/Adapting a Portable SIMULA Compiler to Perkin-Elmer Computers in a UNIX Environment/

by

Gregory L. Dietrich

B.S., Kansas State University, 1979

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

Major Professor

Contents

I. Introduction

This paper describes the effort at Kansas State University to implement a compiler for the language SIMULA-67 [DAH82]. The target machines are those in the Perkin-Elmer 32-bit series, including the 8/32 and 3220 models [PER75, PER79, PER80], resident in the Department of Computer Science. The operating system environment for the SIMULA compiler is UNIX† Version 7 [THO79].

During the Spring 1983 and Spring 1984 semesters, an Implementation Projects class was conducted under the supervision of Professor Rod Bates. Approximately ten graduate students participated in one or both of the classes. S-PORT [JEN83, MIL83], a portable SIMULA system designed by the Norwegian Computing Center (NCC) to facilitate SIMULA compiler implementation, was the basis for the project.

This paper first describes briefly the language SIMULA itself, and some of its interesting features. The features of the portable SIMULA compiler system developed by NCC are then given. I discuss the design of the KSU implementation. My contributions to the compiler are described, and the current state of the project.

† UNIX is a trademark of Bell Laboratories.

II. The SIMULA Language

The language SIMULA was developed in the mid-1960's by the Norwegian Computing Center (NCC). SIMULA is essentially an extension of the language ALGOL 60 [NAU63]. ALGOL is the predecessor of many modern languages such as Pascal [WIR71] and C [RIT78a]. ALGOL's most prominent feature is its block structured definitions, which it introduced. The scoping mechanism used in those 'modern' languages mentioned above has changed little from that used by ALGOL. ALGOL also provides a fairly complete set of control structures including various looping and decision structures similar in scope to Pascal. ALGOL also permits the use of several parameter passing mechanisms, including a rare call-by-name implementation. SIMULA borrows much of its syntax from ALGOL, to the extent that its definitive document, *Common Base Language* [DAH82], only describes the differences between ALGOL 60 and SIMULA.

The intent of the developers of SIMULA was to create a general purpose language suitable for constructing very large software systems. Simulation applications were seen as one example of such programming efforts, and other features were incorporated which cater to the needs of simulations.

The most important of these additions to the ALGOL language are concurrency capability (coroutines) and constructs for simulation entities which may exist with lifetimes independent of the control flow of a program. SIMULA also has defined within the language character and text (string) types and operations, and input/output facilities, which are missing in ALGOL.

Data Abstraction

SIMULA was a pioneer language in that it was the first to implement a type of data abstraction, the class. SIMULA classes were designed to facilitate modular

programming, a necessity for creating large application programs. SIMULA classes encapsulate the definition of a data structure and operations on it. The capability exists in SIMULA to restrict the use of that data in the manner defined within the class. In this respect, SIMULA is similar to Concurrent Pascal [HAN75], which has classes patterned after SIMULA. SIMULA, unlike Concurrent Pascal, allows the programmer to choose which names may not be referenced outside the class by specifying them as *hidden* or *protected*. Concurrent Pascal simply does not allow any access to the data of a class directly, but only through entry routines.

```
[ <prefixClassName> ]
  class <className> [ ( <parameters> ) ]

  begin

    <declarations>

    <statements>

  end <className>;
```

Figure 2.1 Class syntax

```
begin
  class dataClass
  begin

    integer integerData;

    integer procedure integerValue;
      integerValue := integerData;

  end dataClass;


  ref(dataClass) dataPointer;

      comment Create a new class instance;
  dataPointer :- new(dataClass);

      comment Direct reference to variable of the class;
  dataPointer.integerData := 5;

      comment Invoke class procedure;
  if dataPointer.integerValue > 4
  then
    ....

end;
```

Figure 2.2 Class operations

The class syntax and definition in Figures 2.1 and 2.2 illustrate the basic use of SIMULA classes. Using the class definition as a template, a 'client' may create data objects via the *new* operation. *New* takes a parameter which is the identifier of the class type of the object. *New* allocates space for a new class instance from a heap and execution passes to the statement part of the class object. These statements will usually be required to initialize the object's data structure into a consistent state. When the statement part is exhausted, control returns to the statement following the *new* statement. *New* also returns a pointer to the data object. Reference variables were added to the language to hold these pointers. Only pointers to class instances are allowed in SIMULA, not to any simple data type. Furthermore, reference variables may only refer to a class instance of the proper type. Notice in Figure 2.2 that assignment of reference values uses an operator distinct from normal assignment. (':-' for references, and the familiar ':=' for non-references) Comparison of references and other operations also use different notation from that normally used in expressions.

Unless they are hidden, the data definitions within the class may be referenced directly in a manner similar to the fields of a record. In addition, procedures which are defined may be invoked with the familiar dot notation as well.

The lifetime of an class object is normally limited by the lifetime of the object which created it. That is, when the block which created objects terminates, the objects it created are destroyed as well. The created objects are thus 'attached' to their creator. The *detach* statement, when executed by a class object, removes that connection, and the object's lifetime is then indefinite. As long as there exists one or more accessible references to the class object, it continues to exist in the system. *Detach* also causes control to pass immediately from the class's statement part back to its creator.

SIMULA has no mechanism to explicitly return the space used by classes to the heap. Instead, part of the language definition requires a garbage collector to

automatically reclaim unused space. Periodically during the execution of a SIMULA program, the garbage collector may run to determine which class objects are inaccessible, and make their storage available for later *new* calls.

Prefixing is a feature which increases the power of SIMULA classes as data abstraction tools. When a prefix class name is specified as part of a class definition (see Figure 2.1 above), the parameters, declarations and statements of the prefix class are combined with those in the new class. This is useful, for instance, for creating 'generic' data structures, such as a stack, without defining the type of the elements in the structure. The class which defines the operations on the structure may be used as a prefix to other classes which define the elements, allowing the same code to be used to manipulate data structures with different element types. SIMULA itself uses this technique to maintain numerous types of lists in its simulation definitions.

Coroutines

Another SIMULA extension to ALGOL is the coroutine. In SIMULA, two or more detached classes may interleave their execution in an arbitrary manner by using the *resume* command. *Resume* takes one argument, a reference. Execution of the current coroutine is suspended, and control is passed to the specified class. If, at a later time, the first class is resumed, execution will continue with the statement after the *resume* which was previously executed by that class.

The primitive features described above were utilized to define a set of simulation functions which are essentially part of the SIMULA language definition. They are in the form of 'predefined' classes which may be referenced by user programs. The possibility exists, however, for the SIMULA user to define his own simulation system using a different approach altogether using the primitive SIMULA operations.

SIMULA also incorporates in the language definition a method for separate compilation of programs with strict type checking, as well as the possibility of linkage with procedures written in other languages. This philosophy was also adopted in the design of the more recent languages Modula [WIR85] and Ada [ADA79].

III. S-PORT

To aid implementation of SIMULA compilers, the Norwegian Computing Center (NCC) in 1980 introduced S-PORT, a portable system that reduces the effort necessary to write a SIMULA compiler. This chapter describes the components of S-PORT that NCC supplies, as well as the parts which must be supplied by any implementor.

Compared to most languages, SIMULA's run-time behavior is quite complex. Storage management is one of the primary difficulties in implementation. Heap allocation strategies and garbage collection add much complexity to the task of defining the environment of a running SIMULA program. The inherent multitasking that is provided also increases the difficulty of implementation. Furthermore, call-by-name parameter conventions are complicated. The degree of complexity of SIMULA's implementation not only makes the generation of a compiler from scratch very expensive, it also hinders portability of programs written in SIMULA. The more complex a language is by definition, the more easily differing interpretations of its behavior may be incorporated into different compilers.

NCC has eased some of these problems by providing the S-PORT system. It insulates many of the most obscure and complex details of SIMULA from the implementor. Compilers developed using the S-PORT system should be less expensive to produce and will have a higher degree of consistency in their behavior.

The heart of the system is S-Code, a relatively simple low-level language. With only a code generator for S-Code and a package of system-dependent service routines, a working SIMULA compiler may be generated.

NCC provides a front-end compiler, which translates input in the SIMULA language to S-Code. Also provided is a library of run-time support which implements the memory management and other standard SIMULA features. These also are in S-

Code. The implementor provides a code generator for the target machine which accepts S-Code, and a package of system-dependent utility routines that provide support for basic low-level operations involving memory management, input/output, etc.

S-Code

S-Code [JEN83] is a low-level intermediate language which drives the S-Compiler code generation process. The language is not intended for interpretation, and is not suited for that purpose.

It has a loose typing structure which provides several basic types, including the normal INT, REAL, LONGREAL, CHAR, BOOLEAN and others. Types and data quantities are identified by two-byte numbers called tags. The basic data types are predefined tags.

Repetitions of a quantity may be defined, which are allocated essentially as arrays, although no index checking is performed on access, and the group of elements is not generally treated as a whole in S-Code. The basic types may also be combined into structured data types *(records)*, with possible variants. A *record* may also contain a *prefix* tag, which specifies an existing record definition whose fields are to be included in the definition of the new record type.

Constants of any type are allowed, including the structured types. To allow for possible rearranging of fields in records, each field value is associated in a record constant with its field tag, and all fields must have values.

Control flow constructs which are available include *if then else* constructs, conditional and unconditional jumps, and multi-path conditional statements *(switch)*.

Routines may be defined. All parameters are passed by value, except a possible *export* parameter, which may be of arbitrary type. Routines also have the interesting ability to alter their return address, if it is explicitly passed as an *exit* parameter to the

routine. Scope rules for S-Code are simple. Data is either local or global, and routines may not be nested. No recursion is allowed.

Computations are carried out on an expression stack. Some checking is necessary in S-Code expressions. The S-Compiler must check that identical types participate in assignments and parameter passing, for instance. Range checking must also be done on assignment to SINT variables, which allow for space optimization of small INT values. Smaller ranges, which are allowed, and INT values themselves are checked by the Front-end Compiler. No index checking is done on array subscripting by the S-Compiler itself; that also is the responsibility of the translator producing the S-Code. The typing in the language is primarily for the description of data which has already been deemed to be consistently used by the Front-end Compiler.

Separate compilation is allowed, with shared global data, constants and type definitions. The *module* is the construct used for the unit which may be separately compiled. The definitions within a module may be made visible to other modules. A skeleton of the S-Code for a typical module insertion is given in Figure 3.1 below.

```
module x
  <visible declarations>

taglist
  tag t0 0
  . . .
  tag 20 5
  tag 25 6
  . . .
  tag tn n

body . . . endmodule


module y

  insert x 50 50+n
  . . .

body . . . endmodule
```

Figure 3.1

After the visible declarations of constants, global variables, routine profiles and labels, the module specifies tag values for the data and types which may be utilized by other modules. Each visible tag is associated with an integer in the range 0..n, where n+1 is the number of visible tags in this module. An *insert* operator in another module specifies a range of n+1 tags which are available for definition. Each of the visible tags' meanings is associated with a tag in that range, corresponding to the visible tag's index. In the example above, within module y the tag 55 would refer to the quantity defined in module x with the tag value 20.

Front End Compiler (FEC)

The translation of SIMULA into S-Code is performed by this large program, provided in S-Code. The process of translation is done in either two or three passes, whichever the implementor wishes. To date, no one has used the three pass version in an implementation, possibly because the main reason for using the three-pass compiler

would be to lower memory requirements for the system, and the three pass version is not drastically smaller than the two-pass version.

Runtime System (RTS)

Support for some of the most difficult to implement aspects of SIMULA is provided by these S-Code modules. In Release 101 of the S-PORT system, this library is made up of fifteen separately compiled modules. They implement memory management, including garbage collection, much of the input/output, multitasking support for coroutines, and other system-independent low-level functions.

S-Compiler

The implementor must provide a code generator which accepts S-Code and produces viable code for the target machine. This code generator is referred to as the S-Compiler.

Environment Interface

The implementor must also provide 'hooks' to allow the Run-time System and Front-end Compiler to obtain system dependent services such as input/output and memory management. The definition of the S-Code compatible interface routines that the S-PORT system requires is called the Environment Interface [MIL83]. System dependent details that this package must deal with include, among other things, an extensive set of file system and input/output routines and providing the means for the user to communicate his choice of certain compilation options to the front-end compiler.

Another important component of the run-time support is an exception handler which receives control upon any trap or interrupt that a running program may cause. This routine is defined in the Environment Interface document. At the present time,

its job is essentially to determine the cause of an exception when it receives control and communicate that information to the Run-time System, which returns an indication of some action to take. Some arithmetic faults may be resolved by substituting a default value (e.g. zero for floating-point underflow on machines which notice that condition). Other conditions will require a graceful termination of the system, which may involve closing open files and other system-dependent actions.

S-Code provides the capability in the future to have traps to support debugging tools, and NCC is planning to provide a source-level debug utility as part of S-PORT at some time in the future. The exception monitor will be heavily used in such a system. For the present, the use of the exception monitor will be confined to arithmetic and addressing faults.

Development of an S-PORT System.

The development process of a S-PORT system is described below.

1. Write an S-Compiler.

2. Write the S-Code callable Environment Interface routines.

3. Translate the RTS modules producing a library of object code.

4. Translate each FEC pass, producing object code files.

5. Link each translated FEC pass with the Environment Interface and RTS libraries, producing executable FEC passes.

6. SIMULA source may now be translated into S-Code with the FEC passes, and linked in a manner similar to the FEC to yield executable versions.

IV. Kansas State University Implementation of S-port

S-Compiler

The most significant design decision was the general strategy for the implementation of the S-Compiler. Two directions could have been taken in the KSU project: to write the code generator from scratch, or utilize existing similar software.

While early phases of compilation have numerous tools to assist the compiler writer, such as parser generators, code generation is the least automated phase of compilation. Much of the work involved in generating code is tightly related to the particular architecture of the target machine, and performance requirements dictate that output code should be relatively efficient on that machine. Therefore, the task of writing a new code generator, especially for programmers inexperienced in compiler writing, seems quite large. For that reason, another means was chosen to implement the S-Compiler.

The Computer Science Department has a locally produced Pascal compiler, PAS32 [YOU80], which consists of nine passes. It was loosely adapted from a Pascal compiler for the DEC PDP-11 [HAR77]. Passes one through five perform lexical analysis of the Pascal program, checking types and scoping, yielding intermediate code roughly similar in function, if not in form, to S-Code.

To utilize the backend passes of PAS32, a separate translator was needed to transform S-Code into a language closer to that of the input language of pass six of the PAS32 compiler. Also, passes six through nine of PAS32 needed to be modified to provide some functions needed to accommodate S-Code. This section describes the intermediate translator, called Interpass, and some of the changes to PAS32.

Interpass

PAS32's intermediate code for its pass six is in general similar in function to that of S-Code, so much of Interpass's work is relatively trivial. For instance, Figure 4.1 shows in symbolic form the translation of a conditional branch instruction in S-Code to its pass six equivalent. The translation is straightforward, although the opposite interpretation of the conditional expression forces the addition of the *not* operator.

S-Code:

```
expr
fjumpif (conditionalTest,destinationIndex)
    ...
flabel (destinationIndex)
```

Pass six:

```
expr'
compare (P6conditionalTest,P6Type)
not
falsejump (P6Label)
    ...
defLabel (P6Label)
```

Figure 4.1

Some S-Code constructs were somewhat more difficult to translate. For instance, S-Code contains type and data definitions separate from executable code. The reason for this is that the form that data types, particularly records, should take is highly machine dependent. PAS32 was written with a particular architecture in mind, and in pass six's input language, records are treated simply as blocks of bytes at known offsets in a data area and fields are considered byte offsets within those blocks. This data is distributed through the code produced by the PAS32 front-end passes as parameters to the operations on the data. This eliminates the need for a symbol table in the later passes. Interpass, therefore, had to consume record definitions, choose the layout in memory for their fields, and output this information with operations performed on data.

S-Code also includes an *if else endif* construct, which pass six does not have. This construct had to be translated into lower-level operations which accomplish the same result, as shown in Figure 4.2.

```
    S-Code:
expr
if(conditionalTest)
    ...
else
    ...
endif


    PAS32:
expr
not
falsejump(L1)
    ...
jump(L2)
label(L1)
    ...
label(L2)
```

Figure 4.2

One significant difference between S-Code and pass six is the form of addresses and the method of selecting fields within records. An S-Code 'general address' (GADDR) is a two-part quantity consisting of an object base address (OADDR) and an offset into that object (AADDR). A GADDR is represented by two fullwords (32-bit integers) on the pass six stack.

Pass six, however, treats all addresses as single fullword values. In fact, quantities which cannot be expressed by a single word (for example, double precision real values) are not directly placed on the stack by pass six, but their addresses. This situation causes the output of Interpass to be rather convoluted in the cases where GADDR values are manipulated on the pass six stack, since the components of a reference must be added before they are used to fetch a value.

Furthermore, S-Code may use quantities on its expression stack more than once. The *update* operator, for instance, stores a value located on the top of the stack into the address described by the second expression on the stack, popping only the value.

PAS32 always removes its expressions from the stack when used. Its only construct for assignment is the *assign* operator, which acts like the S-Code update, except both its operators are popped after the assignment of the value takes place. Also, S-Code includes a *dup* token which duplicates the top value on the stack. PAS32 has no equivalent operation. Figure 4.3 illustrates several of the issues mentioned above.

```
      S-Code:
expr1
expr2
update

      Pass six:
expr1'
dup(wordPairType)
add(wordType)
expr2'
(possible fetch code)
assign
```

Figure 4.3

Translation of this segment of code requires some backpatching, or insertion of additional code into the stream of code which has already been output. The sequence of events as this S-Code is consumed is described below.

Code is output for expr1, leaving a GADDR on the expression stack. Then expr2's code is output. It may yield either a reference or a value, as fetches are done automatically as necessary in S-Code.

The update operator causes several actions to be taken by Interpass. If expr2 leaves a reference on the top of the stack, code must be emitted to fetch the value, as PAS32 does not allow for reference parameters to its assignment operator. The address expression, expr1, must survive the update. Interpass keeps track of the location in the output stream of the start of all expressions on its stack, so it is possible to patch a dup operator in the output just before the start of the code for expr2. Also, since GADDR's are not useful to PAS32, an *add* operator is output to change the GADDR

copy into a single fullword pointer, leaving the original GADDR value intact.

This backpatching was accomplished not by actually modifying the code which had already been output, but by creating a separate 'map' file which logically splits the output into segments. This allowed Interpass to function using pure sequential output. Therefore, pass six needs to process its input in a non-sequential manner according to the map file.

Some of the most complicated activities of Interpass concerned the exporting of information between separately compiled modules. Definitions which may be made known to other modules are localized in the first section of a module definition, as was seen in the section dealing with S-Code in the previous chapter. There must be some description of that data which survives the compilation of a source file, so that it may be referred to when compiling modules which insert the current module. The scheme chosen was to essentially copy the S-Code of the module header only to two separate files. Those files include the definitions and the list of tags which may be referred to by inserting modules. To perform an insertion, Interpass must read the visible file corresponding to the inserted module, process it to generate its type definitions, and then use the list of tags to update the tag definition table for the inserting module.

PAS32 Backend

S-Code required several functions which the existing Pascal compiler did not provide. Examples cited above included the *dup* token and the associated register allocation and two-word address manipulation on the expression stack.

PAS32 lacks some other capabilities necessary for SIMULA as well. S-Code may have global initialized variables, for instance. PAS32 collects constant values into a contiguous area of memory, but has no initialized variable data. A new area was defined to hold these values. The reason that the existing constant area was not used is

that pass 8 optimizes that area by eliminating duplicate constant values.

S-Code also requires a scheme for keeping track of all pointer values stored in temporary storage (registers or other anonymous storage). This is done so that whenever the garbage collector is invoked while running a SIMULA program, if the objects are relocated to which such pointers refer, the pointers may be adjusted by the Runtime System automatically.

Environment Interface

The Environment Interface library was written in C, since access to UNIX system functions is most easily accomplished from that language. Its design is described in a later chapter.

Other Program Development

Several programs were written to assist the development of the S-Compiler. Since the project was started before we had access to any programs in S-Code, an assembler was written to translate S-Code mnemonics into their binary form, and a dumper to perform the reverse. This allowed test programs to be written in S-Code to debug Interpass. Dumpers were also written for the modified PAS32 intermediate languages.

Recently a program was completed which allows SIMULA source to be automatically formatted.

V. The Environment Interface Package (EI)

In the Spring 1983 and Spring 1984 classes, I worked on the library of routines which provide the support defined in the EI [MIL83]. Since these routines make use of operating system facilities, they were most naturally written in C [KER78]. Except for a small kernel which is written in assembly language, UNIX is written wholly in C, and low-level system functions may be performed very conveniently from that language.

The entire suite of routines that SIMULA requires is shown in Figure 5.1. The bulk of the routines which are defined are input/output, numeric editing/deediting and math functions.

Global variables: status, itemsize, encdrv, curdrv, curins, bioref, tmpqnt

Global constants: maxlen

Type definitions: STRING

Option-setting routines:

getIntInfo	getTextInfo	getRealInfo#
getSizeInfo	giveIntInfo	giveTextInfo

Memory management routines:

defWorkArea	move

Numeric to/from STRING editing routines:

lowten	getInt	getReal
getFrac	putInt	putFix
putLFix	putReal	putLReal
putFrac	putSize	putOaddr
putAaadr	putPaddr	putRaddr

Debugging routines:

getOutermost	getPaddr#	getLineNo
breakpoint#	stmtNote#	dmpObj

Input/output routines:

lookUp#	openDsp	closeDsp
inImage	outImage	locate
getDsName	getDsetSpec	getLpp
newPage	printOutImage	inByte
in2Byte*	outByte	out2Byte*

Mathematical routines:

sqrt	ln	exp
sinus	arctan	cos*
tan*	arcsin*	arccos*
rSqrt	rLn	rExp
rSinus	rArctan	rCos*
rTan*	rArcsin*	rArccos*

Miscellaneous routines:

initialise	terminate	
basicDraw	dateTime	cpuTime

* - optionally provided by implementor
# - unused in the current S-PORT system

Figure 5.1 Environment Interface Definitions

The global variable status is used by the EI package to communicate the success of its functions to the SIMULA RTS. The other variables are not used by the EI, but merely provided for the RTS. A few of the variables have been defined only in the latest release of S-PORT and are intended to be used for future optimizations.

The memory management support SIMULA requires at the S-Code level is very simple. A running SIMULA program may request the allocation or modification of what is called a work area, which is a contiguous address space in memory. If it would be advantageous to allow more than one work area per SIMULA program, the EI may provide for that possibility and the RTS will use multiple areas; otherwise it uses only one work area. The running program may request that a work area be increased in size, decreased, or deleted entirely.

The move routine is intended to allow the implementor to take advantage of any system-dependent facilities for efficient block memory transfers. Unfortunately, the 8/32 processor has no such instructions, although multiple register store/load operations are somewhat more efficient than word-by-word transfer. The 3220, on the other hand, is a more recent model of this architecture, and has extensions to its instruction set which include block memory transfer. Due to the likelyhood of much activity of this sort during garbage collection, it would be wise to provide different versions of this routine which will take advantage of the newer CPUs' block move instructions.

The information handling routines (eg. getIntInfo) are used for communicating needed information between the environment and the FEC and RTS. For instance, the compiler has a large number of options dealing with what type of listing to produce for a compilation, what the file names for all the input, output and intermediate files, and what type of debugging aids to include in its output code. These require a fairly large data structure, and also a means of extracting the pertinent information from the command line used to invoke the SIMULA program. Some coordination between the C

or shell command file driving the compilation process will be needed to present this information to the EI package in a form it can use.

The numeric editing routines took some time to complete. Their function was complicated by the fact that all strings passed to them needed to be parsed. This fact seemed strange at first, since programs which were being compiled had already been parsed. These routines, however, would be called to convert input text from files or the user's terminal. Integer values were not difficult to convert, but floating point values were more of a problem. The method I originally pursued was to calculate the values as I parsed the string. But I decided it would be easier to convert the strings into a form which the similar existing system routine, ecvt, could accept, letting it do the work of conversion and avoiding floating exceptions. Two problems remain. First, the C language in this implementation of UNIX does not support double precision floating point operations. SIMULA does not require double precision arithmetic, but it would be very advisable to provide the added precision, considering SIMULA's applications. Second, ecvt exhibits strange behavior as the values it is calculating near the limits of the machine's range. For instance, '4e74' is converted correctly, but '4.0e74' causes a floating exception. These values are not especially close to the limits of floating point representation, either. The best solution to these problems is to eventually recode these routines in an available language which supports double-precision (eg - Pascal, assembly language).

The file handling operations took most of the balance of the effort. The S-PORT system was developed with a philosophy concerning file operations much different from that of the developers of UNIX [RIT78a, RIT78b]. While the UNIX file system itself generally treats all input/output as simply character-oriented, the EI tends to favor a record-oriented approach. Several types of files are used, whose characteristics are detailed in Figure 5.2.

infile	record-oriented sequential
outfile	record-oriented sequential
printfile	record-oriented sequential
directfile	record-oriented random access
inbytefile	byte-oriented sequential
outbytefile	byte-oriented sequential

Figure 5.2.

Inbytefiles and outbytefiles are similar to normal UNIX files, but are used only by the FEC for binary intermediate files, and are not available from SIMULA itself.

Infiles and outfiles are sequential record-oriented files. S-PORT is flexible as to whether these files may have variable-length records. For text files, the natural way to handle these file types is with newline-terminated variable length records. This reduces space requirements, and since access is sequential, there is no need to be able to find a particular record by its offset within a file. Binary files could not have variable length records, since any byte within a record could have the code for the newline character.

Directfiles are record-oriented random access files. The only way to simply implement this type of file is with fixed length records. Then the standard seek function, which is byte oriented, may be used with a minimum of effort. For variable length records, some complicated indexing scheme would have to be implemented on top of the UNIX file structure, whose efficiency would be suspect.

Printfiles are essentially outfiles with some added routines to aid in printer-oriented operations, such as overstriking lines and page ejects.

Files in S-PORT are referenced by keys, which are small integers assigned with files when they are opened via openDsp. When opening or closing files, a string parameter is provided, which may specify actions to take. Some of the standard actions allowed are tape-oriented, such as REWIND, NEXTFILE and PREVIOUS. File deletion

when closing accomplishes a sort of scratch file capability. Other system-dependent actions may be included by an implementor as they are needed. For our purposes, the only actions implemented were RELEASE, which deletes the file when closed, and APPEND, which positions the file pointer at the end of existing data when the file is opened.

The implementation was generally straightforward. The state of opened files is kept in a table indexed from 0 to the maximum number of open files UNIX allows. Figure 5.3 gives the C definition for the information in each file's entry.

```
typedef struct {
  char name[73];  /* path name passed to openDsp */
  int  isOpen,
    inode,       /* inode of opened file */
    type,        /* S-code file type */
    recl,        /* Record length */
    fixed,       /* Is record length variable? */
    leftover,    /* Used for partial reads */
    position,    /* Current record number */
    fileSize;    /* Number of records in file */
} FENTRY;
```

Figure 5.3

Some fields in the file entry are obvious, such as name, isOpen, type and recl. Fixed is an addition to allow the system to have fixed or variable length records for infiles and outfiles. (Directfiles must have fixed record length). An additional ACTION parameter to openDsp was defined, length. Otherwise, the recl given is considered a maximum length for each line in the file.

SIMULA forbids the same file to be referred to by more than one key at a time. UNIX does not restrict file operations in that manner, so my routines had to enforce that policy. Originally, I checked that the names passed to openDsp were unique, but the names 'xyzzy' and '/usr/greg/xyzzy' may in fact refer to the same file, which would defeat that simple method. I instead saved each file's i-number, which uniquely

identifies the file on a UNIX filesystem. A problem may occur if two files on different filesystems are opened which by chance have the same i-number. Modifying the file table to include the device identification as well as the i-number would correct that problem.

SIMULA allows a smaller buffer to be passed to the read function than the record length of a file, if the implementor can read partial records. That is no problem with the UNIX I/O functions. The leftover field is set to the number of bytes left before the maximum record size is read. Both fixed and variable length records may be read in this manner.

SIMULA places restrictions on seeks within directfiles which UNIX does not. The seek may not place the file pointer beyond existing data. The UNIX system call fills the intervening space in the file with null characters if a seek is beyond the end of the file. Therefore, the number of records in the file is kept in the file's table entry, so that the record number passed to the seek function may be tested.

Three SIMULA files are predefined: SYSIN, SYSOUT, and SYSTRACE. SYSIN and SYSOUT are associated with UNIX stdin and stdout by default, which would allow input and output using these files to be redirected or piped in the usual UNIX manner. SYSTRACE is assumed to be connected with a UNIX file. The S-PORT documentation suggests that it be implemented with a circular buffer to avoid collecting excessive amounts of trace information. That is not implemented, but could be easily done in the future.

Environment switching

PAS32 has no built-in capability to call external C routines, so a mechanism to make use of the EI package had to be created. SIMULA's run-time stack as designed is similar to the PAS32 stack, with minor modifications. PAS32 normally calls functions and procedures with as many parameters in registers as possible, with the remaining parameters passed on the stack. External procedures and functions have all their parameters stored on the stack. Compiled SIMULA programs will always have parameters stored on the stack. Figure 5.4 diagrams the state of the PAS32 stack when an external routine is called.

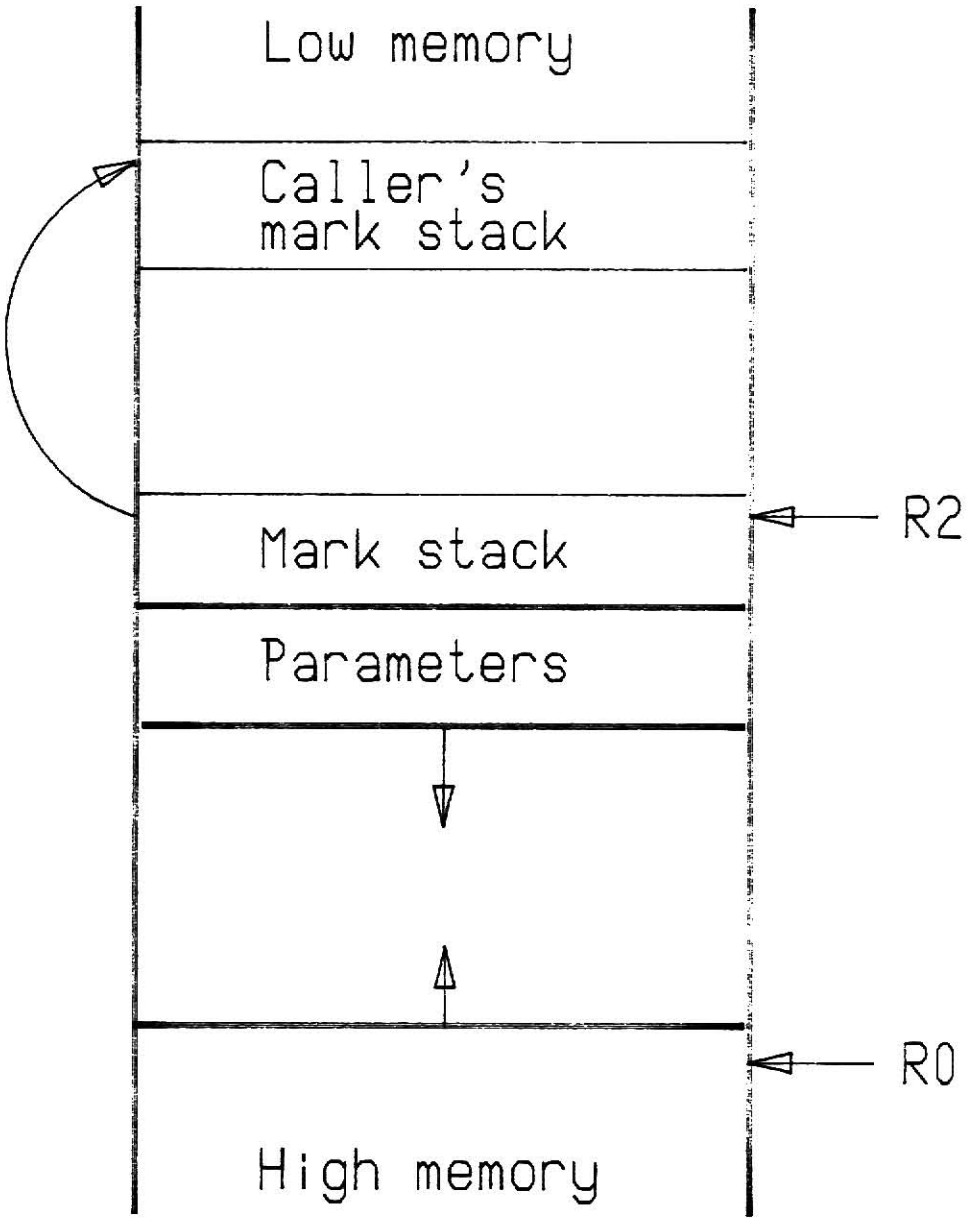

Figure 5.4a PASCAL/32 Stack

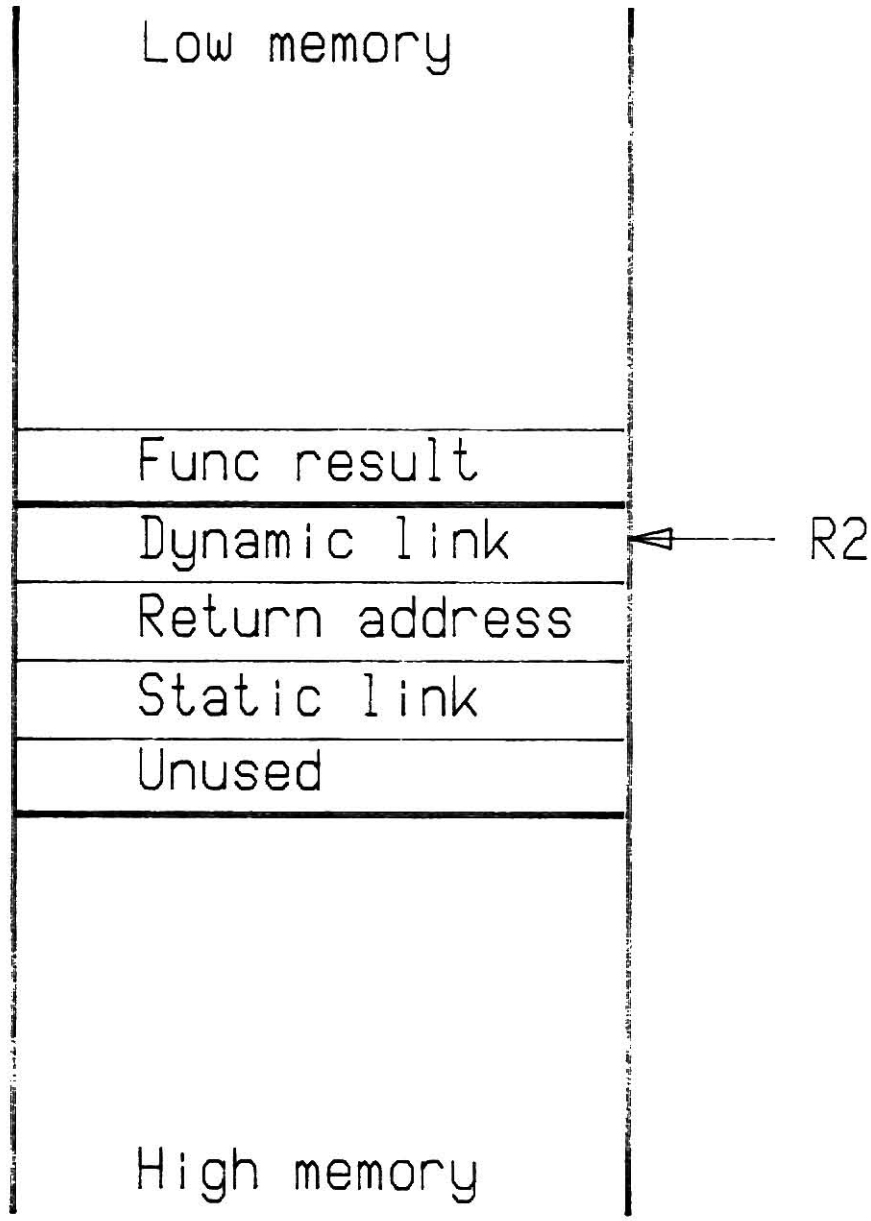

Figure 5.4b PASCAL/32 Activation Record

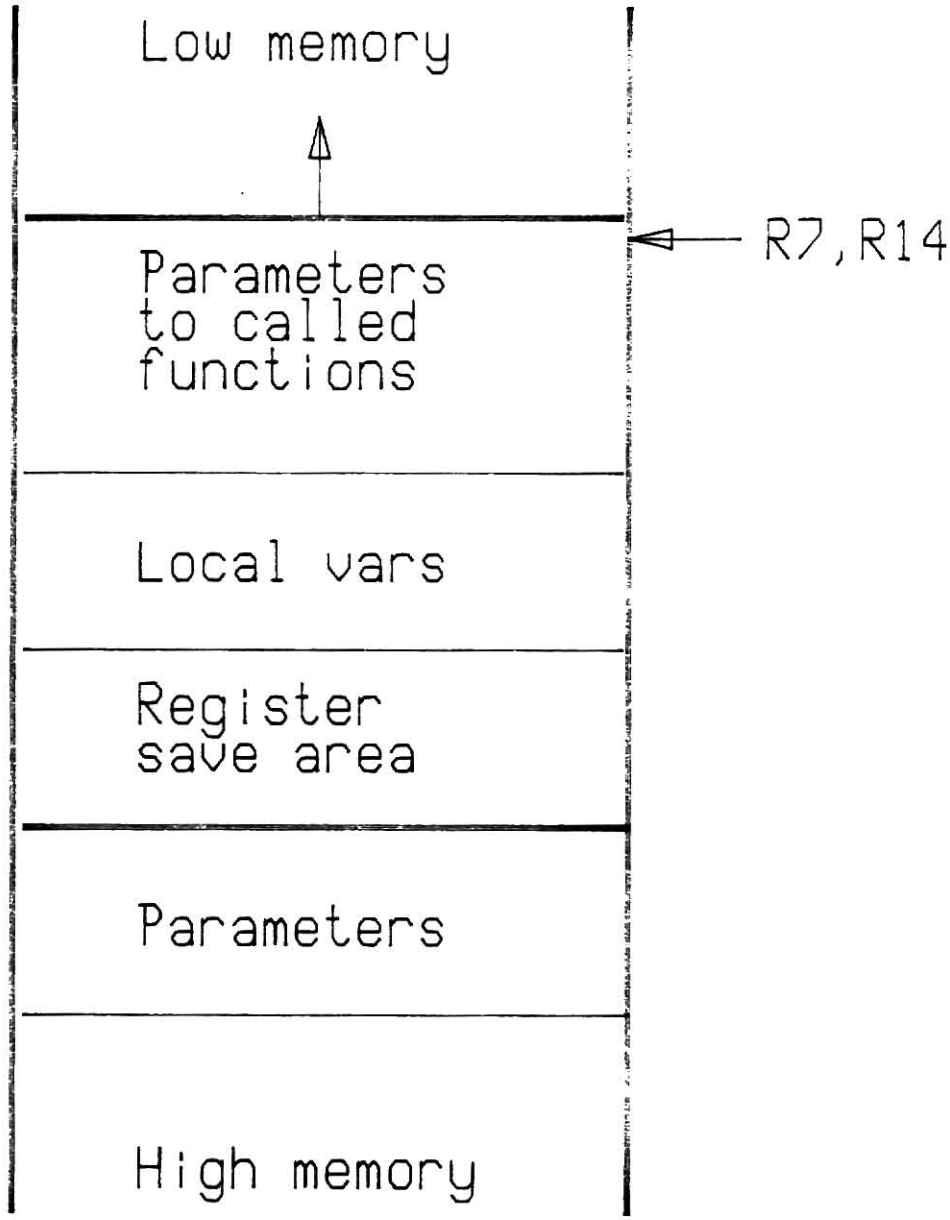

Figure 5.5 C Stack

In Pascal, registers 0 through 2 contain the stack segment status information. The calling routine sets up the stack for the execution of the subroutine, and on return, the called function should have popped its environment off the stack (the dynamic link is modified). If allocation from the PAS32 heap has not occurred, the heapLimit register should be unchanged, and the static link is unchanged in any case. Return values for external functions are left on the stack immediately before the activation record for the called routine. The return address is originally in R14. Other registers may be modified freely.

Copies of value parameters are passed, if the value will fit into a fullword, or if the value is of type real. Arrays and records are always passed by reference. PAS32, unlike standard Pascal, forbids assignment to value parameters altogether, so this saves space and the time needed to make copies of large values.

Figure 5.5 shows the environment of a running C function. On entry, R7 points to the top of the section of parameters to the function, and is adjusted to provide space for the local variables and other workspace. Registers 7 and 14 are used to access values on the stack, with R7 being the stack pointer and R14 a pointer to the current function's local data area. The C stack grows in the opposite direction from the PAS32 stack, but the parameters are stored in the same order in memory, fortunately. Parameters are passed similarly to PAS32 in most respects. Values that will fit into a fullword are passed in fullwords. Real values are single precision only, so they are also passed in one fullword. Arrays are always passed by reference. Structure (record) values, however, are indeed passed by value, and a copy is made on the C stack. Small return values are left in R0, while copies are made of structure values, and a pointer is returned in R0. A summary of registers significant to the two languages follows.

PAS32 registers which need
to survive external function calls:

R0: Heap limit pointer
R1: Static link
R2: Dynamic link
R14: Return address

Significant C
registers on entry to a function:

R0: Return value
R7: Stack top pointer
R15: Return address
(Registers 8-15 are restored upon return.)

Figure 5.6 Register Usage

The differences in environments required that control be passed from PAS32/SIMULA to C via an assembler language interface routine. It saves the Pascal environment and sets up the C environment for the called functions. There was space in the Pascal activation record to save R1 and R14, and two global variables were added to the Pascal driver to save R0 and R2. The original C stack pointer inherited on execution of the Pascal program is accessible in a variable as well. After restoring its value, the parameters are copied from the Pascal stack to the C stack, and a branch to the function desired is made. Upon return from the C function, the Pascal environment is restored as the caller expects. At this time, the EI routines are implemented as procedures, not functions, so the return value is not significant and is ignored.

Unfortunately, PAS32 does not currently include a facility to determine the addresses of external data or functions. For that reason, the interface cannot be told by the calling PAS32 program where to branch when invoking the C function. Therefore, the easiest way for the interface to determine the address of the desired C routine was to provide an entry point in the interface for each EI routine, which loads one register with the address of the routine, and another with the size of the parameter list. The

modifications of the backend passes of PAS32 include automatically prepending two parameters to external C calls: a linker-resolvable reference to the routine's address, and the size of the parameter area. This will allow the interface routine to function with a single entry.

VI. Interpass modifications

In the Spring of 1985, I attempted to integrate the parts of the SIMULA compiler. Interpass was running and had successfully processed the entire RTS. One of the FEC passes required too many tags for the available memory on the 3220, but it was hoped that the 8/32, which had slightly more memory, could handle the entire system. Much work had been completed on pass six of PAS32, and it appeared likely that much progress could be made on actually generating code in a short amount of time.

My tasks were to help with the testing of pass six, repair any deficiencies in IP that emerged, maintain tools which had been developed, and work on the later passes.

Interpass changes

Unfortunately, IP was not as close to being finished as it seemed. NCC had been doing a total revision of S-PORT, in the process changing its name to the SIMULA Standards Group, and the new system arrived soon before I started. Included in the rewrite of S-PORT were some changes to the S-Code language itself. My first job was to revise IP to translate the modified input language.

Interpass was written in S/SL [HOL82] a compiler-oriented language developed from its authors' experience with a type of chart used to express the semantics of languages. S/SL is essentially a pure control-flow language without data. It consumes a stream of tokens (numbers) and emits another token stream. Matching and decisions may be based on the current input token, but all other data operations that may be necessary must be implemented via semantic routines written in a host language, in our case Pascal.

S/SL source is compiled into a table in the form of a Pascal constant definition. PAS32 has been extended to allow constant arrays to be defined. This table is inserted into a Pascal source file which contains the S/SL interpreter and whatever semantic

routines are part of the program. This Pascal file is then compiled to yield a runnable S/SL program.

Interpass is quite large. The S/SL portion is more than 5000 lines, with approximately 100 rules, which are the S/SL equivalent to procedures. The Pascal portion is also greater than 5000 lines, with about 300 semantic routines.

Semantic data structures which were needed in Interpass included an expression stack, called the S-stack, which was used roughly in the manner of the expression stack in S-Code. Descriptors of data were large Pascal records, dynamically allocated, with much information about their S-Code meaning as well as details of how they affected the PAS32 output. Record descriptors' fields were kept in a linked list rooted in the descriptor itself, and parameters to routines linked similarly to the profile descriptor.

The current definition of all tags in the system was kept in a large table containing space in each entry for a pointer to the tag's descriptor, and a variable which specifies the state of the tag (defined or not). Numerous stacks of different data are used, including a string stack, tag stack, count stack to perform computations, and stacks of labels, indexes, input file states, and others.

Though the changes to Interpass in the new S-Port system were mostly not involved, I was unfamiliar with S/SL, IP, and S-Code itself, so the process of learning the function of the system made the initial work progress pretty slowly.

The only change which directly caused a significant alteration to the code for IP was the provision for local quantities in modules and the main program. This change was made to reduce the amount of global data required by the S-PORT system, thereby reducing the (large) number of tags used by the FEC and RTS. A new data structure, an update stack, had to be added to IP, which would be used to facilitate

destroying the meaning of local tags at the end of the module which defined them. The contents of each element of this data structure is a pointer to a descriptor containing the meaning of the tag when the local declaration was encountered, and the tag number affected. At the beginning of a module, a new section on this stack is started, and when locals are defined, the previous meaning for the locals is added to the section. The entire contents of the section are removed from the stack at the end of the module, with the original meaning restored.

While studying the code for IP itself, I noticed some minor mistakes which I corrected. One of these was partially responsible for the memory overconsumption. A stack containing labels for output to Pass six was intended to be twenty entries deep, but the wrong constant was inadvertently used in the declaration, and it was in fact 32,767 entries deep. Since the data in the stack was fullwords, this mistake added about 128K bytes to the size of the running program. The amount of main memory on the 3220 is only about 3/4 megabyte, so this is a significant difference.

In addition to the changes to the S-Code syntax itself, some other modifications have been made to S-PORT which have not been officially reflected in S-Code's definition. Features are still defined which are not currently used, some of which are not likely to be used in the future. In one instance, this is fortunate. For the recompilation of one module to be done without requiring recompilation of other modules in a system, there was provided the *existing* option in a module declaration, which replaced the visible declarations before the module body. This was intended to inform the S-Compiler that the implementation of the module had been changed, but the interface between modules had not been modified, and the same visible information could be reused for this module. IP had not implemented this feature, and the task appeared to be quite difficult because of certain aspects of how the visible files were created. It appears likely that that feature of S-Code will be dropped, as the NCC parties are

questioning its usefulness.

Another part of S-Code which is not used and will probably be discontinued altogether is the use of a display to locate objects in the various scopes of a running program. That feature took much effort to design, and much of the effort to date put into pass six was concerned with it.

These and other currently unused S-Code constructs are now treated as errors by IP.

Other IP modifications.

Initially, IP could not handle the new S-PORT system at all, since the insertion operator's parameters had been changed. When the changes outlined above had been made, IP did somewhat better, translating several of the 15 RTS modules without error. But numerous problems arose which did not occur with the old system. Almost all of them concerned details of the insertion mechanism which the new system revealed to be in error.

One problem, besides insertions, was the way that record and repetition constants were verified. Record constants must be checked by the S-Compiler that all fields are mentioned, and that fields in more than one variant are not mentioned in the same constant. IP set a boolean flag in the descriptor of each attribute tag encountered in a record, including those of prefix records. Then, after all the fields had been processed, it scanned only the main record's body to see that attributes had been initialized consistently, clearing the flags. This left flags set in the fields of all prefix records that were processed. When another record constant of the same type or containing the same prefix record was encountered, the flags which had been left on would cause an error, since IP thought that an attribute was being given two values. A more subtle problem besides the erroneous multiple definition problem is that the proper initialization of

prefix attributes was not checked. The solution to these problems was to extend the verification to include a recursive traversal of all the prefix records, checking that these were properly initialized and clearing their flags.

Constant definitions also presented problems. Their definition is of the following form:

const (tag, type) value(s)

The problem arose when a module in the RTS defined an array of character strings which was shorter than the number of elements in the type part of the const definition. The S-Compiler checked that the number of elements was identical to the defined size of the repetition. That restriction had seemed natural, although it was not explicitly stated in the definition. The check for strict size conformance was removed.

The bulk of the problems, however, were precipitated by deficiencies in the method of including visible definitions from other modules. The documentation for the actions taken for insertions is sketchy at best, possibly because the developers of S-PORT considered the issue too system-dependent to be too explicit. I proceeded on the assumption that the implementation was basically sound for some time, which caused much of my time to be spent treating symptoms of the problem rather than the actual illness.

In general, the modules of S-PORT form a hierarchy, with the COMN module and some other modules defining global information which is used by all the other modules, with other modules containing more specialized definitions which are used by some modules and not others. As a consequence, among the fifteen RTS modules, the modules COMN, SYSR, KNWN, BASE, UTIL and STRG are invariably included in that order (with the S-Code for these modules including each of the previous ones). After these five modules, the rest of the information the modules use is varied, with the

maximum number of modules inserted being ten. The FEC passes use declarations in all RTS modules. A descriptive outline of the S-Code of a 'typical' RTS module follows.

```
module x (parms)
  insert y; insert z; ...

  visible definitions

  tag list

body
  code
endmodule
```

Although the syntax definition of S-Code allows *insert* statements to be located almost anywhere in a program, there is invariably a list of module insertions as the very first items in a module definition, and insertions occur nowhere else in the S-PORT system. The S-Code definition admits that the syntax given is 'descriptive,' and not strict.

In designing the original inclusion mechanism, it was believed that the original chain of inclusions would not have to be reconstructed. That is, for module A to insert module B, the insertions that B performed do not have to be known. If the insertions that were performed in A up to the point of the insertion of B were equivalent to the insertions required by B itself, this would indeed be the case. This approach increased the complexity of IP, but made file handling easier, as will be seen later. The S-Code for the visible part of a module definition was copied almost exactly to a file with the module name suffixed by '.v', and the list of visible tags in a separate file suffixed by '.t'. When inserting a visible file, each tag encountered in the file was immediately translated according to its index in the taglist. The immediate translation was the reason for keeping the separate taglist and visible files. If a tag in the visible file was not

visible the construct it defined was simply skipped. Insert statements were always skipped in the visible files. Therefore, if a module being translated contained n insert statements, exactly n visible files (and n taglist files) would be processed.

This method proceeded without problems for all modules with consistent insertions, but various anomalous behavior emerged when the sequence of insertions began to vary. Not until late in the Spring did I realize that the insertion process that had been used was not workable.

One early problem occurred during the insertion of a routine profile. Profiles identify the type and sequence of parameters to a routine, and the tag of the *body* which contains the executable code for the routine in a separate location. Each parameter has a tag associated with it which is accessible only within the body. The particular profile tag was visible, and its parameters were not. By chance, the profile tag, when translated as above, took on the same value as one of its parameter tags. The data structure for the descriptor of a profile included a linked list of its parameters, and when operating on the parameters, their tag values are manipulated on a stack of tag values. The information describing the parameter is expected to be in a table of descriptors indexed by tag value. The one tag value could not have two meanings simultaneously, so the system failed. The method of implementing profile descriptors, using tag values for the parameters, was suspect in any case, since the parameter tags themselves are not supposed to be meaningful outside the body of the routine anyway. To fix this required redefining the descriptor and code which dealt with parameters to use the descriptors of the parameters rather than their tag values.

After I discovered and fixed several problems of this nature, IP finally could translate the entire RTS. But it crashed when in the midst of inserting RTS modules into the FEC passes. The error was one that could not be mistaken. When PASS1 inserts the module SML, the previous inserts of PASS1 leave tag number 697 with the

meaning of a routine body. The inserts within SML, however, give tag 697 the meaning of a record, which it uses as a prefix for a new record defined in its visible part. Since SML's insertions are assumed to be unnecessary, IP tries and fails to use the tag as PASS1 has defined it.

So when inserting a file, its insertions' effects have to be taken into account. A total redesign of the way insertions are handled is needed to solve this problem.

Two possible directions to take were discussed. The first, and simplest, was to actually perform insertions as they appeared in visible files, thus rebuilding the entire tag table to the state that the inserted file expected. While simple in concept, this method is impractical due to the great amount of file activity it requires. I wrote small program to simulate the visible file processing using the dependencies in the actual S-PORT modules, noting the total number of files read. To translate S-PORT (fifteen RTS modules and the two FEC passes) required 5564 passes through the various visible files. Obviously, this is a heavy performance penalty to pay.

The other option that was possible without changing the practice of keeping visible files in S-Code format was to develop a scheme to add the necessary information to visible files to avoid having to look at the other modules' files upon which they depend. This plan is very complicated, as a method of adding that information to the visible files is not obvious. It is not clear that the performance of such a system would be acceptable, either. It may well be that many redundant file reads would be replaced by much redundant information in fewer, but huge visible files.

As a compromise, I designed a method for performing the insertions that is an optimization of the first method above. While 5564 file reads were needed for the 'brute force' method above, the number of files involved for the entire S-PORT system is only seventeen. If a way could be devised to remember the information in each visi-

ble file as it was processed so that it could be used when needed later by another insert statement, the amount of I/O activity could be mostly eliminated. And if the information were stored in a form that could be easily used (ie - with little CPU time), the performance of Interpass would be comparable to that of the present system.

My design is for a list of repositories of visible definitions, called Visible Save Areas (VSA's). Each time a visible file is processed for the first time in a compilation, a new element in the VSA list is created which will allow the definitions of that module to be later accessed without reference to that modules visible file, or any visible files it may insert. Each VSA is implemented as a list of pointers to the descriptors of the quantities for each index in the tag list. To simulate the processing of a given visible file, with all its associated dependencies, one has only to copy the descriptor pointers in its VSA into the proper locations in the tag table and mark them as defined.

The update stack, which holds lists of tag table entries, is also used in this design. The method of processing inserts is illustrated referring to the code segments below.

```
module A
   <no insertions>

   <definitions>

  taglist
   tag t0 0
   ...
   tag tn n

  body
   <code>
  endmodule

module B
  insert A i i+n+1

  ...

  taglist
   tag t0 0
   ...
   tag tm m

  body
   ...
  endmodule

module C
  insert B j j+m+1

  ...

  taglist
   tag t0 0
   ...
   tag tl 1

  body
   ...
  endmodule
```

These modules must be compiled in the order A, B, C so that the visible files will be available for the insertions. Assuming that A and B have been compiled successfully, the translation of C proceeds as follows:

1. Insert(B, j, j+m+1) is encountered. This module is not represented in any VSA, so its visible file must be read. A new VSA is created, with space for m descriptors. (The size is known from the parameters to the insert operator.) Also the update stack is marked to allow the tagtable to be restored after the insert processing.

The visible file for B is located and prepared for processing.

2. Within B's visible file, insert(A, i, i+n+1) is read. A has no VSA, so one must be created. A new VSA is created like above, processing is suspended on B, the update stack is marked, and A's visible file is readied for reading.

3. A's visible file is processed, and its definitions are placed in the tag table. The old tag table entries are saved on the update stack as they are replaced. (In this case, none of the tags have been defined yet.) When A's tag list is found, at the end of its visible file, each listed tag's description is placed in the corresponding VSA location. At the conclusion of A's tag list, the VSA is marked as complete, and the top section of the update stack is removed with the tag table's meanings restored to the state before this step.

4. Now we resume processing of B's visible file. Using A's new VSA, the tag table is updated to include the proper definitions for tags i to i+n+1. Since we are still processing a visible file, the old meanings for these tags are saved on the update stack. The rest of B.v may now be processed, as the tagtable now holds the definitions it expects.

5. The conclusion of B's visible file is handled in the same manner as that of A in step 3 above. B's VSA is completed and the tag table definitions resulting from B's visible file processing are destroyed.

6. Processing of A's S-code now resumes. B's VSA is used to update the tag table in the range j to j+m+1. The update stack is not used for this since we are operating at the outermost level and the definitions are permanent. Now processing of C may con-

tinue to completion.

In fact, more optimization is included in my design to eliminate even the use of the VSA in the case where that operation would be redundant. A data structure contains information about the insertions performed at each level, so that if an insertion is indicated that is identical to one already reflected in the tag table, the tag table is used without change. The insert statement in question is effectively skipped.

Coding for this design is mostly completed, but has not been added to Interpass at this time. The table below summarizes the effects of the two optimizations on the compilation of the complete S-PORT system.

	brute force	VSA optimization	Eliminating redundant VSA
Modules translated	17	17	17
Visible files read	5546	115	115
Update stack contents(max)	4877 tags	1437 tags	722 tags
Insertion nesting	9 levels	2 levels	2 levels
VSA operations performed	xxxx	411	77

VII. Future work and conclusions

The first thing to accomplish in the future is to finish the revision of Interpass to correctly deal with insertions. It is not clear whether the current translation of the Run-time system is actually correct, or whether by chance no incompatible declarations were encountered. Therefore any results of testing pass six with the current Interpass may be misleading. A review of the design of Interpass itself might be desirable, due to the nature and scope of changes which have been made to it recently.

Changes have been made as well to the input language to pass six of PAS32. Some of these changes are not trivial, and probably a careful review of these changes should be made. The code for the display manipulation, which is not a part of S-PORT, should be removed. In walking through some sample input, it was apparent that the optimizations that pass six performed were not applicable to many of the constructs it is given by Interpass. In fact, some translated S-Code caused the optimizations to lose some code. On the other hand, it was obvious that much other optimization was possible. This would be a good area to explore. It also probably would be advisable to remove a great deal of dead code from pass six. The original intent was to have a single source document which, by conditional compilation, would produce a program which would translate either SIMULA, PAS32, or Concurrent Pascal code. The output from Interpass is dissimilar enough from PAS32 intermediate code that it would be easier to have a separate source for the Pascal and SIMULA languages.

It would seem that the highest priority would be to produce some object code from S-Code programs. It is unfortunate that the design requires so many separate pieces of software to be completed to perform that task. This problem was compounded by the revisions to the language made by the Norwegian Computing Center.

Conclusions

One of the most important design decisions was to use the PAS32 backend passes for code generation rather than writing a new code generator. The major reason for that decision was to enable the code generator for S-Code to be developed with less effort and technical expertise than would be possible otherwise. In the negative sense, though, it complicated the development process in other ways. To generate object code requires that Interpass and the backend PAS32 passes to be written or modified and debugged. This is not nearly the case at the present time. Furthermore, significant changes to the design have been necessary at times due to changes in the input language as well as due to misunderstandings of the semantics of S-Code. The parallel effort on the various parts of the code generator was hindered by the filtering of design changes through the various passes.

The scope of the changes to PAS32's passes was underestimated as well. PASS 6, with which I am most familiar, performs various optimizations to its input code, such as constant folding. These optimizations in many cases do not succeed using the input from Interpass. Not only is the output code from PASS 6 inefficient, it is incorrect. PASS 6 simply throws away some code that Interpass produces. These aberrations are due to the fact that the new input language is really significantly different from that which is produced by the front-end passes of PAS32. A careful study of the output language of Interpass and design of optimizations to be incorporated into PASS 6 will be necessary. In essence, most of the new PASS 6 will be similar in form to the existing Pascal version, but its optimizations will be quite different.

Another negative aspect of the use of PAS32 is performance. PAS32 was designed to be useful on machines with very limited memory. This is a holdover from the PDP-11 implementation of Hartmann, which had severe memory constraints. The sacrifice made to save memory is in speed. Large quantities of disk I/O are performed

in the process of translating programs. In a potentially commercial product, this performance penalty may be too troublesome.

Whether these drawbacks mean that writing a new code generator would have been wiser is an interesting question. Clearly writing a code generator is not an easy task. I tend to think that considering that the compiler was intended to be used in a production environment, it would be advantageous to make it as efficient as possible. A one or two-pass code generator would perform better than the five-pass design that was used. The design would be more unified in such a case, lowering the trickle-down impact of the changes that were made in the process of development. Another avenue that might have been considered would be to make use of the code generator of the portable C compiler [JOH79a], which would allow for greater portability of the S-Port system across various UNIX implementations and machines.

Another important aspect of the design was the use of S/SL for the intermediate translator. S/SL has shown some nice characteristics. It is a simple language, and development of S/SL programs can be rapid. Parsers may be generated routinely from a syntax description, for example. S/SL programs are translated into interpreted code that is more compact than the tables produced by automatic LALR parser generators such as the Lex and Yacc combination [LES79, JOH79b], which is available in the UNIX environment. Interpass, however, required quite involved semantic information to be manipulated underneath S/SL. The Pascal semantic routines numbered in the hundreds. I think that a well-written and documented Pascal or C program would have been easier to understand for a translator of the scope of Interpass.

The S-PORT system itself should be of great help to those who wish to write a SIMULA compiler with the least effort possible. Some of the more useful utilities, such as the Simob run-time debugging system and the SIMULA source formatter, are not currently implemented, but when they are added to the system, it will be a useful

development package.

Some parts of S-PORT's documentation are needlessly obscure, which added to the confusion of the meaning of certain aspects of S-Code (especially the insertion question). Communication of the semantics of complicated features is made more of a problem due to the slowness and expense of dealing with the system's designers over such a long distance. The original design for the insertion mechanism was based on the answers to specific questions about 'transitive insertions,' so either the semantics have changed, or the question was misunderstood by the Norwegian Computing Center, since the method was not in line with the usage of the insert token in S-PORT.

References

ADA79. J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brückner, and B.A. Wichmann
"Rationale for the design of the Ada Programming Language,"
ACM SIGPLAN Notices,
14:6, June 1979

DAH82. O. J. Dahl, B. Myhrhaug, and K. Nygaard,
SIMULA 67 common base language,
Oslo, Norway: Norwegian Computing Center, 1982

FRA77. W. R. Franta,
The Process View of Simulation,
New York, NY: North-Holland, 1977

HAN75. Per Brinch Hansen,
"The Programming Language Concurrent Pascal,"
IEEE Transactions on Software Engineering,
1:2, June 1975, pp 199-207

HAR77. Alfred C. Hartmann,
"A Concurrent Pascal Compiler for Minicomputers,"
Lecture Notes in Computer Science,
New York, NY: Springer-Verlag, 1977

HOL82. R. C. Holt, J. R. Cordy, and D. B. Wortman,
"An Introduction to S/SL Syntax/Semantic Language,"
ACM Transactions on Programming Languages and Systems,
4:2, Apr 1982, pp 149-178

JEN83. P. Jensen, et al.
Definition of S-Code ver 3.1,
Oslo, Norway: Norwegian Computing Center, 1983

JOH79a. Stephen C. Johnson,
"A Tour Through the Portable C Compiler,"
UNIX Programmer's Manual, Seventh Edition,
Murray Hill, NJ: Bell Telephone Laboratories, Jan 1979

JOH79b. Stephen C. Johnson,
"Yacc: Yet Another Compiler-Compiler,"
UNIX Programmer's Manual, Seventh Edition,
Murray Hill, NJ: Bell Telephone Laboratories, Jan 1979

KER78. B. W. Kernighan and D. M. Ritchie,
The C Programming Language,
Englewood Cliffs, NJ: Prentice Hall, 1978

LES79. M. E. Lesk, and E. Schmidt,
"LEX - Lexical Analyzer Generator,"
UNIX Programmer's Manual, Seventh Edition,
Murray Hill, NJ: Bell Telephone Laboratories, Jan 1979

MIL83. G. Millard, O. Myhre, and G. Syrrist,
S-PORT The Environment Interface ver. 3.1,
Oslo, Norway: Norwegian Computing Center, 1983

NAU63. P. Naur, ed
"Revised Report on the Algorithmic Language ALGOL 60"
Communications of the ACM
6:1, 1963, pp 1-17

PER75.
Model 8/32 Processor User's Manual,
Publication 29-428 R01,
Perkin-Elmer Corporation, 1975

PER79.
Model 3220 Processor User's Manual,
Publication C29-693 R00,
Perkin-Elmer Corporation 1979

PER80.
Common Assembly Language (CAL) Programming Reference Manual,
Publication S29-640 R04,
Perkin-Elmer Corporation 1980

RIT78a. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan,
"The C Programming Language,"
The Bell System Technical Journal,
57:6, Jul-Aug 1978, pp 1991-2019

RIT78b. D. M. Ritchie, and B. J. Thompson,
"The UNIX Time-Sharing System,"
The Bell System Technical Journal
575:6, Jul-Aug 1978, pp 1905-1929

THO78. B. J. Thompson,
"UNIX Implementation"
The Bell System Technical Journal
57:6, Jul-Aug 1978, pp 1931-1946

WIR71. Nicholas Wirth,
"The Programming Language Pascal"
ACTA INFORMATICA
1, pp 35-63 (1971)

WIR85. Niklaus Wirth
Programming in MODULA-2, 3rd ed.
New York, New York: Springer-Verlag, 1985

YOU80. Robert Young and Virgil Wallentine,
"PASCAL/32 Language Definition"
Department of Computer Science
Manhattan, KS 1980

Adapting a Portable SIMULA Compiler to Perkin Elmer Computers in a UNIX Environment

by

Gregory L. Dietrich

B.S., Kansas State University, Manhattan KS, 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

The implementation of a compiler for the language SIMULA-67 is described. The target machines are those in the Perkin-Elmer 32-bit series, with the operating system environment Version 7. S-PORT, a portable SIMULA system designed by the Norwegian Computing Center, was the basis for the project. The language SIMULA itself is briefly described, and some of its interesting features. The portable SIMULA compiler system, the design of the KSU implementation, and my contributions to the compiler are then described, as well as the current state of the project.