

303

/THE IMPLEMENTATION OF A SIMULA COMPILER ON THE
KANSAS STATE UNIVERSITY PERKIN-ELMER COMPUTERS/

by

LOWELL RICHARD LINDSTROM

B. S., University of Kansas, 1959

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

LD
2668
.R4
1986
L566
c.2

A11202 663805

TABLE OF CONTENTS

Introduction	1
1. THE SIMULA LANGUAGE -- AN OVERVIEW	2
2. THE S-PORT SYSTEM	5
3. THE PROJECT -- AN OVERVIEW	7
4. OVERVIEW OF PASSES 6, 7, 8, and 9.	9
Pass 6	9
Pass 7	14
Pass 8	20
Pass 9	25
5. NEW AND MODIFIED PROCEDURES FOR SIMULA	27
6. PROJECT STATUS AND FUTURE WORK	44
Bibliography	46
Appendix A: Coroutines	47
Appendix B: A Sample Token Stream for Pass 6	51

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

LIST OF FIGURES

Figure 5.1	New graph structure - DUP token.	32
Figure 5.2	New graph structure - DCOPY token.	32
Figure 5.3	Save Object.	41
Figure A	Coroutines	50
Figure B.1	Treelement	52
Figure B.2	Beginning of Pass 6 stack.	53
Figure B.3	STARTLIST token.	53
Figure B.4	PARM token	53
Figure B.5	ENTER token.	54
Figure B.6	DEFLABEL token	54
Figure B.7	PUSHADDR token	55
Figure B.8	PUSHCONST token.	56
Figure B.9	FIELD token.	57
Figure B.10	ADD token.	58
Figure B.11	PUSHIND token.	59
Figure B.12	PUSHCONST token.	60
Figure B.13	COMPARE token.	61
Figure B.14	NOT token.	62
Figure B.15	FALSEJUMP token.	63

INTRODUCTION

During the Spring semester, 1983, I participated in a project to implement a SIMULA [2] language compiler on the Perkin-Elmer computers at Kansas State University. Dr. Rodney M. Bates was the project director. At this time the project is not complete in that the compiler is not operational. I have been responsible for two of the five originally identified tasks. During the initial semester my responsibilities were to modify passes 6, 8, and 9 of the operational PAS32 Pascal compiler [4] to accept the new SIMULA code and produce machine instructions. One year later I became responsible for modifying pass 7 of that same compiler, in another course. The other three tasks were to write the ENVIRONMENT INTERFACE, to develop testing and debugging tools, and to write an INTERPASS for translating the S-CODE into a token stream acceptable to pass 6 of the PAS32 compiler.

This report contains overviews of the SIMULA language, the portable SIMULA compiler [2] obtained from the Norwegian Computing Center, and the project itself. It then discusses in some detail the data structures and processes used by the four passes of the PAS32 compiler. Finally, the modifications to those passes are discussed, enumerating each new compile time token introduced by the SIMULA system and the resulting changes to existing code needed for each token.

CHAPTER I

THE SIMULA LANGUAGE -- AN OVERVIEW

SIMULA is a block structured language based on ALGOL 60. It has powerful list handling and sequential processing capabilities. Its compiler precludes many run time errors and much run time debugging by recognizing problems and not executing the invalid use of data through referencing based on wrong assumptions. Because of its block structure, decomposition is easy. Decomposition means dividing a large problem into smaller more manageable problem components. Each block of a SIMULA program is a "...formal description, or 'pattern', of an aggregated data structure and the associated algorithms and actions." [6] The block is completely independent of the rest of the program as far as local variables are concerned. This makes the block a smaller, more manageable component of the larger, less manageable problem.

SIMULA's most distinctive features are its CLASS and OBJECT. A CLASS is a block of code which is declared much like a procedure. It contains local variables, some procedures which operate on those variables, and probably some executable statements. OBJECTS are instances of CLASSES. There may be many instances of a CLASS in existence at one time. Unlike a block instance in Pascal, a CLASS instance in SIMULA can remain in existence as long as there is a reference to it. It may even remain in existence after the block instance that called it.

Even after the end of its block of code has been reached it will survive as an item of data.

Identification of individual OBJECTS is through a REFERENCE variable. That is, to allow reference to a generated object the reference to that object must be stored in a variable. Through REFERENCE variables the attributes of a CLASS instance may be assigned to or inspected. An example of how referencing is initiated is as follows:

```
CLASS C(...); ...class body for C...;
  ref(C)X;
....
  X :- new C(...);
```

First the CLASS C is declared, with or without parameters. The REF statement declares X as a reference variable for the CLASS C. Then a new instance of the class C is assigned to X. (":-" is a SIMULA symbol denoting reference assignment.)

One of the strong features of SIMULA is its processing of COROUTINES. COROUTINES do not have the master/subordinate relationship of programs and subroutines. COROUTINES are two objects operating at the same level. A typical COROUTINE object may have this history.

(1) The object is called by some other object. It is said to be "operating and attached". At this point it is subordinate to the calling object.

(2) The object issues a DETACH statement which returns control to the calling object. Reentry to the calling object will be at the point it was exited. The called object is said to be "detached but not yet terminated".

(3) Control returns to the object when a call statement or a RESUME statement is executed, specifying this object by its reference variable. It is said to be "reattached" to the calling block instance if called or to the original caller if RESUMED. If RESUMED, execution will recommence at the statement after the exit point of the DETACH.

(4) The object will also relinquish control when the end of its code is reached. It cannot then be reactivated by a call or RESUME; however, it may still be referenced as an item of data. It is said to be "terminated", but as long as there is reference to it in the program, it will not disappear. When all of the references to it have been used, it will disappear. At this point garbage collection is necessary to reclaim memory. (A more complete explanation of coroutines will be found in Appendix A.)

A series of RESUME statements activating one object after another puts the object, "on the same level", as opposed to the master/subordinate relationship.

There are predefined system CLASSES for special purposes. One such is SIMULATION which is intended to be used for discrete event simulation modeling.

CHAPTER II

THE S-PORT SYSTEM

To implement a SIMULA compiler on the Perkin-Elmer computers it was necessary to obtain from the Norwegian Computing Center a portable SIMULA system. This system is called S-PORT. It consists of a portable front-end compiler and a runtime system. The front-end compiler translates SIMULA source code into a language called S-CODE. S-CODE is an intermediate language meant to be passed to a back-end which generates machine dependent code. The runtime system is linked to the compiled program and at run time provides such services as garbage collection and error handling. Both the front-end and the runtime system are distributed by the Norwegian Computing Center as S-CODE for portability.

The back-end of the compiler must be written specifically for the computer on which the compiler is to be implemented. It must consist of a code generator and an ENVIRONMENT INTERFACE. The code generator is to take S-CODE and translate it into machine language. The ENVIRONMENT INTERFACE is completely described by the Norwegian Computing Center. [3] It is a support system which supplies an interface to the host operating system. It provides workspace, the processing of control data, the input/output routines, the initial interpretation of error conditions, and the processing of diagnostic records. The S_CODE programs contain calls on the procedures in the ENVIRONMENT INTERFACE so that the compiled program can properly interface with the

runtime system and the operating system.

CHAPTER III

THE PROJECT - AN OVERVIEW

The Kansas State University project to implement a SIMULA compiler on its Perkin-Elmer computers involved four major tasks. These were to write the ENVIRONMENT INTERFACE, to write an INTERPASS for translating the S-CODE into a token stream acceptable to pass 6 of the PAS32 compiler, to develop testing and debugging tools, and to modify passes 6, 7, 8, and 9 of the PAS32 compiler. The modification of the PAS32 passes were all accomplished by this author during different semesters. The PAS32 compiler is a Pascal compiler already on the Perkin-Elmer computers. Passes 6, 7, 8, and 9 are the final major passes of that compiler and produce the machine code.

The ENVIRONMENT INTERFACE has been described in an earlier chapter. This package of routines was written in the C language to take advantage of C's ability to interface with the UNIX operating system.

The INTERPASS was necessary to translate the S-CODE into a token stream acceptable to pass 6 of the PAS32 compiler. A major portion of INTERPASS was written in Syntax/Semantic Language (S/SL). [5] The remainder was written in Pascal.

The testing and debugging tools were written in Pascal. They include assemblers and dumpers for the token streams. The assemblers take an input stream of tokens manually developed for

testing a pass and convert it into the binary numbers the compiler expects. The dumpers perform the opposite task of converting the binary numbers output by a pass into mnemonic token streams. Thus test data may be created with mnemonics which are more easily understandable than number strings and then converted to "the language the computer understands". The output may be converted from number strings to mnemonics for easier debugging.

CHAPTER IV

OVERVIEW OF PASSES 6, 7, 8, AND 9

The remainder of this report deals with that portion of this project which were my responsibilities. This chapter describes passes 6, 7, 8, and 9 of the PAS32 compiler. This is a Pascal compiler already operational on the Perkin-Elmer computers. The project design calls for modification of these passes in lieu of writing new code generating passes for the back-end.

PASS 6

Pass 6 of the PAS32 compiler is designed to perform constant folding, i.e., perform operations on constants at compile time rather than run time. It also performs other optimizing operations since it is the first pass of the compiler to build in-memory data structures. We are concerned with those data structures.

Pass 6 builds an in-memory data structure of one procedure at a time. Most optimizations are performed as the data structure is being built. Some optimization is performed as the data structure is being traversed for output to pass 7.

The data structure used is a stack of lists of trees. The basic element in the trees is the TREEELEMENT record. (See Figure B.1, Appendix B) This record contains a next pointer (NP) for linkage on the linear list and left and right pointers (LP & RP) which point to the respective substructures. The TREEELEMENT

record also contains the arguments for each token and the token itself. One TREEELEMENT node is created for each input operator including data item representations such as PUSHVAR or PUSHCONST.

The stack is an array of records. The fields of the records contain pointers to the first and last elements of a list of statements which have been linked with NEXT pointers. The basic operations on the stack are PUSH, POP, APPEND, APPENDSTK and LINK. The PUSH procedure places a new stack element on top of the stack. This element is a record whose fields are HEAD and TAIL. HEAD and TAIL are pointers to the new TREEELEMENT. The POP procedure removes n elements from the stack. APPEND sets the NP of the TREEELEMENT pointed to by TAIL to point to the current TREEELEMENT being processed. APPENDSTK causes the TAIL of the second element on the stack (SOS) to point to the TREEELEMENT pointed to by the HEAD of the top element on the stack (TOS). The TOS is then "popped" leaving its element as part of the linked list of SOS. The LINK procedure accepts the pointer to the element being processed and a count of the number of stack elements being involved in the current operation. If two elements are involved LINK causes the LP of the current operation to point to SOS and the RP to point to TOS. Two elements are popped and the element of the current operation is pushed.

The basic operations on the lists of TREEELEMENTS are performed by UNARYTREE, BINARYTREE, UNARYAPPEND, BINARYAPPEND, APPENDINPUT and PUSHINPUT. Each of these procedures call or cause to be called the READARGS procedure, which creates the new

TREEELEMENT, places the constant token of the operation in the OP field, initializes the NP, LP, and RP fields to nil, and reads and loads the appropriate number of arguments into fields in the element. UNARYTREE and BINARYTREE examine the elements pointed to by TOS and SOS and the current operation to see if optimizations may be performed. If they do not perform optimizations, UNARYTREE will perform a LINK(1). BINARYTREE will perform a LINK(2). UNARYAPPEND and BINARYAPPEND first call UNARYTREE or BINARYTREE respectively, and then perform an APPENDSTK. This causes the tree to be built and it appends the tree to the stack as the TAIL element. APPENDINPUT causes the NP of the TOS TAIL element to point to the current operation, thus adding it as a list member. PUSHINPUT simply pushes the current operation's element on the stack.

Figures B.2 through B.15 in Appendix B illustrate the actions taken by pass 6 as a sequence of tokens is input. Starting with Figure B.2 there are no TREEELEMENTS present. The top of the stack (TOS) is 0 and both HEAD and TAIL fields are initialized to nil. The first token read is STARTLIST which pushes a record onto the stack with its HEAD and TAIL fields at nil. The next token is PARM (Figure B.4). The call on APPENDINPUT causes a new TREEELEMENT to be created by READARGS. LP, RP, and NP are initialized to nil and the five arguments associated with the PARM token are read from the input file and placed in the argument fields of the TREEELEMENT record. Pointers to the PARM TREEELEMENT are placed in TOS.

In Figure B.5 the ENTER token has been processed. Again, READARGS creates a new TREEELEMENT record, initiates its LP, RP, and NP and reads its associated arguments into its argument fields. The UNARYTREE procedure places a pointer to the PARM element into ENTER's LP, pops the TOS which pointed to PARM and pushes a record with HEAD and TAIL pointing to ENTER. The APPENDSTK procedure examines the second record on the stack (SOS). Since both of SOS's fields are nil, pointers to ENTER are placed in SOS's fields and TOS is popped. The stack is now at 0.

The next token in the input stream is DEFLABEL (Figure B.6). After READARGS creates and initializes its TREEELEMENT fields, the APPENDINPUT procedure calls APPEND which, since TOS's fields are not nil, places pointers to DEFLABEL into the TAIL field of TOS and the NP field of ENTER.

Each of the remaining figures illustrate the state of the stack and the state of the tree after each operation has been processed. The procedures invoked by each operation are listed on their respective illustrations. Note in Figure B.15 that when the subtrees of FALSEJUMP are all linked, a pointer to FALSEJUMP is in the TAIL field of TOS, and a pointer to ENTER is in the HEAD field. Always, when a tree is complete, the HEAD field will point to the beginning of the list and the TAIL field will point to the end of the list.

When the data structure has been built, it is traversed by the TRAVERSE TREE and WALKTREE recursive procedures. Because of the NP in the structure this is a modified postorder traversal.

Each node is visited first by visiting the node pointed to by LP. Then the list pointed to by RP is visited. The node itself is output and the NP of the node is visited along with all of its nodes. The result is that the input to pass 7 will be the same as that for pass 6 except where optimizations have occurred to alter the token stream.

PASS 7

Pass 7 generates the machine instructions, with the exception that it can not choose the best form for some of the instructions, in which case that decision is deferred to pass 8. Pass 7 handles register allocation and assignment, and it generates symbolic references to code, procedure, and statement labels, to literal constants and to stack depths.

The output of the generation of machine instructions consists of an instruction op code, a short set which represents any of eight different operation types, and the various fields of the instruction. The instruction op code is one of the Perkin-Elmer machine codes. The different operation types are RX, RR, and RI, which are standard instruction formats, and RXL, RXR, RXC, RXU, and RIS. RXL is an RX instruction which references a code label. RXR is an RX instruction which references a procedure label. (These are referred to in the code as routine labels.) RXC is an RX instruction which refers to a long constant. RXU is an RX instruction which refers to a user defined statement label. RIS is an RI instruction which references a routine's maximum local stack depth.

There are eleven data types which may appear in the output stream in the short set. BYTE indicates a constant of one byte. HALFWORD is a constant of two bytes which must be halfword aligned. FULLWORD is a constant of four bytes which is to be fullword aligned. TWO_REGS and THREE_REGS indicate that an in-

struction contains two or three registers. `STACK_REF` says that the displacement field of the instruction is a stack reference. `ABSOLUTE` says that the displacement field of the instruction is an absolute value. `LITERAL` indicates that the displacement field is a long constant reference. `CODE_LABEL` says that a displacement field is a code label reference. `STMT_LABEL` says the displacement field is an inter-procedural label reference.

To facilitate code generation there are three instruction mapping tables used. The `BASE_OP` table maps an op code into its base code. For example, `L` is the base op code for all of the load instructions and `ST` is the base op code of all the store instructions. The `BASE_TYPE` table maps data types into instruction types. For example `WORD_BASE` is the base type for a word type, set type or structured type instruction. The `BASE_FORM` table maps instruction types into base instruction forms. For example, `RX_BASE` is the base instruction form for `RX`, `RXL`, and `RXR` instructions.

The `INIT_SPECIFIC_OP` procedure initializes each of these tables to their sets of specific values. Thus if `L` represents a constant of value 1, `BASE_OP[L] := L_BASE`, i.e., `BASE_OP[1] := L_BASE`. Similarly `WORD_TYPE` has a constant value of 2; therefore, `BASE_TYPE[2] := WORD_BASE`. An `RX` instruction has a `BASE_FORM RX`; `RX` has a constant value of zero; therefore, `BASE_FORM[0] := RX_BASE`. The values in these three tables are used to index a three dimensional array, `SPEC_OP`.

The `SPEC_OP` table is initialized in the procedures

INIT_SPECIFIC_OP2 and INIT_SPECIFIC_OP. This table contains the constant values of the Perkin-Elmer instruction set. Using the examples above, a SPEC_OP with an index of L_BASE, WORD_BASE, RX_BASE (SPEC_OP[L_BASE, WORD_BASE, RX_BASE]) will have a value of 1 which is the L instruction. The SPEC_OP table is used to generate instructions which have both RX and RR versions, or which have different versions for different register sets.

The SPEC_OP table is referenced by the SPECIFIC_OP function which has been called by the PUT_GEN_OP procedure, which may have been called by any number of procedures, one of which is PUT_RX. These three procedures will be discussed later. First it will be beneficial to know more about the attribute stack.

The attribute stack generally simulates the contents of the virtual stack at runtime. It is implemented as a linked list of ATTRIBUT records. Two pointers, TOP and SECOND are maintained to point to the top and second records on the stack.

Each attribute record contains a link field pointing toward the bottom of the stack, a type field and a kind field. The type field will identify the type of object this attribute describes, i.e., byte, halfword, word, shortreal, real, or structured. The kind field describes the kind of object. If the kind is expression, then the expression value is contained in a register EREG.

If the kind is variable the fields of the attribute are contents, state, register, displacement, and level. The contents

field designates whether the stack element contains the address or value of the variable being described. The state field tells how the object is to be addressed. This can be direct, indexed, or indirect. The register field designates the addressing or index register to be used for addressing the object. The displacement field is the displacement for addressing the object, and the level is the absolute, global, or local addressing level.

If the kind is constant there are two fields to designate the size of a constant. A short constant will be an integer value and a long constant will be a displacement with a value coming from CONST_INDEX.

Code generation is effected by three different procedures. PUT_CODE is the lowest level of these. It is called with the instruction type, op code, three registers, and displacement as arguments. This procedure writes the instruction on its short set into the intermediate output without modification. It outputs the op code exactly as it was received.

PUT_GEN_OP calls PUT_CODE, but before it does it calls the SPECIFIC_OP function so that the right op code will be output. The parameters passed to PUT_GEN-OP are op code, operand type, instruction form, three registers, and displacement. As an example of this sequence PUT_GEN_OP(A2,WORD_TYPE,RX,R1,R2,00) will generate PUT_CODE(RX,A2,R1,R2,0,0).

PUT_RX is a higher level routine in that it is passed a base op code, a register, and an attribute. It uses the kind

field of the attribute to determine whether the object is a variable, set expression or constant. If it is either of the first two, it uses the level field to determine whether to use the global base, procedure base, or local base register. It then uses the state field to determine the index registers. PUT_RX calls PUT_GEN_OP which then calls PUT_CODE to output the completed instruction.

There are three sets of registers available on the Perkin-Elmer: general purpose, real and shortreal. Whether a specific register is available for use at a given time is determined by its state as recorded in the REG_STATE table. This table is two dimensional, register types by register numbers. Each register set contains 16 registers. The state of each register is "free" or "assigned". Except registers which are never available to the programmer, all registers are initialized as "free". The procedure NEW_REG is used to locate available registers. It searches whichever register set that was asked for until it finds a free register. It then calls MARK_REG which will change the register to assigned. The procedure FREE_REG marks assigned registers as free when it is passed the register number and set by a calling procedure.

The allocation of storage for temporaries on the runtime stack is handled by the procedures NEW_TEMP, MARK_TEMP, and SET_TEMP. NEW_TEMP is passed the length and alignment of the temporary to be allocated. It finds the proper alignment for the start of the temporary and returns that as its displacement. It

also increments the global displacement variable TEMP_TOP. TEMP_TOP marks the space allocated for temporaries at any time. MARK_TEMP will return the current value of TEMP_TOP to a calling procedure. SET_TEMP sets TEMP_TOP to a specific value. SET-TEMP is generally considered done when a temporary has been used and is to be released. A call on SET_TEMP with the displacement value of the beginning of this temporary will release the space.

SET_VAR_ATTR, SET_EXPR_ATTR, SET_CONST_ATTR, and SET_SET_ATTR are used to place values in fields of attributes.

GET_IN_REGISTER and GET_IN_GIVEN_REGISTER work together to load the data or address represented by a specified attribute into a register (EREG), free any other registers used by the attribute and update the attribute to reflect the expression. GET_IN_GIVEN_REGISTER actually generates the code to load the register.

The MARK_ADDR procedure uses the fields of an attribute to generate an address.

PASS 8

Pass 8 accepts the code generated by pass 7 and performs a variety of peephole optimizations. It also makes final choices for the instruction sizes. Pass 8 is the first pass to know the size of all machine instructions. It generates six tables to be used in resolving addresses and in placing constants into the constant area.

The basic data structure of pass 8 is a linked list. Pass 8 accepts one block of code at a time. In doing so it builds in memory a list of elements with one element of the list for each instruction input. The fields of the elements of the list are NEXT and LAST pointers to the preceeding and succeeding element respectively, the line number, and the kind of element. Depending on the kind of element there will be a number of other fields in the element. These fields are the instruction field passed from pass 7 as the op code and short set. When new elements are read by pass 8 they are always placed on the end of the list.

For all but a few of the op codes, the elements and their fields are immediately put on the list. Some of the exceptions and the resulting actions are: 1) EOM which updates the stack size to be passed to pass 9. 2) DEF_LAB which checks to see if the label has already been defined. If not, a new element is created and placed on the list. 3) DEFSTLAB which defines a cross routine statement label and places the element on the list.

4) DEF_PROC which defines a new procedure entry and places an element on the list. 5) EXT_PROC which places on the STACKTABLE the size of an external or prefix routine, if this procedure's size is greater than that already on the stack for this procedure or if the size of this procedure had not already been entered. 6) LINE_NO which reads into the current line number variable the line number being processed. 7) LIT_CONST which places long constants into the constant table and updates the LITTABLE with the displacement of this constant. This will be discussed later in more detail. 8) PROC_INSTANCE which increments the STACK_SIZE by the length of this block, taking into account its alignment. 9) EXTID which adds an external name to the ESDTABLE at the displacement found as an argument with the op code. 10) PROC_STACK which initiates most of the pertinent processing of this pass. This will be discussed later.

The six tables used by this pass are the STACKTABLE, BLOCKTABLE, STLABTABLE, ESDTABLE, CONSTTABLE, and LITTABLE. These tables are all records of arrays of integers divided into 100 integer portions. Items are placed in the arrays and their displacements tracked. Should more than 100 integers be placed in one portion of a table, another portion is created.

The STACKTABLE contains the size of each block entered into the array at displacement "procedure number". Each procedure has been identified by a unique number assigned by a previous pass. These numbers start at 1 and are incremented for each procedure identified. Since any procedure number may occur

more than once in the input to pass 8, this latest procedure's size will be entered into the table only if it is the first time pass 8 has seen this procedure or if the size is greater than that previously entered.

The BLOCKTABLE records the value of the instruction counter at the time the block or procedure is being written to the intermediate output. This value is also placed at the displacement of the procedure's number.

The STLABTABLE records the value of the instruction counter as this statement label is being written to the intermediate output. The displacement in the table is the statement label's number.

The ESDTABLE is the external symbol dictionary. This table records the symbol of an external procedure at the displacement of the external procedure's number.

The CONSTTABLE holds all the long constants. They are copied into this area from the input stream by either of two procedures, SORT_IN_LIT or SORT_IN_LARGE_LIT. SORT_IN-LIT accepts the whole long constant as a parameter. It compares the long constant with those already in the constant area, and if it finds that this is a duplicate, it does not enter it. It uses the displacement of the already present long constant to enter into LITTABLE for addressing. SORT_IN_LARGE_LIT is used when the size of the literal exceeds 20 digits. No optimization occurs and the long constant is entered one digit at a time.

The LITTABLE records the displacement of the long constants within the CONSTTABLE. The displacements are placed into LITTABLE at the displacement of the CONST_INDEX, which is like a procedure number.

All of the procedure labels, statement labels, and constants receive their label numbers, procedure numbers, or indexes in a previous pass. This number has also been placed as a field in any instruction which will access one of these constants or labels. Those numbers are used by pass 8 to resolve the final addresses of those labels and constants. This occurs in the procedure PROC_STACK.

The PROC_STACK instruction is received when the whole block has been read into the linked list. This is when the peephole optimization occurs. The procedure first called is TAKE_PROC. TAKE_PROC first enters the size of the procedure in the STACKTABLE, and updates the CURRENT_STACK variable to the current size of the stack. It then calls FIX_REFS. FIX_REFS scans the entire list and adjusts the addresses of all stack references and long constants. For STACK_REF the address in the list element is set to the value of CURRENT_STACK. For long constants the address in the list element is set to the value found in LITTABLE for this long constant. The ADDR field of the element in the list holds the CONST_INDEX value for long constant elements. Thus it is possible to find this long constant in LITTABLE and change the value of ADDR accordingly.

TAKE_PROC next calls OPTIMIZE_PROC. This procedure will

in turn call PHASE1_OPT. Depending on the kind of the element PHASE1_OPT will call one of four optimizing procedures, OPT_CODE, OPT_CODELAB, OPT_BRANCH, and OPT_LAB. There are 21 different procedures which may be called by any of these four. These 21 procedures perform such optimizations as removing redundant and extraneous code, changing op code forms to more efficient forms, removing extraneous instructions, and altering branch chaining to be more efficient.

When the whole list has been scanned for optimization it is written to the intermediate output.

PASS 9

Pass 9 uses the tables produced by pass 8 to replace the label identifiers with addresses. It converts the code from passes 7 and 8 into a format that will be acceptable to the linking loader. The instructions from the previous passes are encoded in words. Pass 9 manipulates these so that they will be in the appropriate format of the Perkin-Elmer instruction set. There are 28 different operators which may appear in the pass 9 input stream. These operators are processed by any one of eleven procedures.

Of those eleven procedures, six significantly affect the final code and addresses. The principle difference between those six procedures is in the type and/or length of the instruction being processed. These procedures perform operations on data types, instructions, or long constants. When called each procedure is passed a parameter that tells what kind of instruction is being processed, i.e., LAB, PROCS, LIT, or ULAB. Four of the six use the RELOC or RELOC2 procedures to handle the final resolution of addresses.

The operations on data are performed by TAKE_DATA2 or TAKE_DATA4. The numbers appended to the procedure labels indicate the length of the instruction in bytes. If the parameter passed is NONE or LAB, there is no address resolution to take place, and the instruction is output to the intermediate file as is. If the kind is PROCS, then BLOCKTABLE is accessed by the called RELOC procedure, and the final address is arrived at using

the displacement returned from BLOCKTABLE and a displacement from the instruction. If kind is LIT, then the data is a long constant and, the final address is the displacement field in the instruction plus LITOFF. LITOFF has the value of the code length as calculated by pass 8. This puts the long constant table at the end of the code and this long constant at some displacement within it.

The TAKE_INSTR2, TAKE_INSTR4, and TAKE_INSTR6 procedures are very similar to the procedures just described. The difference between these procedures is in the length of the instructions. Since the form of the instructions they receive is different, they each will perform the same operations on different fields. These procedures also use RELOC AND RELOC2 for address resolution.

The TAKE_EOM procedure writes the long constants to the LITERALTABLE. As stated before this is at the end of the code file. The long constants are written into this table without alteration.

Pass 9 also prints the source code, the external symbol dictionary, the statement, and data area maps.

CHAPTER V

NEW AND MODIFIED PROCEDURES FOR SIMULA

This chapter contains an explanation of the modifications necessary to each of the four passes of the PAS32 compiler to make it accept S-CODE as modified by INTERPASS, and output machine code for a SIMULA program.

FRAGMENT TABLE HANDLING

Pass 6 of the PAS32 compiler handles one procedure at a time. A procedure begins with an argument list or an ENTER token and ends with an EOM token. Pass 6 is not equipped to handle the nested procedures of S-Code. Nor is it equipped to handle stack entries that are not at the top of the stack, as S-Code allows. Pass 6 expects a polish postfix token stream. It was decided that it would be impractical to alter pass 6 to the extent necessary for it to be able to handle the sequence of an unaltered S-Code token stream.

The solution was to have INTERPASS produce two output files instead of one. The first is the token stream in the S-Code sequence. The second is a table of pairs, each pair pointing to the beginning and end of a fragment of the token stream. This "fragment" table is sorted so that fragments selected sequentially will cause the properly ordered tokens and arguments to be read by pass 6. By reading the fragment table and stepping through the fragments one at a time pass 6 is able to receive the polish postfix token stream it expects.

The modifications to pass 6 involved creating a new procedure, FRAGMENTTABLE, and altering the existing READIFL procedure. The FRAGMENTTABLE procedure reads the fragment table and initializes two global variables, STARTFRAG and ENDFRAG. It steps through the fragment input, one pair each time it is called by READIFL, thus making available the beginning and ending displacements of each fragment. READIFL can determine the page it needs by "startfrag div 128" and the word it needs by "startfrag mod 128". ENDFRAG is used to determine when a new pair of addresses are needed from the fragment table.

THE ALLOCATESAVELONGTEMP TOKEN

Certain temporaries in S_Code cannot be held in a register and therefore pass 7 had to be altered so that these could be stored in memory.

Interpass passes the ALLOCATESAVELONGTEMP token to pass 6 with size and alignment arguments. Pass 6 adds a temporary-id number as an argument, which is the next temporary number in sequence using the same series of numbers as the other temporaries. Pass 6 also maintains a stack of records of each of these tokens. This is necessary so that when the corresponding RELEASELONGTEMP token is processed it will receive same the temporary-id number.

Pass 7 must allocate a temporary of the given length and alignment, generate code to copy from the area pointed to by the top of stack to the allocated temporary and then have the top of stack point to that temporary. This is accomplished by using a routine similar to that in the existing ALLOC_TEMP procedure to

allocate the temporary. From the length argument, the number of words can be computed and the number of registers needed for the load multiple and store multiple instructions can be determined. The starting register is determined because pass 7 already knows the highest numbered register available. Code is generated for the load multiple (LM) and store multiple (STM) instructions using the PUT_RX procedure. The starting register number and the displacement of the temporary are placed in the attribute record so that a MAKE_ADDR procedure may determine the address of the temporary.

THE RELEASELONGTEMP TOKEN

The RELEASELONGTEMP token is created to correspond to the ALLOCATESAVELONGTEMP and LONGFUNCVALUE tokens. This token comes into pass 6 with size and alignment arguments and has added the temporary-id number. Since long temporaries are allocated and released in a last-in first-out order, the temporary-id is easily matched with that assigned to the last ALLOCATESAVELONGTEMP token. After being assigned this number the token is passed through to pass 7. Pass 7 uses the existing RELEASETEMP procedure to free the long temporary space.

THE CALLTOS TOKEN

CALLTOS differs from the existing CALL token in that it will get the address of the routine to call from the stack, instead of computing that address from a procedure label. It has three arguments: mode, parameter length and level. The latter two update the global variables, CALL_PARMLen and CALL_LEVEL, which

track the call lengths and levels.

Pass 6 passes this token through without alteration. Pass 7 computes the address using the MAKE_ADDR procedure on the top element on the attribute stack. The PUT_CODE procedure is used to generate code for a branch and link (BAL) instruction.

THE GOTOTOS TOKEN

This token is to cause an unconditional branch to the address in the top of the stack. It has no arguments and is passed through pass 6 without alteration. In pass 7 the GOTOTOS procedure causes the address of the top attribute to be computed using the MAKE_ADDR procedure. The PUT_GEN_OP procedure is then used to generate code to branch unconditionally to that address.

THE DUP AND NEWPOP TOKENS

The S-Code DUP token requires that the top item on the stack be duplicated. Because pass 6 and pass 7 are designed for Polish postfix token streams which use each result operand only once, the DUP token has made it necessary to introduce an acyclic graph structure which is not a tree to pass 6. For pass 7 this means that some method of keeping registers assigned, and not automatically freed after being used, had to be implemented.

For pass 6 when the DUP token is encountered a new node is created. The first argument of the token is a reference count to be incremented and decremented as the this token and its related tokens are processed. This DUP node is interposed between what was on the top of the pass 6 stack and its TREEELEMENT. The

LINK(1) procedure is used for this. The reference count is set to one. A DCOPY node is created and made to point to the DUP node. The reference count is incremented by one. A pointer to the DCOPY node is pushed on the pass 6 stack.

At this point the structure is: TOS points to the DCOPY node; the DCOPY node points to the DUP node; SOS points to the DUP node; the DUP node points to the expression which started on the top of the stack; and, the reference count is 2. (Figure 5.1).

Should another DUP token be encountered before EOM or NEWPOP, another DCOPY node is created and pushed onto the stack, with its pointer again to the DUP node. The reference count is incremented by one. (Figure 5.2).

The NEWPOP token from S-Code is meant to pop the top item on the stack. Because of the possible existence of a DUP node the subtree(s) of the node pointed to by TOS must be traversed, recursively pruning its elements. In this traversal, should a DUP node be visited its subtree is not pruned unless its reference count is zero. In that case the subtree is pruned and the DUP node discarded. Should a DCOPY node be visited the reference count of the DUP node is decremented by one and the DCOPY node is discarded.

In the example above, should a NEWPOP token be encountered while a second DCOPY node is at TOS, the reference count is decremented by one and that DCOPY node is discarded by popping TOS.

Expr	Dup	Dcopy
Expr	Dup	Dcopy

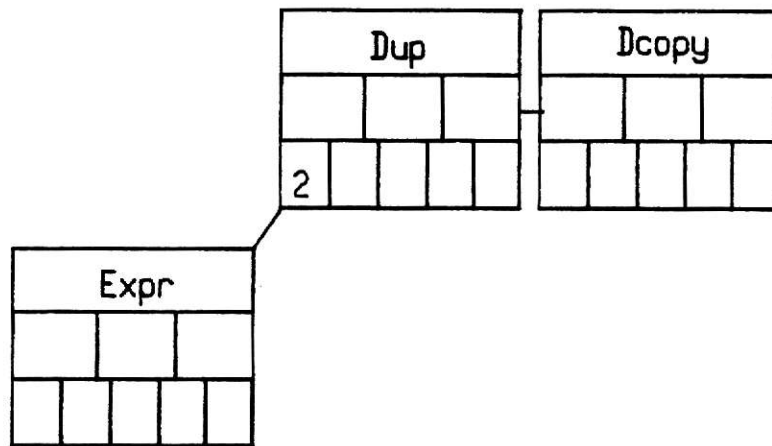


Figure 5.1

Expr	Dup	Dcopy	Dcopy
Expr	Dup	Dcopy	Dcopy

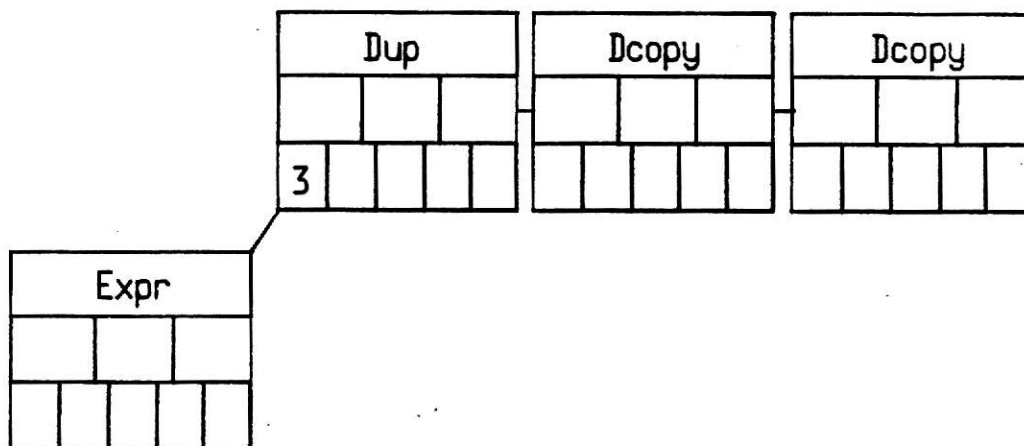


Figure 5.2

Should two more NEWPOP tokens be input before another DUP token, then this reference count would go to zero the DUP node's subtree would be pruned and the node discarded by popping TOS.

In pass 7 when the DUP token is input the object on the top of the attribute stack is duplicated by creating a new object, copying the attributes from the top object to the new object and pushing the new object on the stack.

The presence of the DUP token in pass 7 means that any registers used by the token can not be released until both the DUP token and the object it duplicated have been used. This is accomplished by adding a new array to pass 7, REGS_USED. The array is indexed by the type of register and the register number. Each time the MARK_REG procedure is called the appropriate use count is incremented by one. Whenever the FREE_REG procedure is called, the use count of the appropriate register is decremented by one. The FREE_REG procedure is not allowed to mark the register free unless its use count is zero.

In this respect it is a good time to point out that general purpose register 7 is used by UNIX and is never available to the compiler. Code has been inserted to protect register 7.

THE DUMMYARGLIST TOKEN

The DUMMYARGLIST token's purpose is to take up space at compile time, to skip over parameters which will not actually be passed. That is, to maintain alignment when there is no actual parameter to match a formal parameter.

This token has the same arguments as the existing ARGLIST token. The arguments are mode, displacement, context, type and length. Pass 6 passes DUMMYARGLIST and its arguments through to pass 7. The UNARYAPPEND procedure is used for this. Pass 7 treats this token the same as ARGLIST except in the ARGUMENT_LIST procedure, all code is bypassed after the offset and length have been determined, until the SET_TEMP procedure is called. This will occupy space at runtime in the activation record.

THE LONGFUNCVALUE AND ENDLONGFUNC TOKENS

These two tokens are similar to the existing FUNCVALUE and ENDFUNC tokens. Coming into pass 6 the arguments of the LONGFUNCVALUE token are mode, type, size and alignment. Pass 6 replaces the type argument with a temporary-id number. This is done so that the results of function call may be placed in the temporary area instead of a register. Pass 6 simply places the next sequential temporary number into argument two of the token, and the token is pushed onto the stack.

The ENDLONGFUNC token is not input to pass 6 but is created while the tree structure is being traversed for output. When the WALKTREE procedure recognizes a LONGFUNCVALUE element, it calls the PUTBEGINOP procedure. In this procedure the LONGFUNCVALUE element is output before its subtree is traversed. The token remains in the stream and the traversal of the subtree continues. When all of its nodes have been visited and output, the element is output; however, the table of output tokens causes the ENDLONGFUNC token to be placed in the stream instead of another

LONGFUNCVALUE.

This process will eventually leave a temporary on the compile time stack. Therefore at the time these two tokens are output an additional RELEASELONGTEMP token is also output. This token will have the corresponding temporary-id number as its second argument.

Pass 7 handles these tokens in much the same manner as the FUNCVALUE and ENDFUNC tokens, except the function result can not be stored in a register. The LONGFUNCVALUE procedure allocates space for a temporary, and the ENDLONGFUNC procedure generates code to push the address of the temporary.

THE LONGCONSTADDR TOKEN

For SIMULA long constants can be or can contain addresses of machine instructions. This requires new procedures because the PAS32 compiler is not equipped to handle any such occurrence.

This new token is input to pass 6 with four arguments: location mode, location displacement, value mode, and value. The first argument indicates into which area, global or constant, that the address is to be placed. The second argument is the displacement within this area at which the address is to be placed. The third argument indicates how the address is to be computed. The fourth argument is either a label or a displacement from which the address is to be computed.

Pass 6 will add the constant index argument to these four arguments. This will be used in pass 8 to resolve the location

displacement. Pass 7 will pass this token through unaltered. In pass 8 the true displacement of the constant in which the address is to be placed is resolved. This is done using the constant index, the LITTABLE, which contains the constant area displacement for that constant index, and the ENTR function, which uses the first two to return the desired displacement. The constant index is not used again.

In pass 9 the four arguments are used to locate or compute the address and to place it in the designated area, in the identified constant, at the right displacement. The location mode tells which area contains the constant to be overlayed. The true displacement within that area was computed in pass 8. If the value mode is LCONST or ABSOLUTE, the value argument is the value of the constant which is an address of something else within the constant area or global area, respectively. It may be retrieved using the ENTR function and placed appropriately with the ENTER procedure. If the value mode is PLAB or STLAB, the value argument is a label. The address of the label is retrieved from the BLOCKTABLE with the ENTR function and placed by the ENTER procedure.

THE NEWLONGCONST TOKEN

S-Code has both variables with initialized values and constants. PAS32 can handle constants but not initialized variables as constants. In addition, pass 8, in optimizing the constant area, eliminates duplicates as they are received to be sorted into the constant table. This could mean the elimination of some

initialized variables. The solution to this is the creation of a new constant area by pass 8 where the initialized variables will be placed.

Coming from interpass the long constants will have a mode argument. If the mode is LCONST, it is a true constant. If the mode is ABSOLUTE, it is an initialized variable. New procedures for passing the tokens through passes 6 and 7 are identical to existing procedures with the exception of being able to handle the additional argument.

Pass 8 must not only place the constants into their designated areas, it must put the displacements of those constants into the address fields of the instructions which access the constants. This is complicated by the fact that the mode is never known to those instructions or the tokens that created them. Only the constant index, i.e., serial number, of the constant is known. Therefore, at the time the displacement is placed in the instruction, it is not known which area that displacement pertains to.

To overcome this, these changes are made: 1) The accumulated length of all constants is passed to pass 6 by interpass. Pass 6 determines the lengths of long constants and initialized variables by subtracting the length of each initialized variable from the constant length and accumulating a GLOBALLENGTH. This GLOBALLENGTH is passed through to pass 8. 2) Pass 8 uses code only slightly modified from the existing SORT_IN_LARGE_LIT procedure to sort the initialized variables into their area, the

GLOBALTABLE. The displacement of the variable in that area is computed to be its displacement from the beginning of the GLOBALTABLE plus the total length of the true constants. This computed displacement is placed in LITTABLE and subsequently will be returned as the address of the constant when the FIX_REFS procedure resolves addresses. The address in the instruction will already contain all the offset needed for final address resolution by pass 9.

Pass 9 already initializes a variable, LITOFF, to the code length. It then uses that variable as the base address of the constant area. The base address of the GLOBALAREA will thus be LITOFF plus the constant length.

The constant length at the beginning of pass 8 will change as pass 8 eliminates duplicate constants. This difference in lengths is already available to pass 9 because the variables CONST_A and CONST_B are passed from pass 8 to pass 9. CONST_A is length of the constant area after optimization and CONST_B is the length before optimization. This being known, the resolved displacement of initialized variables may be adjusted by pass 9. If the GLOBALAREA is created after the constant area, the address of the GLOBALAREA is LITOFF plus CONST_A.

THE NEWPUSHLABEL TOKEN

In the PAS32 compiler the PUSHLABEL token is known to be a routine label. In S-Code the label may be a routine or a statement label; therefore, a new NEWPUSHLABEL token is introduced, having a mode argument.

Interpass will input the token to pass 6 with mode and the label as arguments. Pass 6 does not alter this, but passes the token through. Pass 7 uses the mode to determine whether to output a DC instruction with a routine label type or statement label type. The output of pass 7 will be the same as would have been output from pass 7 of the PAS32 compiler. Thus no modifications were made to passes 8 and 9.

THE NEXTCONSTDISPL, NEXTCONSTINDEX, and NEXTLABEL TOKENS

There are three variables in pass 6 which keep track of the length of the constants used so far, the serial number of each constant, and the numbers associated with the labels. These variables were initialized in pass 6 by variables passed to it by pass 5. Since interpass has "replaced" pass 5, initialization of variables does not occur in the same manner. These three tokens have been introduced to initialize the variables mentioned above.

Each of these tokens has one argument, the value needed to properly initialize its corresponding variable. When pass 6 receives one of these tokens, the new procedure INITCONSTVALUES is called, which places the value in the appropriate variable.

THE PUSHLEN, SAVE, TINITO, TGETO, TSETO, and RESTORE TOKENS

These 6 tokens are made necessary by the way intermediate results must be handled by the S-compiler. Because expression evaluation by the S-compiler can cause the creation of new objects, space management can be a problem. At anytime the runtime system may invoke garbage collection which will result in compaction. Because some of the temporaries that exist at that time

may be pointers to existing objects, the garbage collector must have access to them. The pointers may be in registers or long temporaries. The garbage collector must be able determine that the objects they point to are accessible and adjust any pointers to objects which are moved.

The PUSHLEN token is used to tell the runtime system how much space will be necessary for the "save object". This save object is the object into which all temporaries will be stored during the garbage collection. Pass 7 computes this quantity by summing the space required for all registers, the space in use by the temporaries (TEMP_TOP), the additional space required for registers containing pointers, and the space required for all pointers contained within long temporaries.

The pointers contained within long temporaries and in registers have been identified by appropriately placed POINTERTYPE arguments and the LONGTEMPPOINTER tokens. Two new arrays have been created in pass 7, to receive the displacement of pointers within long temporaries and to record the register numbers which contain pointers. If a token has a POINTERTYPE argument, the register number assigned to that token will be made to be entry to the register-pointer array. When a LONGTEMPPOINTER token is received, pass 7 records its displacement argument in the temporary-pointer array. These two arrays will be used latter to place those pointers into the save object.

The SAVE token causes the creation of the save object. It also causes pass 7 to make a copy of the compile time stack. The

Pointer Area	Register Store
Register values	
Pointer values	
Registers	

SAVE OBJECT
Figure 5.3

runtime system has caused the address of the save object to be at the top of the compile time stack. The PUSHLEN token has determined its length. Pass 7 must now generate code to store the values of each register containing a pointer to be placed at the front of the save object. It must also generate the code to place the values of the pointers contained within long temporaries next in the save object. Then load multiple and store multiple instructions are generated to save all of the general purpose, real, and short-real registers.

The first word of the save object contains two pointers, to the beginning of the pointer area and to the beginning of the register store. The displacements of the pointers within the temporaries and the beginning of the temporaries within the save object are saved in global variables.

The entire attribute stack is copied to a new stack, REMEMBER_STACK. The global variable TEMP_TOP and the two arrays containing the descriptions of the pointers are also copied into new global variables.

The TINITO token is used to initialize the scan of the save object. The runtime system has provided the address of the save object to be scanned. The variable SAVE_INDEX is initialized to point to the pointers in the save object.

The TGETO token causes the pointer area of the save object to be consecutively scanned, returning the value of a different pointer with each call. This value, if not zero, is replaced

with the address then on the top of stack.

TSETO causes the value at TOS to be placed into the save area in place of the one just returned. When the value of SAVE_INDEX reaches that of the beginning of the register save area, this process is terminated.

The RESTORE token signals that garbage collection is finished. The registers are restored to their original values. The temporary area is restored using load multiple and store multiple instructions. Then the pointers which may have been altered by the garbage collector are restored, overlaying whatever values might have been restored in the two previous restore steps. The pointers in the temporary area may be located by the displacements in the remembered array and the local base register. The register numbers of the pointers in registers is known from the array which was remembered during the save operation. These two arrays will be restored as will the attribute stack.

THE SIMPLEINDEX TOKEN

SIMPLEINDEX is similar to the existing INDEX token, except that the pass 6 procedure for INDEX does a bounds check. For S-Code no bounds checking is wanted. Where INDEX had upper and lower bounds arguments with an element size, SIMPLEINDEX will have only the element size argument. Pass 6 is modified to bypass the bounds checking. It is also altered to place a zero in the lower bounds argument. The existing pass 7 procedure is also altered to bypass bounds checking.

CHAPTER VI

PROJECT STATUS AND FUTURE WORK

At the time of this report the project has been in the test stage for several months. Pass 6 has been able to accept some input and produces some output. However, it appears that the token stream coming from INTERPASS is not entirely in the order expected by pass 6. Therefore the amount of testing on the new code has been very limited.

None of the new code for passes 7, 8, and 9 has been tested. Reviewing the project as a result of writing this paper has uncovered several errors and omissions which will have to be corrected before any credible testing can take place on any of these three passes.

The Norwegian Computing Center has been contacted about some inconsistencies in the output of S-Code. Until we have their reply, the changes in the input stream cannot be made with any certainty that they will correct problems.

When the Norwegian Computing Center offers some solution to the S-Code problems, testing will resume. Presumably changes will have to be made to INTERPASS. Passes 6, 7, 8, and 9 will have to be thoroughly tested.

CONCLUSIONS

This project provided a learning experience which was more

than an academic exercise. One objective of the project was to produce a functional compiler which could become a useful tool for solving problems. The project provided a better understanding of compiler construction in general and the code generation passes in particular. It was also valuable in providing an experience in the problems of lower level software design and implementation. It clearly demonstrated the problems involved in writing such programs as opposed to higher level applications software.

While there was little opportunity to learn to use the SIMULA language, the project required the acquisition of some knowledge of that language and its potential uses.

BIBLIOGRAPHY

- [1] Dahl, Ole-Johan, and Hoare, C.A.R., "Hierarchical Program Structures", Structured Programming, A.P.I.C., Studies in Data Processing, No. 8, (1972), Academic Press, pp. 175-220.
- [2] Jensen, Peter, Stein, Krogdahl, Ostein, Myhre, Robertson, Peter S., Syrrist, Gunnar, "Definition of S-Code", Norwegian Computing Center, (October, 1982), Forskningsvn, 1B, Blindern, Oslo, Norway.
- [3] Millard, Geoffrey E., Oystein, Myhre, Syrrist, Gunnar, "S-Port, The Environment Interface", Norwegian Computing Center, (June, 1981), Forskningsvn, 1B, Blindern, Oslo 3, Norway.
- [4] Robert Young, "Internal Documentation for the PASCAL/32 Compiler", manuscript, Kansas State University, Manhattan, Kansas, 1978.
- [5] 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 No. 2, pp. 149-178, (April, 1982).
- [6] Robin Hills, "SIMULA 67, An Introduction", a compilation, Robin Hills (Consultants) Limited, 24 Beaufont Road, Camberley, Surrey, England, (1972).

APPENDIX A

COROUTINES

The following example contains an illustration of the use of the SIMULA "call", "detach", and "resume" statements to create a coroutine relationship between procedures. For this illustration there will be four CLASSES: W, X, Y, and Z. The REFERENCE variables for these CLASSES are: A, B, C, and D respectively.

The CLASSES would be defined:

```
class W(...); ...class body for W ...;
class X(...); ...class body for X ...;
class Y(...); ...class body for Y ...;
class Z(...); ...class body for Z ...;
```

The REFERENCE variables would be defined:

```
ref(W)A;
ref(X)B;
ref(Y)C;
ref(Z)D;
```

The main program would be:

```
begin A :- new W;
      B :- new X;
      C :- new Y;
      D :- new Z;
```

```
    call(A);  
end
```

Within the body of A, A would immediately DETACH from M, the main program, and RESUME(B) thus creating a coroutine relationship with B and remaining in a subroutine relationship with M. As long as B has control by virtue of a RESUME command from A, B is responsible for returning control to M if it issues a DETACH or reaches the end of its code. According to the diagram (Figure A) B issues a RESUME(C). C is then responsible for returning control to M if it issues a DETACH or reaches the end of its code. C in turn contains a RESUME(D). D issues a DETACH which returns control to M.

Throughout the operation of the program M remains in control in that any of the coroutines receiving control directly or indirectly from A must accept the responsibility to return control to M. A, B, C, and D have a coroutine relationship in that they operate on an equal level. Any of the coroutines could issue a RESUME to any of the others, i.e., the flow does not have to be in one direction as the diagram shows. In that sense M is not in sole control of the flow of the program. Control may pass between the coroutines with RESUME commands in any required sequence until one of the coroutines issues a DETACH or goes through the end of its code.

Any time a coroutine is called by a RESUME it will be entered at the next instruction after its last RESUME or DETACH.

Each time a coroutine is entered by a RESUME statement, its execution begins as the instruction following its last DETACH or RESUME instruction.

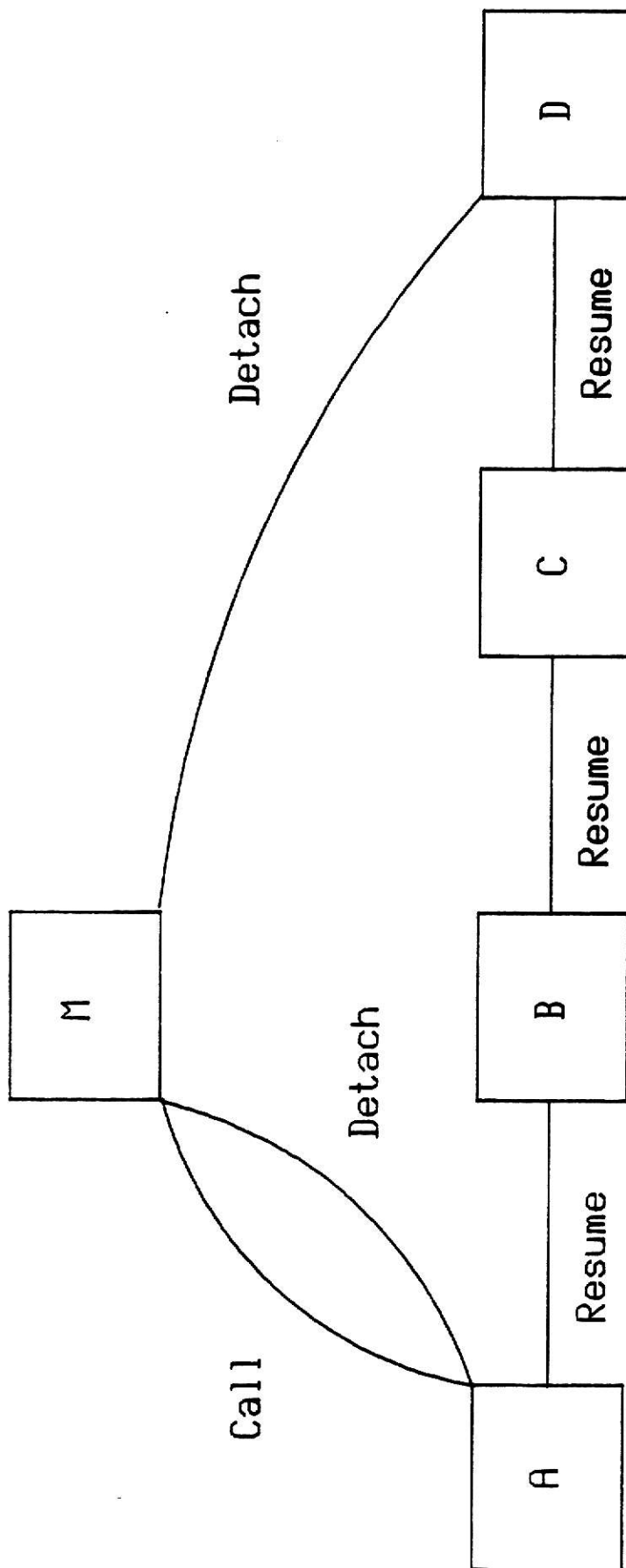


Figure A

APPENDIX B

A SAMPLE INPUT TOKEN STREAM FOR PASS 6

The following is a sample of part of an input token stream to pass 6. On the pages that follow are diagrams of the list and tree structures that would result from this partial stream, before being output to pass 7.

STARTLIST, PARM, ENTER, DEFLABEL, PUSHADDR, PUSHCONST, FIELD,
ADD, PUSHIND, PUSHCONST, COMPARE, NOT, FALSEJUMP ...

Parm				
LP		RP		NP
Mode	8	Con- text	Type	12

TREEELEMENT

Figure B.1

Begin

HEAD	nil
TAIL	nil

Figure B.2

HEAD	nil	nil
TAIL	nil	nil

OPERATION:

Startlist

Figure B.3

HEAD	nil	Parm
TAIL	nil	Parm

OPERATION:

Parm

Procedures Invoked:
AppendInput
Append

Parm				

Figure B.4

HEAD	Enter
TAIL	Enter

OPERATION:

Enter

Procedures Invoked:

Unarytree

Link(1)

Appendstk

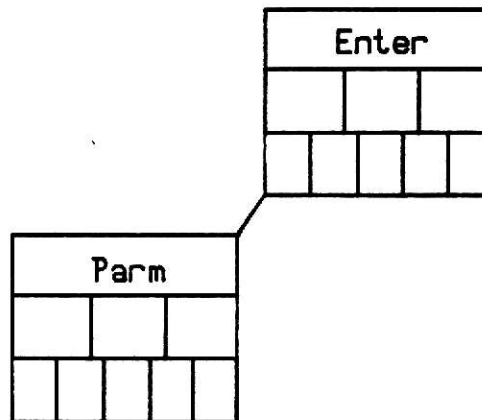


Figure B.5

HEAD	Enter
TAIL	Deflabel

OPERATION:

Deflabel

Procedures Invoked:

AppendInput

Append

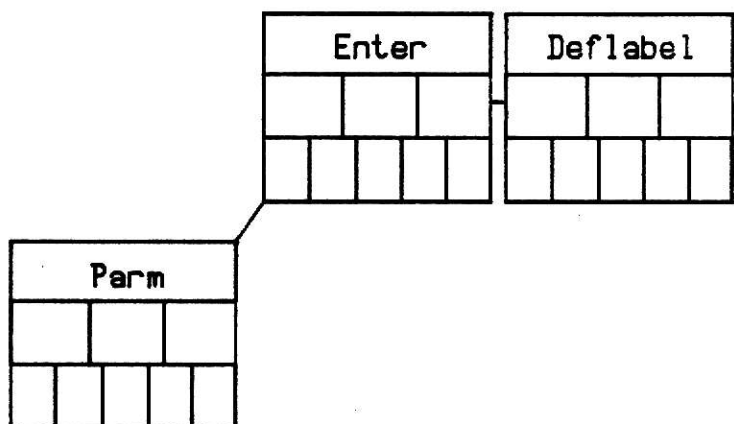


Figure B.6

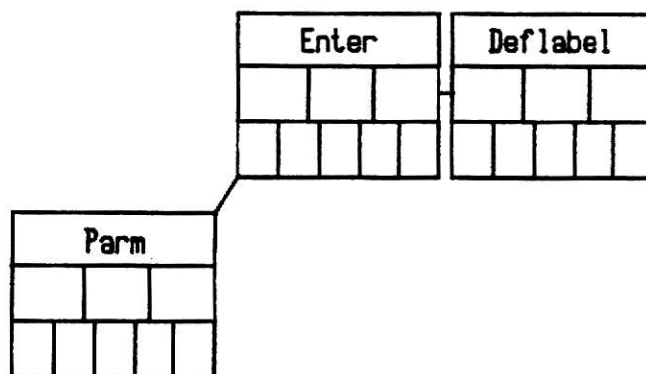
HEAD	Enter	Pushaddr
TAIL	Deflabel	Pushaddr

OPERATION:

Pushaddr

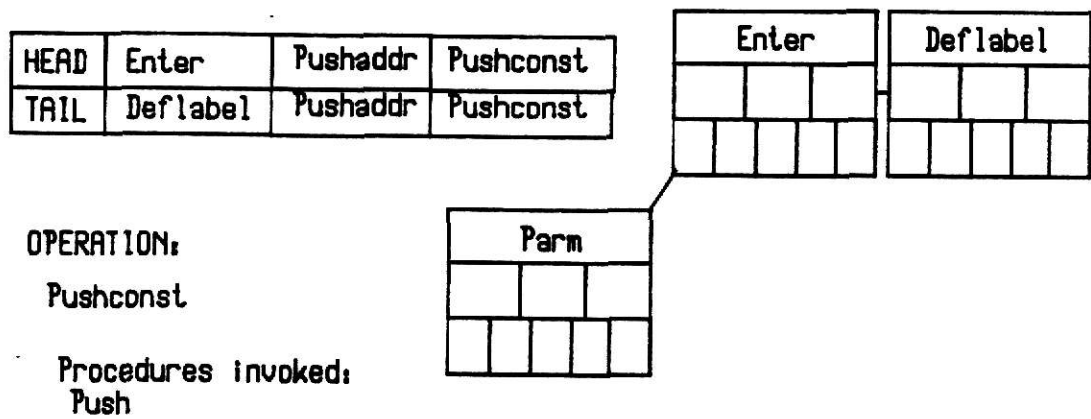
Procedures Invoked:

Push



Pushaddr		

Figure B.7



Pushaddr				

Pushconst				

Figure B.8

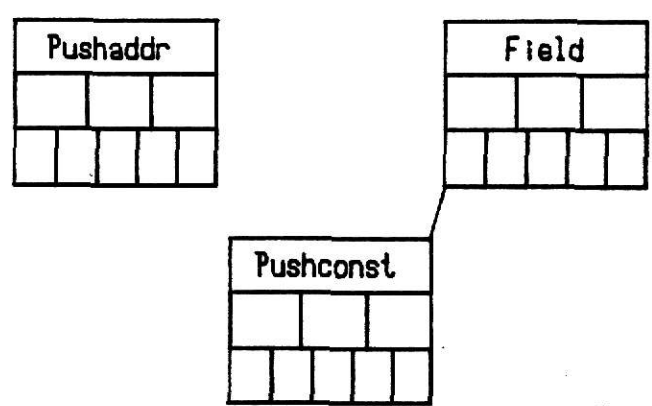
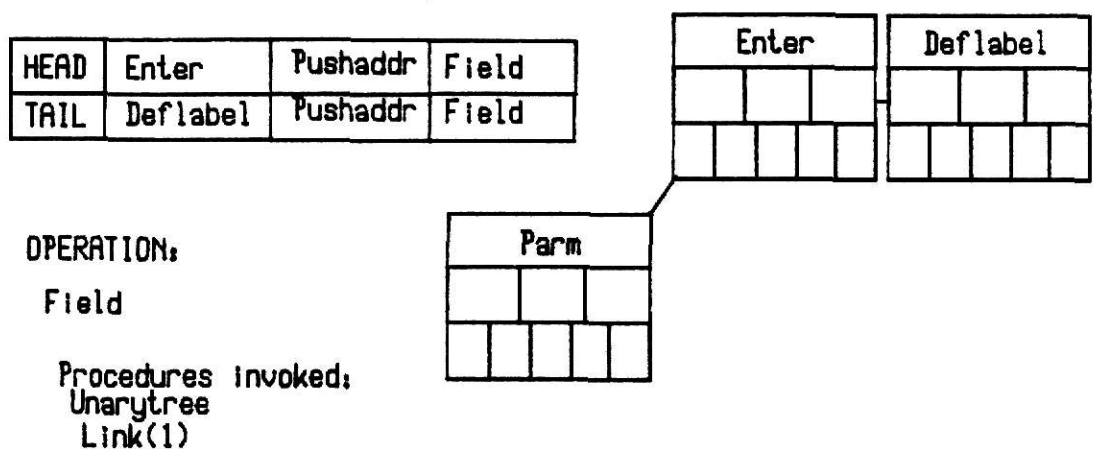


Figure B.9

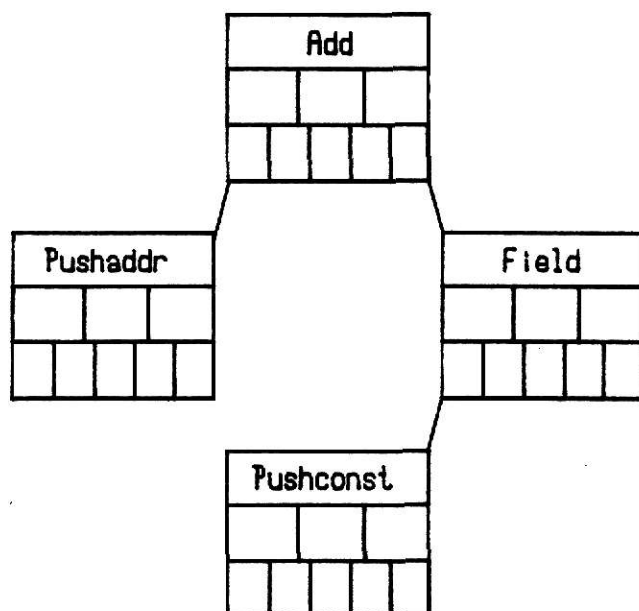
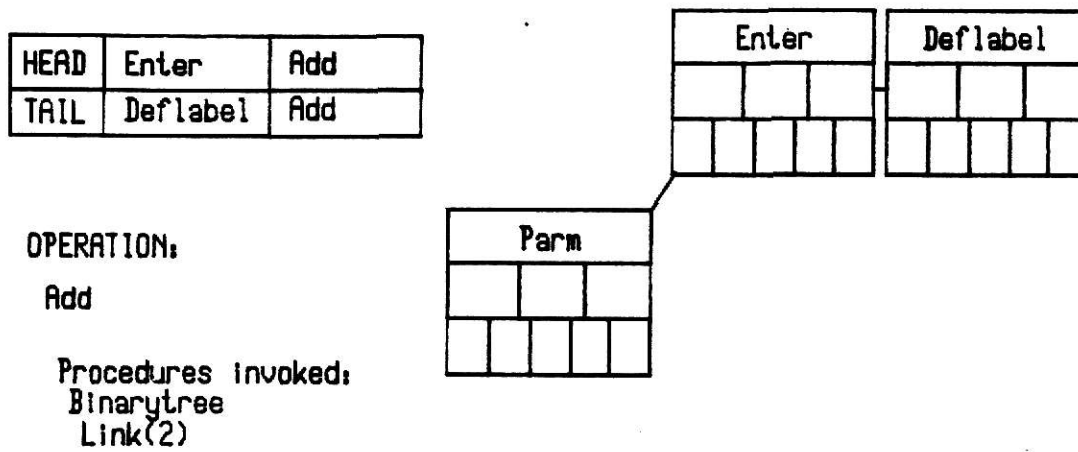


Figure B.10

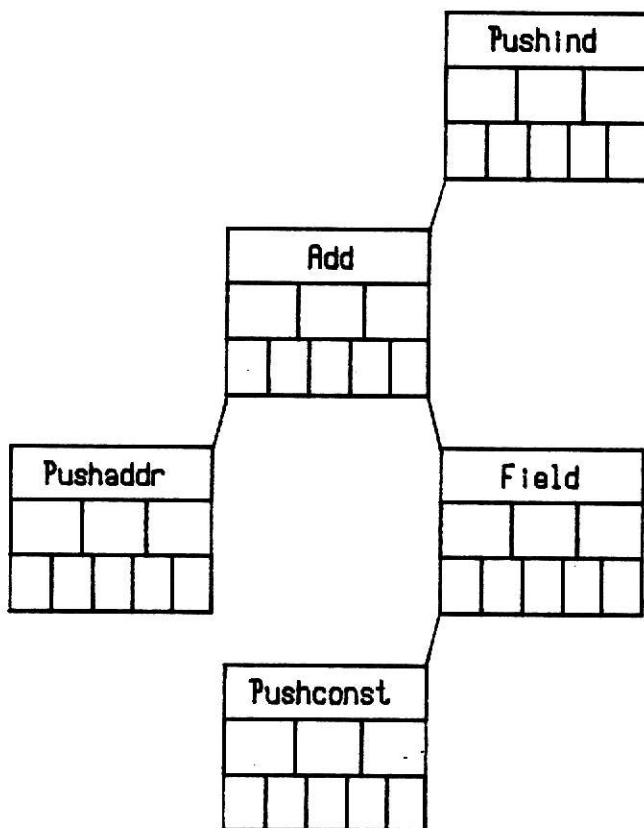
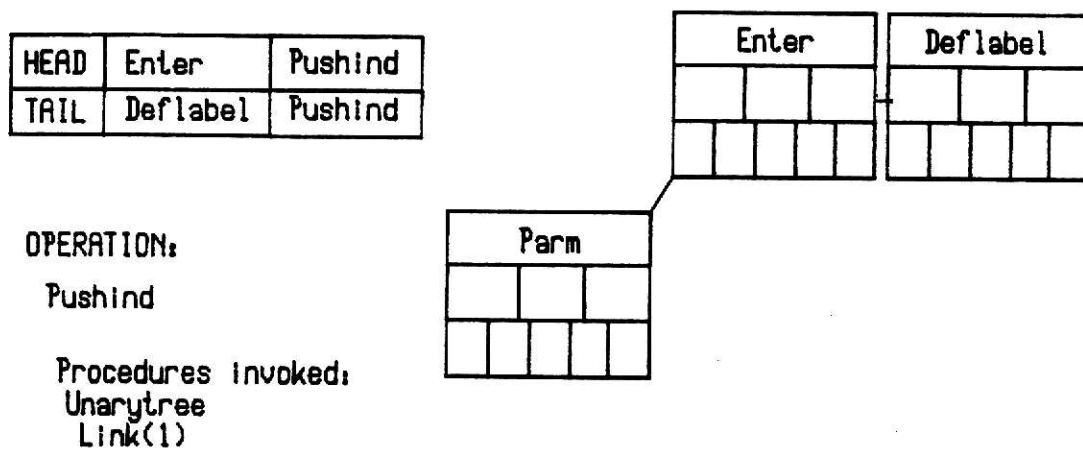


Figure B.11

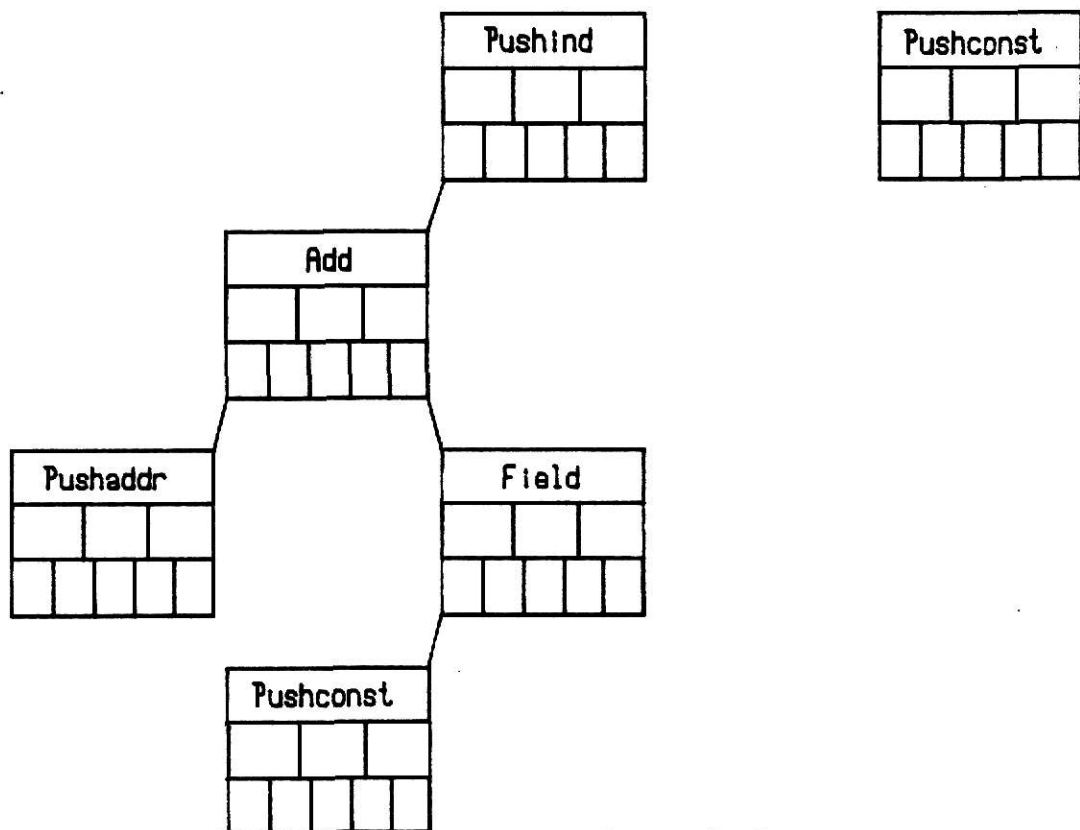
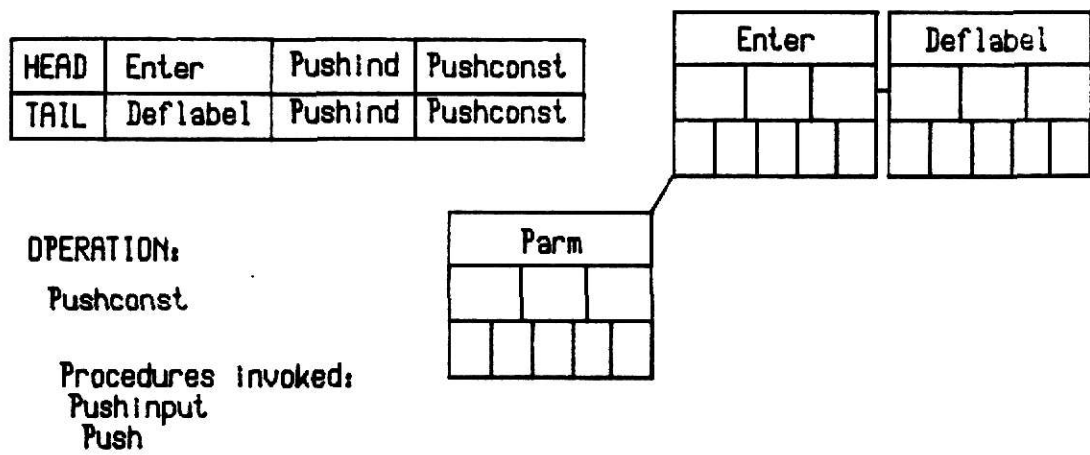


Figure B.12

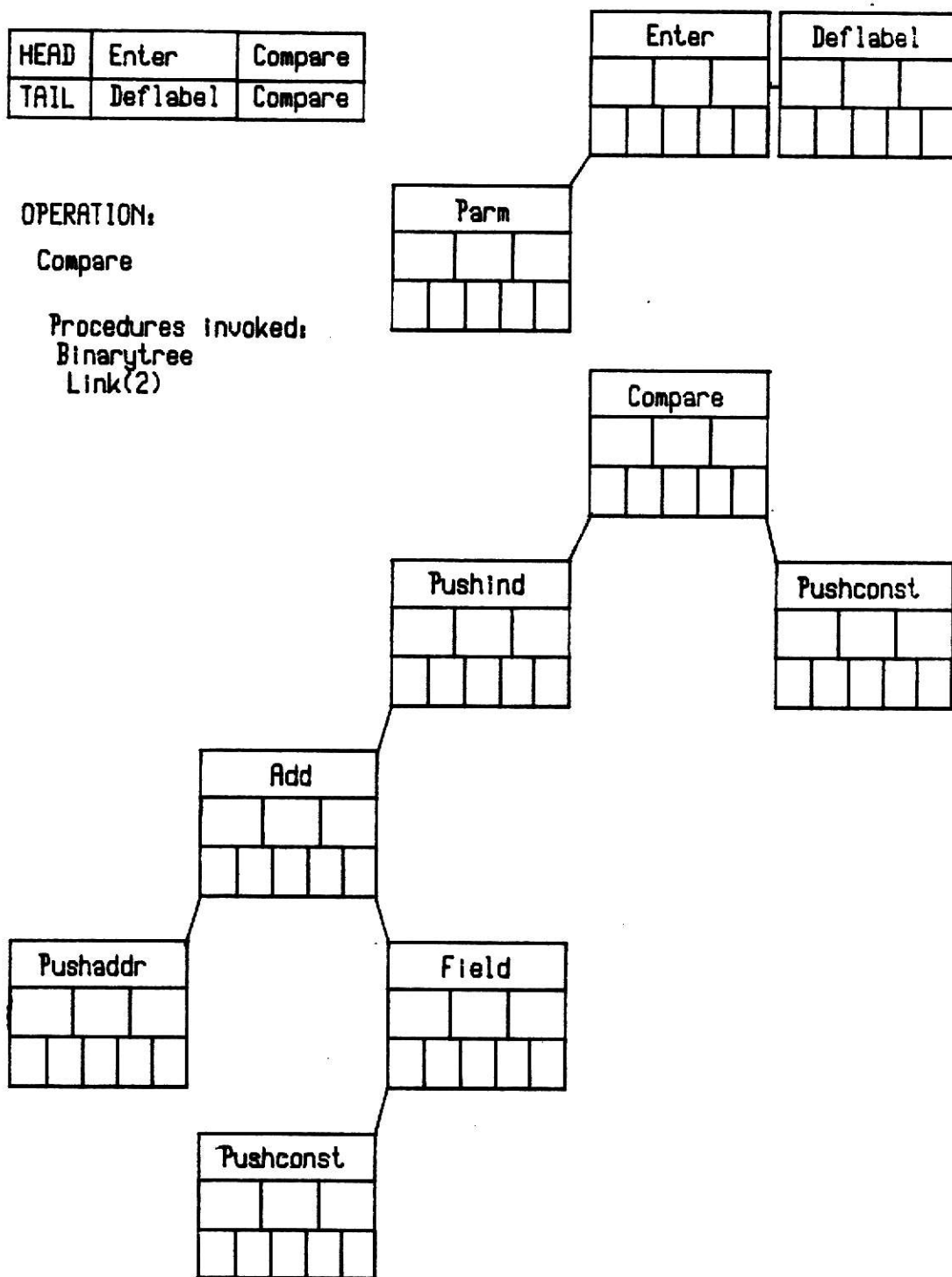


Figure B.13

HEAD	Enter	Not
TAIL	Deflabel	Not

OPERATION:

Not

Procedures Invoked:

Link(1)

Pop(1)

Push

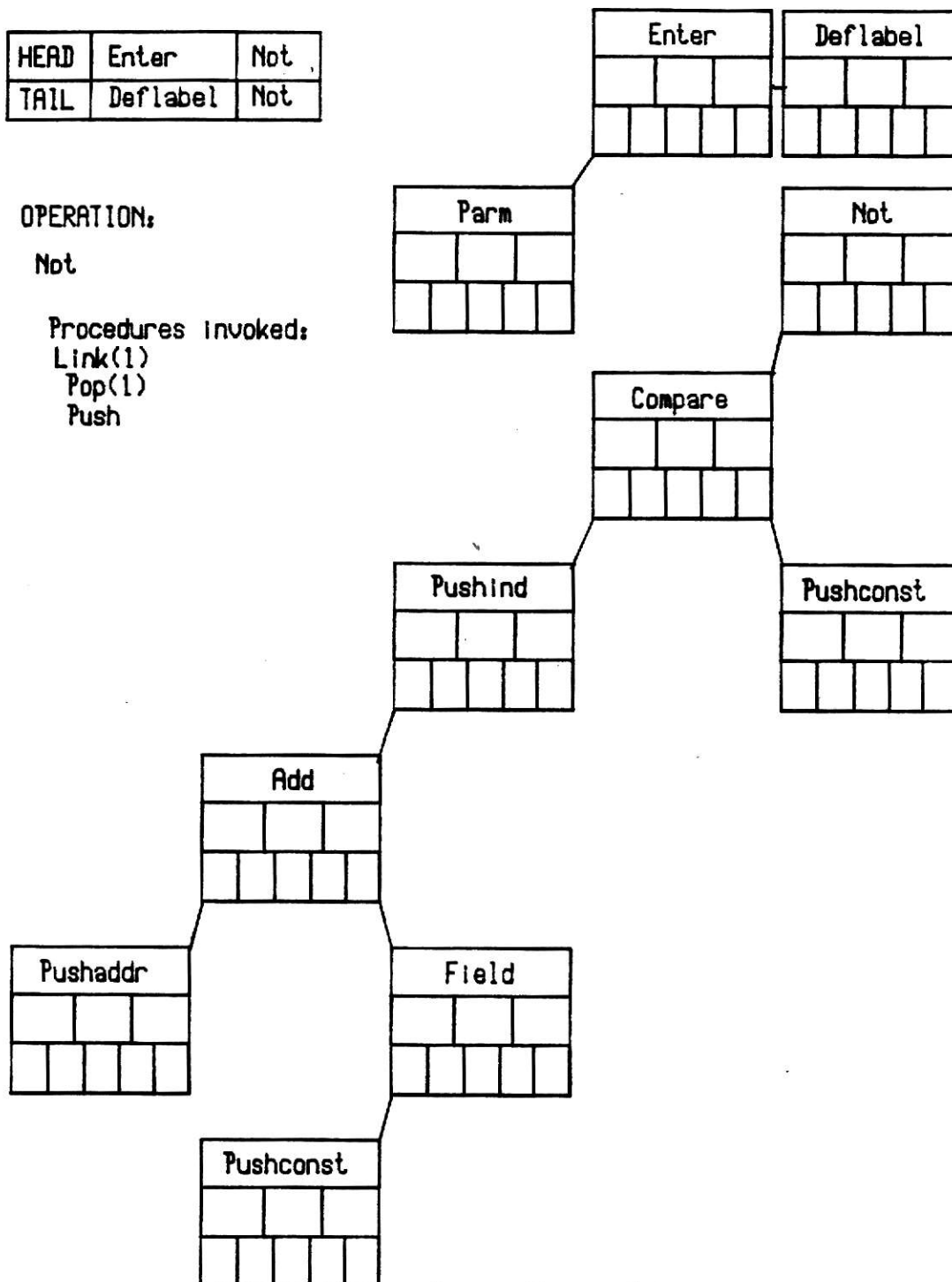


Figure B.14

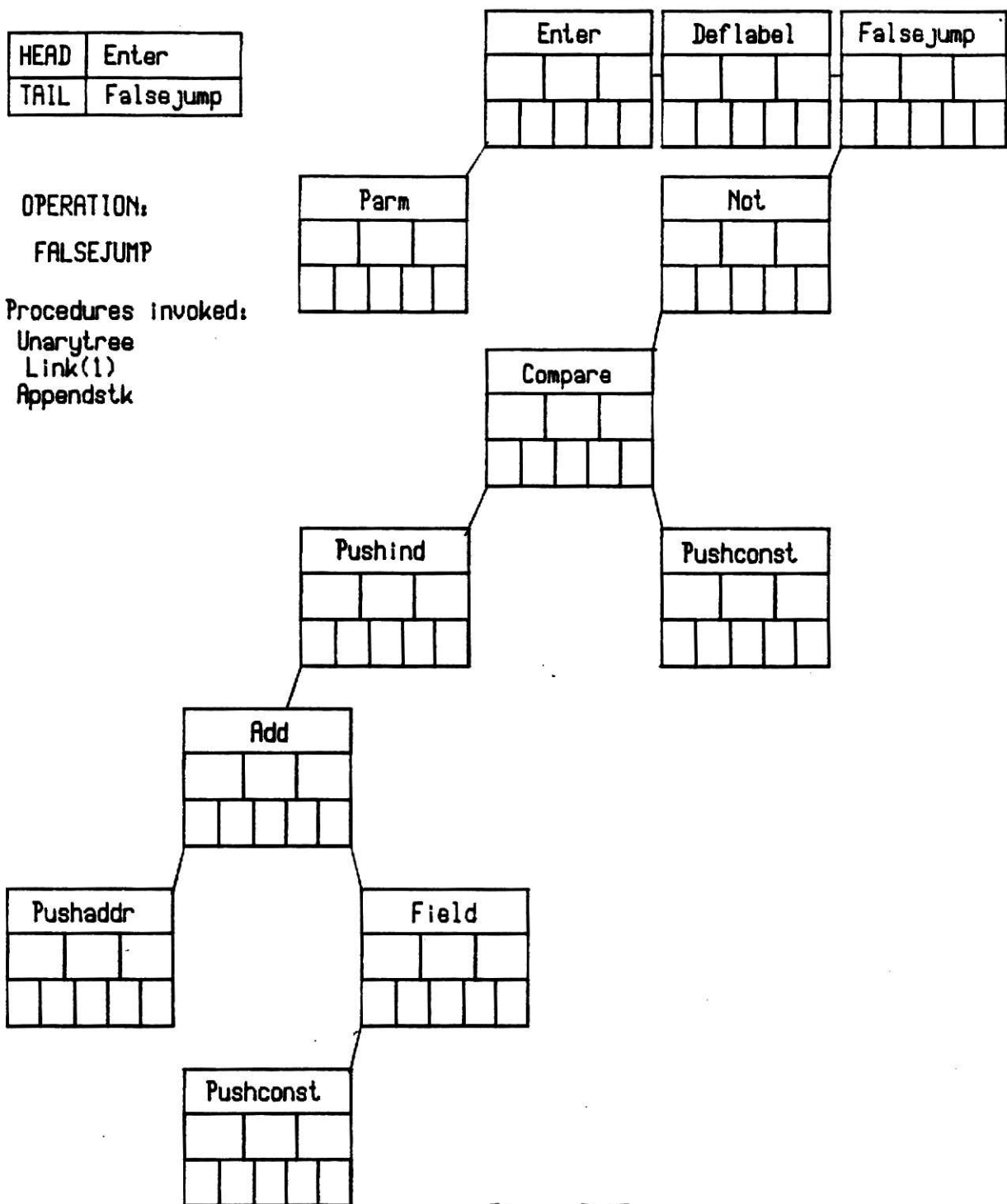


Figure B.15

THE IMPLEMENTATION OF A SIMULA COMPILER ON THE
KANSAS STATE UNIVERSITY PERKIN-ELMER COMPUTERS

by

LOWELL RICHARD LINDSTROM

B. S., University of Kansas, 1959

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

ABSTRACT

THE IMPLEMENTATION OF A SIMULA COMPILER ON THE KANSAS STATE UNIVERSITY PERKIN-ELMER COMPUTERS

This report describes a project to implement a portable SIMULA compiler on the Kansas State University Perkin-Elmer computers. Major portions of the report describe those tasks for which the author was responsible. Those tasks were to modify the code generation passes of an operational Pascal compiler to accept a token stream of a portable SIMULA compiler and to output an executable SIMULA program. The report contains overviews of the SIMULA language, the portable SIMULA compiler, and the project itself. The data structures and processes used by the code generation passes and the modifications to those passes are described in detail.

1430-227
CA96