the project goals stated earlier. It makes use of the Java™ programming language in order to make extensive use of inheritance and code reuse. It is an easily extensible compiler that is also easy to maintain. It makes use of the XML language in order to output a serialized version of the abstract syntax tree that can be passed into a new executor to eliminate an intermediate layer that currently exists in the Timeliner system.

One drawback to the new compiler is that there is no executor in place that makes use of the XML output. However, this compiler provides a great starting point in the creation of a new Timeliner system that is not only easier to maintain, but better performance wise as well.

6.1 Future Work

6.1.1 Executor

The most important thing that needs to be done for this project is the development of a new executor. This executor will be capable of taking in an XML file as the input,

recreate the tree from the XML file, and execute statements by walking the tree. As described in Chapter 2, the executor is also a task scheduler. It must be capable of handling multiple Bundles, scheduling the tasks in the bundles accordingly. Therefore, the executor must keep track of the current tree node in each bundle, and also have a way to walk back up the tree.

The development of a new executor will be able to fully demonstrate the power of the new compiler, and should eventually replace the existing Timeliner implementation.

6.1.2 Error Messages

The goal of the project was to demonstrate the ability to create a compiler that exceeds the current compiler in terms of extensibility and ease of maintenance. Therefore, less emphasis was placed on catching as many errors as possible in one pass. While no erroneous input files passed the compiler without being caught, there are cases where not all of the errors were caught in one pass. Before the compiler can go live, more time should be spent on not only error detection, but also the production of clear, concise error messages.

6.1.3 Optimizations

Currently, the only optimization present in the compiler is algebraic simplification. Some of the semantic errors are also left for the executor to deal with (as with the Ada version of the compiler). Ideally, we should try and move as much of the computation as we can

to the compiler, simplifying the executor. This is beneficial because the input files are compiled on the ground, and then sent to the executor onboard the target system. By simplifying the executor, we lower the amount of resources it takes up in the target system.

6.1.4 Compiler output design

Another topic worth investigating is the XML format of the output file. While the current format is a valid XML document, it seems to be longer than necessary. It would be interesting to see whether it would be possible to modify the format so that it conveys the same information with less code.

6.2 Conclusion

This project was dedicated to the creation of an extensible, object-oriented compiler for the Timeliner User Interface Language. The goals were to create a compiler that was extensible, and made use of modern compiler technologies to make the compiler easier to maintain. In chapter four, we described the steps involved in extending the compiler to accommodate changes in the Timeliner language. In chapter five we showed the dramatic improvements that we were able to achieve through the new version of the compiler both in terms of extensibility but also in terms of reducing the overall number of lines of code. By making the compiler itself extensible, we eliminate the need to write complicated custom adapters for each specific application of Timeliner, thus eliminating a layer of complexity from the entire process.

In addition, we also introduced a new output format that carries the information necessary to rebuild the abstract syntax tree. By passing the tree to the executor instead of a set of data tables, we essentially eliminate another layer of complexity from the Timeliner system. Instead of having to walk the tree to produce the data tables that the executor must comprehend, we have allowed the executor to directly walk the tree and extract the necessary information that way.

Before the system is fully operational, the new executor must be implemented. When this is done, users of the Timeliner system will be able to appreciate the improvements that have come about through this project.

Appendix A

A.1 File Listings

TL_Java_Compiler/
Compiler.java

TL_Java_Compiler/scanner/
RunScanner.java
TimelinerTokenizer (JLex file)

TL_Java_Compiler/parser/
RunParser.java
Timeliner.cup (CUP file)
Sym.java (Generated by CUP)

TL_Java_Compiler/ast/
ArrayExpr.java
ArrayStarExpr.java
BoolArrayFieldDecl.java
BoolExpr.java
BoolFieldDecl.java
Bundle.java
CallStatement.java
CharArrayFieldDecl.java
CharFieldDecl.java
CommandAction.java
CommandParams.java
CommandStatement.java
CondHeader.java
CondStatement.java
DecExpr.java
DeclStatementList.java
Directive.java
DisplayStatement.java
ElseActions.java
ElseIfActions.java
ExpExpr.java
Expr.java
ExprFieldDecl.java
FieldAndSeqDecl.java
FieldDecl.java
FieldDeclList.java
FuncExpr.java
HaltStatement.java
IdExpr.java
InstallParams.java
InstallStatement.java
IntExpr.java
LastElseIfActions.java
LastFieldDeclList.java
LastParamDataList.java
LastSeqDeclList.java
LastStatementList.java
NumArrayFieldDecl.java
NumFieldDecl.java
OpExpr.java
PairElseIfActions.java
PairFieldDeclList.java
PairParamDataList.java
PairSeqDeclList.java
PairStatementList.java
ParamData.java
ParamDataList.java
ParensExpr.java
RemoveStatement.java
ResumeStatement.java
RunAST.java
SeqDecl.java
SeqDeclList.java
SetStatement.java
StartStatement.java
Statement.java

StatementList.java
StopStatement.java
StrExpr.java
TimeExpr.java
TreeNode.java
WaitStatement.java
WhenContinueStatement.java

TL_Java_Compiler/tools/
ArrayType.java
BoolArrayType.java
BoolType.java
CharArrayType.java
CharType.java
CommandLineParser.java
ErrorMsg.java
ErrorType.java
GDB.java
NumArrayType.java
NumType.java
SeqType.java
SubseqType.java
Substring.java
Table.java
Type.java

A.2 Ada Compiler Code Breakdown (Slocs)

Common Modules	**Slocs**
tl_seed_s.ada	47
tl_common_s.ada	540
tl_cdh_adapt_info_s.ada	60

Compiler Modules	
tl_compile_b.ada	42
tl_comp_com_s.ada	120
tl_comp_com_b.ada	131
tl_parser_s.ada	9
tl_parser_b.ada	911
tl_comp_subs_s.ada	22
tl_comp_subs_b.ada	1685
tl_comp_util_s.ada	61
tl_comp_util_b.ada	452
tl_cusser_s.ada	22
tl_cusser_b.ada	257
tl_data_conversion_s.ada	3
tl_data_conversion_b.ada	146
tl_xxx_adapt_constants_s.ada	5
tl_xxx_adapt_data_output_s.ada	5
tl_xxx_adapt_file_write_s.ada	9
tl_xxx_adapt_parsing_s.ada	12

Total Slocs	**4539**

A.3 Java™ Compiler Code Breakdown (Slocs)

File	Slocs
TL_Java_Compiler/	
Compiler.java	33
TL_Java_Compiler/scanner/	
RunScanner.java	114 (declare all the syms)
TimelinerTokenizer	189 (112 + 77 non-; statements)
TL_Java_Compiler/parser/	
RunParser.java	17
Timeliner.cup	260
Sym.java	95
TL_Java_Compiler/ast/	
ArrayExpr.java	26
ArrayStarExpr.java	18
BoolArrayFieldDecl.java	18
BoolExpr.java	14
BoolFieldDecl.java	15
Bundle.java	36
CallStatement.java	13
CharArrayFieldDecl.java	18
CharFieldDecl.java	15
CommandAction.java	24
CommandParams.java	7
CommandStatement.java	39
CondHeader.java	36
CondStatement.java	41
DecExpr.java	14
DeclStatementList.java	24
Directive.java	13
DisplayStatement.java	24
ElseActions.java	18
ElseIfActions.java	4
ExpExpr.java	13
Expr.java	6
ExprFieldDecl.java	25
FieldAndSeqDecl.java	28
FieldDecl.java	4
FieldDeclList.java	4
FuncExpr.java	41

HaltStatement.java	12
IdExpr.java	15
InstallParams.java	4
InstallStatement.java	17
IntExpr.java	15
LastElseIfActions.java	23
LastFieldDeclList.java	14
LastParamDataList.java	15
LastSeqDeclList.java	14
LastStatementList.java	14
NumArrayFieldDecl.java	19
NumFieldDecl.java	15
OpExpr.java	94
PairElseIfActions.java	21
PairFieldDeclList.java	21
PairParamDataList.java	22
PairSeqDeclList.java	21
PairStatementList.java	21
ParamData.java	21
ParamDataList.java	5
ParensExpr.java	18
RemoveStatement.java	11
ResumeStatement.java	13
RunAST.java	24
SeqDecl.java	46
SeqDeclList.java	4
SetStatement.java	27
StartStatement.java	14
Statement.java	4
StatementList.java	4
StopStatement.java	17
StrExpr.java	13
TimeExpr.java	13
TreeNode.java	9
WaitStatement.java	17
WhenContinueStatement.java	16

TL_Java_Compiler/tools/

ArrayType.java	3
BoolArrayType.java	4
BoolType.java	2
CharArrayType.java	3
CharType.java	2
CommandLineParser.java	35
ErrorMsg.java	9
ErrorType.java	2

GDB.java	78
NumArrayType.java	3
NumType.java	2
SeqType.java	2
SubseqType.java	2
Substring.java	6
Table.java	32
Type.java	20
Total Slocs	**2104**

A.4 Makefile

```
JAVAC = javac
JAVAI = java

new: clean all

all: tools compiler

clean:
	rm -rf parser/*.class scanner/*.class ast/*class tools/*.class data_table/*.class *.class

compiler: Compiler.class

Compiler.class:		scanner/RunScanner.class \
			parser/RunParser.class\
			ast\
			dt\
			Compiler.java
			$(JAVAC)   Compiler.java


parser: parser/RunParser.class

scanner: scanner/RunScanner.class

ast: ast/Bundle.class
	(cd ast && $(JAVAC) *.java);

dt: data_table/RunDataTable.class

tools: CommandLineParser.class \
	OutputWriter.class \
	ErrorMsg.class \
	Substring.class \
	Type.class \
	Table.class \
	GDB.class

############# TOOLS ############

CommandLineParser.class: tools/CommandLineParser.java
	(cd tools && $(JAVAC) CommandLineParser.java);

OutputWriter.class: tools/OutputWriter.java
```

```
	(cd tools && $(JAVAC) OutputWriter.java);

ErrorMsg.class: tools/ErrorMsg.java
	(cd tools && $(JAVAC) ErrorMsg.java);

Substring.class: tools/Substring.java
	(cd tools && $(JAVAC) Substring.java);
Type.class: tools/Type.java
	(cd tools && $(JAVAC) Type.java BoolArrayType.java BoolType.java \
	CharArrayType.java CharType.java NumArrayType.java NumType.java \
	SeqType.java SubseqType.java ErrorType.java ArrayType.java);

Table.class: tools/Table.java
	(cd tools && $(JAVAC) Table.java);

GDB.class: tools/GDB.java
	(cd tools && $(JAVAC) GDB.java);

############## Data Table #############
data_table/RunDataTable.class: data_table/GenTable.class
	(cd data_table && $(JAVAC) RunDataTable.java);

data_table/GenTable.class: data_table/GenTable.java
	(cd data_table && $(JAVAC) GenTable.java);

############# Parser ############
parser/RunParser.class: parser/TimelinerParser.class parser/RunParser.java
	(cd parser && $(JAVAC)   RunParser.java);

parser/TimelinerParser.class CUP$Parser$actions.class: parser/TimelinerParser.java
	(cd parser && $(JAVAC)   TimelinerParser.java  sym.java);


parser/TimelinerParser.java parser/sym.java: parser/timeliner.cup scanner/RunScanner.class
	(cd parser && $(JAVAI)   java_cup.Main -parser TimelinerParser < timeliner.cup);

############ Scanner ###########
scanner/RunScanner.class: scanner/TimelinerTokenizer.class scanner/RunScanner.java
	(cd scanner && $(JAVAC) RunScanner.java);

scanner/TimelinerTokenizer.class: scanner/TimelinerTokenizer.java
	(cd scanner && $(JAVAC) TimelinerTokenizer.java);

scanner/TimelinerTokenizer.java: scanner/TimelinerTokenizer
	(cd scanner && $(JAVAI) JLex.Main TimelinerTokenizer);
```

A.5 JLex Specifications File

```
//user code
package scanner;

import java.io.*;
import parser.sym;


%%
//JLex Directives -- these basically name the class, define names
//of functions, etc.

%public
%class TimelinerTokenizer
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%line
%char

//the lines below contain private functions that will be put
//into the actual tokenizer java file
%{
 private int lineNumber = 1;

 private java_cup.runtime.Symbol tok(int kind, Object value) {
  //return new java_cup.runtime.Symbol(kind, yychar, yychar+yylength(), value);
  //use the Symbol.right as the line number
  return new java_cup.runtime.Symbol(kind, yychar, this.getLine(), value);
 }

 public int getLine() {
  return lineNumber;
 }

 private void newline() {
  lineNumber++;
 }

 private Long FixupIntLit(String s) {
        Long returnValue = new Long(0);
        try {
         returnValue = Long.valueOf(s);
        }
```

```
        catch(java.lang.NumberFormatException e) {
                error("Bad integer literal");
        }
        return returnValue;

 }

 private Float FixupDecLit(String s) {
        Float returnValue = new Float(0);
        try {
          returnValue = Float.valueOf(s);
        }
        catch(java.lang.NumberFormatException e) {
                error("Bad decimal literal");
        }
        return returnValue;

 }

 private void error(String s) {
   System.err.println("\nLexical error: "+s+": "+lineNumber+": "+yytext()+"\n");
 }

%}

//this is what the tokenizer will do when it reaches
//the end of the file -- return an EOF token
%eofval{
  return tok(sym.EOF, "EOF");
%eofval}

//I'm defining macros below that help distinguish between tokens

alpha=[A-Za-z_]
digit=[0-9]
str_char=(\\\"|\\\'|\\\\|\\n|[^\'\n])
cha_char=(\\\"|\\\'|\\\\|\\n|[^\"\n])
alpha_num=(({alpha})|({digit}))
int_lit=(({digit})+)
exp_lit=({num_lit}(E|e)(\+|\-)?{int_lit})
dec_lit=(({int_lit})(\.)({int_lit}))
id = {alpha}({alpha_num})*
string_lit=((\")({str_char})*(\")|({char_lit}))
char_lit=(\')({cha_char})*(\')
num_lit=({int_lit}|{dec_lit})
time_lit=({int_lit}\:)+{num_lit}
```

```
//the following are the keywords that I have picked up.  The use
// of the parenthesis are to account for all possible cases
// (Bundle, BuNdLe, etc.)

bundle = (B|b)(u|U)(n|N)(d|D)(l|L)(e|E)
sequence = ((s|S)(e|E)(q|Q)(u|U)(e|E)(n|N)(c|C)(e|E))|((s|S)(e|E)(q|Q))
active = (a|A)(c|C)(t|T)(i|I)(v|V)(e|E)
inactive = (i|I)(n|N)(a|A)(c|C)(t|T)(i|I)(v|V)(e|E)
subsequence =
(((s|S)(u|U)(b|B)(s|S)(e|E)(q|Q)(u|U)(e|E)(n|N)(c|C)(e|E))|((s|S)(u|U)(b|B)(s|S)(e|E)(q|Q)))
close = (c|C)(l|L)(o|O)(s|S)(e|E)
when = (w|W)(h|H)(e|E)(n|N)
whenever = (w|W)(h|H)(e|E)(n|N)(e|E)(v|V)(e|E)(r|R)
every = (e|E)(v|V)(e|E)(r|R)(y|Y)
if = (i|I)(f|F)
before = (b|B)(e|E)(f|F)(o|O)(r|R)(e|E)
within = (w|W)(i|I)(t|T)(h|H)(i|I)(n|N)
otherwise = (o|O)(t|T)(h|H)(e|E)(r|R)(w|W)(i|I)(s|S)(e|E)
else = (e|E)(l|L)(s|S)(e|E)
elseif = ((e|E)(l|L)(s|S)(e|E)(i|I)(f|F))|((e|E)(l|L)(s|S)(i|I)(f|F))
end = (e|E)(n|N)(d|D)
wait = (w|W)(a|A)(i|I)(t|T)
call = (c|C)(a|A)(l|L)(l|L)
continue = (c|C)(o|O)(n|N)(t|T)(i|I)(n|N)(u|U)(e|E)
then = (t|T)(h|H)(e|E)(n|N)
set = (s|S)(e|E)(t|T)
to = (t|T)(o|O)
command = ((c|C)(o|O)(m|M)(m|M)(a|A)(n|N)(d|D))|((c|C)(m|M)(d|D))
message = (m|M)(e|E)(s|S)(s|S)(a|A)(g|G)(e|E)
warning = (w|W)(a|A)(r|R)(n|N)(i|I)(n|N)(g|G)
confirm = (c|C)(o|O)(n|N)(f|F)(i|I)(r|R)(m|M)
query = (q|Q)(u|U)(e|E)(r|R)(y|Y)
disregard = (d|D)(i|I)(s|S)(r|R)(e|E)(g|G)(a|A)(r|R)(d|D)
pause = (p|P)(a|A)(u|U)(s|S)(e|E)
install = (i|I)(n|N)(s|S)(t|T)(a|A)(l|L)(l|L)
remove = (r|R)(e|E)(m|M)(o|O)(v|V)(e|E)
halt = (h|H)(a|A)(l|L)(t|T)
start = (s|S)(t|T)(a|A)(r|R)(t|T)
stop = (s|S)(t|T)(o|O)(p|P)
resume = (r|R)(e|E)(s|S)(u|U)(m|M)(e|E)
declare = (d|D)(e|E)(c|C)(l|L)(a|A)(r|R)(e|E)
boolean = (b|B)(o|O)(o|O)(l|L)(e|E)(a|A)(n|N)
numeric = (n|N)(u|U)(m|M)(e|E)(r|R)(i|I)(c|C)
character = (c|C)(h|H)(a|A)(r|R)(a|A)(c|C)(t|T)(e|E)(r|R)
define = (d|D)(e|E)(f|F)(i|I)(n|N)(e|E)
as = (a|A)(s|S)
```

```
and = (a|A)(n|N)(d|D)
or = (o|O)(r|R)
not = (n|N)(o|O)(t|T)
mod = (m|M)(o|O)(d|D)
in = (i|I)(n|N)
outside = (o|O)(u|U)(t|T)(s|S)(i|I)(d|D)(e|E)
true = (t|T)(r|R)(u|U)(e|E)
on = (o|O)(n|N)
false = (f|F)(a|A)(l|L)(s|S)(e|E)
off = (o|O)(f|F)(f|F)
abs = (a|A)(b|B)(s|S)
sqrt = (s|S)(q|Q)(r|R)(t|T)
sin = (s|S)(i|I)(n|N)
cos = (c|C)(o|O)(s|S)
tan = (t|T)(a|A)(n|N)
arcsin = (a|A)(r|R)(c|C)(s|S)(i|I)(n|N)
arccos = (a|A)(r|R)(c|C)(c|C)(o|O)(s|S)
arctan = (a|A)(r|R)(c|C)(t|T)(a|A)(n|N)
time = (t|T)(i|I)(m|M)(e|E)
directive = ((d|D)(i|I)(r|R)(e|E)(c|C)(t|T)(i|I)(v|V)(e|E))|((d|D)(i|I)(r|R)(e|E)(c|C)(t|T))
step = ((s|S)(t|T)(e|E)(p|P))
hold_at = ((h|H)(o|O)(l|L)(d|D)_(a|A)(t|T))
jump_to = ((j|J)(u|U)(m|M)(p|P)_(t|T)(o|O))


//this section is the section that tells the tokenizer what to
//do when it recognizes something as a token
%%

[" "\t\f\^M]          { }
\n                    { newline(); }
\-\-.*                { }
{bundle}              {return tok(sym.BUNDLE, yytext());}
{sequence}            {return tok(sym.SEQUENCE, yytext()); }
{active}              {return tok(sym.ACTIVE, yytext()); }
{inactive}            {return tok(sym.INACTIVE, yytext()); }
{subsequence}         {return tok(sym.SUBSEQ, yytext()); }
{close}               {return tok(sym.CLOSE, yytext()); }
{when}                {return tok(sym.WHEN, yytext()); }
{whenever}            {return tok(sym.WHENEVER, yytext()); }
{every}               {return tok(sym.EVERY, yytext()); }
{if}                  {return tok(sym.IF, yytext()); }
{before}              {return tok(sym.BEFORE, yytext()); }
{within}              {return tok(sym.WITHIN, yytext()); }
{otherwise}           {return tok(sym.OTHERWISE, yytext()); }
{else}                {return tok(sym.ELSE, yytext()); }
```

```
{elseif}        {return tok(sym.ELSEIF, yytext()); }
{end}           {return tok(sym.END, yytext()); }
{wait}          {return tok(sym.WAIT, yytext()); }
{call}          {return tok(sym.CALL, yytext()); }
{continue}      {return tok(sym.CONTINUE, yytext()); }
{then}          {return tok(sym.THEN, yytext()); }
{set}           {return tok(sym.SET, yytext()); }
{to}            {return tok(sym.TO, yytext()); }
{command}       {return tok(sym.COMMAND, yytext()); }
{message}       {return tok(sym.MESSAGE, yytext()); }
{warning}       {return tok(sym.WARNING, yytext()); }
{confirm}       {return tok(sym.CONFIRM, yytext()); }
{query}         {return tok(sym.QUERY, yytext()); }
{disregard}     {return tok(sym.DISREGARD, yytext()); }
{pause}         {return tok(sym.PAUSE, yytext()); }
{install}       {return tok(sym.INSTALL, yytext()); }
{remove}        {return tok(sym.REMOVE, yytext()); }
{halt}          {return tok(sym.HALT, yytext()); }
{start}         {return tok(sym.START, yytext()); }
{stop}          {return tok(sym.STOP, yytext()); }
{resume}        {return tok(sym.RESUME, yytext()); }
{declare}       {return tok(sym.DECLARE, yytext()); }
{boolean}       {return tok(sym.BOOLEAN, yytext()); }
{numeric}       {return tok(sym.NUMERIC, yytext()); }
{character}     {return tok(sym.CHARACTER, yytext()); }
{define}        {return tok(sym.DEFINE, yytext()); }
{directive}     {return tok(sym.DIRECTIVE, yytext()); }
{as}            {return tok(sym.AS, yytext()); }
{and}           {return tok(sym.AND, yytext()); }
{or}            {return tok(sym.OR, yytext()); }
{not}           {return tok(sym.NOT, yytext()); }
{mod}           {return tok(sym.MOD, yytext()); }
{in}            {return tok(sym.IN, yytext()); }
{outside}       {return tok(sym.OUTSIDE, yytext()); }
{true}          {return tok(sym.TRUE, yytext()); }
{on}            {return tok(sym.ON, yytext()); }
{false}         {return tok(sym.FALSE, yytext()); }
{off}           {return tok(sym.OFF, yytext()); }
{abs}           {return tok(sym.ABS, yytext()); }
{sqrt}          {return tok(sym.SQRT, yytext()); }
{sin}           {return tok(sym.SIN, yytext()); }
{cos}           {return tok(sym.COS, yytext()); }
{tan}           {return tok(sym.TAN, yytext()); }
{arcsin}        {return tok(sym.ARCSIN, yytext()); }
{arccos}        {return tok(sym.ARCCOS, yytext()); }
{arctan}        {return tok(sym.ARCTAN, yytext()); }
```

```
{time}          {return tok(sym.TIME, yytext()); }
{step}          {return tok(sym.STEP, yytext()); }
{hold_at}       {return tok(sym.HOLD_AT, yytext()); }
{jump_to}       {return tok(sym.JUMP_TO, yytext()); }
{int_lit}       {return tok(sym.INTLIT, FixupIntLit(yytext())); }
{dec_lit}       {return tok(sym.DECLIT, FixupDecLit(yytext())); }
{string_lit}    {return tok(sym.STRLIT, yytext()); }
{time_lit}      {return tok(sym.TIMELIT, yytext()); }
{exp_lit}       {return tok(sym.EXPLIT, yytext()); }
"=>"            {return tok(sym.PARAMETER_OP, yytext()); }
\=              {return tok(sym.EQ, yytext()); }
"/="            {return tok(sym.NEQ, yytext()); }
\>              {return tok(sym.GT, yytext()); }
\<              {return tok(sym.LT, yytext()); }
"<="            {return tok(sym.LEQ, yytext()); }
">="            {return tok(sym.GEQ, yytext()); }
\,              {return tok(sym.COMMA, yytext()); }
\.              {return tok(sym.DOT, yytext()); }
".."            {return tok(sym.DOTDOT, yytext()); }
\+              {return tok(sym.PLUS, yytext()); }
\-              {return tok(sym.MINUS, yytext()); }
\*              {return tok(sym.MULT, yytext()); }
V               {return tok(sym.DIVIDE, yytext()); }
"**"            {return tok(sym.MULTMULT, yytext()); }
\(              {return tok(sym.LPAREN, yytext()); }
\)              {return tok(sym.RPAREN, yytext()); }
\{              {return tok(sym.LBRACE, yytext()); }
\}              {return tok(sym.RBRACE, yytext()); }
\[              {return tok(sym.LBRACKET, yytext()); }
\]              {return tok(sym.RBRACKET, yytext()); }
\;              {return tok(sym.SEMICOLON, yytext()); }
{id}            {return tok(sym.ID, yytext()); }
.               {error(yytext()); }
```

A.6 Timeliner Grammar

```
program ::=  BUNDLE ID field_seq_decl CLOSE BUNDLE
        | directive BUNDLE ID field_seq_decl CLOSE BUNDLE
        | BUNDLE ID field_seq_decl CLOSE BUNDLE ID
        | directive BUNDLE ID field_seq_decl CLOSE BUNDLE ID
        | BUNDLE ID STRLIT field_seq_decl CLOSE BUNDLE
        | directive BUNDLE ID STRLIT field_seq_decl CLOSE BUNDLE
        | BUNDLE ID STRLIT field_seq_decl CLOSE BUNDLE ID
        | directive BUNDLE ID STRLIT field_seq_decl CLOSE BUNDLE ID

directive ::= DIRECTIVE ID INTLIT:num

field_seq_decl ::= field_decl_list seq_decl_list
                | field_decl_list
                | seq_decl_list

//--------------- field declarations -----------------//

field_decl_list ::= field_decl
                 | field_decl_list field_decl

field_decl ::= DECLARE var_name BOOLEAN
            | DECLARE var_name BOOLEAN LPAREN INTLIT RPAREN
            | DECLARE var_name NUMERIC
            | DECLARE var_name NUMERIC LPAREN INTLIT RPAREN
            | DECLARE var_name CHARACTER
            | DECLARE var_name CHARACTER LPAREN INTLIT RPAREN
            | DEFINE var_name AS expr

//--------------- sequences -------------------//

seq_decl_list ::= seq_decl
               | seq_decl_list seq_decl

//optional: <seq_name> in close statement
//          <"user_info">
//          [ACTIVE/INACTIVE]

seq_decl ::= SEQUENCE ID declare_statement_list CLOSE SEQUENCE
        |  SEQUENCE ID CLOSE SEQUENCE
        | SEQUENCE ID declare_statement_list CLOSE SEQUENCE ID
        | SEQUENCE ID CLOSE SEQUENCE ID
        | SEQUENCE ID active_status declare_statement_list CLOSE SEQUENCE
```

```
        | SEQUENCE ID active_status CLOSE SEQUENCE
        | SEQUENCE ID active_status declare_statement_list CLOSE SEQUENCE ID
        | SEQUENCE ID active_status CLOSE SEQUENCE ID
        | SEQUENCE ID STRLIT declare_statement_list CLOSE SEQUENCE
        | SEQUENCE ID STRLIT CLOSE SEQUENCE
        | SEQUENCE ID STRLIT declare_statement_list CLOSE SEQUENCE ID
        | SEQUENCE ID STRLIT CLOSE SEQUENCE ID
        | SEQUENCE ID active_status STRLIT declare_statement_list CLOSE
SEQUENCE
        | SEQUENCE ID active_status STRLIT CLOSE SEQUENCE
        | SEQUENCE ID active_status STRLIT declare_statement_list CLOSE
SEQUENCE ID
        | SEQUENCE ID active_status STRLIT CLOSE SEQUENCE ID
        | SUBSEQ ID declare_statement_list CLOSE SUBSEQ
        | SUBSEQ ID CLOSE SUBSEQ
        | SUBSEQ ID declare_statement_list CLOSE SUBSEQ ID
        | SUBSEQ ID CLOSE SUBSEQ ID
        | SUBSEQ ID STRLIT declare_statement_list CLOSE SUBSEQ
        | SUBSEQ ID STRLIT CLOSE SUBSEQ
        | SUBSEQ ID STRLIT declare_statement_list CLOSE SUBSEQ ID
        | SUBSEQ ID STRLIT CLOSE SUBSEQ ID

//--------------- declarations ----------------//
//field_decl_list is exactly the same in both cases, so
//reuse the production from before.

declare_statement_list ::= field_decl_list
                         | statement_list
                         | field_decl_list statement_list

//--------------- statements ------------------//

statement_list ::= statement
                 | statement_list statement

statement ::= COMMAND command_action COMMA command_params
            | COMMAND command_action
            | SET var_name TO expr
            | SET var_name EQ expr
            | CALL ID:methName
            | display_word expr
            | display_word expr PAUSE
            | REMOVE ID:bundle_name
            | HALT ID:bundle_name
            | INSTALL var_name
            | INSTALL var_name COMMA install_params
```

```
            | START var_name
            | STOP
            | STOP var_name:bund_dot_seq
            | RESUME var_name
            | WAIT expr
        //the following case applies only to WHEN
            | condition_header CONTINUE
            | condition_header END cond_word
            | condition_header statement_list END cond_word
            | condition_header elseif_actions END cond_word
            | condition_header statement_list elseif_actions END cond_word
            | condition_header statement_list else_actions END cond_word
            | condition_header statement_list elseif_actions
                else_actions END cond_word


//---------------- for COMMAND and INSTALL --------------------//

command_action ::= ID
                | ID var_name
                | HALT
                | REMOVE
                | START
                | INSTALL
                | STOP
                | RESUME

param_data ::= ID PARAMETER_OP expr

command_params ::= param_data
                | command_params COMMA param_data;

install_params ::= param_data
                | param_data COMMA param_data


//----------------- CONDITIONAL stuff -----------------------//

condition_header ::= cond_word expr
                | cond_word expr THEN
                | cond_word expr secondary_cond expr
                | cond_word expr secondary_cond expr THEN

elseif_actions ::= ELSEIF expr
                | ELSEIF expr statement_list
                | elseif_actions ELSEIF expr
```

```
                | elseif_actions ELSEIF expr statement_list

else_actions ::= ELSE statement_list
             | OTHERWISE statement_list

cond_word ::= WHEN
          | WHENEVER
          | IF
          | EVERY

secondary_cond ::= BEFORE
               | WITHIN

display_word ::=  MESSAGE
              | CONFIRM
              | QUERY
              | DISREGARD
              | WARNING

expr  ::= var_name
      | INTLIT
      | DECLIT
      | STRLIT
      | TIMELIT
      | EXPLIT
      | TRUE
      | ON
      | FALSE
      | OFF
      | MINUS expr
      | NOT expr
      | SQRT expr
      | ABS expr
      | SIN expr
      | COS expr
      | TAN expr
      | ARCSIN expr
      | ARCCOS expr
      | ARCTAN expr
      | expr MULT expr
      | expr PLUS expr
      | expr MINUS expr
      | expr DOTDOT expr
      | expr DIVIDE expr
      | expr MULTMULT expr
```

```
        | expr IN expr
        | expr OUTSIDE expr
        | expr MOD expr
        | expr GT expr
        | expr LT expr
        | expr GEQ expr
        | expr LEQ expr
        | expr EQ expr
        | expr NEQ expr
        | expr AND expr
        | expr OR expr
        | expr COMMA expr
        | LPAREN expr RPAREN
        //TLI.TEST_STR_ARRAY(1,*) case
        | expr COMMA MULT

//var_name is basically a variable that's been declared
var_name ::= ID DOT ID
          | ID
          | ID LPAREN expr RPAREN
          | ID DOT ID LPAREN expr RPAREN

active_status ::= ACTIVE
               | INACTIVE
```

A.7 CUP Specifications File

```
package parser;

import java.lang.String;
import java_cup.runtime.*;
import scanner.*;
import ast.*;
import tools.*;

/* preliminaries to set up and use the scanner */

action code {:

        public int getTokenLine() {
                //uses the getLine() function that was implemented in
                //the lexer
                return ((TimelinerTokenizer)parser.getScanner()).getLine();
        }
:};

parser code {:
        TimelinerTokenizer lexer;

        public TimelinerParser(TimelinerTokenizer llexer) {
          this();
          this.setScanner(llexer);
        }

        //overriding the report_error() to return the symbol and line# instead
        //of the char #
        public void report_error(String message, Object info)
        {
                System.err.print(message);
                if (info instanceof Symbol)
                {
                   if (((Symbol)info).left != -1)
                     {
                          System.err.println(" on symbol " +
                          RunScanner.symnames[(int)((Symbol)info).sym] + "(" +
                          ((Symbol)info).sym + ") with value " + ((Symbol)info).value + " in
                          line #"+ ((TimelinerTokenizer)getScanner()).getLine());
                     }
                   else System.err.println("");
                 }
```

```
                else System.err.println("");
        }

:};

/* terminals (tokens returned by the scanner) */
terminal BUNDLE, SEQUENCE, SUBSEQ, CLOSE, WHEN, WHENEVER,
        EVERY, IF, BEFORE, WITHIN, OTHERWISE, ELSE, ELSEIF, END, CALL,
        CONTINUE, THEN, SET, TO, COMMAND, MESSAGE, REMOVE, HALT,
        DEFINE, AS, AND, OR, NOT, ABS, SQRT, SIN, COS, TAN, START,
        ARCSIN, ARCCOS, ARCTAN, ACTIVE, INACTIVE, WARNING, QUERY,
WAIT,
        EQ, NEQ, GT, LT, LEQ, GEQ, DOTDOT, PLUS, PAUSE, DISREGARD,
CONFIRM,
        MINUS, MULT, DIVIDE, MULTMULT, LPAREN, RPAREN, RESUME, STOP,
        OFF, ON, TRUE, FALSE, CHARACTER, DECLARE, BOOLEAN, IN, COMMA,
        OUTSIDE, NUMERIC, MOD, DOT, SEMICOLON, PARAMETER_OP,
        EXPLIT, INSTALL, TIMELIT, DIRECTIVE, STEP, HOLD_AT, JUMP_TO;

terminal String ID;
terminal String STRLIT;
terminal Long INTLIT;
terminal Float DECLIT;

/* Non terminals */
non terminal program;
non terminal Expr var_name;
non terminal FieldAndSeqDecl field_seq_decl;
non terminal FieldDeclList field_decl_list;
non terminal SeqDeclList seq_decl_list;
non terminal FieldDecl field_decl;
non terminal Expr expr;
non terminal SeqDecl seq_decl;
non terminal Directive directive;
non terminal StatementList statement_list;
non terminal Statement statement;
non terminal CondHeader condition_header;
non terminal ElseActions else_actions;
non terminal String cond_word;
non terminal String secondary_cond;
non terminal String active_status;
non terminal String display_word;
non terminal DeclStatementList declare_statement_list;
non terminal ParamData param_data;
non terminal InstallParams install_params;
non terminal CommandParams command_params;
```

```
non terminal CommandAction command_action;
non terminal ElseIfActions elseif_actions;
non terminal ParamDataList param_data_list;

/* precedences */

precedence left OR;
precedence left AND;
precedence nonassoc EQ, NEQ;
precedence nonassoc LT, GT, GEQ, LEQ;
precedence left COMMA;
precedence nonassoc IN;
precedence nonassoc OUTSIDE;
precedence left DOTDOT;
precedence left PLUS, MINUS;
precedence left MULT, DIVIDE;
precedence left MOD;
precedence left MULTMULT;
precedence nonassoc NOT;


/* the actual grammar for the language */
start with program;

//includes cases for optional args, like <"user_info">
//and <bundle_name> in the CLOSE statement
program ::=  BUNDLE ID:id field_seq_decl:fsl CLOSE BUNDLE
                {: Bundle bundle = new Bundle(id, null, null, fsl, id, idright);
                  RESULT = bundle;
                :}
          | directive:dir BUNDLE ID:id field_seq_decl:fsl CLOSE BUNDLE
                {: Bundle bundle = new Bundle( id, null, dir, fsl,  id, idright);
                  RESULT = bundle;
                :}
          | BUNDLE ID:id1 field_seq_decl:fsl CLOSE BUNDLE ID:id2
                {: Bundle bundle = new Bundle( id1, null, null, fsl,  id2, id1right);
                  RESULT = bundle;
                :}
          | directive:dir BUNDLE ID:id1 field_seq_decl:fsl CLOSE BUNDLE ID:id2
                {: Bundle bundle = new Bundle( id1, null, dir, fsl,  id2, id1right);
                  RESULT = bundle;
                :}
          | BUNDLE ID:id STRLIT:user_info field_seq_decl:fsl CLOSE BUNDLE
                {: Bundle bundle = new Bundle( id,  user_info, null, fsl,  id, idright);
                  RESULT = bundle;
                :}
```

```
        | directive:dir BUNDLE ID:id STRLIT:user_info field_seq_decl:fsl CLOSE
BUNDLE
                {: Bundle bundle = new Bundle( id,  user_info, dir, fsl,  id, idright);
                  RESULT = bundle;
                :}
        | BUNDLE ID:id1 STRLIT:user_info field_seq_decl:fsl CLOSE BUNDLE ID:id2
                {: Bundle bundle = new Bundle( id1,  user_info, null, fsl,  id2, id1right);
                  RESULT = bundle;
                :}
        | directive:dir BUNDLE ID:id1 STRLIT:user_info field_seq_decl:fsl CLOSE
BUNDLE ID:id2
                {: Bundle bundle = new Bundle( id1,  user_info, dir, fsl,  id2, id1right);
                  RESULT = bundle;
                :};



directive ::= DIRECTIVE ID:id INTLIT:num
              {: RESULT = new Directive( id, num.intValue(), idright);
                :};



field_seq_decl ::= field_decl_list:fdl seq_decl_list:sdl
                  {: RESULT = new FieldAndSeqDecl(fdl, sdl, fdl.getLine());
                  :}
                | field_decl_list:fdl
                  {: RESULT = new FieldAndSeqDecl(fdl, null, fdl.getLine());
                  :}
                | seq_decl_list:sdl
                  {: RESULT = new FieldAndSeqDecl(null, sdl, sdl.getLine());
                  :};

//--------------- field decls -----------------//

field_decl_list ::= field_decl:fd
                   {: RESULT = new LastFieldDeclList(fd, fd.getLine());
                   :}
                  | field_decl_list:fdl field_decl:fd
                   {: RESULT = new PairFieldDeclList(fdl, fd, fdl.getLine());
                   :};



field_decl ::= DECLARE var_name:name BOOLEAN
                {: RESULT = new BoolFieldDecl( name.toString(), name.getLine());
```

```
            :}
          | DECLARE var_name:name BOOLEAN LPAREN INTLIT:length RPAREN
            {: RESULT = new BoolArrayFieldDecl( name.toString(), length.intValue(),
name.getLine());
            :}
          | DECLARE var_name:name NUMERIC
            {: RESULT = new NumFieldDecl( name.toString(), name.getLine());
            :}
          | DECLARE var_name:name NUMERIC LPAREN INTLIT:length RPAREN
            {: RESULT = new NumArrayFieldDecl( name.toString(), length.intValue(),
name.getLine());
            :}
          | DECLARE var_name:name CHARACTER
            {: RESULT = new CharFieldDecl( name.toString(), name.getLine());
            :}
          | DECLARE var_name:name CHARACTER LPAREN INTLIT:length RPAREN
            {: RESULT = new CharArrayFieldDecl( name.toString(), length.intValue(),
name.getLine());
            :}
          | DEFINE var_name:name AS expr:e
           {:  RESULT = new ExprFieldDecl( name.toString(), e, name.getLine());

            :};

//--------------- sequences -------------------//

seq_decl_list ::= seq_decl:s
                  {: RESULT = new LastSeqDeclList(s, s.getLine());
                  :}
                | seq_decl_list:sl seq_decl:s
                  {: RESULT = new PairSeqDeclList(sl, s, sl.getLine());
                  :};


//optional: <seq_name> in close statement
//          <"user_info">
//          [ACTIVE/INACTIVE]
seq_decl ::= SEQUENCE ID:i declare_statement_list:slist CLOSE SEQUENCE
                 {: RESULT = new SeqDecl(true,  i, "INACTIVE", null, slist,  i, iright);
                 :}
        |   SEQUENCE ID:i CLOSE SEQUENCE
                 {: RESULT = new SeqDecl(true,  i, "INACTIVE", null, null,  i, iright);
                 :}
        |  SEQUENCE ID:i declare_statement_list:slist CLOSE SEQUENCE ID:i2
                 {: RESULT = new SeqDecl(true,  i, "INACTIVE", null,  slist,  i2, iright);
                 :}
```

```
        | SEQUENCE ID:i CLOSE SEQUENCE ID:i2
                {: RESULT = new SeqDecl(true, i, "INACTIVE", null, null, i2, iright);
                :}
        | SEQUENCE ID:i active_status:as declare_statement_list:slist CLOSE
SEQUENCE
                {: RESULT = new SeqDecl(true, i, as, null, slist, i, iright);
                :}
        | SEQUENCE ID:i active_status:as CLOSE SEQUENCE
                {: RESULT = new SeqDecl(true, i, as, null, null, i, iright);
                :}
      | SEQUENCE ID:i active_status:as declare_statement_list:slist CLOSE SEQUENCE
ID:i2
                {: RESULT = new SeqDecl(true, i, as, null, slist, i2, iright);
                :}
        | SEQUENCE ID:i active_status:as CLOSE SEQUENCE ID:i2
                {: RESULT = new SeqDecl(true, i, as, null, null, i2, iright);
                :}
        | SEQUENCE ID:i STRLIT:user_info declare_statement_list:slist CLOSE
SEQUENCE
                {: RESULT = new SeqDecl(true, i, "INACTIVE", user_info, slist, i, iright);
                :}
        | SEQUENCE ID:i STRLIT:user_info CLOSE SEQUENCE
                {: RESULT = new SeqDecl(true, i, "INACTIVE", user_info, null, i, iright);
                :}
        | SEQUENCE ID:i STRLIT:user_info declare_statement_list:slist CLOSE
SEQUENCE ID:i2
                {: RESULT = new SeqDecl(true, i, "INACTIVE", user_info, slist, i2,
iright);
                :}
        | SEQUENCE ID:i STRLIT:user_info CLOSE SEQUENCE ID:i2
                {: RESULT = new SeqDecl(true, i, "INACTIVE", user_info, null, i2,
iright);
                :}
        | SEQUENCE ID:i active_status:as STRLIT:user_info declare_statement_list:slist
CLOSE SEQUENCE
                {: RESULT = new SeqDecl(true, i, as, user_info, slist, i, iright);
                :}
        | SEQUENCE ID:i active_status:as STRLIT:user_info CLOSE SEQUENCE
                {: RESULT = new SeqDecl(true, i, as, user_info, null, i, iright);
                :}
        | SEQUENCE ID:i active_status:as STRLIT:user_info declare_statement_list:slist
CLOSE SEQUENCE ID:i2
                {: RESULT = new SeqDecl(true, i, as, user_info, slist, i2, iright);
                :}
        | SEQUENCE ID:i active_status:as STRLIT:user_info CLOSE SEQUENCE ID:i2
                {: RESULT = new SeqDecl(true, i, as, user_info, null, i2, iright);
```

```
                :}
        | SUBSEQ ID:i declare_statement_list:slist CLOSE SUBSEQ
                {: RESULT = new SeqDecl(false,  i, "INACTIVE", null, slist,  i, iright);
                :}
        | SUBSEQ ID:i CLOSE SUBSEQ
                {: RESULT = new SeqDecl(false,  i, "INACTIVE", null, null,  i, iright);
                :}
        | SUBSEQ ID:i declare_statement_list:slist CLOSE SUBSEQ ID:i2
                {: RESULT = new SeqDecl(false,  i, "INACTIVE", null, slist,  i2, iright);
                :}
        | SUBSEQ ID:i CLOSE SUBSEQ ID:i2
                {: RESULT = new SeqDecl(false,  i, "INACTIVE", null, null,  i2, iright);
                :}
        | SUBSEQ ID:i STRLIT:user_info declare_statement_list:slist CLOSE SUBSEQ
                {: RESULT = new SeqDecl(false,  i, "INACTIVE", user_info, slist,  i,
iright);
                :}
        | SUBSEQ ID:i STRLIT:user_info CLOSE SUBSEQ
                {: RESULT = new SeqDecl(false,  i, "INACTIVE", user_info, null,  i,
iright);
                :}
      | SUBSEQ ID:i STRLIT:user_info declare_statement_list:slist CLOSE SUBSEQ
ID:i2
                {: RESULT = new SeqDecl(false,  i, "INACTIVE", user_info, slist,  i2,
iright);
                :}
        | SUBSEQ ID:i STRLIT:user_info CLOSE SUBSEQ ID:i2
                {: RESULT = new SeqDecl(false,  i, "INACTIVE", user_info, null,  i2,
iright);
                :};


//--------------- declarations ----------------//
//field_decl_list is exactly the same in both cases, so
//reuse the prod from before.

declare_statement_list ::= field_decl_list:fdl
                        {: RESULT = new DeclStatementList(fdl, null, fdl.getLine());
                        :}
                        | statement_list:sl
                        {: RESULT = new DeclStatementList(null, sl, sl.getLine());
                        :}
                        | field_decl_list:fdl statement_list:sl
                        {: RESULT = new DeclStatementList(fdl, sl, fdl.getLine());
                        :};
```

```
//-------------- statements -----------------//

statement_list ::= statement:s
                 {: RESULT = new LastStatementList(s, s.getLine());
                 :}
               | statement_list:sl statement:s
                 {: RESULT = new PairStatementList(sl, s, sl.getLine());
                 :};



statement ::= COMMAND command_action:ca COMMA command_params:cp
            {: RESULT = new CommandStatement(ca, cp, ca.getLine());
               :}
          | COMMAND command_action:ca
               {: RESULT = new CommandStatement(ca, null, ca.getLine());
               :}
          | SET var_name:name TO expr:e
               {: RESULT = new SetStatement( name, e, name.getLine());
                :}
          | SET var_name:name EQ expr:e
               {: RESULT = new SetStatement( name, e, name.getLine());
               :}
          | CALL ID:methName
               {: RESULT = new CallStatement( methName, methNameright);
               :}
          | display_word:dw expr:e
               {: RESULT = new DisplayStatement( dw, e, false, e.getLine());
               :}
          | display_word:dw expr:e PAUSE
               {: RESULT = new DisplayStatement( dw, e, true, e.getLine());
               :}
          | REMOVE ID:bundle_name
               {: RESULT = new RemoveStatement( bundle_name, bundle_nameright);
               :}
          | HALT ID:bundle_name
               {: RESULT = new HaltStatement( bundle_name, bundle_nameright);
               :}
          | INSTALL var_name:name
               {: RESULT = new InstallStatement( name.toString(), null, name.getLine());
               :}
          | INSTALL var_name:name COMMA install_params:p
               {: RESULT = new InstallStatement( name.toString(), p, name.getLine());
               :}
```

```
        | START var_name:name
             {: RESULT = new StartStatement(name.toString(), name.getLine());
             :}
        | STOP
             {: RESULT = new StopStatement(null, getTokenLine());
             :}
        | STOP var_name:bund_dot_seq
             {: RESULT = new StopStatement( bund_dot_seq.toString(),
bund_dot_seq.getLine());
             :}
        | RESUME var_name:name
             {: RESULT = new ResumeStatement(name.toString(), name.getLine());
             :}
        | WAIT expr:e
             {: RESULT = new WaitStatement(e, e.getLine());
             :}
      //the following case applies only to WHEN
        | condition_header:ch CONTINUE
             {: RESULT = new WhenContinueStatement(ch, ch.getLine());
             :}
        | condition_header:ch END cond_word:cw
             {: RESULT = new CondStatement(ch,  cw, null, null, null, ch.getLine());
             :}
        | condition_header:ch statement_list:sl END cond_word:cw
             {: RESULT = new CondStatement(ch,  cw, sl, null, null, ch.getLine());
             :}
        | condition_header:ch elseif_actions:eif END cond_word:cw
             {: RESULT = new CondStatement(ch,  cw, null, eif, null, ch.getLine());
             :}
        | condition_header:ch statement_list:sl elseif_actions:eif END cond_word:cw
             {: RESULT = new CondStatement(ch,  cw, sl, eif, null, ch.getLine());
             :}
        | condition_header:ch statement_list:sl else_actions:ea END cond_word:cw
             {: RESULT = new CondStatement(ch,  cw, sl, null, ea, ch.getLine());
             :}
        | condition_header:ch statement_list:sl elseif_actions:eif
             else_actions:ea END cond_word:cw
             {: RESULT = new CondStatement(ch,  cw, sl, eif, ea, ch.getLine());
             :};


//---------------- for COMMAND and INSTALL --------------------//


command_action ::= ID:id
             {: RESULT = new CommandAction(id, null, idright);
```

```
:}
| ID:id var_name:name
{: RESULT = new CommandAction(id,  name, idright);
:}
| HALT:h
{: RESULT = new CommandAction("HALT", null, hright);
:}
| REMOVE:r
{: RESULT = new CommandAction("REMOVE", null, rright);
:}
| START:s
{: RESULT = new CommandAction("START", null, sright);
:}
| INSTALL:i
{: RESULT = new CommandAction("INSTALL", null, iright);
:}
| STOP:s
{: RESULT = new CommandAction("STOP", null, sright);
:}
| RESUME:r
{: RESULT = new CommandAction("RESUME", null, rright);
:}
| STEP:s
{: RESULT = new CommandAction("STEP", null, sright);
:}
| HOLD_AT:h
{: RESULT = new CommandAction("HOLD_AT", null, hright);
:}
| JUMP_TO:j
{: RESULT = new CommandAction("JUMP_TO", null, jright);
:}
;


param_data_list ::= ID:id PARAMETER_OP expr:e COMMA
            {: ParamData pd = new ParamData(id, e, idright);
              RESULT = new LastParamDataList(pd, pd.getLine());
            :}
        | param_data_list:pdl ID:id PARAMETER_OP expr:e COMMA
            {: ParamData pd = new ParamData(id, e, idright);
              RESULT = new PairParamDataList(pdl, pd, pdl.getLine());
            :};
```

```
param_data ::= ID:id PARAMETER_OP expr:e
                {: RESULT = new ParamData(id, e, idright);
                :};



command_params ::= param_data:pd
                {: RESULT = new LastParamDataList(pd, pd.getLine());
                :}
                 | param_data_list:pdl param_data:pd
                {: RESULT = new PairParamDataList(pdl, pd, pdl.getLine());
                :};



install_params ::= param_data:pd
                {: RESULT = new LastParamDataList(pd, pd.getLine());
                :}
                 | ID:id PARAMETER_OP expr:e COMMA param_data:pd
                {: ParamData p = new ParamData(id, e, idright);
                  LastParamDataList lp = new LastParamDataList(p, p.getLine());
                  RESULT = new PairParamDataList(lp, pd, lp.getLine());
                :};


//----------------- CONDITIONAL stuff -----------------------//

condition_header ::= cond_word:cw expr:e
                   {: RESULT = new CondHeader( cw, e, e.getLine());
                   :}
                  | cond_word:cw expr:e THEN
                   {: RESULT = new CondHeader( cw, e, e.getLine());
                   :}
                  | cond_word:cw expr:e secondary_cond:sc expr:e2
                   {: RESULT = new CondHeader( cw, e,  sc, e2, e.getLine());
                   :}
                  | cond_word:cw expr:e secondary_cond:sc expr:e2 THEN
                   {: RESULT = new CondHeader( cw, e,  sc, e2, e.getLine());
                   :};



 elseif_actions ::= ELSEIF expr:e
                        {: RESULT = new LastElseIfActions(e, null, e.getLine());
                        :}
                 |  ELSEIF expr:e statement_list:sl
```

```
                        {: RESULT = new LastElseIfActions(e, sl, e.getLine());
                        :}
                | elseif_actions:eif ELSEIF expr:e
                        {: LastElseIfActions lastE = new LastElseIfActions(e, null,
e.getLine());
                          RESULT = new PairElseIfActions(eif, lastE, eif.getLine());
                        :}
                | elseif_actions:eif ELSEIF expr:e statement_list:sl
                        {: LastElseIfActions lastE = new LastElseIfActions(e, sl,
e.getLine());
                          RESULT = new PairElseIfActions(eif, lastE, eif.getLine());
                        :};



else_actions ::= ELSE statement_list:sl
                {: RESULT = new ElseActions("ELSE", sl, sl.getLine());
                :}
              | OTHERWISE statement_list:sl
                {: RESULT = new ElseActions("OTHERWISE", sl, sl.getLine());
                :};



cond_word ::= WHEN
            {: RESULT = "WHEN"; :}
          | WHENEVER
            {: RESULT = "WHENEVER"; :}
          | IF
            {: RESULT = "IF"; :}
          | EVERY
            {: RESULT = "EVERY"; :};

secondary_cond ::= BEFORE
                  {: RESULT = "BEFORE"; :}
                | WITHIN
                  {: RESULT = "WITHIN"; :};

display_word ::=  MESSAGE
                 {: RESULT = "MESSAGE"; :}
               | CONFIRM
                 {: RESULT = "CONFIRM"; :}
               | QUERY
                 {: RESULT = "QUERY"; :}
               | DISREGARD
                 {: RESULT = "DISREGARD"; :}
```

```
            | WARNING
              {: RESULT = "WARNING"; :};

expr ::= var_name:name
          {: RESULT = name; :}
        | INTLIT:i
          {: RESULT = new IntExpr(i.intValue(), iright); :}
        | DECLIT:i
          {: RESULT = new DecExpr(i.floatValue(), iright); :}
        | STRLIT:st
          {: RESULT = new StrExpr(st.substring(1, st.length()-2), stright);:}
        | TIMELIT:time
          {: RESULT = new TimeExpr(time.toString(), timeright); :}
        | EXPLIT:exp
          {: RESULT = new ExpExpr(exp.toString(), expright); :}
        | TRUE:t
          {: RESULT = new BoolExpr(true, tright); :}
        | ON:o
          {: RESULT = new BoolExpr(true, oright); :}
        | FALSE:f
          {: RESULT = new BoolExpr(false, fright); :}
        | OFF:o
          {: RESULT = new BoolExpr(false, oright); :}
        | MINUS expr:e
         {: RESULT = new FuncExpr(FuncExpr.MINUS, e, e.getLine()); :}
        | NOT expr:e
         {: RESULT = new FuncExpr(FuncExpr.NOT, e, e.getLine()); :}
        | SQRT expr:e
         {: RESULT = new FuncExpr(FuncExpr.SQRT, e, e.getLine()); :}
        | ABS expr:e
         {: RESULT = new FuncExpr(FuncExpr.ABS, e, e.getLine()); :}
        | SIN expr:e
         {: RESULT = new FuncExpr(FuncExpr.SIN, e, e.getLine()); :}
        | COS expr:e
         {: RESULT = new FuncExpr(FuncExpr.COS, e, e.getLine()); :}
        | TAN expr:e
         {: RESULT = new FuncExpr(FuncExpr.TAN, e, e.getLine()); :}
        | ARCSIN expr:e
         {: RESULT = new FuncExpr(FuncExpr.ARCSIN, e, e.getLine()); :}
        | ARCCOS expr:e
         {: RESULT = new FuncExpr(FuncExpr.ARCCOS, e, e.getLine()); :}
        | ARCTAN expr:e
         {: RESULT = new FuncExpr(FuncExpr.ARCTAN, e, e.getLine()); :}
        | expr:e1 MULT expr:e2
         {: RESULT = new OpExpr(e1, OpExpr.MULT, e2, e1.getLine()); :}
        | expr:e1 PLUS expr:e2
```

```
  {: RESULT = new OpExpr(e1, OpExpr.PLUS, e2, e1.getLine()); :}
| expr:e1 MINUS expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.MINUS, e2, e1.getLine()); :}
| expr:e1 DOTDOT expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.DOTDOT, e2, e1.getLine()); :}
| expr:e1 DIVIDE expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.DIVIDE, e2, e1.getLine()); :}
| expr:e1 MULTMULT expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.MULTMULT, e2, e1.getLine()); :}
| expr:e1 IN expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.IN, e2, e1.getLine()); :}
| expr:e1 OUTSIDE expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.OUTSIDE, e2, e1.getLine()); :}
| expr:e1 MOD expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.MOD, e2, e1.getLine()); :}
| expr:e1 GT expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.GT, e2, e1.getLine()); :}
| expr:e1 LT expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.LT, e2, e1.getLine()); :}
| expr:e1 GEQ expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.GEQ, e2, e1.getLine()); :}
| expr:e1 LEQ expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.LEQ, e2, e1.getLine()); :}
| expr:e1 EQ expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.EQ, e2, e1.getLine()); :}
| expr:e1 NEQ expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.NEQ, e2, e1.getLine()); :}
| expr:e1 AND expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.AND, e2, e1.getLine()); :}
| expr:e1 OR expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.OR, e2, e1.getLine()); :}
| expr:e1 COMMA expr:e2
  {: RESULT = new OpExpr(e1, OpExpr.COMMA, e2, e1.getLine()); :}
| LPAREN expr:e RPAREN
  {: RESULT = new ParensExpr(e, e.getLine()); :}
//TLI.TEST_STR_ARRAY(1,*) case
| expr:e COMMA MULT
  {: RESULT = new ArrayStarExpr(e, e.getLine()); :};

//var_name is basically a variable that's been declared
var_name ::= ID:id1 DOT ID:id2
            {: RESULT = new IdExpr(id1.concat(".").concat(id2), id1right);
            :}
     | ID:id
            {: RESULT = new IdExpr(id, idright);
            :}
```

```
| ID:id LPAREN expr:e RPAREN
      {: RESULT = new ArrayExpr(id, e, idright); :}
| ID:id1 DOT ID:id2 LPAREN expr:e RPAREN
  {: RESULT = new ArrayExpr(id1.concat(".").concat(id2), e, id1right); :};


active_status ::= ACTIVE
                  {: RESULT = "ACTIVE";
                  :}
                 | INACTIVE
                  {: RESULT = "INACTIVE";
                  :};
```

A.8 XML File Output for Figure 2-2


```
<?xml version="1.0"?>
<Bundle>
   <bundleName>EXAMPLE</bundleName>
   <endName>EXAMPLE</endName>
   <line>1</line>
   <user_info>null</user_info>
   <fsl><FieldAndSeqDecl><line>2</line>
      <fl><LastFieldDeclList><line>2</line>
         <head><NumFieldDecl><line>2</line>
            <id>X</id>
         </NumFieldDecl></head>
      </LastFieldDeclList></fl>
      <sl><LastSeqDeclList><line>3</line>
         <head><SeqDecl><line>3</line>
            <isSeq>true</isSeq>
            <id>ONE</id>
            <endID>ONE</endID>
            <status>INACTIVE</status>
            <userInfo>null</userInfo>
            <dsl><DeclStatementList><line>4</line>
               <sl><LastStatementList><line>4</line>
                  <head><CondStatement><line>4</line>
                     <endWord>WHEN</endWord>
                     <ch><CondHeader><line>4</line>
                        <condWord>WHEN</condWord>
                        <secondCondWord>null</secondCondWord>
                        <condition><OpExpr><line>4</line>
                           <op>10</op>
                           <left><IdExpr><line>4</line>
                              <id>X</id>
                           </IdExpr></left>
                           <right><IntExpr><line>4</line>
                              <value>1</value>
                           </IntExpr></right>
                        </OpExpr></condition>
                     </CondHeader></ch>
                     <slist><LastStatementList><line>5</line>
                        <head><SetStatement><line>5</line>
                           <id><IdExpr><line>5</line>
                              <id>X</id>
                           </IdExpr></id>
                           <value><IntExpr><line>5</line>
                              <value>0</value>
                           </IntExpr></value>
                        </SetStatement></head>
                     </LastStatementList></slist>
                  </CondStatement></head>
               </LastStatementList></sl>
            </DeclStatementList></dsl>
         </SeqDecl></head>
      </LastSeqDeclList></sl>
   </FieldAndSeqDecl></fsl>
</Bundle>
```

Bibliography

[1] Aho, Alfred V. , Ravi Sethi, Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools." Addison-Wesley Publishing Company, 1986.

[2] Appel, Andrew W. "Modern Compiler Implementation in Java." Cambridge University Press, 1999.

[3] Berk, Elliot. "JLex: A Lexical Analyzer Generator for Java™." Department of Computer Science, Princeton University. Version 1.2, May 5, 1997.

[4] Eyles, Don. "A Time-Oriented Language for the writing of Procedures to Sequence the Operation of a Spacecraft and its Systems." Draper Laboratory, Inc.

[5] Hudson, Scott. "CUP LALR Parser Generator for Java™." Graphics Visualization and Usability Center, Georgia Institute of Technology. July 1997.

[6] Pardi, William J. "XML in Action: Web Technology." Microsoft Press, 1999.

[7] "The Timeliner User Interface Language (UIL) System for the International Space Station." NASA, Draper Laboratory.

[8] "User Interface Language Specifications, SSP 30539, Revision F." NASA SSP 30539 Revision F. December, 2000.