PERKIN-ELMER SIMULA SYSTEM : INTERPASS SEMANTIC PROCESSING

by

HEMANGI DHOLAKIA

M.Sc. , Gujarat University, India, 1975

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

1983

Approved by:

Rodney M. Bates

Major Professor

ACKNOWLEDGEMENT

I would like to thank my advisor Dr. Rodney Bates for his valuable guidance and help in the preparation of this paper. I also want to thank Dr. David Gustafson and Dr. Virgil Wallentine for being on my committee. I appreciate the time and help offered by Howard Townsend.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 0

INTRODUCTION

In spring, 1983, the author was one of the participants of an implementation project. The objective of the project was to implement a portable SIMULA compiler to be used in the Perkin Elmer portable SIMULA system. The system is an adaptation of the portable SIMULA system (S-Port) developed by the Norwegian Computing Center (NCC) to the Perkin Elmer 32-bit machines.

The project work was divided into five broad components.

1) The modifications to Pass 6 ,Pass 8 and Pass 9 of Pascal/32 compiler.
2) The modifications to Pass 7 of Pascal/32 compiler.
3) The coding of an independent pass known as Interpass.
4) The design and coding of the Environment Interface Support Package.
5) Code development for the token stream dumpers for Pass 6, Pass 7, Pass 8 and Pass 9 of the Pascal/32 compiler.

The function of Interpass is to translate S-code into the input language for Pass 6. (S-code is a low language control language used by a front end compiler to translate the original SIMULA source program). Interpass includes a main controlling program coded in Syntax/Semantic (S/SL) language and a set of semantic operations coded in Pascal. The author's task was to interpret the underlying data structures of Interpass and to code the semantic procedures to be invoked by the S/SL program. Approximately 1300 lines of Interpass code (Semantic routines) were developed by author during the course of project participation.

This report

1) Provides a brief history of SIMULA language and introduces the S-port system
2) Discusses the important features of the low level language S-code
3) gives a brief description of Syntax/Semantic language
4) summarizes the passes one through nine of the Pascal/32 compiler on Perkin-Elmer machine
5) discusses the components of the Perkin Elmer SIMULA system
6) provides a a detailed description of the major data structures and semantic operations of Interpass and
7) discusses some of the interesting problems encountered in Interpass design.

The report concludes with a summary of the status of the project, the future modifications to be done in Interpass and the author's general impressions of the project on the whole.

CHAPTER 1

SIMULA AND S-PORT

The first part of this chapter briefly describes the history and important features of SIMULA language. The second part discusses the major components of the portable SIMULA system (S-Port).

1.1. SIMULA :

1.1.1. History of SIMULA67 :

SIMULA67 was developed at the Norwegian Computer Center (NCC) during the years 1965-1967 [1]. The language is in effect an extended version of the simulation language SIMULA I. The initiating ideas for SIMULA I can be traced back into late fifties but the major development phases of this language were carried out from 1961 to 1964. Initially, the design of SIMULA I was based on the 'network' concept. A system was described as a network of a fixed number of active components (events) with variable number of passive components acted upon by the active components. But as the language developed, the network concept was found to have shortcomings and the designers of SIMULA I adopted the 'process set' concept.

A system was now viewed as a variable collection of interacting processes. Each process was represented in the form of a stack. The idea of quasi-parallel execution; i.e., the control switching from one process to another by the action of special sequencing was also introduced. The concept was further modified by including process referencing using the pointer variables. At this point, the design was strongly influenced by two languages : ALGOL60 and SIMSCRIPT. The Dynamic block structuring of ALGOL60 and the pointer concept of SIMSCRIPT had been used in the SIMULA I design. The language was basically introduced as a simulation language. After a number of successful applications of SIMULA I to different jobs, in 1965, the designers of SIMULA I envisaged the possibility of modifying SIMULA I into a general purpose language.

Two major observations were made in relation to SIMULA I:

1) The remote attributes accessing machanism was too complex.
2) Several processes were found to be having common properties.

In order to resolve these problems, a new concept was introduced. Hoare's record class construct was included along with the idea of prefixing. Each process was viewed as associated with a link and the link-process pair considered to be a block instance. Further, a block instance was composed of a prefix layer and a main part. With the introduction of these concepts, SIMULA I was transformed into a general purpose language SIMULA 67.

1.1.2 Features of SIMULA :

SIMULA67 is designed to serve dual purposes :

1) It should be used elegently for defining large and complex simulation problems.

2) It should be capable of being used as an algorithmic language for a wide variety of less specialized problems.

If we want to deal with a very large problem, then it is neccessary to decompose the problem into components of mangeable size. Each component can then be understood individually one at a time. Also, In order to manipulate and relate the subcomponents of a large problem, powerful list processing capability is neccessary.

SIMULA67 exhibits the features which are essential for achieving the aforesaid objectives. As the history of the language reveals, ALGOL60 is the base of SIMULA. In ALGOL60, the decomposition is achieved through the 'block' concept. A block is a formal description of an aggregate data structure and the actions involved with this structure. Whenever a block is executed, an instance of block is created. This instance may be in the form of a dynamic memory area allocation for the variables local to that block. Further, several block instances may be generated at the same time. These block instances can interact simultaneously with one another.

In SIMULA, the block concept is modified through the introduction of an object and its class declaration. An object is an independent program segment (block instance) having its local data and actions . A specific data and action pattern is defined in the class declaration. All objects with similar patterns or formal descriptions tend to belong to the same class. A class may also be used as a prefix to another class declaration. The data and actions declared by the prefixed class are used along with the data and actions defined in the new class declaration. The actions defined in a particular class declaration can be executed sequentially by the object of that class. These actions can also be executed as a series of individual subsequences.

Besides decomposing the problem into smaller components, the class/ subclass concept gives sufficient flexibility to both the specialized and non specialized users in the usage of the language. Through the ideas of classes and prefixing SIMULA67 behaves as a powerful application language. A user may prefix his/her program with predefined classes without worrying about the detailed data structures associated with those classes. At the same time, a specialized user may define new classes in the program depending upon the complexity of his/her problem.

SIMULA contains a basic type 'reference' or pointer to identify and relate objects to each other. A reference is qualified by a class name. This means that the given reference can only point to the objects belonging to that class or belonging to the subclasses of that class. Components of data associated with different objects of the same class can be refered to by remote accessing using a dot notation. The reference (ref) type is used to provide list processing to sets of objects. It also provides reference security to the data belonging to a particular class by identifying it at compile time. The list processing facility is implemented in the language through the introduction of system defined class 'Simset'. This class is used as a prefix to the given classes to provide two way links to the set of objects. [2]

1.2. S-PORT :

S-Port is a portable implementation of the SIMULA language. It is developed by the Norwegian Computing Center (NCC). [3]

S-port consists of three components :

1.2.1. Portable front end compiler :

The portable front end compiler translates a source program written in SIMULA into an intermediate language S-code. The compiler itself is written in SIMULA.

1.2.2. Portable Run time system :

This system is used to support the Input/Output handling routines, editing and deediting tasks and mathematical functions library on the machine in question. It is used for the runtime environment manipulation of the machine.

The front end compiler and the run time system are distributed in S-code.

1.2.3. A machine dependent code generator :

In order to build a complete SIMULA system, S-port is to be supported by a back end compiler (S-compiler). The task of S-compiler is to translate the programs in S-code into the machine language of the given machine. It also must establish necessary links to the operating system of the object machine. The environment interface support package provides system dependent services to the S-Port through an environment interface. The interface is system independent and is organised according to the S-code standards.

The S-compiler translates the front end compiler and the run time system into object machine code. These translated systems along with the environment interface insert necessary links to the operating system of the object machine and the front end compiler is thus adapted to the object machine environment. Any SIMULA program can then be translated into S-code by the front end compiler. The translated version can be compiled by the S-compiler and linked with the machine code versions of the run time system and environment interface support package to produce the final object code of the original SIMULA program.

The semiportable SIMULA system provides the front end compiler and the runtime system package. In order to implement the SIMULA system on Perkin-Elmer machines, it is neccessary to develop the S-compiler for these machines and to provide an environment support package compatible with the operating system of these machines [3].

CHAPTER 2

S-CODE

The purpose of this chapter is to describe the important features of the low level language S-Code.[3] This is an intermediate language used as an interface between the front-end compiler and the code generator (S-compiler). The language is *non-interpretable* in the sense that it controls a compilation process, the result of which is an executable form of program.

S-code is a stack and descriptor oriented language. The program translated in S-code is vaguely in the form of Polish postfix notation. The syntax of S-code is described in BNF form. A few key terms used in the definition of S-code are first described in this chapter.

2.1 IMPORTANT DEFINITIONS :

2.1.1. Quantity :

Something which has a specific meaning and which can be manipulated at run time by the executing program.

2.1.2. Descriptor :

An abstraction used by the compiler to describe the properties of the quantities existing at run time. The format of the descriptors depends upon their implementation technique and the target machine. A descriptor defines the actions taken by the compiler when it recognises a specific program element. The action may be data manipulation or code generation for the target machine. There can be a tradeoff between the data manipulation and code generation at certain stages of compilation. But it is more or less decided by the S-compiler. The descriptors themselves donot exist at run time. They provide the information about the objects which can exist during the computation and are used in the compilation process for generating executable code.

S-code defines a stack to perform operations on the descriptors. It is a compile time data structure and is used to describe the effects of S-code instructions. The stack does not have any existance in the executable code obtained from compilation.

2.1.3. Tag :

A primitive syntax symbol used for identifying a descriptor. It exists as a two byte value (Number) which can be greater than or equal to zero. Tags are analogus to the identifiers in high level languages. A 'new tag' is tag value without any association to a quantity. A 'spec tag' is tag value which has been associated in a specification. The tag associated in a specification will later be defined through a declaration of the tag value with the same type.

```
e.g. constspec e REAL

           .

           .

        const e REAL c-real '2.303'
```

'e' is spec tag for constant of type real which is undefined but specified between the two statements. It becomes defined later through the token 'const' and obtains the constant value 2.303.

2.1.4. Index :

A number used for identifying internal labels. An index can be reused to define another label. At any instant, there is a one to one mapping between the index and the instruction. The index may be used to generate the internal labels for backward and forward jumps or to access a particular element in a repetitive quantity. A 'new index' is an index which is undefined. It becomes defined upon its occurance.

2.1.5. Indefinite Repetitions :

The attribute of a record may be defined as a repetitive field. When an attribute contains repetitions, a vector of identical elements is defined. The individual elements are accessed through indexing. A count of zero repetitions indicates that number of elements in the vector is indefinite. Indefinite repetitions are allowed in a record descriptors but only the last attribute or the last attribute in any alternate part may contain indefinite repetitions.

2.1.6. Object Unit :

A data storage unit of implementation defined fixed size. The size consists of integral number of machine addressable storage cells.

2.2 TYPES IN S-CODE :

Every data quantity in S-code belongs to some type. A type decides the internal structure of the quantities and the operations to be performed on them. The global variables, locals, constants, parameters etc. are defined using type. A unique descriptor is associated with each type. This descriptor cannot be used on the stack, therefore types cannot be used dynamically. S-code defines two major types :

1) Prefined types

2) Structured types

2.2.1. Predefined Types :

These are simple types. They are represented as predefined tags in S-code. Tags 0-10 are reserved for the simple types. Simple Types are analogus to the built-in types in high level languages. A simple type can be an arithmetic type, data address type or an instruction address type.

2.2.1.1. Arithmetic types:

Bool, Int, Char, Real, Lreal and Size are arithmetic simple types.

2.2.1.2. Data Address types :

Oaddr, Aaddr and Gaddr are the data address simple types.

2.2.1.3. Instruction Address types :

Paddr and Raddr are the instruction address predefined types.

The following three tables provide the description as well as the estimate of the value ranges of the predefined types.

2.2.2. Structured types :

A record descriptor describes a structured type through the use of a record tag. The generated type can be used in the definition of constants and global or local variables. The structured type can also be used as a prefix to another record or as an attribute type. If the record descriptor contains indefinite repetitions, then such a repetition must be resolved through the use of <u>fixrep</u> token. The token assigns a finite repetition value and therefore a finite number of elements to the associated record descriptor.

A structured type can contain an optional prefix part, common part and an optional alternate part. The prefix refers to a simple or structured type descriptor previously defined. The common part can contain a number of attributes in its body. The allocation order of these attributes and the physical order of the attribute in the record descriptor need not be the same. But, if an attribute includes indefinite repetitions then such an attribute must be declared as the last attribute in the list of attributes. (i.e. just before the end of the record). The record descriptor for the structured type may contain an alternate part too. The concept is analogus to the concept of variant records in Pascal.

Within a given part, the attributes can be packed by the S-compiler in any convenient way. But within a record

TYPE	DESCRIPTION	VALUES
Bool	Can have boolean values	True/False
Int	Can have integer values	Range is machine dependent : Usually half word or word representation
Real	Can have real values	Range is machine dependent : Usually similar to short reals representation on 32-bit machines
Lreal	Can take real values of greater precision	Range is machine dependent : Generally similar to the machine representation of reals on conventional architectures for 32-bit machines
Char	Can have 256 different values	Representation on 32-bit machines as a byte
Size	Describes the size of record or the distance between two machine addresses for two records	Range is machine dependent

TABLE 1. ARITHMETIC TYPES

TYPE	DESCRIPTION	VALUES
Oaddr	Describes the first address of a record	Represents true machine address
Aaddr	Defines the relative address of record component (attribute)	Represents the offset from the base Oaddr : not neccessarily a true machine address
Gaddr	Identifies a particular attribute of the record	Represents an Oaddr and Aaddr pair : Base offset may not be a pure machine address

TABLE 2. DATA ADDRESS TYPES

TYPE	DESCRIPTION	VALUES
Paddr	Defines the address of an instruction in the program	Represents a true machine address
Raddr	Identifies the address of the entry of a routine	Represents true machine address

TABLE 3. INSTRUCTION ADDRESS TYPES

the order of the parts - prefix..common..alternate must be preserved. The size of a structured type is the size of prefix + size of common part + size of the largest alternative. Here is an example of a structured type :

```
record A
   attr  B  INT
   attr  C  CHAR
end record
record  D
  attr  E  BOOL
  attr  F  A  rep 3
end record
record  G  prefix D
  attr H REAL
  attr I REAL rep 0
end record
```

2.3. GLOBAL VARIABLES :

In S-code, a global variable can be declared either through a global definition or a global specification. The declaration results in the static allocation of the global area. A global variable may be initialized by using an <u>init</u> token. If the variable is of structured type, all the components of the record are assigned constant values. An indefinite repetition, if present, is resolved during the initialization. But in general, an uninitialized variable is not allowed to contain indefinite repetition.

In some cases, it may be necessary to refer the variable before it is defined. A global specification is used to bind the tag to the yet undefined variable. Later in the program, the variable must be defined through a global definition.

A global variable is always assigned an integral number of object units. Therefore It can be addressed either by a general address or an object address.

2.4. TAGGED CONSTANTS:

The language can declare a tag refering to a constant area used to hold the value of a constant. A descriptor will be associated with this tag. If it is neccessary to refer to the tagged constant before its value is known then a constant specification is used to bind a tag to the constant of the given type. The constant may later be defined through a constant definition. A constant is also allocated an integral number of object units.

2.5. NAME SCOPE :

S-code allows two levels of scope for the variables or identifiers. The variables or tags declared within a routine have a specific meaning within the body of the routine. The tags lose their meaning at the end of the routine. They may be used with a different meaning in the other routines or at a different point in the program. The scope is confined to two levels because the routines in S-code are nonrecursive, and the static nesting of routines is not allowed.

2.6. ROUTINES:

In S-code, routines are analogus to the procedures in the high level languages. But they exhibit certain restrictive properties. The routines in this language are inherently nonrecursive. The paramaters can only be passed by value. A routine must return (exit) through its end. A return address is available to the routine through an exit definition. The definition identifies a descriptor for the area containing the return address of the routine. Therefore, a routine can change its return address.

A routine definition creates two descriptors, one relating to the profile and the other relating to the body of the routine. A profile defines the input parameter list (<u>import</u> parameters) and the exit descriptors for the routine and the body defines the local descriptors and the instruction sequence within the routine. An import definition declares a local quantity which will be later associated with the profile. The order of the parameters in the profile is important since this order defines the correspondence between the formal parameter locations and parameter value assignments. If an import parameter contains repetition then the call to the routine must contain the count of the maximum values to be transferred.

The <u>Peculiar</u> routines in S-code are of three kinds. A known peculiar routine has its body defined in S-code. The S-compiler can replace the body with another code sequence to achieve optimization. A system routine is used as a part of the interfacing routine set used to provide interface to the run time environment of the S-program. These system routines are not coded in S-code. Therefore, a system routine does not have a body. The calling and parameter passing mechanisms in case of such routines are system dependent. An external peculiar routine is the one which is written in a language other than S-code and its definition is implementation dependent.

2.7. STACK OPERATIONS :

In addition to the above mentioned features, S-code contains instructions relating to the operations on a stack. e.g.

2.7.1. Push :

It is used to push a copy of descriptor of associated with a global constant or local quantity.

2.7.2. Dup :

It is used to duplicate the descriptor on the top of the stack.

2.7.3. Rupdate :

It is used to transfer the value described by the descriptor on the second position of the stack to the location described by the top of the stack. S-code also provide addressing instructions to modify or refer to the areas described by the descriptors of the type GADDR, OADDR or AADDR. e.g.

2.7.4. Deref :

It is used to modify the top of the stack to describe the address of the given area.

The language specifies branches in the program through the use of jump instructions. Specific labels are created for the forward and backward jump destinations using indexes. S-code also provides instructions for handling dynamic quantities and for controlling the segementation in the program.

CHAPTER 3

SYNTAX/SEMANTIC LANGUAGE

This chapter serves as an introduction to Syntax/Semantic language. The important features are described in the following paragraphs. Syntax/Semantic language (S/SL) was developed at the University of Toronto mainly for implementing compilers.[4] The structure of a program written in S/SL resembles a textual notation for syntax diagrams with statements relating to the output actions and calls to the semantic routines inserted. The non determinancy associated with the syntax diagrams is resolved in this language by explicitely including selectors in the choice actions. An S/SL program behaves like a recursive descent parser with output actions and semantic routine calls included in it.[5]

S/SL is a very simple language. The structure of the the language exhibits only five basic features:

3.1. LANGUAGE FEATURES :

1) Input and output of tokens and matching of tokens
2) selection, repetition and sequencing of the statements
3) Rules or subprograms
4) Calls to semantic operations
5) Output of error signals

The semantic operations are written in some base language like Pascal. S/SL is essentially a control language. It does not have any data handling capabilties. Therefore, it does not feature variables or assignments. Data is manipulated only via the semantic operations. Because of the simplicity in structure, S/SL is highly suitable for writing compilers.

Each S/SL program consists of a list of declarations followed by executable subprograms or rules. A rule has three components: name, an optional return type and a list of actions followed by ';'. There are two kinds of rules -- Procedure rules and Choice rules. The Procedure is analogus to the procedures in Pascal and can be recursive. The Choice rule is analogus to the function in Pascal.

3.2. S/SL ACTIONS :

S/SL program generally exhibits eight specific kinds of actions.

3.2.1. Call Action:

This action is used to invoke a procedure rule.The rule name Preceded by the symbol @ signifies a call.

3.2.2. Return Action:

The action is used to indicate a return before reaching the end of the rule. The symbol >> signifies the return. In a Choice rule, the return action must include a value to be returned but in a Procedure rule the return value is not given. This action is generally not used in Procedure rules since a return is implicit at the end of a procedure rule.

3.2.3. Input Action:

The input action reads the next input and matches it against a specific input token. The action is indicated by the presence of an input token name in a rule.It is also called a Match action.

3.2.4. Emit action:

This action is used to output a token in the output stream. It is specified by a dot (.) followed by the token name.

3.2.5. Error action:

This action is used to output the error signal in the special error output stream. It is specified by a (#) followed by the error signal.

3.2.6. Cycle action:

This action is analogus to the loop structure in the high level languages like Pascal. The statements enclosed within '{' and '}' are executed until a return '>>' or one of the cycle exits is encountered. The cycle exit is specified by the symbol '>'.

3.2.7. Exit action:

This action is taken whenever a cycle exit is encountered. The control is transferred from the innermost cycle.

3.2.8. Choice action:

The effect of this action is analogus to a case statement in Pascal. The action is optionally headed by a selector and is followed by a list of labels and the associated actions. The last label may be a '*' which signifies 'otherwise'.

A choice action can be a rule choice, a semantic choice or an input choice. If the selector is the name of an S/SL choice rule then the choice action is a rule choice. If the selector is the name of a semantic operation (routine) then the choice action is considered to be a semantic choice. The rule choice calls a specified choice rule and tries to match the value returned by the rule to any of the labels. A semantic choice calls a semantic routine and tries to match the value returned by it against any of the labels. If the selector is '=' then the next input token is matched against one of the labels. This is called an input choice.

If none of the labels matches with the input token(in case of input choice) or the value returned(in case of a semantic choice) then the alternative following the '*' is taken. The symbol'*' is analogus to the otherwise clause in a Pascal case statement.

The Choice action controls the decision process. The input token or the value returned is only used as a determiner for the selection of a particular set of actions, it does not modify or manipulate any data.

The semantic routines called by the S/SL program are procedures written in a language like Pascal. They can be parameterized or nonparameterized. The parameters passed to the parameterized routines can be only constants, since S/SL does not contain any variables. The entire data handling process is carried out through the execution of these routines. S/SL adheres to a strict data abstraction by separating the algorithm from the data. The algorithm invokes the semantic routines where the data is manipulated.

All input tokens, output tokens, error signals and results returned by semantic procedures used by S/SL rules are defined in the form of sets of values. They may be defined by enumerating their members. Each member is an identifier. The enumeration resembles the enumeration type in Pascal.

3.3. S/SL IMPLEMENTATION :

The S/SL language is implemented in two phases. A translator translates the S/SL program into an intermediate language called S/SL-code. The is a machine- language-like instruction set encoded into sequence of numbers. The result is a Pascal array of integers. The translated S/SL program is then interpreted by an S/SL interpreter. It scans through the array by using a large case statement. Each instruction and semantic operation of S/SL-code has a counterpart label in the case statement. There are additional instructions for handling parameterized semantic operations. The actions of the S/SL-code(S/SL program) are carried out as the instructions are interpreted by the interpreter.

CHAPTER 4

PASCAL/32

This chapter describes the major functions performed by the Passes 1 through 5 and gives a detailed account of the code generation Passes 6 through 9 of the Pascal/32 compiler [5]. This compiler is a nine pass Pascal compiler for the Perkin-Elmer machine. It is an extensive modification of Hartmann's compilers for concurrent and sequential Pascal.[6]

4.1. LEXICAL ANALYSIS (Pass 1) :

The pass transforms the character input from the source program into a sequence of integers known as tokens. Each unique identifier is mapped to a specific spelling index. These spelling indices are then used by the later passes.

4.2. SYNTAX ANALYSIS (Pass 2) :

This pass verifies the syntax of the intermediate code generated by pass 1. The output of this pass is in the form of a polish postfix notation, the operand followed by operators.

4.3. NAME ANALYSIS (Pass 3) :

This pass establishes the scope of the identifiers. It maps each spelling index to a unique name index. Since the same identifier may be used with different meanings in various blocks, pass 3 distinguishes the usage of the identifier by establishing a naming scope.

4.4.DECLARATION ANALYSIS (Pass 4) :

The main function of this pass is semantic processing of the intermediate code generated by Pass 3. It allocates storage to the data by providing data addressing and data type information. The input to this pass is unique name indices which refer to the types, variables parameters or routines. The output refers to the objects by a four tuple consisting of mode, context, displacement and level.

4.5. OPERAND ANALYSIS (Pass 5) :

The pass performs operand type checking on all the operators in the program. It checks the compatibility of operands and their operators. The input to this pass refers to the objects by addressing mode, displacement, context and level. This is the intermediate code generated by pass 4. The output of this pass is a set of assembler instructions for a virtual stack machine.

4.6. PROGRAM LOGIC (Pass 6) :

There are two main functions performed by this pass.

1) Optimisation and rearrangement of the intermediate code supplied by Pass 5 .

2) Operations on the Constants (Constant Folding).

Pass 6 builds an in-core data structure for the procedures from the input stream and in the process carries out the optimisation and constant folding. The data structure is a stack of lists of trees. The fundamental element of the list is a record containing three pointers. The next pointer (NP) points to the next element of the list. The left and right pointers(LP-RP) point to the left and right substructures. The next pointer is used to link a list of statements. A head pointer points to the beginning and a tail pointer points to the end of the list.

In general, one element is created for each input operator. The data structure is built from the input using a stack. Each element of the stack is a list similar to the list to be created. The stack element stores the head and tail pointers. Three basic elements operations are performed on this stack.

1) Pushing a new element on the stack
2) Appending an element to the list
3) Branching one or two lists to the right and left substructures

Constant folding is done during the process of building the data structure. For example, if both operands of an operator are constants, represented by the right and left substructures, then the value of the expression is calculated at compile time. Both operands are popped from the stack and a PUSHCONST entry along with the calculated value is pushed on the stack.

Some other optimizations are also done for generating better code. In order to generate the intermediate code for pass 7, the data structure is traversed recursively in post order. During this traversal, some of the special optimisation cases are handled. For example, in a 'For' loop, if the initial value is greater than the final value, the loop is skipped entirely. Similarly, if the value of the expression in a 'Case' statement is known at compile time then the case statement is replaced by a jump to the particular label.

4.7. CODE GENERATION (Pass 7) :

The main function of the pass is to generate code for the 8/32 interdata machine. The input to this pass is a set of instructions for the virtual stack machine (generated by Pass 5) modified in structure by Pass 6. The output is a set of 8/32 instructions in unformatted form.

pass 7 allocates general, single precision and double precision floating point registers. It also generates symbolic references to code labels, procedure labels, statement labels, stack depths and literals. One of the main data structures of this pass is an attribute stack. The stack is in the form of a linked list of records. It simulates the contents of run time stack of virtual stack machine. The fundamental element of this stack is an attribute record. Two pointers are maintained on the stack. The TOP pointer points to the top most record and Second pointer points to the second top most record. An attribute record itself contains a backward link field, type of the object being described and a kind field describing the kind of the object.

Another data structure used by Pass 7 is a set of structures associated with register allocation. General, real and short real are the three register sets available to the user. The current state of each register is stored in an array and each register is either marked free or allocated.

Pass 7 does not distinguish between the instructions which perform similar functions but whose sizes are varying. The intermediate code generated by the pass contains all such instruction pairs.

4.8. CODE OPTIMISATION (pass 8) :

The main function of this pass is to make machine dependent optimizations. It makes instructions choices depending upon the length of the instruction. Pass 8 maps the code labels to the actual machine addresses by generating a table. It performs machine dependent optimisations using the peephole technique.

4.9. FINAL ASSEMBLY (pass 9) :

The main function of this pass is to convert the machine code into the format accepted by the linking loader. It uses the table supplied by Pass 8 to replace the labels with the addresses.

CHAPTER 5

PERKIN-ELMER SIMULA SYSTEM

The major objective of this chapter is to describe the components of Perkin-Elmer SIMULA System. This system is going to be implemented as a complete SIMULA system on perkin-Elmer machine . It is to be run under the Unix(Tm) operating system. S-Port or the Semi-Portable SIMULA system provides a portable front end compiler and a portable run time system. It also outlines the machine independent environment interface. In order to implement the system on the Perkin-Elmer machine, It is neccesary to provide a machine dependent code generater. The task involves the generation of an S-compiler (the code generator) and an environment interface support Package. The package is machine dependent. The S-compiler written for the Perkin Elmer Machine consists of two components.

5.1. S-COMPILER COMPONENTS:

1) An independent pass known as interpass
2) A set of modified code generator (passes 6 through 9) of the Pascal/32 compiler.

5.1.1. Interpass :

This pass translates the token stream of S-code into the intermediate language of a modified version of pass 6 of the Pascal/32 compiler. The input to this pass is a token stream of S-code. It is a file of integers in binary byte code form.

Interpass recognizes the operations indicated by the S-code and releases appropriate tokens for the input stream of modified pass 6. It also allocates data space for the constants and global or local variables within the respective areas. Unlike the initial passes of Pascal/32, interpass , in certain cases, does not release the tokens at the end of the output stream. Instead through the use of pointers the tokens are inserted in the middle of the output stream . The main controlling language of interpass is S/SL, but the operations associated with the manipulation of data are written in Pascal. The S/SL program invokes these semantic operations. The S/SL interpreter for the interpass is also written in Pascal.

5.1.2. Modifications to pass 6 through pass 9 :

5.1.2.1. Pass6:

1) New tokens must be included in the input token stream of this pass for the language features which are present in S-code but not in Pascal. E.g. S-code contains the terminal tokens update and rupdate for transferring the value described by one operand to the location designated by the second operand. Unmodified Pass 6 does not contain counterpart tokens for these two operations.

2) Pass 6 contains the mechanism for handling the true constants but not initialized variables which behave like constants. S-code has initial constant values

for the variables of the arithmetic type, data address type and instuction type. Pass 6 is therefore modified to include one more area which can hold the initialized global variables of S-code.

3) The design of Pass 6 supports a polish postfix notation and the basic data structure is a tree. Each result operand is used only once and so this structure is suitable. The dup token of S-code requires the duplication of its operand. It is not possible to represent this operation by a tree. Therefore, algorithms for generating graphs other than a tree are included in the design of Pass 6.

5.1.2.2. Pass 7:

This pass is designed to hold the descriptors of the runtime operands on the stack as long as they are not computed. The register containing the operand is freed after the computation of the result is complete. In order to process the tokens like dup in S-code, it is necessary to save the contents of the register or the temporary location after the result is computed. Pass 7 has thus to be modified to allow for the prevention of the automatic deallocation of the register or the temporary location.

Besides the dup token processing, there are various other cases which need multiple uses of the registers. E.g. for an efficient handling of dynamic quantities. Pass 7 is also modified to include operations to process the general (GADDR) type of S-code. The object address (OADDR) and attribute address (AADDR) pair of GADDR type is treated as a single item in S-code. But two full 32 bit words are required to hold these components. Existing Pass 7 generates code for all such large values, and the value not being held in single register, is manipulated in the memory.

Such an arrangement is considerably slow for addressing operations involving GADDR type. The Pass 7 is modified to treat the GADDR values as a pair of values on the stack of its token stream. Pass 6 and Pass 7 contain the facility for array bound checks and in case of lower bound, the substraction of a lower value other than zero. S-code assumes a lower bound of zero for the list of repetitive quantities. Also, bounds checking is not required in S-code. These passes are modified for the zero lower and no bounds checking.

5.2. ENVIRONMENT INTERFACE SUPPORT PACKAGE :

The package is used to facilitate the access to the UNIX operating system. Since the base language of the Unix operating system is 'C', the package is written in C. In particular, the package provides the system dependent interface for the environment in which the Portable SIMULA system is to be implemented. Since the runtime environment of the Pascal code generator is different from that of UNIX, the support package contains procedures for the neccessary environment switch between the Pascal and SIMULA environments and the C environment. In particular, this includes saving of stack pointers for both C stack and

Pascal stack and copying of the parameters. The package translates system independent environment functions into UNIX functions. In addition to this, UNIX supplied editing and deediting services are modified by introducing new functions in the package.

CHAPTER 6

DATA STRUCTURES AND SEMANTIC ROUTINES FOR INTERPASS

The author was one of the participants of the Perkin Elmer Simula system project. The Project work was devided into five broad components. Coding of interpass was one of the major components. Interpass includes a main controlling program coded in Syntax/semantic language (S/SL) and a set of semantic operations coded in Pascal. The author's task was to interpret the underlying data structures and to code the semantic procedures to be invoked by the S/SL program. The outline of the data structures and the semantic operations was provided by the project director, Dr. Rodney Bates. This chapter gives a detailed description of the major data structures and the semantic operations of Interpass. The role of these semantic operations in the design of interpass is also discussed.

6.1. DATA STRUCTURES :

6.1.1. S-stack :

This is a global stack which corresponds to the compile time stack defined in the S-code. It is a stack of pointers pointing to the descriptors. The stack is implemented as a Pascal array of pointers.

6.1.2. Tag-stack

The tag stack is a stack of tag values associated with the descriptors. It is implemented as an array of integers within a predefined subrange.

6.1.3. Count-stack :

The count stack is used to temporarily hold various integer values such as displacements, length etc during the compilation. Compile time arithmetic is done using the count stack. It is implemented as an array of integers.

6.1.4. Update stack :

This stack stores the global meanings of tags which are redefined with local meaning in a routine body. At body exit, the stack entries are used to restore the former meanings of the tags.

6.1.5. Reachable stack :

This is a global stack of boolean values. It is used to indicate whether the current point in S-code is reachable or not. Initially, it has only one entry with value false. The top is turned false by the tokens associated with backward jumps, forward jumps, unconditional gotos and switches. The top of the stack is turned true by backward destinations, forward destinations, switch destinations and the labels <u>endif</u> and <u>else.</u> The stack is pushed when statically nested procedures are entered and popped when they are exited.

6.1.6. Mark stack :

This is a stack of marks pointing to sections of S-stack. The entries of the mark stack store the subscripts to the S-stack. Each entry of the mark stack points to the element just below the bottom of the section. The bottom

element of this stack points to the bottom of the entire S-stack. The stack grows upwards (see chapter7 for the significance of sections).

6.1.7. Fragment stack :

This is a stack of pointers pointing to the roots of lists of fragment pointers. The fragment pointers point to the starting and ending locations of a fragment within the output file (see chapter 7 for the description of fragment pointers {file pointers } and output file).

6.1.8. Tag Table :

This table is an array of tag records. A tag record stores two kinds of information about a tag. One field of the record stores the tag state. A tag may be undefined, referenced or defined. The other field points to the descriptor associated with the given tag. The table is used to locate the descriptor when the tag naming it appears in the input. The table entries are modified at compile time when the tags are declared or undeclared.

6.1.9. Descriptor Kinds:

The descriptors contain information about the objects which exist during the compilation process. In this implementation, seven major descriptor kinds are defined. The descriptors corresponding to these descriptor kinds are implemented as a type of variant record with the record fields stating the information relevent to the specific kind.

6.1.9.1. DATA DESCRIPTOR KINDS :

Descriptors of this kind represent the information about the data objects. Such descriptors may be used to define the areas associated with the global variables, local variables and constants or to manipulate the parameters in a routine profile. The data descriptors can be of the following kinds :

1) Global descriptor kind
2) Local descriptor kind
3) Constant descriptor kind
4) Import descriptor kind
5) Export descriptor kind
6) Exit descriptor kind

A tag is associated with each descriptor which is one of the above kinds. A data descriptor may also contain information about the results obtained on account of runtime operations. Such a data descriptor belongs to the resultdescriptor kind. No tag names can be associated with the descriptors of this kind.

6.1.9.1.1. Datadescriptor Information :

6.1.9.1.1.1. Output pointer:

The pointer points the spot in the output file immediately following the token which produced the given data descriptor. The descriptor must be on the S-stack.

6.1.9.1.1.2. Mode:

The mode of the descriptor can be a VAL mode, REF mode or a VALADDRESSDESCRIPTOR mode. The third kind is neccessary to describe the data which is of val mode according to the S-code definition but is of a type which pass 6 of the Pascal compiler references with an address.

6.1.9.1.1.3. Type Tag:

This is a unique tag for the type associated with the given descriptor.

6.1.9.1.1.4. Displacement:

The descriptor holds the displacement of the global or local variables or constants with in their respective areas. The displacement has no meaning for descriptors of the resultdescriptor kind.

6.1.9.1.1.5. External ID:

If the data is external then the identification of the data is stored in the descriptor.

6.1.9.1.1.6. External procedure Label:

For the descriptor defining the external procedure, the information about the pass 6 procedure label is stored in the descriptor.

6.1.9.1.1.7. Identification:

The identification string of the tag associated with the data item is stored in this field .

6.1.9.1.1.8. SIMULA Line Number:

For the data descriptor of structured type, the Simula line number associated with the tag is stored in the descriptor.

6.1.9.1.1.9. Fix Repetition count:

If the data descriptor type includes an indefinite repetition, then the number of repetitions within this instance of the type is resolved and stored. The field has a meaningful value only if the has fix repetition field contains true.

6.1.9.1.1.10. Has Fix Repetitions:

The data descriptor contains the information about the presence or absence of a fix repetition value. This boolean field holds true if the fixed repetition value is present.

6.1.9.1.1.11. Repetition Count:

The number of repetitions in the data item type is stored in the descriptor.

6.1.9.1.1.12. Range Lower Bound:

The data item may be of a type which which is restricted to a certain range. The lower bound of this range is stored here. The field holds a meaningful value only data is restricted to a certain range.

6.1.9.1.1.13. Range Upper Bound:

For the range restricted data item, similar to the lower bound, the value of the upper bound is also stored.

6.1.9.1.1.14. Has Range:

This boolean field of descriptor denotes the presence of a range.

6.1.9.1.1.15. Has Pass 6 Counter Part:

A compile time stack of pass 6 holds a descriptor which corresponds to this data descriptor then the data descriptor has a Pass 6 counter part. This boolean field in the variant record will be a true if such a descriptor is present in Pass 6.

6.1.9.2. TYPE DESCRIPTOR KIND :

A descriptor is associated with every declared type and a built-in type. The following information is neccessary to describe a given type :

6.1.9.2.1. Typedescriptor Information :

6.1.9.2.1.1. TypeLength :

The length of the variables of type is stored in this field. In case of a structured type, if there are more than one alternatives present,the length of the largest alternate is stored. Also, if the type contains indefinite repetitions, the repetitions are assumed to be zero times.

6.1.9.2.1.2. Record alternative length :

For given type, alternate with indefinite repetition is chosen and the repetitions are assumed to occur zero times. The length of the alternate is stored in the record field. This length and the type length are synonymous if no alternates are present.

6.1.9.2.1.3. TypeAlignment :

The alignment value of a variable of the given type is stored. It can be 8,4,2 or 1 bytes for the types in S-code and Perkin-ELmer.

6.1.9.2.1.4. Prefix Tag:

If the record type has a prefix then the tag of the prefix is stored in the descriptor record field.

6.1.9.3. PROFILE DESCRIPTOR KIND :

The profile of a routine is used to define the import and export parameters as well as exit descriptors for the routine. A descriptor of this kind holds the following information :

6.1.9.3.1. Profiledescriptor Information :

6.1.9.3.1.1. Profile kind :

The profile may define a known routine , external routine, system routine, interface routine or ordinary routine. The routine kind is thus stored in the profile descriptor record field.

6.1.9.3.1.2. Body Tag :

For known, system and external profiles there is only one body and it is defined. The tag associated with the body is stored in this field.

6.1.9.3.1.3. Identification :

Peculiar routines associated with the profile are identified by a string. The string is therefore stored in the descriptor record.

6.1.9.3.1.4. Nature ID :

For external routines associated with the profile, the nature of the routine is specified in the form of a character string. This information is held in the descriptor record.

6.1.9.3.1.5. Export parameter :

If an export parameter is present in the profile then the tag associated with it is stored here. There canbe atmost one export parameter.

6.1.9.3.1.6. Exit parameter :

The tag associated with an exit parameter is stored, if any exit parameter is present. There canbe at most one exit parameter.

6.1.9.3.1.7. Activation Record Displacement :

The profile descriptor record field contains the displacement (one word) within the activitation record for the position of next parameter or next local variable. The value changes during compilation as local declarations are processed.

6.1.9.3.1.8. Import parameter List:

The root of the list of import parameter tags is stored in this field.

6.1.9.3.1.9. Current parameter:

The pointer to the current parameter in the import parameter is stored in the descriptor record.

6.1.9.4. BODY DESCRIPTOR KIND :

The routine body is used to compile a set of instructions and a descriptor is associated with the same. A body descriptor may hold the following information :

6.1.9.4.1. Bodydescriptor Information :

6.1.9.4.1.1. Pass 6 procedure Label:

The pass6 proc label for the sequence of instructions defining the routine body is stored here.

6.1.9.4.1.2. Activation Record Displacement:

The displacement within the activation record for the next local variable is stored in this field. The value changes during compilation.

6.1.9.4.1.3. Profile Tag:

The tag of the profile of the given routine body is stored here.

6.1.9.5. LABEL DESCRIPTOR KIND :

General labels in the S-code are used for the unconditional transfer of control within the S-program. A descriptor is associated with each label. The label descriptor contains the pass6 statement label to which the given S-code label will be translated.

6.1.9.6. SWITCH DESCRIPTOR KIND :

A switch in the S-program identifies a set of destinations and a jump may be made to the specific destination within this range. The switch descriptor holds the following information :

6.1.9.6.1. Switchdescriptor Information :

6.1.9.6.1.1. Minimum Pass6 Label:

The minimum (lowest numbered) pass 6 label in the pass6 case table which corresponds to the lower destination value is stored in this field.

6.1.9.6.1.2. Maximum Pass6 Label:

The maximum pass6 label in the pass6 case table is stored. This corresponds to the upper bound of the destinations.

6.1.9.7. ATTRIBUTE DESCRIPTOR KIND :

A descriptor is associated with every attribute of a record. A record attribute in S-code is analogus to the record field in Pascal. When a value of the attribute appears in the value of the surrounding record, a check is made to see if the attribute is already initialized or not. The descriptor contains a boolean field to hold this information.

6.1.10. Index Table :

The specific labels generated as a result of Jump instructions are not referenced by tags. Instead, they are accessed through the use of a kind of label called index. The index may be defined,referenced or undefined.Defined means the destination for the jump has occured during compilation but the jump itself has not. referenced means jump has occured but the destination is yet undefined. The undefined refers to a new index with no specific meaning.

An index table is a record with two fields :

6.1.10.1. Index state entry :

This stores the current state of the index.(undefined, referenced or defined)

6.1.10.2. Index Pass 6 label entry :

This field stores the pass6 label which the given index will be translated into. The index must be defined or referenced.

There is an Index table array with one element for each legal index. Each element of the array is an index table record. Initially, all elements have undefined index entry state.

6.1.11. Minimum Tag :

Upon entering a routine body, the destination indices and label tags locally acquire a new meaning. Minimum tag is a compile time global variable which is used to store the minimum tag defined locally within the body. At body exit, this value is used to undefine all the local tags.

6.1.12. Value area :

This is a compile time area used to store the values of long reals and record types. It is implemented as an array of bytes.

6.1.13. Constant, global and local areas :

These are runtime areas allocated at compile time. The displacement within each of the next available locations is stored. A three element array subscripted by area names is used to hold these displacements.

6.2. SEMANTIC OPERATIONS :

6.2.1. Semantic operations associated with the S-stack :

As the compilation process in Interpass progresses, the descriptors are continuously created and destroyed on the S-stack. The stack semantics for assignment and addressing instructions in the S-program require that certain conditions on the descriptors held in the stack are valid. For example, the correct interpretation of the assign instruction and therefore valid code generation is possible only if the second topmost descriptor on the stack is of ref mode. At runtime the value held in the topmost descriptor is to be transferred to the location designated by the second topmost descriptor. The addressing instructions modify or replace the top descriptor on the stack. This means a new result descriptor must occupy the top of the stack at the end of translation.

It is also neccessary to set the mode, type or outputpointer fields of the new result descriptor, if it is a data descriptor. Push and set operations are used to accomplish this. The emission of the right tokens in the output stream is possible only by examining the information (mode or type) in a descriptor record. Choice operations on the S-stack descriptors are needed for this purpose. Here are some of the possible operations which are used in the correct interpretations of stack, assignment, addressing and arithmetic instructions.

6.2.1.1. Verification operations :

1) Check the kind of descriptor which is on the top of the S-stack.
2) Check the mode of the top most or the second top most descriptor on the S-stack.
3) Verify that the types of the top most and the second top most descriptors are identical.
4) Verify the type of the top most or the second top most descriptor on the S-stack.

6.2.1.2.. Replacement operations :

1) Set the mode of the topmost descriptor to a given value.
2) Set the type of the topmost descriptor to the value obtained from the tag stack.
3) Remember the position in the output stream corresponding to the token for the descriptor on the

s-stack.(i.e. Set the output pointer in the token stream)

6.2.1.3. Push Operations :

1) Push a copy of the topmost descriptor on the S-stack.
2) Push a new descriptor.

6.2.1.4. Choice Operations :

1) Return the type of the descriptor which is on the top of the stack.
2) Return the mode of the topmost descriptor on the s-stack.
3) Return a boolean value indicating whether the topmost descriptor has a Pass6 counter part.
4) Return the kind of the topmost descriptor.

Here is an example of how the S/SL program will invoke some of the associated semantic routines:

<u>dist</u> :

This is a dyadic addressing S-code instruction. The topmost and the second topmost descriptors describe address values. Code is generated to compute the signed difference from second topmost descriptor to the top descriptor measured in object units. The two descriptors are replaced by a descriptor for this difference. The operations involved are :

1) If top descriptor is of REF mode then change the top of the stack to hold the contents of the area refered to by this descriptor.
2) Verify that the new descriptor on the top of the stack

is of type OADDR.

3) If the second topmost descriptor is of REF mode then change the second top of the stack to hold the contents of the area refered to by the descriptor.
4) Verify that the new descriptor is of type OADDR.
5) Emit a pass 6 sub token (for the subtraction between the two addresses) in the output token stream.
6) Pop the two topmost descriptors from the stack.
7) Push the descriptor associated with the result obtained.
8) Set the type of the descriptor to 'Size'.

S/SL will invoke the following semantic routines for interpreting the dist instruction.

@Force TOS value	[S/sl rule]
oVerifySMDataDotType(OADDR)	[Semantic routine]
@Force SOS value	[S/sl rule]
oVerifySSosMDataDotType(OADDR)	[Semantic routine]
oEmittoken(sub)	[Semantic routine]
oPopS	[Semantic routine]
opopS	[Semantic routine]
oPushSNewResultDescriptor	[Semantic routine]
oSetSMDataDotType (Size)	[Semantic routine]

The S/SL rule Force TOS value invokes the following semantic routines :

oChooseSDotMode

oChooseSMDataDotType

oSetSDotMode (Mode)

oEmitToken (Pass6 token)

oEmitTagMTypeDotP6Type

6.2.2. Semantic Operations associated with count stack :

During the compilation of an S-program, the count stack is used to temporarily hold the constant values which may appear in various program elements. In particular, the count stack may store :

1) Number of repetitions in a repetition field of a quantity of given type.
2) Number of fixed repetitions in the quantity of resolved type.
3) Size of an S-code type.
4) Size of the S-code structured type having an alternate with indefinite repetition.
5) Displacements with in the activation records for routine profile and routine body.
6) Lower and upper bound of a range.
7) Count of jump indices for the jump instructions.
8) Byte, number or string (converted) values for the char, int, real or long real types.
9) Parameter size of a profile.
10) Displacements within the constant, global and local areas.

A constant value is placed on the count stack through push operations. There are various semantic routines which push the appropriate constant value on the count stack.

While processing tokens whose interpretation involves these operations, it is some times neccessary to increment or decrement the constant value on the count stack. Compile time addition or multiplication of the top two constants on the stack is also needed in some translations. Therefore simple semantic routines for these operations are introduced. E.g. while proccessing an <u>sdest</u> instruction in the S-program, it is neccessary to locate the destination (within a specified range) where a jump is to be made. This involves the addition of lower bound of destination size to the number included in the argument of the instruction. The S/SL program at this point will invoke the appropriate semantic routine.

Constant values for the character, real types etc must be emitted to output stream following a Pass6 push constant token. Semantic routines exist for the emission of the constant values from the count stack.

The count stack always holds the values temporarily. Once the constant has been used, it is discarded through a pop operation. There is a semantic routine for this operation too.

6.2.3. Semantic Operations associated with tag stack and tag table :

Whenever a tag is read from the input stream, it is held temporarily in the tag stack. Since tags may be specified upon their definition, it is neccessary to verify the state of the tag whenever the tag is refered to. The tag value stored in the tag stack is used to create a new descriptor relating to that tag. The pointer to the descriptor will be stored in the tag table at the tag value.

As the arguments to a token are scanned further, the appropriate fields of the descriptor refering to the tag are set up. The setting operations may use the constant values being held in the count stack temporarily. During the Processing of S-code tokens like <u>profile</u> the arguments to the token contain more than one new tags. This requires the use of tags stored at a position one or two elements deep in the tag stack. The semantic operations are sometimes carried out on descriptors pointed to by the second topmost tag on the tag stack. While interpreting the type conversion instructions it is neccessary to examine the tag held in the tag stack in order to determine whether the conversion will be valid or not. Choice operations are needed to accomplish this. The choice operations are also neccessary to act as determiners in the S/SL selector statements when more than one option is available in the compilation process. In particular, during emission of constant values, the emitted tokens can differ depending upon the type (stored in the tag stack) of the constant.

6.2.3.1. Replacement or set operations :

There is a group of semantic operations which is used to set the fields of:

1) Data descriptor record

2) Profile descriptor record

3) Body descriptor record

4) Switch descriptor record

5) Label descriptor record

For the information held in the fields of these records, the reader is directed to refer to the section 6.1 .

6.2.3.2. Verification operations :

1) Verify the state of the tag on the top of the tag stack. The state can be undefined, defined or specified.
2) Verify that the tag refers to an S-code type.
3) Verify that an attribute is not initialized. The tag will be associated with an attribute descriptor.
4) Verify that all attributes in a record type are initialized.

6.2.3.3. Choice Operations :

1) Return the tag value associated with the top of the tag stack.
2) Return the tag state corresponding to the topmost tag.

6.2.4. Semantic Operations associated with the index stack :

The backward and forward jump instructions access the jump labels through indexing. When a jump instruction is encountered an index entry relating to the destination of the jump is changed. In particular, the state of the index will change depending on the kind of jump. The index will be undefined if the jump is forward jump. A set of pass 6 labels is associated with the destination indices. It is neccessary to allocate a new pass6 label every time the destination is defined or referenced. Further, once a jump has occured the corresponding index should be made available for reuse. But the Pass 6 labels can never be reused. A new index table is pushed when the compile time control enters a new scope level. Following Semantic operations operate on the index stack :

6.2.4.1. Push operation :

Push a new index table on the index stack

6.2.4.2. Allocation operation :

Allocate a new pass 6 label for the destination defined or referenced in the index stack.

6.2.4.3. Set operation :

Set the state of the index in the index table entry subscripted by the top value on the count stack.

6.2.4.4. Pop operation :

Pop the index table entry off the index stack.

6.2.5. Semantic operations associated with the emission of tokens :

When an output action is required to be taken in the S/SL program, the program invokes an emit routine. In Interpass, emit operations exist for the emission of Pass6 tokens, values, Pass6 statement labels, Pass 6 procedure labels and Pass6 general labels. There are also emit operations which are used to emit the mode, displacement, Type length or Pass 6 type.

6.2.6. Semantic operation associated with the Pass6 label stack :

The only operations needed are for pushing a new label, popping the existing label and swapping the top two labels.

6.2.7. Semantic Operations associated with reachable stack :

It is required to push, pop, set or return a boolean to indicate the reachability of the S-code. Appropriate operations exist for accomplishing this task.

6.2.8. Semantic Operations associated with the constant, global and local areas :

A Semantic operation is used to align the area displacement for a given type. There is also a semantic operation which increments the area displacement as the constant, local or global is placed in the respective area. The routines corresponding to these operations are used in the interpretation of constant and global specifications.

6.2.9. Semantic Operations associated with the Value area:

Pass 6 long constant values and address constants are accumulated in the value area. There are semantic routines which are used to store the constant values of long reals, reals, text strings, integers, characters and booleans in the value area. The displacement within the area is temporarily held on the count stack and is used for the placement of the constant value in the area. A packed value is stored for the address constants relating to the global, constant, body or local descriptors through the use of a semantic routine. The values are packed in a full word.

Pass 8 eliminates duplicate constants. The packed addresses are created to ensure the distinction among the constants when Pass8 processes them, because constants which may appear similar till the Pass8 compilation may differ when the actual addresses are assigned. Once the value area is filled, the contents of the area are emitted in the output stream. A semantic operation is used for such emission.

6.2.10. Semantic Operations associated with Segmentation and S-stack sections :

The reader is directed to refer to chapter 7 (Interesting problems), for the detailed description of these semantic operations.

6.2.11 Miscellenous Semantic operations :

Interpass includes a semantic routine which prints error messages and terminates the compilation process when ever error conditions arise during compilation. The front end compiler translates *SIMULA into S*-program. At this stage, all the errors relating to SIMULA are detected by the front end compiler. Normally, therefore, the S-program tobe compiled by Interpass should not contain errors. Detection of errors by Interpass indicates that S-program is incorrect. Immediate termination of the compilation process is therefore called for. There also exist semantic routines which operate on the global minimum tag variable.

See Appendix I and Appendix II for the data structure declarations and code for some of the semantic routines.

CHAPTER 7

INTERESTING PROBLEMS

Some of the interesting issues which arose during the design of interpass are discussed in this chapter. The solutions adopted in each case are also described.

7.1. SEGMENTATION AND NESTED PROCEDURES :

S-code allows for the existence of out of sequence code in the S-program. A segment instruction in S-code uses bseg and eseg tokens to enclose 'thunks' of code which will be located somewhere in the program and which cannot be generally scanned in strict sequential order. Further, the bseg and eseg tokens may be nested thus allowing for nested procedures. Pass 6 of the Pascal/32 compiler has no provision for handling such nested procedures. A direct handling of the nonsequential code can result in a waste of memory space. Therefore, the solution adopted was to use files for the treatment of code in the nonsequential order.

An output file called the main file is written physically in the order the tokens are produced. There is another file known as map file which allows Pass 6 to read the tokens of the main file in the correct logical order. Because of segmentation, the logical order of the tokens will be different from their physical order. The main file is divided into fragments. It is assumed that, within a fragment, the physical and the logical ordering of the tokens is the same. The map file contains a list of pointers to these fragments. The physical ordering of the pointers is kept the same as the logical ordering of fragments to which they point. Pass 6 will read the map file sequentially but this sequential scan will result in random access to the main file as and when required.

A segment stack is defined. It is a stack of segments which are entered but not exited yet. Each entry of the stack is the root of a list of pointers pointing to the fragments in the segment. The list itself is a doubly circular linked list of fragmentPointer nodes. Each node stores a pair of main file pointers and two pointer fields which are used as the right and left link fields for the linked list. The pair of file pointers consists of a start-of-fragment pointer and an end-of-fragment pointer. The start-of-fragment pointer points to the first word of the fragment and the end-of-fragment pointer points to the word beyond the last word of the fragment.

The fragments reside in the main file. The main file contains tokens which are not in order on the whole. But the tokens within a single fragment are internally in order. Further, the fragments have no inherent ordering, they all can be scattered within the main file.

The index or map file is list of fragment pointers in the order in which the fragments should be processed. Placing the fragment pointers in the map file and writing of the tokens in the main file is implemented in the following manner :

At the beginning of a segment, the fragments of this segment are linked in a list as long as the segment is not exited.

1) The segment stack is checked to see if it is empty or not.
2) If the stack is not empty then the last fragment pointer of the list corresponding to the top entry of the segment stack is 'closed'.
3) A new segment is pushed onto the segment stack. This segment contains (initially empty) fragment list.
4) The new list is set to grow by adding an empty fragmentpointer node and 'opening' it.

When the segment ends (eseg token is encountered), the pointers to its fragments are written to the map file. This is implemented in the following way :

1) The last fragment pointer of the list corresponding to the top of the segment stack is 'closed'.
2) The list of fragment pointers attached to the top entry of the segment stack is written to the map file.
3) After fragment pointers from the list are written to the file, the top segment stack entry is popped.
4) If the segment stack is not empty, then a new fragmentpointer node is added to list pointed by the new top of the segment stack and is 'opened'.
5) The last fragment pointer of top list is always open

because the unfinished fragment is currently in the process of being built. That means tokens are being accumulated in a serial order with in this growing fragment.

7.1.1. OPENING:

To open a fragment, the start-of-fragment pointer of the new fragment is set to the next main file position. The current token pointer is updated every time a new token is written. So the next token stream position is the spot pointed to by the current token pointer.

7.1.2. CLOSING :

To close a fragment, the current token pointer is stored in the end-of-fragment pointer of the open fragment list node. This marks the end of the current fragment.

It is to be noted that the start of the next fragment and the end of the current fragment point to the same spot in the main file.

7.2. FORCE VALUE OPERATIONS :

The descriptors on the S-stack can be of REF mode. A force value operation on such a descriptor requires the insertion of a pass6 pushindirect token at a place in the token stream just after the instruction token which generated this descriptor. The instruction token may be residing at a position other than the end of the token stream and the force value operations can be performed on a descriptor which is below the top of the S-stack. This requires inserting the Pushindirect token at a spot other

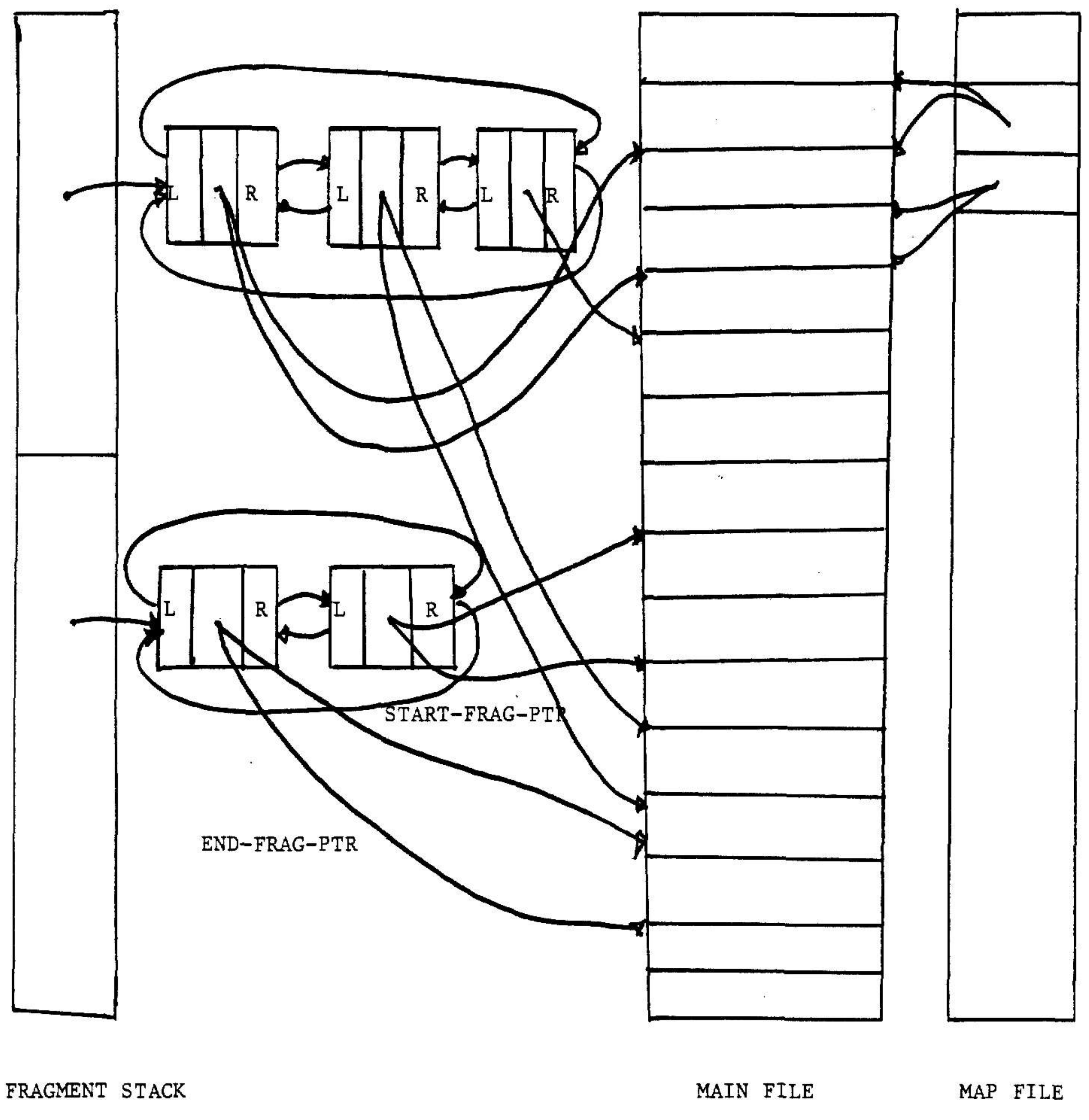

FIGURE 2. INTERPASS SEGMENTATION

than at the end of the token stream.

The solution adopted is to insert the token logically in the middle of the stream by splitting a fragment. The insertion mechanism is implemented as follows :

1) The descriptor record on the S-stack contains a field which stores the pointer to the spot in the output stream where the token which produced this descriptor is located.
2) The fragment containing this token is split into two fragments. This is done by first searching for that fragment through a scan of the fragment pointers in the list attached to the top entry of the segment stack.
3) The found fragment is split into two, and a new fragment is linked between the two pieces. The inserted fragment contains the pointers to the token tobe inserted. The token itself is written physically at the end of the main token file.

In this way, the logical ordering of tokens is arranged without disturbing the physical ordering of the emitted tokens in the output stream.

Some special cases may arise in the insertion process:

1) The token to be followed by Pushindirect may be found exactly at the end of the fragment. In such a case, a new fragmentpointer node is created and linked to the existing list.
2) If the token to be followed by Pushindirect lies at

the end of the token file then it belongs to the last open fragment and is therefore appended to that fragment.

7.3. CONSTANTS :

Another problem which arose during the interpass design was the treatment of constants. In Pass6, the constants which can reside in the memory are placed in the longconstant tokens. These tokens will later be used to construct the contents of the constant area. This is done in the order the longconstants actually occur. The displacements within the constant area are included in the tokens which operate on these constants.

In S-code, the constants can be specified through a constant specification. The displacements within the area must be determined at the time of specification in order to agree with the layout of the constant area. But no constant value is available at the time of specification. This requires inserting a token at the point of specification, when at a later stage in the program the constant is defined through a constant definition. A mechanism similar to the one used in handling force value operations is used here. The token is inserted after the specification token by using the insertion mechanism.

7.4. S-STACK SECTIONS :

The controlling structure in S-code for the If-statement, posed yet another interesting problem in the interpass design. Because of the control mechanism of

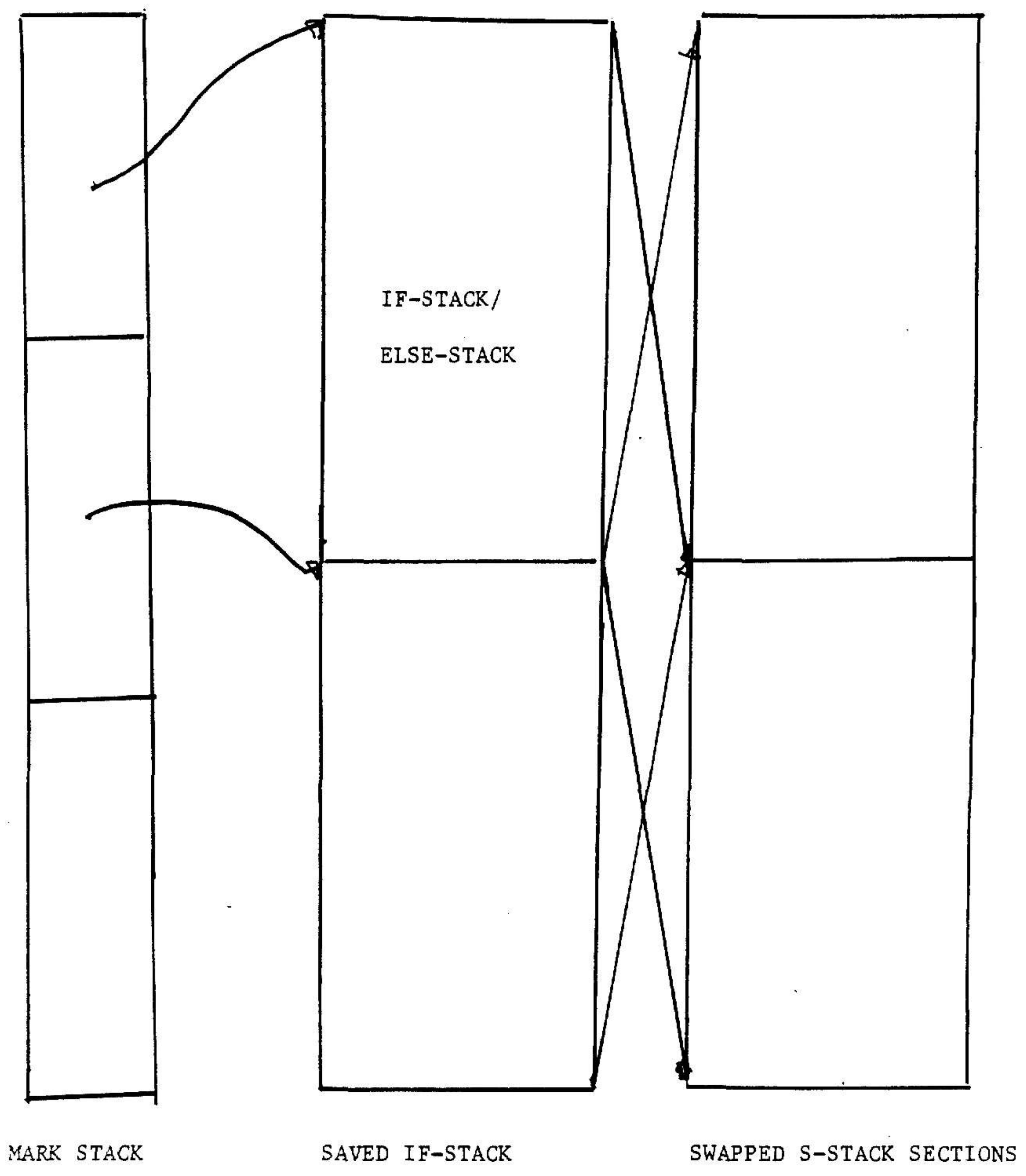

FIGURE 1. SWAPPING OF S-stack SECTIONS

S-code, it was neccessary to divide the S-stack into sections. It was also neccessary to identify the boundaries to the various sections. A mark stack was therefore added in the interpass data structure. The mark stack stores the subscripts (markers) to the S-stack and is used to divide the S-stack into sections. When the <u>if</u> token appears in the S-program, the relation following the if ,is checked by examining the top two entries of the S-stack. Code is generated which at runtime will transfer control to the else label if the relation is false. At this point, a copy of the complete state of the S-stack (and the associated descriptors) is saved as an IF stack.

This is implemented in the following manner :

1) The marker pointing to the top of the S-stack is pushed on the mark stack.
2) A copy of the entire stack from the marker below the one just pushed is pushed on the S-stack.

When the <u>else</u> token is encountered, a branch is generated to the endif label and a copy of the entire S-stack is saved as an ELSE stack. The stack itself is restored to the original state of the saved if-stack.

This is implemented as follows :

1) The topmost and the second topmost sections of the s-stack are swapped. The top most section is the section between the top of the s-stack down to the first marker and the second top most section is the section between the first and the second marker.

2) When an endif label is reached, the current stack and the saved stack are merged. The merge involves taking the corresponding descriptors and checking that they are identical. The descriptors can only differ in mode, in that case the VAL mode of the descriptor is changed to REF mode.

This is implemented as follows :

1) The number of elements in the sections of the s-stack in top and second top sections are counted.
2) If the descriptors differ only in the mode, the descriptor with the VAL mode is changed to ref by inserting a pass6 push indirect token at the appropriate place. The insertion mechanism is used here.
3) After the merge, the top section and the top mark are popped from the s-stack and the mark stack respectively.

The merge is neccessary to ensure that the state of s-stack is same after taking either of the branches in the if statement.

CHAPTER 8

CONCLUSIONS

As a result of the participation in Portable SIMULA project work, the author got an opportunity to study the components involved in the adaptation of semiportable systems. The author also received an exposure to the low level intermediate language S-code. The involvement with the task of coding Interpass resulted in a fascinating indepth study of the underlying data structures of Interpass.

It was observed that the selection of Syntax/Semantic language as the main control language was made because the language structure of S/SL is suited well for the control structure of S-code. It was also observed that the restricted size of the S-code control instruction set aided in keeping the S/SL and Pascal components of Interpass to a comparatively small size. This will result into a more efficient implementation of the SIMULA system.

8.1. PROJECT STATUS (Interpass) :

The coding work of Interpass which includes the Coding of S/SL program and coding of semantic routines in Pascal is about half complete. No testing is, however, yet done on the code sofar developed.

8.2. FUTURE WORK :

The design of Interpass, at present, does not include the treatment of initialised attributes of a record type in S-code. Future modifications to the design will include some additions to the related data structures. In particular, this will involve addition of extra fields in the record descriptor. Corresponding semantic operations will also be introduced and coded at a later stage.

Interpass design will also be modified to include the code for utility routines. The routines will involve the reading of S-code tokens from Interpass input stream, writing the tokens to Interpass output stream, allocating and freeing the descriptors during the compilation process and converting the text strings and input literals from S-code into internal form.

Finally, the coding of the S/SL interpreter remains to be done for interpretating the S/SL program along with the semantic routines which are called by it.

APPENDIX I

DECLARATIONS FOR SEMANTIC OPERATIONS

```
; const max_s_stack = 50
  ; max_count_stack = 50
  ; max_update stack = 50
  ; max_reachable_stack = 50
  ; max_tag_stack = 50
  ; max_tag_count = 100
  ; max_bound = 100
  ; max_p6_proc_label_count =100000
  ; max_sim_lin_num = maxint
  ; max_word = xxxx
  ; max_mark = 50
  ; maxareas = 3
  ; maxbytes = 255
          * ERROR MESSAGE CODES
  ; eCheckDataDescriptor = 1
  ; eVerifySDotMode = 2
  ; eVerifySSosDotMode = 3
  ; eVerifySTosSosMDataDotTypesMatch = 4
  ; eSStackOverflow = 5
  ; eSStackUnderflow = 6
  ; eVerifySMDataDotType = 7
  ; eVerifySSosMDataDotType = 8
  ; eVerifyTagDotState = 9
  ; eVerifyTagDotKindType = 10
  ; ePushTagWithSMdataDotType = 11
  ; ePushTagWithSSosMDataType = 12
  ; eChooseTagMTypeDotHasRepField = 13
  ; eSetTagSosTagMDataDotTypeToTag = 14
  ; eSetTagMDataDotDisplacementToCount = 15
  ; eSetTagSosMDataOutputPointer = 16
  ; eSetTagSosMdataDotHadFixRep = 17
  ; eSetTagSosMDataDotRepToCount = 18
  ; eSetTagSosMDataDotLowerBoundToCount = 19
  ; eSetTAgSosMDataDotUpperBoundToCount = 20
  ; eSetTagSosMDataDotHasRange = 21
  ; eVerifyCountValue = 22
  ; eCountStackOverflow = 23
  ; eCountStackUnderflow = 24
  ; eCheckSwitchDescriptor = 25
  ; eVerifyDataDescriptor = 26
  ; eVerifyTagDotKindType = 27
  ; eVerifyTagDotProfileKindType = 28
  ; eVerifyTagDotBodyKindType = 29
  ; eVerifIndexStateSubCountDotState = 30
  ; eVerifyAllIndexesUndefined = 31
  ; eIndexStackOverflow = 32
  ; eIndexStackUnderflow = 33
  ; eP6LabelStackOverflow = 34
  ; eP6LabelStackUnderflow = 35
; eReachableStackOverflow = 36
; eReachableStackUnderflow = 37
; eFragmentStackOverflow = 38
; eFragmentStackUnderflow = 39
; eMarkstackOverflow = 40
; eMarkstackUnderflow = 41
; eTagstackOverflow = 42
; eTagstackUnderflow = 43
; eNoTwoSections = 44
; eNotenoughelementsonSstack = 45
              Descriptor Kinds
; globalDescrKind = 0
; localDescrKind = 1
; constantDescrKind = 2
; importDescrKind = 3
; exportDescrKind = 4
; exitDescrKind = 5
; resultDescrKind = 6
; declaredTypeDescrKind = 7
; built_inTypeDescrKind = 8
; attributeDescrKind = 9
; labelDescrKind = 10
; swtichTableDescrKind = 11
; profileDescrKind = 12
; bodyDescrKind = 13
                    Descriptor Modes
; refmode = 0
; valmode = 1
; valAddrDescrmode = 2
               Tag entry state values
; undefinedtagState = 0
; specifiedTagState = 1
; definedTagState = 2
                Index entry state values
; undefinedIndexState = 0
; definedIndexState = 1
; referencedIndexState = 2
        Profile Descriptor Profile Kinds
; knownProfileKind = 0
; systemProfileKind = 1
; externalProfielKind = 2
; interfaceProfileKind = 3
; ordinaryProfileKind = 4
  ** Comparison values for Count TOS and SOS **
; Count_less_than = 0
; Count_greater_than = 1
; Count_Equal_to = 2
              * S-CODE  TYPES *
; Bool = 0
; Char = 1
; Int = 2
; Real = 3
; Lreal = 4
; Size = 5
```

```
  ; Oaddr = 6
  ; Aaddr = 7
  ; Gaddr = 8
  ; Raddr = 9
  ; Paddr = 10
  ; Recrd = 11
              *  Area  Names *
  ; GlobalArea = 1
  ; ConstantArea = 2
  ; LocalArea = 3

; type s_pointer = @ descriptorTyp
  ; tag_typ = 0 .. max_tag_count
  ; tag_table entry_typ = 0 .. max_tag_count
  ; tagstateTyp = undefinedTagState .. definedTagState
  ; ScodeTyp = Bool .. Recrd
  ; descriptorPointerTyp = @ descriptortyp
  ; datadescriptorTyp
    = globalDescrKind .. resultDescrKind
  ; DataDescrptorSetTyp = Set of datadescriptorTyp
  ; S_code_SetTyp = Set of ScodeTyp
  ; descriKindTyp = globalDescrKind .. bodyDescrKind
  ; outputPointertyp = 0 .. max_output_count
  ; descrModeTyp = refmode .. valAddrDescrmode
  ; displacementTyp = 0 .. max_displac_num
  ; areadisplacementTyp = 0 .. max_displac_num
  ; stringTyp = Packed array (. 1 .. max_char .) of char
  ; SIMULALinenumberTyp = 0 .. max_sim_lin_num
  ; SRepTyp = 0 .. max_rep_num
  ; boundTyp = 0 .. max_bound
  ; p6LabelTyp = 0 .. max_p6_label count
  ; p6procLabelTyp = 0 .. max_p6_proc_label count
  ; profileKindTyp
    = knownprofileKind
      .. ordinaryProfileKind indexstateTyp
      = undefinedIndexState ..  referencedIndexState
  ; filepointerTyp = 0 .. max_word
  ; fragmentlistNodepointerTYP = ^ fragmentListNode
  ; fragmntpointerTyp
    = Record
        fragmentStartpointer : filepointerTyp
      ; fragmentEndpointer : filepointerTyp
      end
  ; fragmntListNodeTyp
    = Record
        fragmentListLeftLink
        : fragmentlistNodepointerTyp
      ; fragmentListRightLink
        : fragmentlistNodepointerTyp
      ; fragmentListNodeFragmentpointer
        : fragmntpointerTyp
      end
  ; frag_pnterTyp = fragmentlistNodepointerTyp
  ; s_stackTyp
  = array (. 1 .. max_s_stack .) of s_pointer
; cnt_stackTyp
  = array (. 1 .. max_count_stack .) of integer
; reechable stackTyp
  = array (. 1 .. max_reach_count .) of boolean
; p6lbel_stackTyp
  = array (. 1 .. max_p6_label_count .) of integer
; Mrk_stackTyp = array (. 1 .. max_mark .) of integer
; fragmnt_stackTyp
  = array (. 1 .. max_fragments .) of frag_pnterTyp
; gl_con_loc_Typ
  = array (. 1 .. maxareas .) of areadisplacementTyp
; value_area Typ = array (. 1 .. maxbytes .) of Byte
; tag_tbleTyp
  = record
      tagentrytate : tagStateTyp
    ; tagentryDescriptorPointer : descriptorPointerTyp
    ;
    end
; tag_tabl arrayTyp
  = array (. 1 .. max_tag_count .) of tag_tbleTyp
    { the array of tags ,subscripted by tag values }
; updat_recTyp
  = record
      updateStackTagvalue : tagTyp
    ; updateStackoldTagTableEntry : tag_tbleTyp
    ;
    end
; updat_stackTyp
  = array (. 1 .. max_upd_count .) of updat_rec
; descriptrTyp
  = record
      case descrKind : descrKindTyp
      of globalDescrKind , localDescrKind
         , constantDescrKind , importDescrKind
         , exportDescrKind , exitDescrKInd
         , resultDescrKind
      : ( dataDescrOutputPointer : outputpointerTyp
        ; dataDescrMode : descrModeTyp
        ; dataDescrTypeTag : tagTyp
        ; dataDescrDisplacement : displacementTyp
        ; dataDescrExternalId : externalIdTyp
        ; dataDescrExternalProclabel : p6proclableTyp
        ; dataDescrId : stringTyp
        ; dataDescrLineNumber : SIMULALineNumberTyp
        ; dataDescrFixrep : SRepTyp
        ; dataDescrHasFixrep : boolean
        ; dataDescrLowerBound : boundTyp
        ; dataDescrUpperBound : boundTyp
        ; dataDescrHasrange : boolean
        ; dataDescrHasp6counterpart : boolean
        )
      ; attributeDescrKind
      : ( attributeDesrInitialized : boolean
              )
            ; labelDescrKind
            : ( labelDescrp6Stlab : integer )
            ; switchTableDescrKind
            : ( switchDescrLowerp6Label : p6LabelTyp
              ; switchDescrUpperp6Label : p6LabelTyp
              )
            ; declaredTypeDescrKind , built inTypeDescrKind
            : ( typeDescrLength : displacementTyp
              ; typeAlignment : displacementTyp
              ; typeDescrIndefAlterLength : displacementTyp
              ; typeDescrIndefrep : tagTyp
              ; typeDescrPrefixtag : tagTyp
              ; typeDescrHasRepField : boolean
              )
            ; profileDescrKind
            : ( profileDescrprofileKind : profileKindTyp
              ; profileDescrBodyTag : tagTyp
              ; profileDescrId : stringTyp
              ; profileDescrNature : stringTyp
              ; profileDescrExportParameter : tagTyp
              ; profileDescrExitparameter : tagTyp
              ; profileDescrparameterARDisplacement
                : displacemenTyp
              ; profileDescrImportparameterList : tagListTyp
              ; profileDescrCurrentparameter : tagListTyp
              )
            ; bodyDescrKind
            : ( bodyDescrp6lab : p6plabTyp
              ; bodyDescrARDisplacement : displacementTyp
              ; bodyDescrprofileTag : tagTyp
              )
          End { case descriptors }
      ; IndxTablentryTyp
        = record
            indexentrystate : indexStateTyp
          ; indexp6Label : p6LabelTyp
          end
      ; des_Pointer = descriptorPointerTyp

    ; var DataDescriptorSet : DataDescrptorSetTyp
      ; S_Code_Type Set : S_code SetTyp
      ; Descriptor : DescriptrTyp
      ; s_stack : s_stackTyp
      ; Tag_table : Tag_tabl arrayTyp
      ; count_stack : cnt_stackTyp
      ; reachable stack : reechable_stackTyp
      ; p6label_stack : p6lbel stackTyp
      ; Mark_stack : Mrk_stackTyp
      ; fragmentListNode : fragmntListNodeTyp
      ; frag_pointer : frag_pnterTyp
      ; fragment_stack : fragmnt_stackTyp
      ; tag_table rec : tag_tbleTyp
      ; updat_rec : updat_recTyp
```

```
; update_stack : updat_stackTyp
; IndexTablentry : IndxTablentryTyp
; glob_const_loc_array : gl con_loc_TYp
; val_area_array : value area_TYp
; s_stack_top , count_stack_top : integer
; reachable_stack_top , P6label stack_top : integer
; fragment_stack_top , mark_stack_top : integer
; tag_stack_top : integer
; current_output_pointer : integer
```

APPENDIX II

```
            {************************************

               Semantic Operations Associated With
               S Stack ::

               S_Stack :: A stack of pointers pointing to
                          the Descriptors
            ***********************************************}
    Procedure oPushSWithTagMDataDescriptor
    {Push a copy of the descriptor associated with tag top
     of stack(tos) onto the S-stack. The tag must be defined
     and must be of data descriptor kind }

    ; var pointer : DescriptorPointerTyp

    ; begin
        if Tag_stack (. tag_stack_top .) .
tagentryDescriptorPointer ^
             . Descrkind
           IN DataDescriptor Set
              - (. importDescrKind , exportDescrKind ,
exitDescrKind
                , resultDescrKind .)
              + (. ParameterDescrKind , ProfileDescrKind .)
        then if Tag_stack (. tag_stack_top .) .
tagentrystate
                = specifiedtagstate
          then begin
            GETDESCRIPTOR ( Pointer )
          ; Pointer ^
            := Tag_stack (. tag_stack_top .)
                 . tagentryDescriptorPointer ^
          ; oPushS ( Pointer )
          ;
          end
      ;
      end

  ; Procedure oSetSMDataDotMode ( DescriptorMode )
    { Set the mode field of the top descriptor on
      S-stack to the parameter value               }


    ; begin
        if S Stack (. s_stack_top .) ^ . DescrKind
           IN DataDescriptor_Set
        then begin
          S_Stack (. s_stack_top .) ^ . dataDescrMode
          := DescriptorMode
        ;
        end
      end

  ; Procedure oSetSMDataDotHasP6Counterpart
```

```
    { Set the has pass 6 counterpart field of the
      top descriptor on the S-stack to true      }


    ; begin
        if S Stack (. s_stack_top .) ^ . DescrKind
           IN Datadescriptor_Set
        then S Stack (. s_stack_top .) ^ .
dataDescrHasP6Counterpart
          := true
      end

  ; Procedure oSetSMDataDotOutputPointer
    { Sets the output pointer field to the top
      descriptor on S-stack to the current spot
      in the output stream .                  }


    ; begin
        If S Stack (. s_stack_top .) ^ . DescrKind
           IN Datadescriptor_Set
        then S Stack (. s_stack_top .) ^ .
dataDescrOutputPointer
          := current_output_pointer
      end

  ; Procedure oSetSDotDescriptorKind ( descriptorKind )
    { Sets the descriptor kind field of descriptor
      on top of the S-stack to the parameter value  }


    ; begin
        S_Stack (. s_stack_top .) ^ . DescrKind :=
descriptorKind
      end

  ; Procedure oPushSWithDuplicate
    { Duplicate the copy of descriptor on top
      of S-stack and push it on the S-stack  }


    ; var param : descriptorPointerTyp

    ; begin
        GETDESCRIPTOR ( param )
      ; param ^ := S_Stack (. s_stack_top .) ^
      ; oPushS ( param )
      end

  ; Procedure oChooseEmpty
    { Returns false (zero) if S-stack not empty
      otherwise true (One)                     }
    ; begin
        if s_stack_top = 0
        then HANDLE_CHOICE ( ord ( false ) )
        else HANDLE_CHOICE ( ord ( true ) )
      end

  ; Procedure oChooseSMDataDotType
    { Returns the type field of the descriptor
      which is on the top of the S-stack.    }

    ; begin
        if S Stack (. s_stack_top .) ^ . descrKind
           IN DataDescriptor_Set
        then HANDLE CHOICE
             ( S_Stack (. s_stack_top .) ^ . dataDescrTypeTag
)
        else oAbort ( eCheckDataDescriptor )
      end

  ; Procedure oChooseSMDataDotMode
    { Returns the mode field of the descriptor
      which is on the top of the S-stack    }


    ; begin
        if S Stack (. s_stack_top .) IN DataDescriptor Set
        then HANDLE CHOICE
             ( S_Stack (. s_stack_top .) ^ . dataDescrMode )
        else oAbort ( eCheckdataDescriptor )
      end

  ; Procedure oChooseSDotHasP6Counterpart
    { Returns the Has pass 6 counterpart field
      of top descriptor on the S-stack        }

    ; begin
        if S Stack (. s_stack_top .) ^ . descrKind
           IN DataDescriptor_Set
        then HANDLE CHOICE
             ( ord
                 ( S_Stack (. s_stack_top .) ^
                     . dataDescrHasP6Counterpart
                 )
             )
        else oAbort ( eCheckdataDescriptor )
      end

  ; Procedure oVerifySDotMode ( DescriptorMode )
    { Aborts if the mode of the descriptor on the
      top of S-stack is not equal to the parameter }

    ; begin
        if S_Stack (. s_stack_top .) ^ . descrKind
           IN DataDescriptor_Set
        then begin
          if S Stack (. s_stack_top .) ^ . dataDescrMode
             <> DescriptorMode
          then oAbort ( eVerifySDotMode )
        end
        else oAbort ( eCheckdataDescriptor )
      end

  ; Procedure oVerifySSosDotMode ( DescriptorMode )
    { Aborts if the mode of the second topmost
      descriptor on S-stack is not equal to the
      parameter.                                    }

    ; begin
        if S Stack (. pred ( s_stack_top ) .) ^ . descrKind
           IN DataDescriptor_Set
        then begin
          if S Stack (. s_stack_top .) ^ . dataDescrMode
             <> DescriptorMode
          then oAbort ( eVerifySSosDotMode )
        end
        else oAbort ( eCheckdataDescriptor )
      end

  ; Procedure oVerifySTosSosMdataDotTypesMatch
    { Aborts if the types of the topmost and second
      topmost descriptors on the S-stack are unequal. }


    ; begin
        if ( ( S_Stack (. s_stack_top .) ^ . descrKind
               IN DataDescriptor_Set
             )
             AND ( S_Stack (. pred ( s_stack_top ) .) ^ .
descrKind
                   IN DataDescriptor_Set
                 )
           )
        then begin
          if S Stack (. s_stack_top .) ^ . dataDescrTypeTag
             <> S_Stack (. pred ( s_stack_top ) .) ^
                  . dataDescrTypeTag
          then oAbort ( eVerifySTosSosMdataDotTypesMatch )
        end
        else oAbort ( eCheckdatadescriptor )
      end

  ; Procedure oPopS
    { pops the top entry from the S-stack}

    ; begin
        if s_stack_top = 0
        then oAbort ( eSStackUnderflow )
```

```
          else begin
            FREEDESCRIPTOR ( S Stack (. s_stack_top .)
          ; s_stack_top := pred ( s_stack_top )
          end
        end

    ; procedure oPushS ( descrPointer : descriptorPointerTyp )
      { Pushes a (Pointer to) descriptor on the S-stack}

      ; begin
          if succ ( s_stack_top ) > max_s_stack
          then oAbort ( eSStackOverflow )
          else begin
            s_stack_top := succ ( s_stack_top )
          ; S_Stack (. s_stack_top .) := descrPointer
          end
        end

    ; Procedure oChooseSDotDescriptorKind
      { Returns the descriptor kind of the topmost
        descriptor of the S-stack.                }


      ; begin
          HANDLE_CHOICE ( S_Stack (. s_stack_top .) ^ .
descrKind )
        end

    ; Procedure oVerifySMDataDotType ( SCodeType )
      { Aborts if the type of the top descriptor
        of S-stack does not belong to the S-code type}


      ; begin
          if S Stack (. s_stack_top .) ^ . descrKind
             IN DataDescriptor_Set
          then begin
            if S Stack (. s_stack_top .) ^ . dataDescrTypeTag
               <> ScodeType
            then oAbort ( oVerifySMDataDotType )
          end
          else oAbort ( eCheckDataDescriptor )
        end

    ; Procedure oVerifySSosMDataDotType ( SCodeType )
      { Aborts if the type of the second topmost
        descriptor of the S-stack is not an S-code type. }


      ; begin
          If S Stack (. pred ( s_stack_top ) .) ^ . descrKind
             IN DataDescriptor_Set
          then begin
            if S Stack (. pred ( s_stack_top ) .) ^ .
```

```
dataDescrTypeTag
             <> SCodeType
          then oAbort ( oVerifySSosMDataDotype )
        end
        else oAbort ( eCheckDataDescriptor )
      end

   ; Procedure oChooseSMDataDotHasFixrep
     { Returns the fix repetition value of the
       topmost descriptor of the S-stack.    }


     ; begin
        if S Stack (. s_stack_top .) ^ . descrKind
           IN DataDescriptor_Set
        then begin
          HANDLE CHOICE
            ( ord ( S_Stack (. s_stack_top .) ^ .
dataDescrHasFixrep )
        end
        else oAbort ( eCheckDataDescriptor )
      end

   ; procedure oChooseSMDataDotHasRange
     { Returns the Has range field value of the
       topmost descriptor of the S-stack       }


     ; begin
        if S Stack (. s_stack_top .) ^ . descrKind
           IN DataDescriptor_Set
        then HANDLE CHOICE
            ( ord ( S_Stack (. s_stack_top .) ^ .
dataDescrHasrange )
            )
        else oAbort ( eCheckDataDescriptor )
      end

   ; Procedure oPushSNewResultDescriptor
     { Pushes a new descriptor on S-stack }

     ; var d_Pointer : des_Pointer

     ; begin
        GET_DESCRIPTOR ( d_Pointer )
      ; s_stack_top := succ ( s_stack_top )
      ; s_stack (. s_stack_top .) := d_Pointer
      ; with s_stack (. s_stack_top .) ^
        do begin
          DescrKind := ResultDescrKind
        ; DataDescriHasP6counterpart := true
        end { with }
      end
```

```
    ; Procedure oSwapSCountDepthIntoS
      { Swap topmost descriptor with the one located
        at a distance indicated by the topmost value
        of count stack deep in the S-stack.        }


      ; var temp : des_Pointer

      ; begin
          temp := s_stack (. count_stack (. count_stack_top .)
.)
        ; s_stack (. count_stack (. count_stack_top .) .)
          := s_stack (. s_stack_top .)
        ; s_stack (. s_stack_top .) := temp
        end

    ; Procedure oSReverseTopCountElements
      { Reverse the top certain number of elements
        on the S-stack. The number is stored on the
        top of the count stack.                  }


      ; var temp : des_Pointer
        ; topelements , I : integer

      ; begin
          topelements := count_stack (. count_stack_top .)
          if s_stack_top < topelements
          then oAbort ( eNotenoughelementsonSstack )
          else begin
            for I := 1 to ( topelements div 2 )
            do begin
              temp := s_stack (. I .)
            ; s_stack (. I .) := s_stack (. topelements + 1 -
I .)
            ; s_stack (. topelements + 1 - I .) := temp
            end { for }
          ;
          end { else }
        ;
        end

    ; Procedure oSetSMDataDotTypeToTag
      { Set the type field of the topmost descriptor
        to the top element value on the tag stack. }


      ; begin
          If S Stack (. s_stack_top .) ^ . descrKind
             IN DataDescriptor_Set
          then S Stack (. s_stack_top .) ^ . dataDescrTypeTag
            := Tag_stack (. tag_stack_top .)
          else oAbort ( eCheckDataDescriptor )
        end
```

```
                {***********************************

                  Semantic Operations Associated with
                  TAG_STACK ::

                  Tag_stack :: A semantic stack of tag values

                *******************************************}

   ; Procedure oGetDescriptorPointer
       ( Table_Pointer : Integer ; var d_Pntr : des_Pointer )
     { Gets the pointer to the descriptor associated
       with a given tag                               }


     ; var table Entry : tag_tabl_Typ

     ; begin
         table entry := tag_table (. Table Pointer .)
       ; d_Pntr := table_entry . tagentryDescriptorPointer
       end

   ; Function oCheckDataDescriptorKind ( pntr : Integer ) :
Boolean
      { Checks if the descriptor associated with the tag
          is a data descriptor or not                   }


     ; begin
         If tag_table (. Pntr .) . TagentryDescriptorPointer
^

              . DescrKind
            IN DataDescriptor_Set
         then oCheckDataDescriptorKind := true
       end

   ; procedure oPushTagWithInputTag
     { Reads a tag from the input and pushes it
       on the tag stack                        }

     ; var tag : tagTyp

     ; Begin READ_NUMBER ( tag ) ; oPushTag ( tag ) end

   ; procedure oVerifyTagDotState ( tagState )
     { Abort if tag tos unequal to parameter }


     ; Begin
         If Tag_table (. Tag_stack (. tag_stack_top .) .)
              . TagEntryState
            <> tagState
         then oAbort ( everifyTagDotState )
```

```
        ;
        end

    ; Procedure oSetTagDotState ( tagState )
      { Set the state tag tos to the parameter value }


      ; Begin
          Tag_table
            (. Tag_stack (. tag_stack_top .) . TagEntryState =
tagState
        end

    ; Procedure oChooseTagDotState
      { Return the state of tag tos       }


      ; var State_param : TagStateTyp

      ; begin
          State_param
          := Tag_table (. Tag_stack (. tag_stack_top .) .)
               . TagEntryState
        ; HANDLE CHOICE ( ord ( State_Param ) )
        end

    ; Procedure oChooseTag
      { Return the tag tos }


      ; begin HANDLE CHOICE ( Tag_stack (. tag_stack_top .) )
end

    ; Procedure oVerifyTagDotKindIsType
      { Abort if tag tos does not name
        an S-code type                }


      ; var DescriptorPointer : des_Pointer

      ; begin
          oGetDescriptorPointer
            ( Tag_stack (. tag_stack_top .) ,
Descriptorpointer )
        ; If DescriptorPointer ^ . DescrKind NOT IN
S_Code_Type_Set
          then oAbort ( everifyTagDotKindType )
        end

    ; Procedure oPushTagWithSMDataDotType
      { S tos must be a data kind descriptor.
        push the s-code type field on the tag stack }
    ; begin
        if S stack (. s_stack_top .) ^ . DescrKind
           IN DataDescrKind_set
        then oPushtag
            ( S_stack (. s_stack_top .) ^ . dataDescrTypeTag
)
        else oAbort ( epushTagWithSMDataDotType )
      end

  ; Procedure oPushTagWithSSosMDataType
    { S Sos must be a data kind descriptor.
      Push the S-code type field of this descriptor
      onto the tag stack                         }


    ; begin
        if S stack (. pred ( s_stack_top ) .) ^ . DescrKind
           IN DataDescrKind_set
        then oPushtag
            ( S_stack (. pred ( s_stack_top ) .) ^ .
dataDescrTypeTag
            )
        else oAbort ( epushTagWithSSosMDataType )
      ;end
  ; Procedure oChooseTagMTypeDotHasRepField
    { Return boolean indicating whether the
      type defined by Tag tos has a rep
      field.  Tag tos must name an S-code Type }

    ; var DescriptorPointer : des_Pointer

    ; begin
        if oCheckScodeTypeDescrKind ( Tag_stack (.
tag_stack_top .) )
        then begin
          oGetDescriptorPointer
            ( Tag_stack (. tag_stack_top .) ,
Descriptorpointer )
        ; HANDLE CHOICE
            ( ord ( Descriptorpointer ^ . typeDescrHasrep )
)
        end
        else oAbort ( eChooseTagMTypeDotHasRepField )
      end

  ; Procedure oPushTag ( tg : tagTyp )
    { Pushes a tag on the tag stack   }


    ; begin
        If tag_stack_top > max_tag_stack
        then oAbort ( etagStackOverflow )
        else begin
          tag_stack_top := succ ( tag_stack_top )
        ; Tag_stack (. tag_stack_top .) := tg
        end
      end

  ; procedure opopTag
    { Pop the tag stack    }


    ; begin
        If tag_stack_top > 0
        then tag_stack_top := pred ( tag_stack_top )
        else oAbort ( etagstackunderflow )
      end

  ; Procedure oSetTagSosMDataDotTypeToTag
    { Set the type field of the descriptor
      named by the Tag Sos to Tag Tos, Tag
      Sos must name a data descriptor    }

    ; var d_pointer : des_pointer

    ; begin
        If oCheckDatadescriptorKind
             ( Tag_stack (. pred ( tag_stack_top ) .) )
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. pred ( tag_stack_top ) .) ,
d_pointer )
        ; d_pointer ^ . dataDescrTypeTag
          := Tag_stack (. tag_stack_top .)
        end
        else oAbort ( oSetTagSosMDataDotTypeToTag )
      end

  ; procedure oSetTagMDataDotDisplacementToCount
    { Set the displacement field of the descriptor named
      by the top most tag to the current displcacement
      obtained from the top of the count stack, the top
      of tag stack must name a datadescriptor         }


    ; var d_pointer : des_pointer

    ; begin
        if oCheckDatadescriptorKind ( Tag_stack (.
tag_stack_top .) )
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. tag_stack_top .) , d_pointer )
        ; d_pointer ^ . dataDescrDisplacement
          := Count_stack (. count_stack_top )
        end
        else oAbort ( oSetTagMDataDotDisplacementToCount )
      end
  ; Procedure oSetTagSosMDataOutputpointer
    { Set the output pointer field of the
      descriptor named by the top tag to the
      current output point                }

    ; var d_pointer : des_pointer

    ; begin
        if oCheckdatadescriptorKind
             ( Tag_stack (. pred ( tag_stack_top ) .) )
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. pred ( tag_stack_top ) .) ,
d_pointer )
        ; d_pointer ^ . dataDescrOutputpointer :=
Current_output_point
        end
        else oAbort ( eSetTagSosMDataOutputPointer )
      end

  ; Procedure oSetTagSosMDataDotFixrepToCount
    { Set the fix repetition field of the data
      descriptor named by the second most tag
      to a value taken from the top of the count
      stack                                     }

    ; var d_pointer : des_pointer

    ; begin
        if oCheckdatadescriptorKind
             ( Tag_stack (. pred ( tag_stack_top ) .) )
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. pred ( tag_stack_top ) .) ,
d_pointer )
        ; d_pointer ^ . dataDescrFixrep
          := Count_stack (. count_stack_top .)
        end
      ;
    procedure oSetTagSosMDataDotHasFixrep
    { Set the has fix repetition field of the
      data descriptor named by the second topmost
      tag to true                             }

    ; var d_pointer : des_pointer

    ; begin
        if oCheckdatadescriptorKind
             ( Tag_stack (. pred ( tag_stack_top .) )
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. pred ( tag_stack_top ) .) ,
d_pointer )
```

```
          ; d_pointer ^ . dataDescrHasFixrep := true
          end
          else oAbort ( eSetTagSosMDataDotHAsFixrep )
        end

    ; procedure oSetTagSosMDataDotRepToCount
      { Set the repetition field of the data descriptor
        to the value obtained from the top of the count
        stack                                          }

      ; var d_pointer : des_pointer

      ; begin
          if oCheckdatadescriptorKind
               ( Tag_stack (. pred ( tag_stack_top ) .) )
          then begin
            oGetDescriptorpointer
              ( Tag_stack (. pred ( tag_stack_top ) .) ,
d_pointer )
          ; d_pointer ^ . dataDescrRep
            := Count_stack ( count_stack_top )
          end
          else oAbort ( eSetTagSosMDataDotRepToCount )
        end

    ; procedure oSetTagSosMDataDotLowerBoundToCount
      { Set the lower bound field of the data descriptor
        named by the second topmost tag to the value
        obtained from the top of the count stack     }

      ; var d_pointer : des_pointer

      ; begin
          if oCheckdatadescriptorKind
               ( Tag_stack (. pred ( tag_stack_top ) .) )
          then begin
            oGetDescriptorpointer
              ( Tag_stack (. pred ( tag_stack_top ) .) ,
d_pointer )
          ; d_pointer ^ . dataDescrLowerBound
            := Count_stack ( count_stack_top )
          end
          else oAbort ( eSetTagSosMDataLowerBoundToCount )
        end

    ; procedure oSetTagSosMDataDotUpperBoundToCount
      { Set the upper bound field of the data
        descriptor named by the second topmost tag
        to the value obtained from the top of the
        count stack                               }

      ; var d_pointer : des_pointer

      ; begin
```

```
        if oCheckdatadescriptorKind
             ( Tag_stack (. pred ( tag_stack_top ) .) )
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. pred ( tag_stack_top ) .) ,
d_pointer )
        ; d_pointer ^ . dataDescrUpperBound
          := Count_stack ( count_stack_top )
        end
        else oAbort ( eSetTagSosMDataDotUpperBoundToCount )
      end

  ; Procedure oSetTagSosMDataDotHasRange
    { set the has range field of the data
      descriptor named by the second topmost
      tag to true                          }

    ; var d_pointer : des_pointer

    ; begin
        if oCheckdatadescriptorKind
             ( Tag_stack (. pred ( tag_stack_top ) .) )
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. pred ( tag_stack_top ) .) ,
d_pointer )
        ; d_pointer ^ . dataDescrHasrange := true
        end
        else oAbort ( eSetTagSosMDataDotHasRange )
      end

        {******************************************
         Semantic operations associated with COUNT STACK

         COUNT STACK
              : Stack of Integers


         ******************************************}

  ; procedure oVerifycountValue ( value )

  { Abort if the count tos is not
    equal to the value of the parameter  }

    ; begin
        if count_stack (. count_stack_top .) <> value
        then oAbort ( eVerifycountValue )
      end

  ; procedure oChooseCountValue

  { returns the count tos value }
```

```
    ; begin HANDLE ChOICE ( Count_stack (. count_stack_top
.) ) end

  ; procedure oChooseSosValue

{ return the tag sos value     }

    ; begin
        HANDLE CHOICE ( Count_stack (. pred (
Count_stack_top ) .) )
      end

  ; procedure oIncrementCount

{ Add one to count tos        }

    ; begin
        Count_stack (. count_stack_top .)
        := Count_stack (. count_stack_top )  + 1
      end

  ; procedure oDecrementCount

{ subtract one from count tos             }

    ; begin
        count_stack (. count_stack_top .)
        := count_stack (. count_stack_top .) - 1
      end

  ; procedure oAddCount

 { replace Count tos and Count sos by their sum     }

    ; begin
        count_stack (. pred ( count_stack_top ) .)
        := count_stack (. count_stack_top .)
           + count_stack (. pred ( count_stack_top ) .)
      ; pop_count
      end

  ; procedure oMultiplyCount

{ replace count tos and sos by their product        }

    ; begin
        count_stack (. pred ( count_stack_top ) .)
        := count_stack (. count_stack_top .)
           * count_stack (. pred ( count_stack_top ) .)
      ; pop_count
      end

  ; procedure oChooseCountComparison
```

```
{ compare and pop count tos and count sos.
  Return one of the values - lessthan,
  equalto or greaterthan                      }

    ; begin
        If count_stack (. count_stack_top .)
           < count_stack (. pred ( count_stack_top ) .)
        then HANDLE CHOICE ( Count_Less_Than )
        else If count_stack (. count_stack_top .)
                > count_stack (. pred ( count_stack_top ) .)
          then HANDLE CHOICE ( count_Greater_Than )
          else HANDLE CHOICE ( count_Equal To )
      end

  ; Procedure oPushCountWithValue ( value )

{ push the value of the of the parameter
  on the count stack                        }

    ; var localval : integer

    ; begin localval := value ; PushCount ( LocalVal ) end

  ; Procedure oPushCountWithDuplicate

{ push a copy of the Count tos onto the count stack   }

    ; var localval : integer

    ; begin
        localval := count_stack (. count_stack_top .)
      ; PushCount ( localval )
      end

  ; Procedure oPushCountWithInputByte

{ read a byte from the input and
  push it on the count stack          }

    ; var Byte_num : integer

    ; begin READBYTE ( Byte_num ) ; PushCount ( Byte_num )
end

  ; Procedure oPushCountWithInputNumber

 {read a number from the input and
  push on the count stack                   }

    ; var number : integer

    ; begin READNUMBER ( number ) ; PushCount ( number ) end

  ; Procedure oPushCountWithInputLiteral
```

```
    { Read a string from the input, interpret
      as an integer and push it on the stack }

    ; var Literal num : integer

    ; begin READSTRING ( Literal_num ) ; PushCount (
Literal_num ) end

  ; Procedure oPopCount

    { Pop the count stack    }

    ; Begin
        If count_stack_top = 0
        then oAbort ( eCountStackUnderflow )
        else count_stack_top := pred ( count_stack_top )
      end

  ; Procedure oPushCountWithTagMSwitchFirstp6Label

    { Tag tos must be a switch descriptor. Push
      the first P6label on the count stack        }

    ; var param : integer
      ; d_pointer : des_pointer

    ; begin
        If oCheckSwitchDescrptorKind ( Tag_stack (.
tag_stack_top .))
        then begin
          oGetDescriptorPointer
            ( Tag_stack (. tag_stack_top .) , d_pointer )
        ; param := d_pointer ^ . switchDescrLowerp6label
        ; oPushCount ( param )
        end
        else oAbort ( eCheckSwitchDescriptor )
      end

  ; Procedure oPushCountWithTagSwitchLastP6Label

    { Push the last label value in the range
      on the count stack                    }

    ; var param : integer
      ; d_pointer : des_pointer

    ; begin
        If oCheckSwitchDescrptorKind ( Tag_stack (.
tag_stack_top .) )
        then Begin
          oGetDescriptorPointer
            ( Tag_stack (. tag_stack_top .) , d_pointer )
        ; param := d_pointer ^ . switchDescrUpperP6label
        ; oPushCount ( param )
        end
        else oAbort ( eCheckSwitchDescriptor )
      end

  ; Procedure oPushCountWithTagMdataDotDisplacement
    { Tag tos must name a data descriptor,
      push displacement field of this descriptor
      on the count stack                   }


    ; var param : integer
      ; d_pointer : des_pointer

    ; begin
        If oCheckDataDescriptorKind ( Tag_stack (.
tag_stack_top .) )
        then begin
          oGetDescriptorPointer
            ( Tag_stack (. tag_stack_top .) , d_pointer )
        ; param := d_pointer ^ . dataDescrDisplacement
        ; oPushCount ( param )
        end
        else oAbort ( eCheckDataDescriptor )
      end

  ; Procedure oPushcountWithSMDataDotDisplacement
    { Push the displacement field of the top most
      s-stack descriptor on the count stack      }

    ; var param : integer
      ; d_pointer : des_pointer

    ; begin
        If s_stack (. s_stack_top .) ^ . descrKindTyp
           IN Descriptor_set
        then begin
          param := s_stack (. s_stack_top .) ^ .
dataDescrDisplacement
        ; oPushCount ( param )
        end
        else oAbort ( everifydatadescriptor )
      end

  ; Procedure oPushCountWithSMDataFixrep
    { Push the fix rep field of the topmost
      s-stack descriptor on the count stack  }

    ; var param : integer

    ; begin
        If s_stack (. s_stack_top .) ^ . descrKindTyp
           IN DataDescriptor_set
        then begin
```

```
          param := s_stack (. s_stack_top .) ^ .
dataDescrFixrep
         ; oPushCount ( param )
         end
         else oAbort ( eVerifydatadescriptor )
       end

   ; Procedure oPushCountWithSMDataDotRep
     { Push the repetition field of the
       topmost s-stack descriptor on the count stack }

     ; var param : interger

     ; begin
         If s_stack (. s_stack_top .) ^ . descrKindTYp
            IN DataDescriptor_set
         then begin
          param := s_stack (. s_stack_top .) ^ .
dataDescrRep
         ; oPushCount ( param )
         end
         else oAbort ( eVerifydatadescriptor )
       end

   ; Procedure oPushCountWithSMDataDotLowerBound
     { Push the lower bound field of the topmost
       s-stack descriptor on the count stack      }

     ; var param : integer

     ; begin
         If s_stack (. s_stack_top .) ^ . descrKindTyp
            IN DataDescriptor_set
         then begin
          param := s_stack (. s_stack_top .) ^ .
dataDescrLowerBound
         ; oPushCount ( param )
         end
         else oAbort ( eVerifydatadescriptor )
       end

   ; Procedure oPushCountWithSMDataDotUpperBound
     Push the upper bound field of the topmost
     s-stack descriptor on the count stack      }

     ; var param : integer

     ; begin
         if s_stack (. s_stack_top .) ^ . descrkindTyp
            IN DataDescriptor_set
         then begin
          param := s_stack (. s_stack_top .) ^ .
dataDescrUpperBound
         ; oPushCount ( param )
```

```
      . else oAbort ( eVerifydatadescriptor )
      end.

  ; Procedure opushCountWithTagTypeDotlength
    { Tag tos must name a s-code type descriptor
      Push the length field of this descriptor on
      the count stack                             }

    ; var param : integer

    ; begin
        oGetDescriptorPointer
          ( tag_stack (. tag_stack_top .) ,
Descriptorpointer )
      ; if Descriptorpointer ^ . Descrkind IN
s_code_type set
        then begin
          param := Descriptorpointer ^ . typeDescrLength
        ; oPushCount ( param )
        end
        else oAbort ( eVerifyTagDotKindType )
      end

  ; Procedure oPushCountWithTagMTypeDotHasIndefRepField
    { Tag tos must name a type descriptor, push
      the HasIndefrepfield (1 or 0) on the count
      stack                                        }

    ; var param : integer

    ; begin
        oGetDscriptorpointer
          ( tag_stack (. tag_stack_top .) ,
descriptorpointer )
        if Descriptorpointer ^ . DescrKind IN
s_code_type set
        then begin
          param
          := Descriptorpointer ^ .
typedDescrIndefAlternsteLength
        ; oPushCount ( param )
        end
        else oAbort ( eVerufyTagDotKindType )
      end

  ; Procedure oPushCountWithTagMProfileDotARDisplacement
    { Tag tos must name a profile descriptor. Push
      the activation record displacement field on the
      count stack                                   }

    ; var Param : integer
      ; d_pointer : des_pointer

    ; begin
```

```
        If oCheckprofileDescriptorKind ( Tag_stack (.
tag_stack_top .)
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. tag_stack_top .) , d_pointer )
        ; param := d_pointer ^ .
profileDescrParameterARDisplacement
        ; oPushCount ( param )
        end
        else oAbort ( eVerifyTagDotProfileKindTyp )
      end

  ; Procedure oPushCountWithTagMBodyDotARDisplacement
    { Tag tos must name a body descriptor, push
      activation record displacement field on
      the count stack                             }

    ; var param : integer
      ; d_pointer : des_pointer

    ; begin
        if oCheckprofileDescriptorKind ( Tag_stack (.
tag_stack_top .)
        then begin
          oGetDescriptorpointer
            ( Tag_stack (. tag_stack_top .) , d_pointer )
        ; param := d_pointer ^ . bodyDescrARDisplacement
        ; oPushCount ( param )
        end
        else oAbort ( eVerifyTagDotBodyKindTyp )
      end

      {*****************************************


        Semantic operations associated with the
        Index table stack :

        Index table stack : stack of s-code
        jump indexes


        *****************************************}

  ; Procedure oSetIndexSubCountDotState ( indexState )
    { Topmost count stack entry is the table
      subscript, set its state to the parameter value  }

    ; begin
        Index_Table stack
          (. count_stack (. count_stack_top .) ) .
indexEntryState
        := IndexState
      end
  ; Procedure oVerifyIndexStateSubCountDotState ( indexState
)
    { Topmost count stack is the table subscript, verify
      its index state.                                  }

    ; begin
        If Index_Table stack (. count_stack (.
count_stack_top .) .)
               . indexEntryState
             <> IndexState
        then OAbort ( eVerifyIndexStateSubCountDotState )
      end

  ; Procedure oSetIndexSubCountDotp6LabelToNewLabel
    { Topnmost count stack entry is the table
      subscript, allocate a new P6 label and
      store it in the p6 label field of the
      table entry                                }

    ; var label : integer

    ; begin
        oCreateP6Label ( label )
      ; Index_Table_Stack (. Count_stack (. count_stack_top
.) .)
          . IndexP6Label
        := label
      end

  ; Procedure oVerifyAllIndexesUndefined
    { Verify that all entries in the Index
      table are undefined, abort if not    }


    ; var j : integer

    ; begin
        For j := 1 to max_index_table
        do If Index_Table Stack (. j .) . IndexEntryState <>
undefined
          then oAbort ( eVerifyAllIndexesUndefined )
      end

  ; Procedure oPushDownIndexes
    { Push a new Index table on the Index
      table stack with all entries undefined    }

    ; begin
        if Succ ( index_stack_top ) > max_index_stack
        then oAbort ( eIndexstackoverflow )
        else begin
          index_stack_top := succ ( index_stack_top )
        ; Index_stack (. index_stack_top ) . IndexEntryState
```

```
          := undefined
        end
      ;
      end

  ; procedure oPopIndexes
    { Pop the Index table stack   }

    ; begin
        if index_stack_top = 0
        then oAbort ( eIndexstackunderflow )
        else begin index_stack_top := pred ( index_stack_top
) end
      end

     {*************************************

      Semantic operations associated with
      P6label stack


      **************************************}

  ; Procedure oPushP6LabelWithNewLabel
    { Create a new P6label and push it
      on the P6label stack            }

    ; begin
        If succ ( P6label_stack_top ) > max_labels
        then oAbort ( eP6labelstackoverflow )
        else begin
          p6label_stack_top := succ ( p6label stack_top )
        ; p6label_stack ( p6label stack_top ) :=
p6label_stack_top
        end
      ;
      end

  ; procedure oPopP6Labels
    { Pop the P6label stack       }

    ; begin
        if p6label_stack_top = 0
        then oAbort ( eP6labelstackunderflow )
        else begin P6label_stack_top := pred (
P6label_stack_top ) end
      end

  ; Procedure oSwapP6Labels
    Swap the top two labels on the P6label stack }

    ; var temp : integer

    ; begin
```

```
          temp := p6Label stack (. P6label stack_top .)
        ; p6Label_stack (. P6label stack_top .)
          := P6LAbel_stack (. pred ( P6label stack_top ) .)
        ; P6Label_stack (. pred ( P6label stack_top ) .) :=
temp
        end

        {*****************************************************

          Semantic operations associated with the
          Reachable stack

          : stack of booleans used to determine when the
          s-code is reachable. ( i.e. When the runtime
          control can reach the current point

          *****************************************************

    ; Procedure oPushReachableWithTrue
      { Push true on the reachable stack    }

      ; begin
          if Succ ( reachable_stack_top ) >
max_reachable_stack
          then oAbort ( eReachablestackoverflow )
          else begin
            reachable_stack_top := succ ( reachable stack_top
)
          ; Reachable stack (. reachable_stack_top .) := true
          end
        end

    ; Procedure oPushReachableWithFalse
      { Push false on the reachable stack     }

      ; begin
          if Succ ( reachable_stack_top ) >
max_reachable_stack
          then oAbort ( eReachablestackoverflow )
          else begin
            reachable_stack_top := succ ( reachable_stack_top
)
          ; Reachable stack (. reachable_stack_top .) := false
          end
        end

    ; Procedure oPopReachable
      { Pop the reachable stack      }

      ; begin
          if reachable_stack_top = 0
          then oAbort ( eReachablestackunderflow )
          else begin
            reachable_stack_top := pred ( reachable_stack_top
)
        end
      end

  ; Procedure oSetReachableToTrue
    { Set the top entry of the reachable stack
      to true                                }

    ; begin Reachable_stack (. reachable_stack_top .) :=
true end

  ; procedure oSetReachableToFalse
    { Set the top entry of the reachable
      stack to false                   }

    ; begin Reachable_stack (. reachable_stack_top .) :=
false end

  ; Procedure oChooseReachable
    { Return the value of the top entry
      of reachable stack              }

    ; begin
        HANDLE CHOICE
          ( ord ( Reachable_stack (. reachable_stack_top .)
)
      end

      {*******************************************************

        Semantic operations associated with Global, local
        and constant areas


       *******************************************************

  ; Procedure oAlignAreaToTagType ( areaName )
    { The area displacement named by the parameter
      is aligned to the alignment of the type named
      by the topmost descriptor pointed by the tag
      stack                                       }

    ; var d_Pointer : des_Pointer

    ; begin
        oGetDescriptorPointer
          ( tag_stack (. tag_stack_top .) , d_Pointer )
      ; If d_pointer ^ . DescrKind Not IN S_code type set
        then oAbort ( everifyTagDotKindIsType )
        else glob_const_loc_array (. areaName .)
          := ( ( glob_const_loc_array (. areaName .) - 1 )
               DIV ( d_Pointer ^ . tagAlignment )
               + 1
             )
```

```
                * d_Pointer ^ . tagAlignment
        end

    ; Procedure oIncrAreaDisplacementByCount ( areaName )
      { Increment the current displacement of the
        area named by the parameter by adding the
        top value from the count stack                     }

      ; begin
          glob_const_loc_array (. areaName .)
          := gloc_const loc_array (. areaName .)
             + count_stack (. count_stack_top .)
        end

    ; Procedure oSetAreaToCount ( areaName )
      { Set the area displacement named by the
        parameter to the top value from the
        count stack                                  }

      ; begin
          glob_const loc_array (. areaName .)
          := count_stack (. count_stack_top .)
        end

    ;
```

```
        SEMANTIC OPERATIONS FOR INTERPASS SEGMENTATION
       procedure oBegSegment

       ; begin
           if fragment_stack_top > 0
           then begin
             fragment_stack (. fragment_stack_top .) ^
               . fragmentListNodeLeftLink
               . fragmentListNodeFragmentpointer .
fragmentendpointer
             := current_output_pointer
           ;
           end { then }
         ; push segment
         ; With fragment_stack (. fragment_stack_top .) ^
           do begin
             fragmentListNodeRightlink
             := fragment_stack (. fragment_stack_top .)
           ; fragmentListNodeLeftLink
             := fragment_stack (. fragment_stack_top .)
           ; fragmentListNodepointer . fragmentStartpointer :=
0
           ; fragmentListNodepointer . fragmentEndpointer := 0
           end { with }
           { add one fragment to the newly pushed list    }
         ; getnode
             ( fragment_stack (. fragment_stack_top .) ^
                 . fragmentListNodeLeftLink
             )
         ; With fragment_stack (. fragment_stack_top .) ^
           do begin
             fragmentListNodeLeftLink ^ .
fragmentListNodeLeftLink
             := fragment_stack (. fragment_stack_top .)
           ; fragmentListNodeRightLink :=
fragmentListNodeLeftLink
           ; fragmentListNodeLeftLink ^ .
fragmentListNodeRightLink
             := fragment_stack (. fragment_stack_top .)
           ; fragmentListNodeLeftLink ^ .
fragmentListNodefragmentpointer
               . fragmentStartPointer
             := current_output_pointer
           end { with }
         ;
         end  { oBegsegment }

     ; procedure push segment

       ; begin
           if succ ( fragment_stack_top ) > max_fragment_stack
           then eFragmentStackOverflow
           else begin
             fragment_stack_top := succ ( fragment_stack_top )
```

```
          ; getnode ( fragment_stack (. fragment_stack_top .)
          end
        ;
        end { push_segment}

    ; procedure getnode ( var p : fragmentListNodepointerTyp )

      ; begin
          if avail = nil
          then new ( p )
          else begin
            p := avail
          ; avail := p ^ . fragmentListNodeRightLink
          end
        ;
        end { GET NODE }

    ; procedure releasenodelist ( p :
fragmentListNodePointerTyp )

      ; var temp : fragmentListNodepointerNodeTyp

      ; begin
          temp := p ^ . fragmetListNodeRightLink
        ; p ^ . fragmentListNodeRightLink := avail
        ; avail := temp
        end  { RELEASE NODE LIST}

    ; procedure oWriteList ( var refptr :
fragmentListNodePointerTyp )

      ; var P : fragmentListNodepointerTyp

      ; begin
          P := refptr
        ; p := p ^ . fragmentListNodeRightLink
        ; while p <> refptr
          do begin
            Write ( map_file )
            := p ^ . fragmentListNodefragmentpointer
                 . fragmentStartpointer
          ; fragment_pointer := succ ( fragment_pointer )
          ; Write ( map_file )
            := p ^ . fragmentListNodefragmentpointer
                 . fragmentEndPointer
          ; fragment_pointer := succ ( fragment_pointer )
          ; p := p ^ . fragmentListNodeRightLink
          end { while }
        ;
        end  { OWRITELIST  }

    ; procedure oPopSegment

      ; var param : fragmentListNodepointerTyp
      ; begin
          if fragment_stack_top = 0
          then eFragmentstackunderflow
          else begin
            param
            := fragment_stack (. fragment_stack_top .) ^
                 . fragmentListNodeLeftLink
          ; releasenodelist ( param )
          ; fragment_stack_top := Pred ( fragment_stack_top )
          end
        ;
        end { opopSegment }

    ; procedure oEndsegment

      ; var swap_pointerl : fragmentListNodePointerTyp

      ; begin
          with fragment_stack (. fragment_stack_top .) ^
          do fragmentListNodeLeftLink ^
              . fragmentListNodefragmentpointer .
fragmentEndpointer
            := current_output_pointer
     { WRITE TOS's LIST TO FRAGMENT POINTER FILE          }
        ; oWriteList ( fragment_stack (. fragment_stack_top .)
)
        ; oPopSegment
     { ATTACH A NEW FRAGMENT TO END OF tos LIST AND OPEN IT
}
        ; if Fragment_stack_top > 0
          then begin
            swap_pointerl
            := fragment_stack (. fragment_stack_top .) ^
                 . fragmentListNodeLeftLink
          ; getnode
              ( fragment_stack (. fragment_stack_top .) ^
                  . fragmentListNodeLeftLink
              )
          ; with fragment_stack (. fragment_stack_top .) ^
            do begin
              fragmentListNodeLeftLink ^ .
fragmentListNodeRightLink
              := fragment_stack (. fragment_stack_top .)
            ; fragmentListNodeLeftLink ^ .
fragmentListNodeLeftLink
              := swap_pointerl
            ; swap_pointerl ^ . fragmentListNodeRightLink
              := fragmentListNodeLeftLink
            ; fragmentListNodeLeftLink ^
                . fragmentListNodefragmentpointer .
fragmentStartpointer
              := current_output_pointer
            end { with }
    ;
    end { oEndsegment }
  ;
```

```
      Procedure PushMark ( marker : integer )

      ; begin
          if succ ( mark_stack_top ) > max_mark
          then oAbort ( eMarkstackoverflow )
          else begin
            mark_stack_top := succ ( mark_stack_top )
          ; mark_stack ( mark_stack_top ) := marker
          end
        end

    ; Procedure oMarkAndCopySSection

      ; var upper_mark , lower_mark : integer
        ; count : integer

      ; begin
          PushMark ( s_stack_top )
        ; upper_mark := Mark_stack (. mark_stack_top .)
        ; lower_mark := Mark_stack (. pred ( mark_stack_top )
.)
        ; For count := succ ( lower_mark ) to upper_mark
          do begin
            s_stack_top := succ ( s_stack_top )
          ; S_Stack (. s_stack_top .) ^ := S_Stack (. count .)
^

          end
        ;
        end

    ; Procedure oSwapMarkedSSections

      ; var sec_limit_3 , sec_limit_2 , sec_limit_1 ,
ref_point , sub
            , j
            : integer
        ; temp_stack : array (. 1 .. max_s_stack .) of
s_pointer

      ; begin
          sec_limit_3 := s_stack_top
        ; sec_limit_2 := mark_stack (. mark_stack_top .)
        ; If pred ( mark_stack_top ) < min_mark
          then oAbort ( eNoTwosections )
          else begin
            sec_limit_1 := mark_stack (. pred ( mark_stack_top
) .)
          ; sub := 1
          ; For j := sec_limit_3 to succ ( sec_limit_2 )
            do begin
              temp_stack (. sub .) := s_stack (. j .)
            ; sub := succ ( sub )
            end
          ; ref_point := sec_limit_3
```

```
  ; For j := sec_limit_2 downto succ ( sec_limit_1 )
    do begin
      s_stack (. ref_point .) := s_stack (. j .)
    ; ref_point := pred ( ref_point )
    end
  ; mark_stack (. mark_stack_top .) := ref_point
  ; sub := 1
  ; For j := ref_point downto sec_limit_1
    do begin
      s_stack (. ref_point .) := temp_stack (. sub .)
    ; sub := succ ( sub )
    end
  ;
  end { else }
;
end { swap sections }
```

BIBLIOGRAPHY

1 Nygaard, K., and Dahl, O., The development of SIMULA Language, ACM SIGPLAN Notices, 13,8, August 1978, 245-271.

2 Hills, R., SIMULA67 - AN Introduction, Robin Hills(Consultants) Limited, 1972.

3 Jensen, P. and Krogdahl, S. and Myhre, O. and Robertson, P., The definition of S-code, Norwegian Computing Center,Oslo, 1980.

4 Hartmann, A., A concurrent Pascal compiler for minicomputers, Ph.D. Thesis, California Institute of Technology, 1975.

5 Holt, R. and Cordy, J. and Wortman, D., An Introduction to S/SL: Syntax-Semantic language, ACM Transactions on Programming Languages and Systems, 4,2, April 1982, 149-178.

6 Department of Computer Science, Kansas State University, Internal documentation of Pascal/32 Compiler, 1977.

PERKIN-ELMER SIMULA SYSTEM : INTERPASS SEMANTIC PROCESSING

by

HEMANGI DHOLAKIA

M.Sc., Gujarat University,India, 1975

AN ABSTRACT OF MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

ABSTRACT

Perkin-Elmer SIMULA system is an adaptation of the Portable SIMULA system (S-port) developed by Norwegian Computing Center (NCC) to the Perkin-Elmer 32-bit machines. This report mainly concerns with the description of data structures and semantic operations of Interpass.Interpass is a part of the compiler to be used in the Perkin Elmer portable SIMULA system.

The Portable SIMULA system (S-Port) consists of a portable front end compiler and runtime system.The front end compiler translates the program written in SIMULA to an intermediate language called S-code. In order to implement the portable SIMULA system (S-port) on the Perkin-Elmer machine, it is neccessary to add a machine dependent code generater (S-Compiler) to S-Port. S-compiler translates the program in S-code into the machine language of interdata machine.Code generating mechanism and intermediate token stream languages of the last four passes are used in the construction of S-compiler. Also, a new pass , known as Interpass is written to translate the S-code into the input token stream for the Pascal code generater.Interpass is written in Pascal and S/SL(syntax/semantic) language.The program written in S/SL resembles a recursive descent Parser with output actions and semantic procedures called in.The semantic routines are written in Pascal.

This report presents a brief history of the SIMULA language.It introduces the S-port system and low introduction to Syntax/Semantic language , describes the major features of the pass 1 through pass 9 of Pascal/32 compiler, discusses the components of Perkin-Elmer SIMULA system and provides a detailed description of the major data structures and semantic operations of Interpass.The report also discusses some of the problems encountered in the design of Interpass.