SEMANTIC MECHANISMS FOR A PORTABLE SIMULA SYSTEM

by

MIN-FENG CHANG

B.A., UNIVERSITY OF CHINESE CULTURE, TAIWAN, 1979

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

Approved by:

Rodney M. Bates

Major Professor
Rodney M. Bates
I wish to express my sincere appreciation to my major professor, Dr. Rodney M. Bates, for his patience, guidance, and direction in completion of this report. Thanks are also given to my other committee members, Dr. David Gustafson and Dr. David Schmidt. I also wish to express my appreciation to my auntie, Mei-Lan Hwang, my uncle, Dr. C. L. Hwang, and their family. Finally, specially gratitude is given to my family, for their encouragement and the sacrifices they have made for me.
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>General Overview of an S-PORT System</td>
<td>20</td>
</tr>
<tr>
<td>2.</td>
<td>The Structure of The Perkin-Elmer SIMULA System</td>
<td>23</td>
</tr>
<tr>
<td>3.</td>
<td>The Structure of The S-CODE Assembler</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>And S-CODE Dumper</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>The Structure of The S Stack</td>
<td>28</td>
</tr>
<tr>
<td>5.</td>
<td>The Structure of The String Stack</td>
<td>30</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENT

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>i</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ii</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Purpose</td>
<td>2</td>
</tr>
<tr>
<td>1.3 S-PORT Project Goals</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Organization</td>
<td>4</td>
</tr>
<tr>
<td>2. The SIMULA Language</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Class Declarations and Object Generation</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Remote Accessing</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Concatenation and Hierarchies</td>
<td>13</td>
</tr>
<tr>
<td>3. The S-PORT System</td>
<td>17</td>
</tr>
<tr>
<td>3.1 The Background of S-PORT</td>
<td>17</td>
</tr>
<tr>
<td>3.2 General Overview</td>
<td>17</td>
</tr>
<tr>
<td>4. The Perkin-Elmer SIMULA System</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Components</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Interpass</td>
<td>22</td>
</tr>
<tr>
<td>4.3 S-CODE Assembler and S-CODE Dumper</td>
<td>24</td>
</tr>
<tr>
<td>4.4 PASCAL Code Generation Passes Used For The Perkin-Elmer SIMULA System</td>
<td>25</td>
</tr>
<tr>
<td>5. Interpass Semantic Data Structures</td>
<td>27</td>
</tr>
<tr>
<td>5.1 Naming Conventions</td>
<td>27</td>
</tr>
<tr>
<td>5.2 Stacks</td>
<td>27</td>
</tr>
<tr>
<td>5.2.1 The S Stack</td>
<td>28</td>
</tr>
<tr>
<td>5.2.2 The Update Stack</td>
<td>29</td>
</tr>
</tbody>
</table>
5.2.3 The String Stack .................................. 29
5.2.4 The Input File Stack .............................. 30
5.2.5 Some Miscellaneous Stacks ....................... 31
5.3 Tables .............................................. 31
  5.3.1 The Tag Table .................................. 31
  5.3.2 The Index Table ................................ 32
  5.3.3 The Taglist Table ............................... 33
  5.3.4 The Type Convert Table ........................ 33
5.4 Descriptors ......................................... 33
  5.4.1 Data Descriptor Fields .......................... 34
  5.4.2 Attribute Descriptor Fields .................... 35
  5.4.3 Type Descriptor Fields ........................ 37
  5.4.4 Profile Descriptor Fields ....................... 38
  5.4.5 Label, Switch, and Body Descriptor Fields ... 39
5.5 Some Miscellaneous Data Structures ............... 40
  5.5.1 Value Area and Area Displacement Array ....... 40
  5.5.2 Output Fragments ............................... 40
  5.5.3 Inside Routine and Compiler Switch Array ...... 42
  5.5.4 Output Copy File and Output Copy Flag ......... 42
5.6 Examples of Interpass Processing Routines ....... 42
  5.6.1 Example 1 - VerifySEnty ....................... 43
  5.6.2 Example 2 -
      SetTagSosMDatadLenToCount ..................... 44
  5.6.3 Example 3 -
      PushUpperCaseWithTagAndUndefine .............. 45
6. Conclusion ........................................... 47

References .............................................. 49

Appendices
  Appendix A: The Interpass Semantic Operations ....... 51
  Appendix B: The Code of Interpass Processing Routines. 60
  Appendix C
    C.1: Description of S-CODE syntax rules ........... 88
    C.2: The syntax of the S-CODE language ........... 89
Chapter 1

Introduction

1.1 Background

SIMULA is a programming language developed by Dahl and Nygaard in Norway. This language is ALGOL-like in form and has simulation as a major application domain.

Over the years a number of people have desired to use SIMULA (SIMULA 67) on computer equipment where an implementation has not existed. The amount of work involved in establishing an independent implementation would be somewhere in the region of 4 to 12 man years. In the summer 1979, the Norwegian Computing Center and Program Library Unit at the University of Edinburgh initiated the "S-PORT" project: the implementation of a portable SIMULA system. According to their estimation, the efforts to adapt the portable system to a new machine is about one manyear.

The portable SIMULA system (S-PORT) consists of a portable front-end compiler and a portable run time support system. The front-end compiler translates SIMULA source programs into an intermediate language called S-CODE. Both the front-end compiler and run time support system are distributed in S-CODE. The programs in S-CODE are to be translated by a back-end translator, called the S-Compiler, into the machine language of the object machine. S-CODE is the intermediate language used to transmit an
analyzed SIMULA program from the front-end compiler to the code generator. It has been designed as a practical tool to facilitate the production of SIMULA compilers on as wide range of computers as possible.

The Perkin-Elmer SIMULA system project is to adapt the portable SIMULA system to Perkin-Elmer 32-bit machines. It is being developed to run under the UNIX operating system. The project is undertaken in the Computer Science Department of Kansas State University, Manhattan, Kansas, under the direction of Dr. Rodney M. Bates. It was the subject of a course titled Implementation Projects, taught in spring semester 1983 and spring semester 1984. Dr. Bates taught the course and provided design documentation for the components to be developed.

1.2 Purpose

The purpose of this project is to design and implement a portable SIMULA system (S-PORT) for Perkin-Elmer 32-bit machines, such as 3220 and 8/32, under the UNIX version 7 operating system in the Kansas State University Computer Science Department. This report is a part of the project whose purpose is to design a new pass, called the interpass, to translate S-CODE into the input token stream for an existing PASCAL code generator. The interpass program is written in two different languages, Syntax/Semantic Language (S/SL) and PASCAL. My major work on this project involved the design of the semantic processing routines of the
interpass PASCAL portion. About half of the interpass PASCAL portion routines had been done during the spring semester 1983, and I did the rest of them.

1.3 S-PORT Project Goals

One major interest in the project is an essential reduction of the effort required for production of SIMULA systems for new machines. Through the development of a portable SIMULA system, a large part of the maintenance will become independent, both of machine architecture and changes in operating systems; furthermore, the maintenance of the target dependent part will be greatly simplified due to this separation.

Users of these systems will benefit in several ways. Larger resources may be concentrated on the machine independent portable parts, e.g., on language oriented optimization and tuning of the run time system. Also the portability of SIMULA programs, already far above most comparable languages, will be improved due to the fact that there will be a common syntactical and semantic checker/analizer. Several program development tools will become available, such as symbolic dump, interactive debugging and control and data flow analysis, in a standardized manner, not varying from system to system.
1.4 Organization

This paper is organized into six chapters. Chapter 1 contains explanations of the motivation for designing a S-PORT system, the purpose of the Perkin-Elmer SIMULA system project, and the goals of the S-PORT project. Chapter 2 provides a summary of the SIMULA language concepts, the class declarations, object generation, and the remote access types. Chapter 3 provides the reader with a general overview of the S-PORT system. Chapter 4 explains the Perkin-Elmer SIMULA system, its components, interpass program, and code generation passes used for the Perkin-Elmer SIMULA system. Chapter 5 provides the semantic data structures of interpass. Chapter 6 is then the conclusion.

Finally, appendix A contains the semantic operations of those processing routines that I wrote for the interpass program PASCAL portion, appendix B contains the code of those processing routines, and appendix C contains the description of the S-CODE syntax rules and the syntax of the S-CODE language.
Chapter 2

The SIMULA Language

2.1 Introduction

SIMULA (SIMulation Language) is a language designed to facilitate formal description of the layout and rules of the operation of systems with discrete events. It is an ALGOL 60 derivative which tried to incorporate useful primitives for simulation. SIMULA was designed by Dahl, Myhrhaug, and Nygaard in Norway as a general purpose language that is general enough to serve as the base of the definition of a simulation language.

The SIMULA language retains the spirit of ALGOL 60 and includes that language as a subset. It begins with ALGOL and extends the block concept to allow the generation and naming of blocks which can coexist as coroutines. Such blocks are known as objects and are generated from templates known as class declarations. We can, therefore, consider SIMULA to be ALGOL plus constructs necessary to

1. Specify class declarations
2. Generate coroutine objects from class declarations
3. Name generated objects and manipulate variables associated with named generated objects
4. Concatenate class declarations to realize a hierarchical structure of class declarations
2.2 Class Declarations and Object Generation

As stated concisely by Dahl et al. [1972],

"... a procedure which is capable of giving rise to block instances which survive its call is known as a class; and the instances will be known as objects of that class."

A class declaration can have the following form:

```plaintext
<class declaration> ::= class <class identifier>
                     <formal parameter part>; <specification part>;
                     <class body>

<class body> ::= <statement>
```

As with procedures, the class body is delineated with `begin`-`end` pairs. The formal parameters, variables, and procedures declared within the body constitute the attributes of that class.

Thus, considering the class body to be a block, we can give an example of a class declaration as follows:

```plaintext
class C1 (FP1); SP1;
begin
  D1;
  E1;
end C1;
```
where

\[ C1 = \text{class identifier} \]
\[ FP1 = \text{formal parameter list of the class } C1 \]
\[ SP1 = \text{the list of specifications of the parameters } FP1 \]
\[ D1 = \text{attribute declarations (includes variables and procedures)} \]
\[ E1 = \text{executable statements of class declaration, given via arbitrary block structuring.} \]

The expression

\[ \text{new } C1(...) \]

Where \( C1 \) is a class, is an object generator. When encountered it creates an object which is an instance of class \( C1 \) and starts to execute the executable statements of \( C1 \). The execution continues until the end of the class body is encountered, at which time the execution is terminated. Execution will be suspended, however, if a call to the procedure DETACH is encountered. Execution of detach returns processor control to the creating object, and in particular to the continued execution of the statement which generated the object or its successor. This suspension capability makes it possible for many non-terminated objects to exist simultaneously.

In order to subsequently refer to generated objects, the value returned by the functional designator \text{new} must be retained. For this purpose, a special variable type is necessary because the value is not of type INTEGER, REAL, BOOLEAN, and so on, but
in reality a reference pointer to the object data structure. The special variables will have type ref and their declaration specifies (as a qualification) the class of objects to which the variable may refer. Formally

\[
<\text{reference variable declaration}> ::= \\
\text{ref}(<\text{qualification}>)<\text{identifier list}>
\]

\[
<\text{qualification}> ::= <\text{class identifier}>
\]

The qualification of a reference variable is used to test for the validity of reference assignments to that variable. It provides reference security.

In addition, we have four special operations:

\[
:- \quad \text{To denote reference assignments}
\]

\[
== \quad \text{To denote reference equality (same object)}
\]

\[
=/= \quad \text{To denote reference inequality}
\]

\[
is, in \quad \text{To test the class membership of a referenced object}
\]

and the reference value "\text{none}" which points to no object, and is additionally the initial value of all reference variables.

\textbf{EXAMPLE:} To retain reference to generated class objects, the value returned by the functional designator \texttt{new} must be assigned to a reference variable using the reference assignment operator. Thus if given

\[
\text{ref(ships)}\text{tug, boat, skow};
\]
and the declaration

```plaintext
class ships;
  ;
end ships;
```

then

```plaintext
tug :- new ships;
boat :- new ships;
```

generates two ships objects and retains the references in the variables tug and boat. Further they refer to different objects so that the relational expression

```plaintext
tug == boat
```

would be false. An object may be referenced by several variables, so that execution of

```plaintext
skow :- tug
```

would cause tug and skow to reference the same object. Further, reference variables can be subscripted and collected in arrays. Thus the declaration

```plaintext
ref(ships) array fleet (1:50);
```

would provide for reference to fifty ships objects. A fleet of ships could then be generated (with retained references) via

```plaintext
for i := 1 step 1 until 50 do
  fleet [i] :- new ships;
```
Procedures may also be declared as type \texttt{ref}. For example, a functional procedure named, \texttt{pick}, which calculates and returns as its value a reference to a class \texttt{ship} object, would have the header:

\begin{verbatim}
    ref(ship) procedure pick;
\end{verbatim}

2.3 Remote Accessing

An object whose statements are being obeyed by the processor is said to be active. For the moment, the state of all other existing objects must be considered suspended and probably detached. Much convenience and flexibility can be achieved if the active object can base its behavior on the attributes of certain suspended objects. This capability requires special mechanisms, because an object’s attributes are generally accessible only by that object when active.

One mechanism is provided to facilitate such references. It is known as remote access. The term remote access is used for the accessing of attributes of other objects. It is performed in two steps: the first step is the selection of a particular object, and the second step is the determination of the attribute of the selected object.

Two types of remote access are used to satisfy the security requirement the qualification of the object. They are a remote identifier and the connection.
A remote identifier is composed of a reference variable, a dot, and an attribute identifier. For example, suppose an object named \( X \) is of class \( C \) where \( C \) contains a local variable \( Y \). A second object (of the same or different class) can manipulate the variable \( Y \) of the object referenced by \( X \) with access of the form \( X.Y \).

The construct has the interpretation "the \( Y \) belonging to the object referenced by \( X \).".

An expression

\[ X.U.V.W.Z \]

may be a legal access. Since expressions are evaluated from left to right, such expressions are meaningful only if \( X, U, V, W \) are reference variables, whereas the type of \( Z \) is arbitrary but determines the type associated with the access.

A local qualification may be placed on a reference variable for accesses which would not otherwise be permitted. To eliminate the access restrictions and provide complete access freedom, the operator \( \text{qua} \) is provided. The \( \text{qua} \) operator has the form

\[
<\text{qualified reference}> ::= \\
<\text{reference expression}> \text{qua} <\text{class identifier}>
\]

If \( X \) is a reference expression with qualification \( C \), then "\( X \text{ qua D} \)" is legal only if \( D \) is \( C \) or a subclass of \( C \). By using \( \text{qua} \), any attribute of an object may be referenced.
The other mechanism for accomplishing remote access is connection. The connection is a convenient type of access when several remote accesses to the attributes of a given object must be made. It is provided by "inspect" which, when executed by one object, allows it to reference all attributes of the inspected object without using the dot notation. Formally, the connection statement has the format as follows:

\[
<\text{connection block}> ::= <\text{statement}>
\]
\[
<\text{connection clause}> ::= \text{when } <\text{class identifier}> \quad \text{do } <\text{connection block}>
\]
\[
<\text{otherwise clause}> ::= <\text{empty} > \quad \text{otherwise } <\text{statement}>
\]
\[
<\text{connection part}> ::= <\text{connection clause}>
\]
\[
<\text{connection statement}> ::= \text{inspect} <\text{reference expression}> \quad \text{do}
\]
\[
<\text{connection block} > <\text{otherwise clause}>
\]
\[
<\text{connection part} > <\text{connection clause}>
\]

Let Ci denote class identifiers, X a reference expression, and Si a statement. The typical usages include

(a) \text{inspect } X \text{ do } S1;

(b) \text{inspect } X \text{ do } S1 \text{ otherwise } S2;

(c) \text{inspect } X \text{ when } C1 \text{ do } S1 \text{ otherwise } S2;

(d) \text{inspect } X \text{ when } C1 \text{ do } S1

\hspace{1cm} \text{when } C2 \text{ do } S2
\hspace{1cm} \cdot
\hspace{1cm} \cdot
\hspace{1cm} \text{when } Cn \text{ do } Sn
\hspace{1cm} \text{otherwise } T;
where the otherwise clause is executed if the relational expression "X == none" is true or if X does not reference a class C1, C2, ... or Cn object. A void otherwise clause can be assumed if none is specified. When X is an object of class C1, the statement Si may freely contain accesses to attribute of objects of class C1 as referenced by X and obviates the remote access notation. Variables used in Si which are not declared in Ci are taken from the block (or surrounding block) in which the reference is made, using the the normal scoping rules.

2.4 Concatenation and Hierarchies

SIMULA allows the user to develop program structures from simpler ones in a hierarchical fashion. It allows one to do so in a most convenient manner with the operation of concatenation, a binary compile time operation defined between two class declarations C1 and C2 or a class C1 and a block B concatenation produces as a result the declaration of a new class or block. The resulting class or block declaration is formed first by merging the attributes of both components, and then combining their executable statements.

The operation of concatenation is specified by prefixing a block or class declaration with the name of a previously defined class name C. Suppose that C1 has declaration as given previously (in page 6). Then a declaration of the form
C1 class C2 (FP2); SP2;

begin
  D2;
  E2;
end C2;

results in a class C2 which has the equivalent declaration

class C2 (FP1, FP2);
SP1; SP2;

begin
  D1; D2;
  E1;
  E2;
end C2;

An object of class C2 is considered a compound object. Furthermore, class C2 is considered a subclass of class C1; that is, the prefixed class is considered a subclass to the prefix class. A hierarchy of classes may be generated with declarations of the form

class C1; ...;
C1 class C2; ...;
...;
Cn-1 class Cn; ...;

Then the entire list C1, ... Cn-1 must be considered a prefix to Cn and its structure follows immediately by considering the sin-
gle prefix case; that is, the nesting of $D_i$, $E_i$ is continued. It is important to note that a single class may be used to prefix several different concatenated classes. For example, if we have

```plaintext
class C1; ...;
C1 class C2; ...;
C1 class C3; ...;
```

it will be desirable to allow the body of the prefix class to be split by use of the statement "inner", which has the following effect. Let the definition of class $C_2$ be as given earlier, but change the definition of $C_1$ to

```plaintext
class C1 (FP1); SP1;
begin
D1;
E11;
inner;
E12;
end;
```

For objects of class $C_1$, inner acts as a dummy statement and the order of execution is "$E_{11}; E_{12}$". For objects of class $C_2$, however, the order of execution is "$E_{11}; E_2; E_{12}$", which implies that class $C_2$ has the equivalent definition
class C2 (FP1, FP2);
    SP1; SP2;

begin
    D1; D2;
    E11;
    E2;
    E12;
end;

The split body feature is readily used to provide common initializing actions E11 and terminating actions E12 to a variety of actions E2.
Chapter 3

The S-PORT System

3.1 The Background of S-PORT

S-PORT is the implementation of a portable SIMULA system, developed by Norwegian Computing Center (NCC) in 1979. The development of a portable SIMULA system (S-PORT) was established as a subproject under the general project System Construction and Application Languages (SCALA). The general aims of the SCALA project have been to study methods in program development, e.g. for portable compiler systems. Based on the SIMULA system, the effort required for maintaining existing systems and implementing new systems will be greatly reduced.

3.2 General Overview

In order to implement an easily portable SIMULA system, the target dependent parts of the implementation were carefully defined and separated from the language dependent, machine independent parts. The machine independent part of the language system consists of a front-end compiler and a set of modules comprising the run time support system. The front-end compiler translates SIMULA source programs into an equivalent program
represented in an intermediate language called "S-CODE". The front-end compiler is written in SIMULA. The run time modules, which are written in a language called SIMULETTA, are also translated into S-CODE.

S-CODE is not seen as a set of instructions which can be executed or interpreted to perform the task specified by the source program. Instead S-CODE controls a compilation process which is an executable form of the program. It is intended for further translation to some form accepted by the relevant machine, not for direct interpretation. This is in contrast to e.g. P-CODE for PASCAL, which is compromised both for interpretation and further compilation. The detail description of the S-CODE syntax rules and the syntax of the S-CODE language are shown in appendix C.

To construct a complete SIMULA system from S-PORT, we have to add a back-end translator called the S-Compiler and an Environment Interface Support Package. The S-Compiler translates programs in S-CODE into the machine language of the object machine. The Environment Interface Support Package provides system-dependent services through a system-independent interface (the environment interface).

To bootstrap the portable SIMULA system onto the object machine, first we have to use the S-Compiler to translate the front-end compiler and run time support system to the machine language code of the object machine. Then, we must link these two machine language programs and the compiled version of the
environment interface support package together. The result is a front-end compiler which can be run on the object machine.

The front-end compiler can then be used to translate any SIMULA program to S-CODE. The result is then translated by the S-Compiler and linked with the object machine code versions of the run time support system and environment interface support package to give an object machine code version of the original SIMULA program. The general overview of the S-PORT system is shown in Figure 1.
Figure 1. General Overview of an S-PORT System
Chapter 4

The Perkin-Elmer SIMULA System

4.1 Components

The Perkin-Elmer SIMULA System is a project to adapt the portable SIMULA system to Perkin-Elmer 32-bit machines such as 3220 and Interdata 8/32. It is being developed to run under the UNIX version 7 operating system. The Perkin-Elmer SIMULA System consists of a front-end compiler, a run time support system, an S-Compiler, and an environment interface support package.

The front-end compiler and run time support system which we get from Norwegian Computing Center are originally written for S-PORT. The front-end compiler translates SIMULA source programs into S-CODE. Both the front-end compiler and run time support system are distributed in S-CODE.

In order to construct a complete Perkin-Elmer SIMULA System, we have to design an S-Compiler and an environment interface support package. Those are written in the PASCAL, S/SL, and C programming languages. The environment interface support package is written in C in order to make the full range of operating system services directly accessible. The S-Compiler draws extensively on an existing code generator for the Perkin-Elmer machines. This is actually the last four passes of the PASCAL PAS/32 compiler. These are written in PASCAL. A moderate amount of modification to
these code generation passes and their intermediate token stream languages is required to fit the SIMULA case.

In addition, a new pass, called the interpass, is necessary to translate S-CODE into the necessary input token stream for the PASCAL code generator. The interpass program is written in two different languages, Syntax/Semantic Language (S/SL), and PASCAL. The following section will describe the structure of the interpass.

Figure 2 briefly shows the structure of the Perkin-Elmer SIMULA System.

4.2 Interpass

The interpass translates S-CODE programs into the necessary input token stream for the existing PASCAL code generation passes.

The main control program for interpass is written in Syntax/Semantic Language (S/SL). S/SL is a language which was developed at the University of Toronto as a tool for implementing compilers. It closely resembles a textual notation for expressing syntax diagrams with output actions and calls on semantic routines inserted. An S/SL generated program is, in fact, a recursive descent parsing program with output actions and semantic routine calls embedded.
Figure 2. The Structure Of The Perkin-Elmer SIMULA System
S/SL is a very simple language and allows high productivity in programming. It has no variables or assignments. The implementation of S/SL is also highly efficient both in speed and in size. First, the S/SL program is translated into a table form via an S/SL assembler. This table can be stored in a PASCAL array. Hence, it is accessed by a PASCAL interpreter program called a table walker. The interpreter walks through the table, executing instructions and thus carrying out the actions of the S/SL program. Modest modifications to the implementation of S/SL are necessary to support the Perkin-Elmer SIMULA System. These do not involve the language itself, but only the environment in which S/SL is to be used. For a complete description of S/SL, see [3].

The remainder of the interpass is written in PASCAL. This is actually a list of the necessary semantic processing routines. Those semantic processing routines manipulate stacks, tables, descriptors, and some miscellaneous data structures. Details of interpass data structures will be described in Chapter 5.

4.3 S-CODE Assembler and S-CODE Dumper

Several tool programs are also being developed. These include an S-CODE assembler and an S-CODE dumper. The S-CODE assembler accepts a symbolic form of an S-CODE token stream and converts it to the binary byte code form accepted as input by the interpass. The S-CODE dumper draws on an existing token stream.
dumpers for the various intermediate token streams used by the PASCAL code generation passes. It translates the S-CODE token stream into a printable output.

Symbolic form of S-CODE token stream
( Text Input )

S-CODE
: V
-------------------
| S-CODE Assembler |
-------------------

Binary byte code form
( Internal Format )

: V
-------------------
| Interpass ! |
-------------------

S-CODE
: V
-------------------
| S-CODE Dumper |
-------------------

Printable Output

Figure 3. The Structure of The S-CODE Assembler and S-CODE Dumper

The S-CODE assembler and S-CODE dumper will be used to create test S-CODE token streams initially and to examine and modify S-CODE streams produced by the front-end compiler. The existing PASCAL code generator passes’ token stream dumpers will need to be modified to support the Perkin-Elmer SIMULA System.

4.4 PASCAL Code Generation Passes Used For The Perkin-Elmer SIMULA System

The code generation passes used for the Perkin-Elmer SIMULA System are the sixth through ninth of an existing nine-pass
PASCAL PAS/32 compiler for Perkin-Elmer machines. It is a modified and extended version of PASCAL compiler originally written by Hartmann [4].

Pass 6 receives a postfix token stream with all static errors and declarations removed, run time address assigned, etc. The basic operation of pass 6 is to build a linked data structure for each procedure, perform certain machine-independent optimizations on this representation, and emit optimized output in a language nearly the same as its input language for Pass 7.

Pass 7 actually generates the Perkin-Elmer machine instructions. It performs register allocation for the general, single precision floating point and double precision floating point registers. It selects machine instructions but does not select instruction formats. It generates symbolic reference to code labels, procedure labels, statement labels, literal constants, and stack depths. These are all resolved by passes 8 and 9.

Pass 8 uses the peephole technique to perform a variety of machine-dependent optimizations. It is the first pass to actually know the size of all machine instructions, so it generates the table of correspondences between code labels and actual machine addresses. Pass 9 uses this table to replace the labels with addresses and converts the code to the format accepted by the linking loader.
Chapter 5

Interpass Semantic Data Structures

5.1 Naming Conventions

The interpass semantic data structures was originally supplied by Dr. Bates as the documentation for the design of the interpass program. Before we look at the interpass data structures, several naming conventions have to be pointed out. Those naming conventions are used through the entire interpass program.

"Typ" in an interpass identifier means a PASCAL type for some interpass compile-time type. It normally appears only at the end of the identifier, and only types will thus end. "Type" used anywhere within an identifier refers to an S-CODE type. Record field names always start with the record type name, minus "Typ". Value names (PASCAL constants) always end with the type name, minus "Typ", of the type they are value of.

5.2 Stacks

Stack is one of the most important data structures used in the interpass. All stack top pointers point to the top element actually present in the stack. There are several different stacks used in interpass. Those are described below.
5.2.1 The S Stack

This stack is the one described in the S-CODE definition [3]. It is implemented as an array of pointers point to descriptors. There is also a stack of subscripts to the S stack, called the mark stack, which divides the S stack into sections. Each entry of the mark stack points to the element below the bottom element of the section it delimits. The bottom element of the mark stack always points one below the bottom of the entire S stack. Figure 4 illustrates the structure of the S stack.

![Figure 4. The Structure of The S Stack](image)

All semantic operations which do not refer explicitly to sections view only the top S stack section as the entire S stack.
5.2.2 The Update Stack

This is a stack of global meanings of tags which have been redefined with local meaning. Whenever a tag is redefined, its old tag table entry and its tag are pushed onto the update stack. Then the tag table entry is changed to the new meaning. This occurs at body entry, at which time, all defined label tags are pushed down and given a new meaning of undefined. At body exit, all entries of the update stack are used to restore the corresponding tag table entries to their former meaning.

The update stack consists of two entry fields, updateEntryTagValue and updateEntryOldTagTableEntry. The tag for updateEntryTagValue entry is an old meaning. The updateEntryOldTagTableEntry is a copy of the old tag table entry.

5.2.3 The String Stack

Abstractly, the string stack is a stack of strings. In implementation, there is a stack of characters onto which strings of varying length can be pushed. It is named stringSpaceStack. There is also a stack of pointers point into the stack of characters which segments the latter into strings. It is named stringPointerStack. The top string starts with the character above the one pointed to by the top pointer on the pointer stack and ends with the character pointed to by the character top of stack pointer. The pointer stack will have to always have one pointer on its bottom which points to one character below the bottom of
the character stack to make this work. Figure 5 shows the structure of the string space stack and the string pointer stack.

![String Space Stack Diagram](image)

String Space Stack

Figure 5. The Structure of The String Stack

5.2.4 The Input File Stack

This is a stack of input files. Actually, there is just one input file open at a time. For this, there is a file, input buffer, and necessary buffer pointers. These are for the file which is on the top of the abstract input file stack. A real stack exists which holds the file name and buffer pointers for each file which is below the top of the abstract stack. When a file is to be pushed, the information about the current input file is pushed onto the real stack. Then it is closed and reopened with a new file name and positioned at the beginning of file. When a file is popped, the current file is closed (it
should always be at the end of file), the information about the
previous file is popped from the real stack and used to reopen
the old file name. The buffer pointers are used to position the
file where it was when it was pushed down. All S/SL input is done
from the file on the top of the abstract input file stack (i.e.
the file which is open). Initially, the input file is one with a
standard name.

5.2.5 Some Miscellaneous Stacks

There are some other stacks used in interpass. The tag stack
is a semantic stack of tag values. The reachable stack is a sim-
ple stack of booleans used to determine when the S-CODE is reach-
able, i.e. when run time control can reach the current pointer.
The P6 label stack is a stack of Pass 6 code labels for use in
translating S-CODE if and skip statements. And the count stack is
a stack of integers to handle the compile time addressing comput-
tation.

5.3 Tables

The tables used in interpass are implemented as array struc-
ture. Those arrays are subscripted by some particular values.
There are semantic operations that refer to those tables.
5.3.1 The Tag Table

This is an array, subscripted by the tag values. The tag table consists of two entry fields, tagEntryState and tagEntryDescriptorPointer. The values of tagEntryState are undefinedTagState, specifiedTagState, and definedTagState. All entries are initially undefined. The tagEntryDescriptorPointer points to the descriptor associated with this tag. It is meaningful only when tagEntryState value is not equal to undefinedTagState.

5.3.2 The Index Table

The index table is an array, subscripted by S-CODE index. The S-CODE index is one kind of code label used in S-CODE. There is actually a stack of such tables (only two are required). A newly pushed index table must have indexState value equal to undefinedIndexState, for all entries.

The index table has two entry fields, indexEntryState and indexEntryP6Label. The values which indexEntryState can take are undefinedIndexState, definedIndexState, and referencedIndexState. Undefined means the index currently has no meaning. Defined means the destination has occurred but the jump to it at compile time has not. Referenced means the jump to it at compile time has occurred but the destination has not.

The indexEntryP6Label is the Pass 6 label which this index
is to be translated into. It is meaningful only when the indexState value equals definedIndexState or referencedIndexState.

5.3.3 The Taglist Table

This is a table of translations between external numbers and tags. It is implemented as an array of tags, subscripted by external number. There are semantic operations to make it empty, add a new element, and translate a count stack value to a tag and push it.

5.3.4 The Type Convert Table

This is a table of translations between the S-CODE types and the Pass 6 types. There are semantic operations to convert the S-CODE types to Pass 6 types. The convert table between the S-CODE type and the Pass 6 type is shown in below.

<table>
<thead>
<tr>
<th>S-CODE</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>P6 halfword</td>
</tr>
<tr>
<td>CHAR</td>
<td>P6 byte</td>
</tr>
<tr>
<td>INT</td>
<td>P6 word</td>
</tr>
<tr>
<td>REAL</td>
<td>P6 shortreal</td>
</tr>
<tr>
<td>LREAL</td>
<td>P6 real</td>
</tr>
<tr>
<td>ADDR</td>
<td>P6 word</td>
</tr>
<tr>
<td>GADDR</td>
<td>P6 wordpair</td>
</tr>
<tr>
<td>OADDR</td>
<td>P6 pointer</td>
</tr>
<tr>
<td>PADDR</td>
<td>P6 word</td>
</tr>
<tr>
<td>RADDR</td>
<td>P6 word</td>
</tr>
<tr>
<td>SIN</td>
<td>P6 halfword</td>
</tr>
<tr>
<td>SIZE</td>
<td>P6 word</td>
</tr>
</tbody>
</table>
5.4 Descriptors

The descriptors contain information about various objects which will exist during the computation. The descriptors themselves do not exist at run time; they are compile time objects used to control the generation of executable code. Once a descriptor has been defined, it remains unaltered, but the tagged descriptors are used to create and modify anonymous descriptors held in a stack.

There are several different descriptor kinds used inside the interpass program, globalDescrKind, localDescrKind, constantDescrKind, importDescrKind, exportDescrKind, exitDescrKind, resultDescrKind, attributeDescrKind, labelDescrKind, switchDescrKind, profileDescrKind, bodyDescrKind, declaredTypeDescrKind, and builtinTypeDescrKind. The declaredTypeDescrKind is a type declared in the S-Program. The builtinTypeDescrKind can be one of the predefined S-CODE types INT, REAL, SHORTREAL, etc.

There are two fields present in all descriptors, descrKind and descrId. The descrKind can be any one of the descriptor kinds that are listed above. The descrId is the identifier string associated with the tag.

5.4.1 Data Descriptor Fields:

The data descriptor can be any one of globalDescrKind,
localDescrKind, constantDescrKind, attributeDescrKind, importDescrKind, exportDescrKind, exitDescrKind, and resultDescrKind. It consists of the following fields: dataDescrOutputPointer, dataDescrMode, dataDescrTypeTag, dataDescrDisplacement, dataDescrLength, dataDescrExternalId, dataDescrExternalProcLabel, dataDescrLineNumber, dataDescrFixrep, dataDescrHasFixrep, dataDescrRep, dataDescrLowerBound, dataDescrUpperBound, dataDescrHasRange, dataDescrHasP6CounterPart.

The dataDescrOutputPointer points to the spot in the output immediately following the token which produced this descriptor. It is meaningful only for descriptors on the S-stack.

The dataDescrMode is the MODE, as described in the S-CODE definition. There is a third value valAddrDescrMode. It describes data which is VAL insofar as what is legal to do with it, but which is of a type which Pass 6 always references with an address.

The dataDescrTypeTag is a tag for the type of the data. The dataDescrDisplacement is the displacement of the data and the dataDescrLength is the length of the data. The displacement and length of the data are not meaningful for resultDescrKind. Also the length of a quantity is not simply the length of its type because of possible rep and fixrep values.

If the data is external, then the dataDescrExternalId is the external name and the dataDescrExternalProcLabel is the Pass 6 procedure label which has been defined for Pass 6 for it.
The `dataDescrFixRep` resolves the number of repetitions of the indefinite repetition field within the type. It is meaningful only if `dataDescrHasFixRep` is true. The `dataDescrHasFixRep` field has boolean type. It is true if the data has a `fixRep` value.

The `dataDescrLowerBound` is the lower bound of the range and the `dataDescrUpperBound` is the upper bound of the range, if there is a range. The `dataDescrHasRange` has boolean type to control whether there is a range or not. It has true value if there is a range associated with the data item.

The `dataDescrLineNumber` is the SIMULA line number associated with the tag. The `dataDescrRep` is the number of repetitions of the type which are in this data item. The `dataDescrHasP6CounterPart` is a boolean type. It is true if the dataDescriptor corresponds to a Pass 6 descriptor on Pass 6's compile time stack.

5.4.2 Attribute Descriptor Fields

The attribute descriptor has the following fields: `attributeDescrParentTypeTag`, `attributeDescrInitialized`, `attributeDescrLink`, `attributeDescrAlternativeNumber`. Those descriptor fields are meaningful only for attribute descriptors.

The `attributeDescrParentTypeTag` is the tag for the record type of which this is an attribute. The `attributeDescrLink` is a link pointer for the list of attribute descriptors belonging to the parent type of this attribute. The
attributeDescrAlternativeNumber is the number of the alternative to which this attribute belongs. Alternatives are numbered with zero being the common part and the others being 1, 2, ... .

The attributeDescrInitialized has boolean type. Normally it has value false. It is set when a value for this attribute appears in a value for the surrounding record type. At the end of the record value, attributes are scanned to see what every attribute has been initialized. The boolean is also turned off during the scan.

5.4.3 Type Descriptor Fields

The type descriptor has the following fields: typeDescrInfo, typeDescrLength, typeDescrPrefixFixRep, typeDescrRepLength, typeDescrIndefAlternativeLength, typeDescrHasIndefRepField, typeDescrAlignment, typeDescrPrefixTag, typeDescrIndefRepTag, typeDescrAttributeList, typeDescrCurrentAttribute.

The typeDescrInfo is an encoding of the S-CODE info value supplied in the type definition. The typeDescrPrefixFixRep is the fixrep value which this type applies to its prefix type.

The typeDescrLength is the length of the prefix portion. This can differ from the length of the prefix type, because the prefix can have fixrep and rep attached.

The typeDescrRepLength is the length of the type, assuming the indefinite rep field (if any) is repeated zero times. If
there are multiple alternatives, this value is for the longest alternate.

The typeDescrIndefAlternativeLength is the length of the type, assuming the alternative which has the indef repetition is chosen and that the indefinite repetition is repeated zero times. If there is no indefinite repetition or no alternatives, this value will be the same as typeDescrLength.

The typeDescrHasIndefRepField has boolean type. It has true value if the type has an indefinite repetition field. The typeDescrIndefRepTag is the tag for the type of the indefinite repetition field of this type. A special tag value will assign to it if there is no indefinite repetition field.

The typeDescrAlignment is the required alignment for this type. Value will be one of [1,2,4,8]. The typeDescrPrefixTag is the tag of the type of the prefix for this type. A special tag value will be assigned to it if there is no prefix.

The typeDescrAttributeList is a pointer to the first of a list of attribute descriptors for this type. The typeDescrCurrentAttribute is a pointer point to the current attribute descriptor in the list. This moves through the list at compile time, while processing attribute lists.

5.4.4 Profile descriptor fields

The profile descriptor has the following fields: pro-

The profileDescrProfileKind can be one of knownProfileKind, systemProfileKind, externalProfileKind, internalProfileKind, and ordinaryProfileKind. It tells what kind of routine this profile defines. The profileDescrBodyTag can have only one body and which know its tag. It is meaningful only for known, system, and external profiles.

The profileDescrNature is the nature associated with the profile, if any. The profileDescrExitParameter is the tag of the exit parameter, if any. The profileDescrParameterARDisplacement is the displacement within the activation record where the next parameter will go.

The profileDescrParameterList is the root of the list of tags for the import parameters. The profileDescrCurrentParameter is a pointer to the current import parameter in the list. This moves through the list at compile time, while processing actual parameter lists.

5.4.5 Label, Switch, and Body Descriptor Fields

A label descriptor has labelDescrP6stlab field. It is the Pass 6 statement label which this S-CODE label is translated into.
A switch descriptor consists of two fields, switchDescrLowerP6Label and switchDescrUpperP6Label. The switchDescrLowerLabel is the minimum Pass 6 label used in the Pass 6 case table and the switchDescrUpperLabel is the maximum Pass 6 label used in the Pass 6 case table.

A body descriptor has bodyDescrP6plab, bodyDescrLocalARDisplacement, and bodyDescrProfileTag fields. The bodyDescrP6plab is the Pass 6 procedure label for the code defined by this body. The bodyDescrLocalARDisplacement is the displacement within the activation record where the next local will go. The bodyDescrProfileTag is a tag of the profile for this body.

5.5 Some Miscellaneous Data Structures

5.5.1 Value Area and Area Displacement Array

The value area is a large compile time area where S-CODE values of reals and record types are accumulated. There is also a compile time global output pointer which shows where in the output token stream the contents of the area should be inserted when they are complete.

For each of several areas, there is a compile time global variable which is the next available displacement within the area. These are arranged in an array, area displacement array, subscripted by area name.
5.5.2 Output Fragments

There is an output fragment stack, whose entries are roots of lists of fragment pointers. Fragment pointers give the beginning and ending location within the output file of a fragment. Ultimately, output fragment pointers are written to an output file called the map file, in the order that the fragments themselves are to be read by Pass 6. Fragment file addresses are zero-origin (32-bit) word numbers within the fragment file. Minus one in the first file pointer of a fragment pointer in the map file means the end of file.

The fragment pointer fields include a fragmentStartPointer and a fragmentEndPointer. The fragmentStartPointer is a pointer point to the first word of the fragment, within the fragment file. The fragmentEndPointer is a pointer point to the word beyond the last word of the fragment, within the fragment file.

The fragment list node fields consists of a fragmentListLeftLink, a fragmentListRightLink, and a fragmentListNodeFragmentPointer. The fragmentListLeftLink and fragmentListRightLink are the left and right link fields for the double, circularly linked list of fragment list nodes. The fragmentListNodeFragmentPointer is the fragment pointer.

The fragment stack entry is just a fragmentListNode which is a list header for the double, circularly linked list. Its fragment pointer field is not used. The fragment stack entry has a currentOutputPointer which is a pointer to the next word to be
written to the main output file. This is a physical file position.

5.5.3 Inside Routine and Compiler Switch Array

A global boolean variable is used to keep track of when interpass is inside a routine. When it is, some program elements are illegal in S-CODE.

The compiler switch array is an array of bytes which holds compiler option information.

5.5.4 Output Copy File and Output Copy Flag

The output copy file is an extra file which is used to record data on separately compiled modules. There are semantic operations which open and close it. When opened and when the copy flag is on, all bytes read from the input file are automatically copied to this file. In addition, there is a semantic operation which writes its parameter to this file only.

The output copy flag is a global boolean variable. There are semantic operations to turn it on and off. When off, it prevents copying to the output copy file, even though it may be open.

5.6 Examples of Interpass Processing Routines

According to those data structures listed above, there are
semantic operations which correspond to a number of processing routines to construct the interpass PASCAL portion. The entire semantic operations are given in [8]. Part of the semantic operations and their corresponding processing routines' codes are listed in appendix A and B, which are the parts that I wrote for the interpass.

Three processing routines are given here to show how they work inside the interpass. Each routine is given by its PASCAL language code and an explanation.

5.6.1 Example 1 -- VerifySEmpty

The first example illustrate the VerifySEmpty routine. From the routine name, we can figure out that this routine is used to verify that the top section of the S stack is empty. The code corresponding to this routine is shown below:

```pascal
; procedure VerifySEmpty

; begin

  VerifySHasOneElement ( sStackTop )

; VerifyMarkHasTwoElements ( markStackTop )

; if sStack [ sStackTop ] <> markStack [ markStackTop ]

  then Abort ( eVerifySEmpty )

end
```

As described in section 5.2.1, the mark stack is a stack of subscripts to the S stack which divides the S stack into sections. The only condition to check if the S stack is empty is to verify
that the value in the top of the S stack is equal to the value in the top of the mark stack.

This procedure starts with VerifySHasOneElement and VerifyMarkHasTwoElements to check if the S stack has at least one element and the mark stack has at least two elements. The reason why we need to check if the mark stack has at least two elements is because the bottom element of the mark stack always points one below the bottom of the entire S stack.

If the S stack has at least one element and the mark stack has at least two elements, then verify the equalization of the values in the top of the S stack and top of the mark stack. If they are not equal, it means the S stack is not empty, and abort using the error code eVerifySEmpty.

5.6.2 Example 2 -- SetTagSosMDataDotLengthToCount

The procedure SetTagSosMDataDotLengthToCount is used to set the length field of a data descriptor to the value on the top of the count stack. This procedure's code is shown below:

```plaintext
; procedure SetTagSosMDataDotLengthToCount
; var dPointer : descriptorPointerType
; begin
    VerifyTagHasTwoElements ( tagStackTop )
; VerifyCountHasOneElement ( countStackTop )
; if CheckDataDescriptorKind
    ( tagStack [ pred ( tagStackTop ) ] )
```
then  begin
    GetDescriptorPointer
    ( tagStack [ pred ( tagStackTop ) ], dPointer )
    ; dPointer ^  . dataDescrLength
    := countStack [ countStackTop ]
end
end

This procedure has a local variable "dPointer", which is declared as descriptor pointer type. The procedure starts with VerifyTagHasTwoElements and VerifyCountHasOneElement to verify that the tag stack has at least two elements and the count stack has at least one element. Then it calls the function CheckDataDescriptorKind to check whether the tag sos names a data descriptor. If the tag sos names a data descriptor, then use the tag sos as subscript to get the pointer from the tag table to the descriptor associated with the tag sos. And assign the value in the top of the count stack to the dataDescrLength field of the descriptor pointer.

5.6.3 Example 3 -- PushUpdateWithTagAndUndefine

The procedure PushUpdateWithTagAndUndefine is used to push the tag table entry named by tag tos onto the update stack. As described in section 5.2.2, a tag's old tag table entry and the tag are pushed onto the update stack when the tag is redefined. This procedure contains the following code:
; procedure PushUpdateWithTagAndUndefine

; begin

    VerifyTagHasOneElement ( tagStackTop )

; PushUpdateStack ( tagStack [ tagStackTop ],
                   tagTable [ tagStack [ tagStackTop ] ] )

; tagTable [ tagStack [ tagStackTop ] ] . tagEntryState
  := undefinedTagState

end

First, the procedure verifies that the tag stack has at least one element, then it pushes the tag table entry named by the top of the tag stack onto the update stack. Finally, it sets the tagEntryState field of the tag table named by the tag tos to undefinedTagState.
Chapter 6

Conclusion

In the previous chapters, we have summarized the basic concepts of the SIMULA language, the portable SIMULA system, and the Perkin-Elmer SIMULA system. To date, the SIMULA language has turned out to be a general, high level programming language and a system description language. It has had an important impact on modern programming languages by virtue of its introduction of the class and concatenation features. SIMULA itself has made it possible to do jobs within operations research and data processing. The very successful implementations on several systems are the reason for the increased use of the SIMULA language in recent years.

An adaptation of the portable SIMULA system to a target system is carried out by writing a translator from S-CODE to a language accepted by the target system, and writing the bodies of a number of processing routines that define the interface to the target environment. The front-end compiler and run time system, which are distributed in S-CODE, are translated, and the modules are linked together, giving a complete system.

For the Perkin-Elmer SIMULA System project, the design and coding of the S-Compiler and environment interface support package has been done. Many of the environment interface support package routines have been tested using artificial test
environments. The coding for the S/SL and PASCAL portions of interpass are both complete. That includes those processing routines that I coded for the interpass PASCAL portion. The interpass has been tested to translate the run time system.

Coding of the modifications to Pass 6 and Pass 7 is about 30% complete, with no testing done. No coding of the modifications to Pass 8 and Pass 9 is done. The S-CODE assembler and S-CODE dumper are both coded and thoroughly tested. The modifications to the token stream dumper for the Pass 6 input token stream is completed and thoroughly tested. And the modifications to the token stream dumpers for the Pass 7 through Pass 9 input token streams are 30% complete, with no testing done.

Some further work is needed in order to complete the whole system. This includes complete testing on interpass, coding and testing of the modifications to Pass 6 through pass 9, and coding and testing of the modifications to the token stream dumper for the Pass 7 through Pass 9 input token streams.

After we finish this work, we will choose several sample programs to compile and run under the system. If we can get the results we expect, then the Perkin-Elmer SIMULA System will be ready for the user to run the simulation programs using the SIMULA language.
REFERENCES


[5]. Robert Young; "Internal Documentation For The PASCAL/32 Compiler"; manuscript; Kansas State University, Manhattan, Kansas.


[7]. Rodney M. Bates; "Interpass Semantic Data Structures"; manuscript; Computer Science Department, Kansas State University, Manhattan, Kansas; 1984.

[8]. Rodney M. Bates; "Interpass Semantic Operations"; manuscript; Computer Science Department, Kansas State University, Manhattan, Kansas; 1984.
APPENDICES
Appendix A

Interpass Semantic Operations

The interpass semantic operations are provided by Dr. Bates as the documentation for the design of the interpass PASCAL portion processing routines. In here, only those semantic operations and processing routines that I wrote for the interpass PASCAL portion are listed below. The other semantic operations that are not listed in here can be seen in the interpass semantic operations documentation [8].

There are some naming conventions in the interpass semantic operations. All semantic operation names begin with "o". They generally begin with a verb, followed by the name of the data structure they change, or the main structure they examine.

A stack name by itself means the top element on that stack. For other stack elements, "tos", "3os", and "4os" follow the stack name. "Dot" is like the "." in PASCAL, but sometimes a little bit more abstract. E.g., "tagDotLength" refers to the "length" field of the descriptor pointed to by tag tons, indirectly through the tag table. "M" is an abbreviation for "must be" and denotes a condition which must hold if the S-program and interpass are correct.

For ease in debugging, all such conditions are explicitly verified by the semantic operations, and interpass is terminated with an error message if one of them fails.
Parts of the interpass semantic operations are listed below. The numbers which at the end of each operation name are used to indicate the line numbers of which the processing routine associate with this semantic operation locate in appendix B.

### S Stack ###

oVerifySDotDescrKind (DescriptorKind)  { 9-20 }
Verify that the descriptor on S tos has the kind given in the parameter. Abort if not.

oSetSDotIdToString  { 22-28 }
Set the identifier field of S tos to string tos.

oSetSMDataDotOutputPointertoCount  { 30-37 }
S tos must be a data descriptor. Set the output pointer field of S tos to count tos.

oVerifySEmpty  { 39-48 }
Verify that the top section of the S stack is empty. Abort if not.

oPopMarkedSSectionMEEmpty  { 50-52 }
The top section of the S stack must be empty. Pop this section by popping the mark stack.

### Emit Operations ###

oEmitToken (P6Token)  { 57-62 }
Write the P6 token to the main file.

oEmitTokenBeforeSMDataDotOutputPointer (P6Token)  { 64-72 }
Like oEmitToken, but insert into output stream at S dot output pointer.

oEmitTokenBeforeSSosMDataDotOutputPointer (P6Token)  { 74-82 }
Like oEmitTokenBeforeSMDataDotOutputPointer, but insert before S sos dot output pointer.

oEmitValue (integer)  { 84-89 }
Write the integer value to the main file.

oEmitValueBeforeSMDataDotOutputPointer (integer)  { 91-99 }
Like oEmitValue, but insert into output stream at S dot output pointer.

oEmitValueBeforeSSosMDataDotOutputPointer (integer)  { 101-109 }
Like oEmitValueBeforeSMDataDotOutputPointer, but insert before S sos dot output pointer.
oEmitTagMTypeDotIndefAlternativeLength  { 111-124 }
Tag tos must name a type descriptor. Emit its
indefAlternativeLength field.

oEmitTagMTypeDotP6TypeBeforeSMDataDotOutputPointer  { 126-157 }
Tag tos must be a S-CODE type tag. S tos must be a
data descriptor. Convert the type to P6 type and
emit it at the point in the output token stream
identified by the output pointer field of S tos.

oEmitTagMSwitchSubCountDotP6Label  { 159-176 }
Tag tos must name a switch descriptor. Use count
tos to select one of the elements of the switch.
Do the necessary arithmetic to convert this to the
appropriate Pass 6 label. Emit this label.

oEmitTagMProfileDotId8Chars  { 178-204 }
Tag tos must name a profile descriptor. Emit its
identifier field packing exactly 8 characters into
two successive words in the output file.

oEmitValueAreaCountByteBeforeSMDataDotOutputPointer  { 206-237 }
Emit the contents of the value area, from beginning,
with length taken from Count tos, into the token
stream at S tos dot output pointer.

### Tag Stack ###

oIsTagMTypeStructured  { 242-255 }
Tag tos must name a type descriptor. Return a boolean
indicating whether it is a structured type.

oSetTagSosMTypeDotPrefixTagToTag  { 257-271 }
Tag sos must name a type descriptor. Set its prefix
tag field to tag tos.

oSetTagSosMTypeDotIndefRepTagToTag  { 273-287 }
Tag sos must name a type descriptor. Set its
indefRepTag field to tag tos.

oSetTagSosMTypeDotPrefixLengthToCount  { 289-304 }
Tag sos must name a type descriptor. Set its prefix
length field to count tos.

oSetTagSosMTypeDotPrefixFixrepToCount  { 306-321 }
Tag sos must name a type descriptor. Set its prefix
fixrep field to count tos.

oSetTagMTypeDotLengthToCount  { 323-337 }
Tag tos must name a type descriptor. Set its length
field to count tos.
oSetTagSosMTypeDotIndefAlternativeLengthToCount   { 339-354 }
Tag sos must name a type descriptor. Set its indefinite alternative length field to count tos.

oSetTagSosMTypeDotAlignmentToCount   { 356-371 }
Tag sos must name a type descriptor. Set its alignment field to count tos.

oTurnOnTagSosMTypeDotHasIndefRepField   { 373-385 }
Tag tos must name a type descriptor. Turn on its hasIndefRepField field.

oTurnOnTagSosMTypeDotHasIndefRepField   { 387-400 }
Tag sos must name a type descriptor. Turn on its hasIndefRepField field.

oIsTagMTypeDotCurrentAttributeNil   { 402-419 }
Tag tos must name a type descriptor. Return true if and only if the current attribute field is nil.

oSetTagSosMDataDotLengthToCount   { 420-435 }
Tag sos must name a data descriptor. Set the length field of this descriptor to count tos.

oSetTagMAttributeDotParentTypeToTagSos   { 437-451 }
Tag tos must name an attribute descriptor. Set its parent type tag field to tag sos.

oSetTagMAttributeDotAlternativeNumberToCount   { 453-468 }
Tag tos must name an attribute descriptor. Set the alternative number field of this descriptor to count tos.

oSetTagMProfileDotNature (profileNature)   { 470-484 }
Tag tos must name a profile descriptor. Set its nature field to the value of the parameter.

oSetTagMTypeDotInfoToInputString   { 486-544 }
Tag tos must name a type descriptor. Read an input string, translate it to a typeDescInfo value using the table below. Store the result in the Info field of the type descriptor.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'TYPE'</td>
<td>typeInfo</td>
</tr>
<tr>
<td>'DYNAMIC'</td>
<td>dynamicInfo</td>
</tr>
<tr>
<td>others</td>
<td>invalidInfo</td>
</tr>
</tbody>
</table>

oIsTagMProfileDotCurrentParameterNil   { 546-563 }
Tag tos must name a profile descriptor. Return true if and only if the current parameter field is nil.

oAppendTagSosMTypeDotAttributeListWithTag   { 565-590 }
Tag sos must name a type descriptor. Tag tos and the descriptor it points to through the tag table are added to the end of the attribute list of the type descriptor.
oInitializeTagMTypeDotCurrentAttribute  { 592-607 }
Tag tos must name a type descriptor. The currentAttribute
field of this descriptor is set to the first attribute
tag of the type. If there is no attribute, it is set to nil.

oAdvanceTagMTypeDotCurrentAttribute  { 609-626 }
Tag tos must name a type descriptor. The current
attribute field of this descriptor is changed to be
the attribute following the one it currently names.
If there is no additional attribute, it is set to nil.

oPushTagWithTagMTypeDotCurrentAttributeTag  { 628-642 }
Tag tos must name a type descriptor. The current
attribute field of this descriptor must not be nil.
The tag of the taglist node pointed to by the
current attribute field is pushed onto the tag stack.

oAppendTagSosMProfileDotParameterListWithTag  { 644-671 }
Tag sos must name a profile descriptor. Tag tos and
the descriptor it points to through the tag table are
added to the end of the import parameter list of the
profile descriptor.

oInitializeTagMProfileDotCurrentParameter  { 673-690 }
Tag tos must name a profile descriptor. The current
parameter field of this descriptor is set to the
first import parameter tag of the profile. If there
is no import parameter, it is set to an illegal tag
value.

oAdvanceTagMProfileDotCurrentParameter  { 692-712 }
Tag tos must name a profile descriptor. The current
parameter field of this descriptor is changed to be
the import parameter following the one it currently
names. If there is no additional import parameter,
it is set to nil.

oPushTagWithTagMProfileDotCurrentParameterTag  { 714-732 }
Tag tos must name a profile descriptor. The current
parameter field of this descriptor must not be nil.
The tag of the taglist node pointed to by the current
parameter field is pushed onto the tag stack.

oVerifyTagHasThreeElements  { 734-741 }
Verify that the tag stack has at least three elements.
Abort if not.

### Count Stack ###

oPushCountWithSMDotDataDotLength  { 746-753 }
Push length field of S tos onto the count stack.
S tos must be a data descriptor.
oPushCountWithSSosMDDataDotFixrep { 755-763 }
Push fixrep value of S sos onto the count stack.
S sos must be a data descriptor.

oPushCountWithSSosMDDataDotRep { 765-773 }
Push rep field of S sos onto the count stack.
S sos must be a data descriptor.

oPushCountWithSMDataDotOutputPointer { 775-782 }
Push output pointer field of S sos onto the count stack.
S sos must be a data descriptor.

oPushCountWithSSosMDDataDotOutputPointer { 784-793 }
Push output pointer field of S sos onto the count stack.
S sos must be a data descriptor.

oPushCountWithTagOfTypeDotAlignment { 795-809 }
Tag tos must be a S-CODE type. Push the alignment of
this type onto the count stack.

oVerifyCountHasThreeElements { 811-819 }
Verify that the count stack has at least three elements.
Abort if not.

oVerifyCountHasFourElements { 821-829 }
Verify that the count stack has at least four elements.
Abort if not.

oSwapCount { 831-841 }
Swap the top two elements of the count stack.

oRotateCount3osToTos { 843-855 }
Move count 3os to the top, letting tos and sos shift down.

oRotateCount4osToTos { 857-871 }
Move count 4os to the top, letting tos, sos, and 3os
shift down.

oRotateCountTosTo3os { 873-885 }
Remove count tos and insert it beneath the count sos
(which was the count 3os initially).

oRotateCountTosTo4os { 887-901 }
Remove count tos and insert it beneath the count 3os
(which was the count 4os initially).

oMaxCount { 903-912 }
Replace count tos and count sos with the larger of these.

oComputeRangeSizeAndAlignmentOnCount { 914-934 }
Count sos and tos contain a lower and upper bound,
respectively. Replace them by a size and alignment for
the smallest P6 type which will hold this range.
*** Inside Routine ***

cTurnOnInsideRoutine { 939-941 }
   Turn on the inside routine global variable.

cTurnOffInsideRoutine { 943-945 }
   Turn off the inside routine global variable.

cVerifyInsideRoutineIsOff { 947-954 }
   Abort if the inside routine global variable is not off.

*** Tag Table ***

cCreateDeclaredTypeDescriptorAtTag { 959-976 }
   Create a new descriptor of declaredTypeDescKind with
   hasIndefRepField = false, prefixTag = noTag,
   info = noRecordInfo, and attributeList = nil.
   Store the pointer to it at tag table sub tag tos.

*** String Stack ***

cPushStringWithInputString { 981-983 }
   Read a string from the input and push it onto the
   string stack.

cPopString { 985-994 }
   Pop the string stack.

cSwapString { 996-1032 }
   Swap string tos and string sos.

cSkipInputString { 1034-1036 }
   Read a string from the input and throw it away.

cVerifyStringEqualsInputString (errorCode) { 1038-1085 }
   Verify that string tos equals the string in the input.
   If not, abort using errorCode. The input string is
   consumed.

cVerifyStringPointerHasTwoElements { 1087-1095 }
   Verify that the string pointer stack has at least two
   elements. Abort if not.

cVerifyStringPointerHasThreeElements { 1097-1105 }
   Verify that the string pointer stack has at least three
   elements. Abort if not.

*** Compiler Switch Array ***

cSetSwitchSubCountSosToCount { 1110-1117 }
   Count sos is used as a subscript to the compiler switch
   array. The selected element is set to the value of count
tos.
### Input File Stack ###

**oPushAndOpenInputFileWithStringVisible** { 1132-1163 }
String tos is transformed into a visible file name by appending ".v". The name and position of the current input file are pushed down, and file with the constructed name is opened. It is positioned at the beginning, and becomes the file from which S-CODE is now read.

**oPushAndOpenInputFileWithTagList** { 1165-1196 }
Just like **oPushAndOpenInputFileWithStringVisible**, except the file is a taglist file, whose name is constructed by appending ".t".

### Output Copy File ###

**oOpenOutputCopyFileWithStringVisible** { 1201-1232 }
String tos is transformed into a visible file name by appending ".v". The file which the constructed name is opened as the output copy file. The output copy file must currently be closed. If the output copy flag is turned on, all input read from the current input file will automatically be copied to this file.

**oOpenOutputCopyFileWithTagList** { 1234-1265 }
Just like **oOpenOutputCopyFileWithStringVisible**, except the file is a taglist file, whose name is constructed by appending ".t".

**oTurnOnOutputCopying** { 1267-1269 }
**oTurnOffOutputCopying** { 1271-1273 }
The output copy flag is turned on or off. When on, and when there is an open output copy file, all input read from the current input file will automatically be copied to this file.

**oEmitStringToOutputCopyFile** { 1275-1299 }
String tos is written to the output copy file. The file must be open, but the output copy flag has no effect.

### Taglist Table ###

**oMakeTagListTableEmpty** { 1304-1311 }
Make the taglist table empty, i.e., it contains no translation rules.

**oAddTagListEntryFromTagAndCount** { 1313-1329 }
This routine does arithmetic on tags. Tag contains, top toward bottom, an external tag, taglimit, and tagbase. Count tos contains an external number. Verify that tagbase + external number <= taglimit. Abort if not. Then construct a translation rule which translates external tag to tagbase + external number. Put this rule into the taglist table.
oTranslateTagUsingTaglistTable  { 1331-1339 }
Use tag tos as a search key to search the taglist table.
If no translation is found, abort. Otherwise, replace
tag tos with the translated value.

### Update Stack ###

oPushUpdateStack  { 1344-1358 }
Push old tag table value and the tag onto the update stack.

oVerifyUpdateHasOneElement  { 1360-1366 }
Verify that the update stack has at least one element.

oPushUpdateWithAllDefinedLabelsAndUndefine  { 1368-1386 }
Each defined tag with descriptor kind of labelDescrKind
is pushed onto the update stack along with its tag table
entry. The tag state in the tag table becomes
undefinedTagState.

oPushUpdateWithTagAndUndefine  { 1388-1395 }
The tag table entry named by tag tos is pushed onto the
update stack. The tag table entry state becomes
undefinedTagState.

oPopUpdateAndRestoreTag  { 1397-1406 }
The top entry of the update stack is copied back into
the tag table at the tag named in the entry.

oPopAllUpdatesAndRestoreTags  { 1408-1416 }
Do oPopUpdateAndRestoreTag until the update stack is empty.

### Miscellaneous Semantic Operation ###

oIdentifyLineNumber  { 1122-1127 }
This operation tells the semantic routines that count tos
is the value of the current line number. Semantic routines
use this information to put the line number in error
message.
Appendix B

The Code of Interpass Processing Routines

{ The code of Interpass processing routines associate with
  the semantic operations which are listed in Appendix A. 
  These Interpass PASCAL portion processing routines are 
  those that I wrote for the Interpass program }

{ *** S Stack *** }

; procedure VerifySDotDescrKind
( DescriptorKind : descrKindTyp )

{ Abort if the descriptor on S tos is not equal to 
  the value of the parameter }

; begin
VerifySHasOneElement ( sStackTop )
; if sStack [ sStackTop ] ^= descrKind
<> DescriptorKind
  then Abort ( eVerifySDotDescrKind )
end

; procedure SetSDotIdToString

; begin
VerifySHasOneElement ( sStackTop )
; sStack [ sStackTop ] ^= dataDescrId
:= stringStack [ stringStackTop ]
end

; procedure SetSMDataDotOutputPointerToCount

; begin
VerifySHasOneElement ( sStackTop )
; if CheckDataDescriptorKind ( sStack [ sStackTop ] )
  then sStack [ sStackTop ] ^= dataDescrOutputPointer
     := countStack [ countStackTop ]
end

; procedure VerifySEmpty

{ Abort if the top section of the S stack is not empty }

; begin
VerifySHasOneElement ( sStackTop )
  ; VerifyMarkHasTwoElements ( markStackTop )
  ; if sStack [ sStackTop ] <> markStack [ markStackTop ]
  then Abort ( eVerifySEmpty )
end

; procedure PopMarkedSSectionMEmpty

; begin VerifySEmpty ; Popmark end

[ *** Emit Operations *** ]

; procedure EmitToken ( P6Token : integer )

; begin
  writeFile ( mainFile , P6Token )
  ; currentOutputPointer := currentOutputPointer + 4
end

; procedure EmitTokenBeforeSMDDataDotOutputPointer
( P6Token : integer )

; begin
  VerifySHasOneElement ( sStackTop )
  ; InsertToken
    ( sStack [ sStackTop ] " . dataDescrOutputPointer
    , P6Token )
end

; procedure EmitTokenBeforeSSosMDataDotOutputPointer
( P6Token : integer )

; begin
  VerifySHasTwoElements ( sStackTop )
  ; InsertToken
    ( sStack [ pred ( sStackTop ) ] " .
    dataDescrOutputPointer , P6Token )
end

; procedure EmitValue ( value : integer )

; begin
  writeFile ( mainFile , value )
  ; currentOutputPointer := currentOutputPointer + 4
end

; procedure EmitValueBeforeSMDDataDotOutputPointer
( value : integer )

; begin
  VerifySHasOneElement ( sStackTop )
  ; InsertToken
    ( sStack [ sStackTop ] " . dataDescrOutputPointer
; procedure EmitValueBeforeSSosMDotDataDotOutputPointer
  ( value : integer )

    ; begin
    ; VerifySHasTwoElements ( sStackTop )
    ; InsertToken
    ; ( sStack [ pred ( sStackTop ) ] ° .
    ; dataDescrOutputPointer , value )
    ; end

; procedure EmitTagMTypeDotIndefAlternativeLength

    ; var dPointer : descriptorPointerType

    ; begin
    ; VerifyTagHasOneElement ( tagStackTop )
    ; if CheckTypeDescriptorKind
    ; ( tagStack [ tagStackTop ] )
    ; then begin
    ; GetDescriptorPointer ( tagStack [ tagStackTop ]
    ; , dPointer )
    ; EmitToken ( dPointer ° .
    ; typeDescrIndefAlternativeLength )
    ; end

; procedure EmitTagMTypeDotP6TypeBeforeSMDDataOutputPointer

    ; var SCodeBuiltInType : SCodeBuiltInTypeTyp
    ; P6Token : P6TokenType

    ; begin
    ; VerifyTagHasOneElement ( tagStackTop )
    ; VerifySHasOneElement ( sStackTop )
    ; if tagStack [ tagStackTop ] in SCodeBuiltInType
    ; then if sStack [ sStackTop ] ° . descrKind
    ; in dataDescriptorKinds
    ; then begin
    ; TypeConvertTable [ BOOL ] := P6Token [ P6halfword ]
    ; TypeConvertTable [ CHAR ] := P6Token [ P6byte ]
    ; TypeConvertTable [ INT ] := P6Token [ P6word ]
    ; TypeConvertTable [ REAL ] := P6Token [ P6shortword ]
    ; TypeConvertTable [ LREAL ] := P6Token [ P6real ]
    ; TypeConvertTable [ ADDR ] := P6Token [ P6word ]
    ; TypeConvertTable [ ADDR ] := P6Token [ P6wordpair ]
    ; TypeConvertTable [ ADDR ] := P6Token [ P6pointer ]
    ; TypeConvertTable [ ADDR ] := P6Token [ P6word ]
    ; TypeConvertTable [ ADDR ] := P6Token [ P6word ]
    ; TypeConvertTable [ ADDR ] := P6Token [ P6halfword ]
    ; TypeConvertTable [ ADDR ] := P6Token [ P6word ]

    ; for SCodeBuiltInType := BOOL to SIZE
    ; do InsertToken
152 ( sStack [ sStackTop ] ~ . dataDescrOutputPointer
153 , TypeConvertTable [ SCCodeBuiltInType ] )
154 end
155 else Abort ( eVerifyDataDescriptor )
156 else Abort ( eVerifySCCodeType )
157 end
158
159 ; procedure EmitTagMSwitchSubCountDotP6Label
160
161 ; var switchValue : integer
162 ; dPointer : descriptorPointerTyp
163
164 ; begin
165 VerifyTagHasOneElement ( tagStackTop )
166 ; VerifyCountHasOneElement ( countStackTop )
167 ; if CheckSwitchDescriptorKind ( tagStack [ tagStackTop ] )
168 then begin
169 GetDescriptorPointer ( tagStack [ tagStackTop ]
170 , dPointer )
171 ; switchValue
172 := countStackTop + dPointer ~ . switchDescrLowerP6Label
173 ; if switchValue <= dPointer ~ . switchDescrUpperP6Label
174 then EmitToken ( switchValue )
175 end
176
177 ; procedure EmitTagMProfileDotId8Chars
178
179 ; var dPointer : descriptorPointerTyp
180 ; charList : array [ 1 .. 4 ] of char
181 ; count , i : integer
182
183 ; begin
184 VerifyTagHasOneElement ( tagStackTop )
185 ; if CheckProfileDescriptorKind
186 ( tagStack [ tagStackTop ] )
187 then begin
188 GetDescriptorPointer ( tagStack [ tagStackP ]
189 , dPointer )
190 ; count := 1
191 ; for i := 1 to 8
192 do begin
194 profileDescrId [ i ]
195 ; if count = 4
196 then begin
197 write4Chars ( charList )
198 ; count := 1
199 end
200 else count := count + 1
201 end
202 end
203
204 end
205
; procedure EmitValueAreaCountBytesBeforeSMDapDotOutputPointer

; var byteList : array [ 1 .. 4 ] of byte
; length, extra, i, j, k, m, n : integer

; begin
  VerifyCountHasOneElement ( countStackTop )
  ; length := countStack [ countStackTop ] div 4
  ; extra := countStack [ countStackTop ] mod 4
  ; j := 1
  ; i := 1
  ; k := 1
  ; while j <= length
    do begin
      byteList [ k ] := valueArea [ i ]
      ; if k = 4
        then begin
          write4Bytes ( byteList )
          ; k := 1
          ; j := j + 1
          end
        else k := k + 1
      ; i := i + 1
    end
  ; for m := 1 to 4 do byteList [ m ] := 0
  ; for n := 1 to extra
    do begin
      byteList [ n ] := valueArea [ i ]
      ; i := i + 1
    end
  ; write4Bytes ( byteList )
end

{ *** Tag Stack *** }

; procedure IsTagMTypeStructured

{ Return true if the tag tos descriptor is a structured type }

; var structuredType : boolean

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; structuredType := true
  ; if CheckTypeDescriptorKind ( tagStack [ tagStackTop ] )
    then if tagStackTop >= lastSCodeBuiltInType
      then HandleChoice ( ord ( structuredType ) )
    else HandleChoice ( ord ( not structuredType ) )
  end

; procedure SetTagSosMTypeDotPrefixTagToTag

; var dPointer : descriptorPointerType
; procedure SetTagSosMTypeDotIndefRepTagToTag

; var dPointer : descriptorPointerType

; begin
    VerifyTagHasTwoElements ( tagStackTop )
; if CheckTypeDescriptorKind
    ( tagStack [ pred ( tagStackTop ) ] )
then begin
    GetDescriptorPointer
    ( tagStack [ pred ( tagStackTop ) ] , dPointer )
; dPointer ^ . typeDescrIndefRepTag
    := tagStack [ tagStackTop ]
end
end

; procedure SetTagSosMTypeDotPrefixLengthToCount

; var dPointer : descriptorPointerType

; begin
    VerifyTagHasTwoElements ( tagStackTop )
; VerifyCountHasOneElement ( countStackTop )
; if CheckTypeDescriptorKind
    ( tagStack [ pred ( tagStackTop ) ] )
then begin
    GetDescriptorPointer
    ( tagStack [ pred ( tagStackTop ) ] , dPointer )
; dPointer ^ . typeDescrPrefixLength
    := countStack [ countStackTop ]
end
end

; procedure SetTagSosMTypeDotPrefixFixrepToCount

; var dPointer : descriptorPointerType

; begin
    VerifyTagHasTwoElements ( tagStackTop )
; VerifyCountHasOneElement ( countStackTop )
; if CheckTypeDescriptorKind

( tagStack [ pred ( tagStackTop ) ] )
then begin
  GetDescriptorPointer
  ( tagStack [ pred ( tagStackTop ) ] , dPointer )
  ; dPointer ^= typeDescrPrefixFixRep
  := countStack [ countStackTop ]
end end

; procedure SetTagSosMTypeDotLengthToCount

; var dPointer : descriptorPointerType

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; VerifyCountHasOneElement ( countStackTop )
  ; if CheckTypeDescriptorKind ( tagStack [ tagStackTop ] )
  then begin
    GetDescriptorPointer ( tagStack [ tagStackTop ]
                        , dPointer )
    ; dPointer ^= . typeDescrLength :=
    countStack [ countStackTop ]
  end end

; procedure SetTagSosMTypeDotIndefAlternativeLengthToCount

; var dPointer : descriptorPointerType

; begin
  VerifyTagHasTwoElements ( tagStackTop )
  ; VerifyCountHasOneElement ( countStackTop )
  ; if CheckTypeDescriptorKind
    ( tagStack [ pred ( tagStackTop ) ] )
  then begin
    GetDescriptorPointer
    ( tagStack [ pred ( tagStackTop ) ] , dPointer )
    ; dPointer ^= . typeDescrIndefAlternativeLength
    := countStack [ countStackTop ]
  end end

; procedure SetTagSosMTypeDotAlignmentToCount

; var dPointer : descriptorPointerType

; begin
  VerifyTagHasTwoElements ( tagStackTop )
  ; VerifyCountHasOneElement ( countStackTop )
  ; if CheckTypeDescriptorKind
    ( tagStack [ pred ( tagStackTop ) ] )
  then begin
    GetDescriptorPointer
    ( tagStack [ pred ( tagStackTop ) ] , dPointer )
; dPointer ^ . typeDescrAlignment
    := countStack [ countStackTop ]
end

; procedure TurnOnTagMTypeDotHasIndefRepField

; var dPointer : descriptorPointerType

; begin
  VerifyTagHasOneElement ( tagStackTop )
  if CheckTypeDescriptorKind ( tagStack [ tagStackTop ] )
    then begin
      GetDescriptorPointer ( tagStack [ tagStackTop ] , dPointer )
      dPointer ^ . typeDescrHasIndefRepField := true
    end
end

; procedure TurnOnTagSosMTypeDotHasIndefRepField

; var dPointer : descriptorPointerType

; begin
  VerifyTagHasTwoElements ( tagStackTop )
  if CheckTypeDescriptorKind
    ( tagStack [ pred ( tagStackTop ) ] )
    then begin
      GetDescriptorPointer
        ( tagStack [ pred ( tagStackTop ) ] , dPointer )
      dPointer ^ . typeDescrHasIndefRepField := true
    end
end

; procedure IsTagMTypeDotCurrentAttributeNil

{ Return true iff the current attribute field is nil }

; var dPointer : descriptorPointerType

; begin
  VerifyTagHasOneElement ( tagStackTop )
  if CheckTypeDescriptorKind ( tagStack [ tagStackTop ] )
    then begin
      GetDescriptorPointer ( tagStack [ tagStackTop ] , dPointer )
      ; HandleChoice
        ( ord ( dPointer ^ . typeDescrCurrentAttributeTag = nil )
        )
    end
end

; procedure SetTagSosMDataDotLengthToCount
; var dPointer : descriptorPointerType

; begin
  VerifyTagHasTwoElements ( tagStackTop )
  ; VerifyCountHasOneElement ( countStackTop )
  ; if CheckDescriptorKind
    ( tagStack [ pred ( tagStackTop ) ] )
    then begin
      GetDescriptorPointer
        ( tagStack [ pred ( tagStackTop ) ] , dPointer )
      ; dPointer ^ . dataDescrLength :=
        countStack [ countStackTop ]
    end
end

; procedure SetTagMAttributeDotParentTypeToTagSos

; var dPointer : descriptorPointerType

; begin
  VerifyTagHasTwoElements ( tagStackTop )
  ; if CheckAttributeDescriptorKind
    ( tagStack [ tagStackTop ] )
    then begin
      GetDescriptorPointer ( tagStack [ tagStackTop ] , dPointer )
      ; dPointer ^ . DescrParentTypeTag
        := tagStack [ pred ( tagStackTop ) ]
    end
end

; procedure SetTagMAttributeDotAlternativeNumberToCount

; var dpointer : descriptorPointerType

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; VerifyCountHasOneElement ( countStackTop )
  ; if CheckAttributeDescriptorKind
    ( tagStack [ tagStackTop ] )
    then begin
      GetDescriptorPointer ( tagStack [ tagStackTop ] , dpointer )
      ; dpointer ^ . attributeDescrAlternativeNumber
        := countStack [ countStackTop ]
    end
end

; procedure SetTagMProfileDotNature
    ( profileNature : profileNatureTyp )

; var dPointer : descriptorPointerType

; begin
VerifyTagHasOneElement ( tagStackTop )
  ; if CheckProfileDescriptorKind
  ( tagStack [ tagStackTop ] )
  then begin
    GetDescriptorPointer ( tagStack [ tagStackTop ]
      , dPointer )
    ; dPointer ~> . profileDescrNature := profileNature
  end
end

; procedure SetTagMTypeDotInfoToInputString

var dPointer : descriptorPointerType
  ; Info : array [ 1 .. maxStringLength ] of char
  ; TString , DString : packed array [ 1 .. 4 ] of char
  ; length , i , j : integer
  ; match : boolean

begin
  VerifyTagHasOneElement ( tagStackTop )
  ; if CheckTypeDescriptorKind ( tagStack [ tagStackTop ] )
  then begin
    GetDescriptorPointer ( tagStack [ tagStackTop ]
      , dPointer )
    ; ReadString
    ; length := 0
    ; for i
      := ( stringPointerStack
        [ pred ( stringPointerStackTop ) ]
          + 1
           )
    to stringPointerStack [ stringPointerStackTop ]
    do begin
      length := length + 1
      ; Info [ length ] := stringSpaceStack [ i ]
    end
    ; TString := 'TYPE'
    ; DString := 'DYNAMIC'
    ; if length = 4
    then begin
      match := true
      ; j := 1
      ; while match and ( j <= length )
      do begin
        if Info [ j ] <> TString [ j ]
          then match := false
        else j := j + 1
      end
      ; if match
      then dPointer ~> . typeDescrInfo := typeRecordInfo
      else dPointer ~> . typeDescrInfo := invalidRecordInfo
    end
    else if length = 7
    then begin
match := true
; j := 1
; while match and ( j <= length )
do begin
  if Info [ j ] <> DString [ j ]
  then match := false
  else j := j + 1
end
; if match
  then dPointer ^ . typeDescrInfo := dynamicRecordInfo
  else dPointer ^ . typeDescrInfo := invalidRecordInfo
end
else dPointer ^ . typeDescrInfo := invalidRecordInfo
end

; procedure IsTagMPopCurrentParameterNil
{ Return true iff the tag tos current parameter field is nil }

; var dPointer : descriptorPointerType

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; if CheckProfileDescriptorKind
    ( tagStack [ tagStackTop ] )
    then begin
      GetDescriptorPointer ( tagStack [ tagStackTop ]
        , dPointer )
      ; HandleChoice
        ( ord ( dPointer ^ . profileDescrCurrentParameter = nil )
        )
    end
end

; procedure AppendTagSosMTypeDotAttributeListWithTag

; var dPointer : descriptorPointerType
; tagNode : tagNodePointerType
; currentTagNode : tagNodePointerType

; begin
  VerifyTagHasTwoElements ( tagStackTop )
  ; if CheckTypeDescriptorKind
    ( tagStack [ pred ( tagStackTop ) ] )
    then begin
      GetDescriptorPointer
        ( tagStack [ pred ( tagStackTop ) ] , dPointer )
        ; new ( tagNode )
    tagNode ^. tagNodeTag := tagStack [ tagStackTop ]
    tagNode ^. tagNodeLink := nil
    currentTagNode := dPointer ^ . typeDescrAttributeTagList
    if currentTagNode = nil
    then dPointer ^= typeDescrAttributeTagList := tagNode
else begin
    while currentTagNode <> tagNodeLink
    do currentTagNode := currentTagNode^.tagNodeLink
    ; currentTagNode^.tagNodeLink := tagNode
end
end
end

; procedure InitializeTagMTypeDotCurrentAttribute

; var dPointer : descriptorPointerTyp

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; if CheckTypeDescriptorKind ( tagStack [ tagStackTop ] )
  then begin
    GetDescriptorPointer ( tagStack [ tagStackTop ]
    , dPointer )
    ; if dPointer^.typeDescrAttributeTagList = nil
    then dPointer^.typeDescrCurrentAttributeTag := nil
    else dPointer^.typeDescrCurrentAttributeTag
    := dPointer^.typeDescrAttributeTagList
  end
end

; procedure AdvanceTagMTypeDotCurrentAttribute

; var dPointer : descriptorPointerTyp

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; if CheckTypeDescriptorKind ( tagStack [ tagStackTop ] )
  then begin
    GetDescriptorPointer ( tagStack [ tagStackTop ]
    , dPointer )
    ; if dPointer^.typeDescrCurrentAttributeTag
    = nil
    then dPointer^.typeDescrCurrentAttributeTag := nil
    else dPointer^.typeDescrCurrentAttributeTag
    := dPointer^.typeDescrCurrentAttributeTag
    . tagNodeLink
  end
end

; procedure PushTagWithTagMTypeDotCurrentAttributeTag

; var dPointer : descriptorPointerTyp

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; if CheckTypeDescriptorKind ( tagStack [ tagStackTop ] )
  then begin
    GetDescriptorPointer ( tagStack [ tagStackTop ]
    , dPointer )
; if dPointer = typeDescrCurrentAttributeTag <> nil then PushTag ( dPointer =
         . typeDescrCurrentAttributeTag )
end end

; procedure AppendTagSosMProfileDotParameterListWithTag
var dPointer : descriptorPointerType
; tagNode : tagNodePointerType
; currentTagNode : tagNodePointerType

; begin
VerifyTagHasTwoElements ( tagStackTop )
; if CheckProfileDescriptorKind ( tagStack [ pred ( tagStackTop ) ] )
then begin
GetDescriptorPointer ( tagStack [ pred ( tagStackTop ) ] , dPointer )
; new ( tagNode )
; tagNode = . tagNodeTag := tagStack [ tagStackTop ]
; tagNode = . tagNodeLink := nil
; currentTagNode := dPointer =
   . profileDescrParameterTagList
; if currentTagNode = nil then dPointer =
   . profileDescrParameterTagList := tagNode
else begin
   while currentTagNode = . tagNodeLink <> nil
do currentTagNode := currentTagNode = . tagNodeLink
   ; currentTagNode = . tagNodeLink := tagNode
end
end

; procedure InitializeTagMProfileDotCurrentParameter
var dPointer : descriptorPointerType

; begin
VerifyTagHasOneElement ( tagStackTop )
; if CheckProfileDescriptorKind ( tagStack [ tagStackTop ] )
then begin
GetDescriptorPointer ( tagStack [ tagStackTop ] , dPointer )
; if dPointer = . profileDescrParameterTagList = nil then dPointer =
   . profileDescrCurrentTagParameter := nil
else dPointer = . profileDescrCurrentTagParameter := dPointer =
   . profileDescrParameterTagList
end
end
; procedure AdvanceTagMProfileDotCurrentParameter

; var dPointer : descriptorPointerType

; begin

  VerifyTagHasOneElement ( tagStackTop )

  ; if CheckProfileDescriptorKind

   ( tagStack [ tagStackTop ] )

  then begin

    GetDescriptorPointer ( tagStack [ tagStackTop ]

                      , dPointer )

    ; if dPointer ^= . profileDescrCurrentTagParameter

     . tagNodeLink

      = nil

     then dPointer ^=

        . profileDescrCurrentTagParameter := nil

     else dPointer ^= . profileDescrCurrentTagParameter

      := dPointer ^= . profileDescrCurrentTagParameter

     . tagNodeLink

  end

end

; procedure PushTagWithTagMProfileDotCurrentParameterTag

; var dPointer : descriptorPointerType

; begin

  VerifyTagHasOneElement ( tagStackTop )

  ; if CheckProfileDescriptorKind

   ( tagstack [ tagStackTop ] )

  then begin

    GetDescriptorPointer ( tagStack [ tagStackTop ]

                      , dPointer )

    ; if dPointer ^= . profileDescrCurrentTagParameter <> nil

      then PushTag

      ( dPointer ^= . profileDescrCurrentTagParameter )

     else Abort

      ( ePushTagWithTagMProfileDotCurrentParameterTag )

  end

end

; procedure VerifyTagHasThreeElements ( tagStackTop : integer )

{ Verifies that the Tag Stack has at least three elements }

; begin

  if tagStackTop <= 2

      then Abort ( eVerifyTagHasThreeElements )

end

{ *** Count Stack *** }
; procedure PushCountWithSMDataDotLength

; begin
    VerifySHasOneElement ( sStackTop )
    ; if CheckDataDescriptorKind ( sStack [ sStackTop ] )
    then PushCount ( sStack [ sStackTop ] . dataDescrLength )
end

; procedure PushCountWithSSosMDataDotFixrep

; begin
    VerifySHasTwoElements ( sStackTop )
    ; if CheckDataDescriptorKind
        ( sStack [ pred ( sStackTop ) ] )
    then PushCount
        ( sStack [ pred ( sStackTop ) ] . dataDescrFixrep )
end

; procedure PushCountWithSSosMDataDotRep

; begin
    VerifySHasTwoElements ( sStackTop )
    ; if CheckDataDescriptorKind
        ( sStack [ pred ( sStackTop ) ] )
    then PushCount
        ( sStack [ pred ( sStackTop ) ] . dataDescrRep )
end

; procedure PushCountWithSMDataDotOutputPointer

; begin
    VerifySHasOneElement ( sStackTop )
    ; if CheckDataDescriptorKind ( sStack [ sStackTop ] )
    then PushCount
        ( sStack [ sStackTop ] . dataDescrOutputPointer )
end

; procedure PushCountWithSSosMDataDotOutputPointer

; begin
    VerifySHasTwoElements ( sStackTop )
    ; if CheckDataDescriptorKind
        ( sStack [ pred ( sStackTop ) ] )
    then PushCount
        ( sStack [ pred ( sStackTop ) ] . dataDescrOutputPointer )
end

; procedure PushCountWithTagMTypeDotAlignment

; var SCodeBuiltInType : SCodeBuiltInTypeTyp
    ; dPointer : descriptorPointerType
; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; if tagStack [ tagStackTop ] in SCodeBuiltInType
  then begin
    GetDescriptorPointer ( tagStack [ tagStackTop ]
      , dPointer )
    ; PushCount ( dPointer ^= typeDescrAlignment )
  end
  else Abort ( eVerifySCodeType )
end

; procedure VerifyCountHasThreeElements
  ( countStackTop : integer )
{
  Verify the count stack has at least three elements }

; begin
  if countStackTop <= 2
  then Abort ( eVerifyCountHasThreeElements )
end

; procedure VerifyCountHasFourElements
  ( countStackTop : integer )
{
  Verify the count stack has at least four elements }

; begin
  if countStackTop <= 3
  then Abort ( eVerifyCountHasFourElements )
end

; procedure SwapCount

; var temp : integer

; begin
  VerifyCountHasTwoElements ( countStackTop )
  ; temp := countStack [ countStackTop ]
  ; countStack [ countStackTop ]
  := countStack [ pred ( countStackTop ) ]
  ; countStack [ pred ( countStackTop ) ] := temp
end

; procedure RotateCount3tosToTos

; var temp : integer

; begin
  VerifyCountHasThreeElements ( countStackTop )
  ; temp := countStack [ countStackTop ]
  ; countStack [ countStackTop ]
  := countStack [ countStackTop - 2 ]
  ; countStack [ countStackTop - 2 ] := countStack [ countStackTop - 1 ]
; countStack [ pred ( countStackTop ) ] := temp
end

; procedure RotateCount4osToTos

; var temp : integer

; begin
  VerifyCountHasFourElements ( countStackTop )
  ; temp := countStack [ countStackTop - 3 ]
  ; countStack [ countStackTop - 3 ] := countStack [ countStackTop - 2 ]
  ; countStack [ countStackTop - 2 ] := countStack [ countStackTop - 1 ]
  ; countStack [ countStackTop - 1 ] := countStack [ countStackTop ]
  ; countStack [ countStackTop ] := temp
end

; procedure RotateCountTosTo3os

; var temp : integer

; begin
  VerifyCountHasThreeElements ( countStackTop )
  ; temp := countStack [ countStackTop ]
  ; countStack [ countStackTop ] := countStack [ countStackTop - 1 ]
  ; countStack [ countStackTop - 1 ] := countStack [ countStackTop - 2 ]
  ; countStack [ countStackTop - 2 ] := temp
end

; procedure RotateCountTosTo4os

; var temp : integer

; begin
  VerifyCountHasFourElements ( countStackTop )
  ; temp := countStack [ countStackTop ]
  ; countStack [ countStackTop ] := countStack [ countStackTop - 1 ]
  ; countStack [ countStackTop - 1 ] := countStack [ countStackTop - 2 ]
  ; countStack [ countStackTop - 2 ] := countStack [ countStackTop - 3 ]
  ; countStack [ countStackTop - 3 ] := temp
end

; procedure MaxCount

; begin
  VerifyCountHasTwoElements ( countStackTop )
  ; PopCount
; if countStack [ countStackTop ]
  >= countStack [ countStackTop - 1 ]
then PushCount ( countStack [ countStackTop ] )
else PushCount ( countStack [ countStackTop - 1 ] )
end

; procedure ComputeRangeSizeAndAlignmentOnCount
begin
  VerifyCountHasTwoElements ( countStackTop )
  ; if ( countStack [ pred ( countStackTop ) ] >= 0 )
  and ( countStack [ countStackTop ] <= 255 )
then begin
  countStack [ pred ( countStackTop ) ] := 1
  ; countStack [ countStackTop ] := 1
  end
else if ( countStack [ pred ( countStackTop ) ] >= -32768 )
  and ( countStack [ countStackTop ] <= 32767 )
then begin
  countStack [ pred ( countStackTop ) ] := 2
  ; countStack [ countStackTop ] := 2
  end
else begin
  countStack [ pred ( countStackTop ) ] := 4
  ; countStack [ countStackTop ] := 4
  end
end

{ *** Inside Routine *** }

; procedure TurnOnInsideRoutine
begin insideRoutineFlag := true end

; procedure TurnOffInsideRoutine
begin insideRoutineFlag := false end

; procedure VerifyInsideRoutineIsOff
{ Abort if the insideRoutine global variable is not off }
begin
  if NOT insideRoutineFlag
then Abort ( eVerifyInsideRoutineIsOff )
end

{ *** Tag Table *** }

; procedure CreateDeclaredTypeDescriptorAtTag
begin
  var dPointer : descriptorPointerType
begin
VerifyTagHasOneElement ( tagStackTop )

GetDescriptor ( declaredTypeDescriptorKind , dPointer )

with dPointer do begin
  typeDescrHasIndefRepField := false
  typeDescrPrefixTag := noTag
  typeDescrInfo := noRecordInfo
  typeDescrAttributeTagList := nil
end

; tagTable [ tagStack [ tagStackTop ] ]
  . tagEntryDescriptorPointer
:= dPointer
end

{ *** String Stack *** }

; procedure PushStringWithInputString

; begin ReadString end

; procedure PopString

; begin
VerifyStringPointerHasTwoElements
  ( stringPointerStackTop )

; stringPointerStackTop :=
  pred ( stringPointerStackTop )

; stringsSpaceStackTop
:= stringPointerStack [ stringPointerStackTop ]
end

; procedure SwapString

; var stringStack : array [ 1 .. 200 ] of char
; count , index , temp1 , temp2 , i : integer

; begin
VerifyStringPointerHasThreeElements
  ( stringPointerStackTop )

; count := 1

; index :=
  stringPointerStack [ stringPointerStackTop - 2 ] + 1

; for temp1 := index
  to stringPointerStack
  [ pred ( stringPointerStackTop ) ]
  do begin
    stringStack [ count ] := stringsSpaceStack [ temp1 ]

    ; count := count + 1
    end

  for temp2
  := ( stringPointerStack


[ pred ( stringPointerStackTop ) ]
+ 1
)
to stringPointerStack [ stringPointerStackTop ]
do begin
  stringSpaceStack [ index ] :=
  stringSpaceStack [ temp2 ]
; index := index + 1
end
; stringPointerStack [ pred ( stringPointerStackTop ) ]
:= index - 1
; for i := 1 to ( count - 1 )
do begin
  stringSpaceStack [ index ] := stringStack [ i ]
  ; index := index + 1
end
end

; procedure SkipInputString

; begin ReadString ; PopString end

; procedure VerifyStringEqualsInputString
( errorCode : integer )

; var length , index1 , index2 , count : integer
; match : boolean

; begin
  VerifyStringPointerHasTwoElements
  ( stringPointerStackTop )
  ; ReadString
  ; length
  := stringPointerStack [ stringPointerStackTop ]
  - stringPointerStack
  ; if ( stringPointerStack
  [ pred ( stringPointerStackTop ) ]
  - stringPointerStack
  [ stringPointerStackTop - 2 ]
  = length
  then begin
    index1
  := stringPointerStack
  [ pred ( stringPointerStackTop ) ]
  + 1
  ; index2
  := stringPointerStack
  [ stringPointerStackTop - 2 ]
  + 1
  ; match := true
  ; count := 1
  ; while ( count <= length ) and match
do begin
  if stringSpaceStack [ index1 ]
    = stringSpaceStack [ index2 ]
  then begin
    index1 := index1 + 1
    index2 := index2 + 1
    count := count + 1
  end
  else match := false
end

; if match
  then PopString
  else Abort ( errorCode )
end

else Abort ( errorCode )
end

; procedure VerifyStringPointerHasTwoElements
( stringPointerStackTop : integer )

{ Verify that string pointer stack has at least two elements }

; begin
  if stringPointerStackTop <= 1
  then Abort ( eVerifyStringPointerHasTwoElements )

; procedure VerifyStringPointerHasThreeElements
( stringPointerStackTop : integer )

{ Verify that string pointer stack has at least three elements }

; begin
  if stringPointerStackTop <= 2
  then Abort ( eVerifyStringPointerHasThreeElements )

{ *** Compiler Switch Array *** }

; procedure SetSwitchSubCountSosToCount

; begin
  VerifyCountHasTwoElements ( countStackTop )
  ; CompilerSwitches
  [ countStack [ pred ( countStackTop ) ] ]
  := countStack [ countStackTop ]
end

{ *** Miscellaneous Semantic Operation *** }

; procedure IdentifyLineNumber
begin
  VerifyCountHasElement ( countStackTop )
  inputLineNumber := countStack [ countStackTop ]
end

{ *** Input File Stack *** }

; procedure PushAndOpenInputFileWithStringVisible
; var filename : identifier
; i, count : integer

; begin
  VerifyStringPointerHasTwoElements
  ( stringPointerStackTop )
  if ( stringPointerStack [ stringPointerTop ]
      - stringPointerStack
      [ pred ( stringPointerStackTop ) ]
      + 2
    ) >= idlength
then begin
  count := 1
  for i := ( stringPointerStack
      [ pred ( stringPointerStackTop ) ]
      + 1
  )
to stringPointerStack [ stringPointerStackTop ]
do begin
    filename [ count ] := stringSpaceStack [ i ]
    ; count := count + 1
  end
end
else Abort ( eFilenameTooLong )
end

; procedure PushAndOpenInputFileWithStringTagList
; var filename : identifier
; i, count : integer

; begin
  VerifyStringPointerHasTwoElements
  ( stringPointerStackTop )
  if ( stringPointerStack [ stringPointerStackTop ]
      - stringPointerStack
      [ pred ( stringPointerStackTop ) ]
      + 2
    )
< idlength
then begin
  count := 1
  for i := ( stringPointerStack
      [ pred ( stringPointerStackTop ) ]
    + 1
  )
to stringPointerStack [ stringPointerStackTop ]
do begin
  filename [ count ] := stringSpaceStack [ i ]
  ; count := count + 1
  end
  ; filename [ count ] := ','
  ; filename [ count + 1 ] := 't'
  ; PushAndOpenInputFile ( filename )
end
else Abort ( eFilenameTooLong )
end

{ *** Output Copy File *** }

; procedure OpenOutputCopyFileWithStringVisible

; var filename : identifier
; i, count : integer

; begin
  VerifyStringPointerHasTwoElements
    ( stringPointerStackTop )
  ; if ( stringPointerStack [ stringPointerStackTop ]
    - stringPointerStack
      [ pred ( stringPointerStackTop ) ]
    + 2
  )
  <= idlength
then begin
  count := 1
  for i := ( stringPointerStack
    [ pred ( stringPointerStackTop ) ]
  + 1
  )
to stringPointerStack [ stringPointerStackTop ]
do begin
  filename [ count ] := stringSpaceStack [ i ]
  ; count := count + 1
  end
  ; filename [ count ] := '.'
  ; filename [ count + 1 ] := 'v'
  ; OpenOutputCopyFile ( filename )
end
else Abort ( eFilenameTooLong )
end

; procedure OpenOutputCopyFileWithStringTagList

; var filename : identifier
; i, count : integer

; begin
  VerifyStringPointerHasTwoElements
    ( stringPointerStackTop )
  ; if ( stringPointerStack [ stringPointerStackTop ]
    = stringPointerStack
    [ pred ( stringPointerStackTop ) ]
    + 2
    )
    <= idLength
  then begin
    count := 1
    ; for i
      := ( stringPointerStack
        [ pred ( stringPointerStackTop ) ]
        + 1
      )
    to stringPointerStack [ stringPointerStackTop ]
    do begin
      filename [ count ] := stringSpaceStack [ i ]
      ; count := count + 1
    end
    ; filename [ count ] := '.'
    ; filename [ count + 1 ] := 't'
    ; OpenOutputCopyFile ( filename )
  end
  else Abort ( eFilenameTooLong )
end

; procedure TurnOnOutputCopying

; begin outputCopyingFlag := true end

; procedure TurnOffOutputCopying

; begin outputCopyingFlag := false end

; procedure EmitStringToOutputCopyingFile

; var length, i : integer

; begin
  VerifyStringPointerHasTwoElements
    ( stringPointerStackTop )
  ; if outputCopyOpenFlag
    then begin
      length
    := stringPointerStack [ stringPointerStackTop ]
- stringPointerStack
  [ pred ( stringPointerStackTop ) ]
  ; writeByteToOutputCopyFile ( length div 256 )
  ; writeByteToOutputCopyFile ( length mod 256 )
  for i := ( stringPointerStack
  [ pred ( stringPointerStackTop ) ]
  + 1
  )
  to stringPointerStack [ stringPointerStackTop ]
  do writeByteToOutputCopyFile
  ( stringSpaceStack [ i ] )
end
end

{ ### Taglist Table ### }

; procedure MakeTagListTableEmpty
var i : integer

; begin
  for i := 1 to maxTagListDepth do
    tagListTable [ i ] := noTag
end

; procedure AddTagListEntryFromTagAndCount
var externalTag, tagLimit, tagBase
  , externalNumber : integer

; begin
  VerifyTagHasThreeElements ( tagStackTop )
  ; VerifyCountHasOneElement ( countStackTop )
  ; externalTag := tagStack [ tagStackTop ]
  ; tagLimit := tagStack [ pred ( tagStackTop ) ]
  ; tagBase := tagStack [ tagStackTop - 2 ]
  ; externalNumber := countStack [ countStackTop ]
  ; if ( tagBase + externalNumber ) <= tagLimit
    then tagListTable [ externalTag ] :=
      tagBase + externalNumber
    else Abort ( eAddTagListEntryFromTagAndCount )
end

; procedure TranslateTagUsingTagListTable

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; if tagListTable [ tagStack [ tagStackTop ] ] = noTag
    then Abort ( eTranslateTagUsingTagListTable )
  else tagStack [ tagStackTop ]
    := tagListTable [ tagStack [ tagStackTop ] ]
end
{ *** Update Stack *** }

; procedure PushUpdateStack ( tagValue : tagTyp )
  ( tagValue : tagTyp, oldTagTableEntry : tagEntryTyp )
  ; begin
    if updateStackTop = maxUpdateStackDepth
    then Abort ( eUpdateStackOverflow )
    else begin
      updateStackTop := succ ( updateStackTop )
      ; with updateStack [ updateStackTop ] do begin
        updateEntryTagValue := tagValue
        ; updateEntryOldTagTableEntry :=
          oldTagTableEntry
      end
    end
  end

; procedure VerifyUpdateHasOneElement
  ( updateStackTop : integer )
  ; begin
    if updateStackTop < 1
    then Abort ( eVerifyUpdateHasOneElement )
  end

; procedure PushUpdateWithAllDefinedLabelsAndUndefine

; var index : integer

; begin
  for index := 1 to maxTagCount
  do if ( tagTable [ index ] . tagEntryState = definedTagState )
       or ( tagTable [ index ] . tagEntryState = specifyTagState
       )
  then if tagTable [ index ] . tagEntryDescriptorPointer
       = labelDescrKind
  then begin
    PushUpdateStack ( index, tagTable [ index ] )
    ; tagTable [ index ] . tagEntryState
      := undefinedTagState
  end
 end

; procedure PushUpdateWithTagAndUndefine

; begin
  VerifyTagHasOneElement ( tagStackTop )
  ; PushUpdateStack ( tagStack [ tagStackTop ],
tagTable [ tagStack [ tagStackTop ] ]
; tagTable [ tagStack [ tagStackTop ] ] . tagEntryState
:= undefinedTagState
end

; procedure PopUpdateAndRestoreTag

; begin
VerifyUpdateHasOneElement ( updateStackTop )
; tagTable
[ updateStack [ updateStackTop ] . updateEntryTagValue ]
:= updateStack [ updateStackTop ]
. updateEntryOldTagTableEntry
; updateStackTop := pred ( updateStackTop )
end

; procedure PopAllUpdateAndRestoreTags

; var i : integer

; begin
VerifyUpdateHasOneElement ( updateStackTop )
; i := updateStackTop
; while i > 0 do
begin PopUpdateAndRestoreTag ; i := i - 1 end
end

Appendix C

C.1 Description of S-CODE syntax rules

The S-CODE syntax is described in BNF with the following additions:

-- Meta symbols are written in lower case without brackets, terminal symbols are underlined lower case, and upper case is used for predefined tags.

-- Alternative right hand sides for a production may be separated by ::= as well as |.

-- Productions may be annotated with comments enclosed in parentheses.

-- Part of a right hand side may be enclosed in angular brackets followed by one of the characters ?, #, or + with the following meanings:

   < symbol string >?   ( "symbol string" is optional; it may occur zero times or once. )

   < symbol string >#   ( "symbol string" may occur zero or more times at this point. )

   < symbol string >+   ( "symbol string" must occur one or more times at this point. )

-- Spaces and line breaks are used simply to separate various parts of a production; they have no other significance, in particular they will not occur in the S-program.

-- Particular instances of a meta-symbol may be given a prefix, separated from the symbol by a colon, e.g. body:tag. The prefix (body) has the sole purpose of identifying the meta-symbol (tag) in the accompanying description; it has no syntactical significance whatsoever.
C.2 The syntax of the S-CODE language.

(For each production is given reference to its definition)

S-program
::= program program_head:string
    program_body endprogram

program_body
::= interface_module
::= macro_definition_module
::= <module_definition>"#
::= main <program_element>"#

program_element
::= instruction
::= label_declaration
::= routine_profile ! routine_definition
::= skip_statement ! if_statement
::= protect_statement
::= goto_statement ! insert_statement
::= delete_statement

instruction
::= constant_declaration
::= record_descriptor ! routine_specification
::= stack_instruction ! assign_instruction
::= addressing_instruction ! protect_instruction
::= temp_control ! access_instruction
::= arithmetic_instruction ! convert_instruction
::= jump_instruction
::= if_instruction ! skip_instruction
::= segment_instruction ! call_instruction
::= area_initialization ! eval_instruction
::= info_setting ! macro_call

simple_type
::= BOOL ! CHAR
::= INT ! REAL ! LREAL ! SIZE
::= OADDR ! AADDR ! GADDR ! PADDR ! RADDR

type
::= structured_type ! simple_type

resolved_type
::= resolved_structure ! simple_type

quantity_descriptor
::= resolved_type < rep count: number >?
::= range < rep count: number >?
range
  ::=  INT range lower:number upper:number
  ::=  SINT

record_descriptor
  ::=  record record_tag:newtag <record_info>?
       <prefix_part>? common_part
       <alternate_part>? endrecord

record_info
  ::=  info "TYPE" | info "DYNAMIC"

prefix_part
  ::=  prefix resolved_structure

common_part
  ::=  <attribute_definition>?

alternate_part
  ::=  alt <attribute_definition>?

attribute_definition
  ::=  attr newtag quantity_descriptor

resolved_structure
  ::=  structured_type < fixrep count:ordinal >?

structured_type
  ::=  record_tag:tag

value
  ::=  boolean_value  !  character_value
  ::=  integer_value   !  size_value
  ::=  real_value      !  longreal_value
  ::=  attribute_address ! object_address
  ::=  general_address  !  program_address
  ::=  routine_address  !  record_value

repetition_value
  ::=  <boolean_value>+  !  text_value
  ::=  <character_value>+  !  size_value+
  ::=  <integer_value>+   !  <longreal_value>+
  ::=  <real_value>+      !  <object_address>+
  ::=  <attribute_address>+  !  <program_address>+
  ::=  <general_address>+  !  <record_value>+

text_value
  ::=  text long_string

boolean_value
  ::=  true  !  false
character_value
::= c-char byte

integer_value
::= c-int integer_literal:string

real_value
::= c-real real_literal:string

longreal_value
::= c-lreal real_literal:string

size_value
::= c-size type | nosize

attribute_address
::= (c-dot attribute:tag) # c-aaddr attribute:tag
::= anon

object_address
::= c-oaddr global_or_const:tag
::= onone

general_address
::= (c-dot attr:tag) # c-gaddr global_or_const:tag
::= gnone

program_address
::= c-paddr label:tag | nowhere

routine_address
::= c-raddr body:tag | nobody

record_value
::= c-record structured_type
    [attribute_value]* endrecord

attribute_value
::= attr attribute:tag type repetition_value

constant_declaration
::= constant_specification | constant_definition

constant_specification
::= constspec const:new tag quantity_descriptor

constant_definition
::= const const:spectag
    quantity_descriptor repetition_value

stack_instruction
::= push obj:tag | pushv obj:tag
::= pushc value | pushlen | dup | pop | empty
::= popall byte
assign_instruction ::= assign | update | rupdate

addressing_instruction ::= fetch | refer | resolved_type | deref
 ::= select | attribute:tag | selectv | attribute:tag
 ::= remote | attribute:tag | remotew | attribute:tag
 ::= index | indexv
 ::= inco | deco
 ::= dist | dsize | structured_type | locate

protect_statement ::= save <program_element># restore

temp_control ::= t-init | t-geto | t-seto

access_instruction ::= setobj | getobj
 ::= accessa | oindex:byte | attribute:tag
 ::= accessv | oindex:byte | attribute:tag

arithmetic_instruction ::= add | sub | mult | div | rem | neg
 ::= and | or | xor | imp | eqv | not
 ::= compare | relation


convert_instruction ::= convert | simple_type

label_declaration ::= label_specification | label_definition

label_specification ::= labelspec | label:newtag

label_definition ::= label | label:spectag

goto_statement ::= goto

jump_instruction ::= forward_jump | forward_destination
 ::= backward_jump | backward_destination

forward_jump ::= switch | switch:newtag | size:number
 ::= fjumpl | relation | destination:newindex
 ::= fjump | destination:newindex
forward_destination
  ::= sdest switch:tag which:number
  ::= fdest destination:index

backward_jump
  ::= bjump destination:index
  ::= bjumpl relation destination:index

backward_destination
  ::= bdest destination:newindex

skip_statement
  ::= skipif relation <program_element> endskip

skip_instruction
  ::= skipif relation <instruction> endskip

if_statement
  ::= if relation <program_element> else_part

else_part
  ::= else <program_element> endif
  ::= endif

if_instruction
  ::= if relation <instruction> i_else_part

i_else_part
  ::= else <instruction> endif
  ::= endif

segment_instruction
  ::= bseg <program_element> eseg

routine_profile
  ::= profile profile:newtag peculiar
  ::= <import_definition>
  ::= <export_or_exit>
  ::= endprofile

peculiar
  ::= known body:newtag kid:string
  ::= system body:newtag sid:string
  ::= external body:newtag nature:string xid:string
  ::= interface pid:string

import_definition
  ::= import parm:newtag quantity_descriptor

export_or_exit
  ::= export parm:newtag export_type
  ::= exit return:newtag
export_type
  ::= resolved_type | range

routine_specification
  ::= routine spec body: newtag profile: tag

routine_definition
  ::= routine body: spectag profile: tag
     <local_quantity> | <instruction> |
     endroutine

local_quantity
  ::= local var: newtag quantity_descriptor

call_instruction
  ::= connect_profile <parameter_eval> |
     connect_routine

connect_profile
  ::= precall profile: tag
  ::= asscall profile: tag
  ::= reproc n: byte profile: tag

connect_routine
  ::= call body: tag | <instruction> | call-tos

parameter_eval
  ::= <instruction> | asspar
  ::= <instruction> | assrep n: byte

module_definition
  ::= module module_id: string check_code: string
     visible existing
     body <program_element> | endmodule

visible existing
  ::= <visible> | tag_list | existing

visible
  ::= record_descriptor | routine_profile
  ::= routine_specification | label_specification
  ::= constant_specification | insert_statement
  ::= info_setting

tag_list
  ::= < tag internal: tag external: number > |

interface_module
  ::= global module module_id: string check_code: string
     <global_interface> | tag_list
     body < init global: tag type repetition_value > | endmodule
global INTERFACE
 ::= record_definition
 ::= constant_definition < system sid:string >?
 ::= global_definition < system sid:string >?
 ::= routine_profile
 ::= info_setting

global_definition
 ::= global internal:newtag quantity_descriptor

macro_definition_module
 ::= macro module module_id:string check_code:string
   <macro_definition>#: endmodule

macro_definition
 ::= macro macro_name:byte macro_parcount:byte
   known maco_id:string
   <macro_body_element>#: endmacro

macro_body_element
 ::= mark macro_sequence:string
 ::= mpar macro_parnumber:byte

macro_call
 ::= mcall macro_name:byte <actual_parameter:string>#

insert_statement
 ::= insert module_id:string check_code:string
   tagbase:newtag taglimit:newtag

area_initialization
 ::= zeroarea
 ::= initarea resolved_type
 ::= dinitarea structured_type

eval_instruction
 ::= eval

delete_statement
 ::= delete from:tag

info_setting
 ::= info string
 ::= line line:integer
 ::= setswitch switch:byte setting:byte

byte
 ::= an (8-bit) unsigned integer value in the range 0..255
number ::= a two byte value greater than or equal to zero. A number
(or any other multi-byte structure) will always be
transmitted with the most significant byte first. Let
the bytes be \(<B_1>B_2>\) in that sequence; the value will
be \(256 \times B_1 + B_2\).

ordinal ::= a number with value greater than zero.

tag ::= An ordinal (the "tag-value") associated with a descriptor.
     ::= The number zero followed by an ordinal (the "tag-value")
     and an identifying string.
     This second form is intended for debugging purposes, and
     is used to associate an identification to the tag.

newtag ::= A tag-value with no association.

spectag ::= A tag-value which, if not undefined, must have been given
an association in a specification (labelspec, constspec,
routinespec).

index ::= A byte within an implementation-defined range which
identifies an internal label. An important side effect
of its occurrence is that it looses its meaning. Thus
an internal label can be the destination of exactly one
jump-instruction.

newindex ::= An index-byte which is undefined; it becomes defined as
an internal label through its occurrence.

string ::= A byte with the value \(N\) followed by \(N\) "data bytes". The
character count \(N\) must be greater than zero, thus a
string cannot be empty (neither can a long_string, q.v.).

long_string ::= An ordinal with the value \(N\) followed by \(N\) "data bytes".
SEMANTIC MECHANISMS FOR A PORTABLE SIMULA SYSTEM

by

MIN-FENG CHANG

B.A., UNIVERSITY OF CHINESE CULTURE, TAIWAN, 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984
SIMULA is a programming language for simulation which extends the concepts of ALGOL 60 to ideas of the class and coroutine concepts, the associated reference variables, fully defined text handling, and input/output facilities.

In the summer of 1979, the Norwegian Computing Center established the portable SIMULA system (S-PORT) as a portable compiler system to implement the SIMULA language. The major interest in the S-PORT project is an essential reduction of the effort required for production of SIMULA systems for new machines.

The Perkin-Elmer SIMULA System is a project undertaken in the Computer Science Department of Kansas State University, under the direction of Dr. Rodney Bates, to adapt the portable SIMULA system to the Perkin-Elmer 32-bit machines. It is being developed to run under the UNIX operating system. To construct a complete Perkin-Elmer SIMULA System, we got the front-end compiler and run time support system from Norway and designed and coded our own back-end translator (S-Compiler) and environment interface support package.

To translate the compiled versions of the front-end compiler and run time system to the S-Compiler, a translator program called the interpass is needed. The major part of the project that I was involved in was the design and coding, part of the processing routines for the PASCAL portion of the interpass program.