

CS 98

ALGOL W IMPLEMENTATION

BY

H. BAUER

S. BECKER

S. GRAHAM

TECHNICAL REPORT NO. CS 98

MAY 20, 1968

COMPUTER SCIENCE DEPARTMENT  
School of Humanities and Sciences  
STANFORD UNIVERSITY



ALGOL W IMPLEMENTATION\*

By

H. Bauer

S. Becker

S. Graham

---

\*This research is supported in part by the National Science Foundation under grants GP-4053 and GP-6844.

# ALGOL W IMPLEMENTATION

## Table of Contents

I.	Introduction . . . . .	1
XI.	General Organization . . . . .	3
III.	Overall Design . . . . .	4
	A. Principal Design Considerations . . . . .*	4
	B. Run-Time Program and Data Segmentation . . . . .	6
	C. Pass One . . . . .	7
	D. Pass Two . . . . .	8
	1. Description of Principles and Main Tasks . . . . .	8
	2. Parsing Algorithm . . . . .*	8
	3. Error Recovery . . . . .	9
	4. Register Analysis . . . . .*	11
	5. Tables . . . . .	13
	6. Output . . . . .	13
	E. Pass Three . . . . .	16
IV.	Compiler Details . . . . .	17
	A. Run-Time Organization . . . . .	17
	1. Program and Data Segmentation . . . . .	17
	2. Addressing Conventions . . . . .	20
	3. Block Marks and Procedure Marks . . . . .	20
	4. Array Indexing Conventions . . . . .I.#	22
	5. Base Address Table and Linkage to System Routines . . . . .	23
	6. Special Constants and Error Codes . . . . .	24
	7. Register Usage . . . . .	27
	8. Record Allocation and Storage Reclamation . . . . .	27
	B. Pass One . . . . .	33
	1. Table Formats Internal to Pass One . . . . .	34
	2. The Output String Representing an ALGOL W Program . . . . .	36
	3. The Table Output of Pass One . . . . .	39
	4. Introducing Predefined Identifiers . . . . .	41

Table of Contents (cont.)

C.	Pass Two .....	43
	1. Storage Allocation .....	43
	2. Value Stack .....	45
	3. Interpretation Rules .....	45
	4. Pass Two Tables .....	51
	5. Output of Pass Two .....	54a
D.	Pass Three .....	62
	1. Register Allocation.....	62
	2. Block Entry.....	65
	3. Block Exit .....	69
	4. Procedure Statements and Typed Procedure Designators .....	70
	5. Procedure Entry .....	73
	6. Procedure Exit .....	80
	Formal Parameters in Expressions and Assignments .....	81
	8. Array Declaration .....	83
	9. Subscripted Variables .....	89
	10. Passing Subarrays as Parameters ....	91
	11. Arithmetic Conversion .....	94
	12. Arithmetic Expressions .....	97
	13. Logical Expressions .....	108
	14. String Expressions.....	115
	15. Bit Expressions.....	117
	16. Record Designators .....	118
	17. Field Designators .....	119
	18. Case Statements and Case Expressions .....	120
	19. If Statement, If Expression, and While Statement .....	122
	20. For Statement .....	123
	21. Goto Statement .....	127
	22. Assignment Statements .....	130
	23. Card Numbers .....	134
E.	Trace Facilities .....	0.0.0 ..... 136
Appendix I	Example .....	138
Appendix HI	Simple Precedence Grammar .....	144

Figures

1.	Reserved Word Tables .....	35
2.	Identifier Tables.....	36
3.	Pass One Output Codes .....	38
4.	Example of BLOCKLIST and NAMETABLE .....	41
5.	Format of NAMETABLE and field contents after Pass Two .....	52 ff
6.	Pass Two Output Vocabulary.....	56 ff

## I. INTRODUCTION

In writing a compiler of a new language (ALGOL W) for a new machine (IBM System/360) we were forced to deal with many unforeseen problems in addition to the problems we expected to encounter. In a few instances, we gave in to temptation and changed the language; in many others we would have liked to have been able to change the machine. This report describes the final version of the compiler. Not surprisingly, there are several things that in retrospect we would do differently, both in design of the language and in design of the compiler. We will not discuss these after-thoughts here.

The implemented language ALGOL W<sup>1)</sup> is based on the Wirth/Hoare proposal<sup>2)</sup> for a successor to ALGOL 60. The major differences from that proposal are in string definition and operations and in complex number representation. Consideration was given to including both parallelism and data file facilities in the language but both ideas were abandoned because their inclusion would have necessitated substantial changes in those parts of the compiler that had already been written,

The project was initiated and directed by Professor Niklaus Wirth, who proposed many of the ideas incorporated in the compiler and suggested ways to bring them about. Joseph W. Wells, Jr. and Edwin H. Satterthwaite,

---

1) Bauer, H.R., Becker, S. and Graham, S.L. ALGOL W Language Description, Report CS 89, Computer Science Department, Stanford University (March 1968).

2) Wirth, Niklaus and Hoare, C.A.R. A Contribution to the Development of ALGOL. Comm. ACM 9 (June 1966), pp. 413-431.

Jr. wrote the PL360 System in which the compiler is embedded, the linkages to the compiler, and the loader. Although the authors did the bulk of the programming for the compiler, valuable contributions were made by Larry L. Bumgarner, Jean-Paul Rossiensky, Joyce B. Keckler, Patricia V. Koenig, John Perine, and Elizabeth Fong. We are grateful **also** for the many helpful comments and suggestions made by the faculty and students of the Computer Science Department. **Finally**, we **gratefully** acknowledge the support given us by the **National Science Foundation** under grants GP-4053 and GP-6844 and the computer time made available by the Stanford Linear Accelerator Center and the Stanford **Computation** Center.

## II. GENERAL ORGANIZATION

The compiler is divided into three passes,

Pass One is a scanner. It reads the source program, converts the symbols to internal codes, deletes comments and blanks, converts numeric constants to internal form, builds a block-structured name-table and lists the source program.

Pass Two does a complete syntactic analysis and extensive error checking. It does all static storage allocation. The output of Pass Two is the completed nametable and a binary tree representing those parts of the program for which code is to be generated.

Pass Three generates the object program in reentrant machine code.

The three passes are written in PL360<sup>1)</sup> as separate programs. The passes use a common data area for data shared by them. This area remains in core if sufficient room is available; otherwise the tree output of Pass Two is written on secondary storage and read segment-by-segment by Pass Three.

The discussion is divided into two sections. Part III describes the design of the three passes. Part IV provides information about the details of the compiler and is devoted primarily to a discussion of the run-time organization and the object code generated by the compiler.

---

<sup>1)</sup> Wirth, Niklaus. "PL360, A Programming Language for the 360 Computers," Journal of the ACM 15 (January 1968), pp. 37-74.



### III. OVERALL DESIGN

#### A. Principal Design Considerations

Following are the main features we wished to incorporate and some of the ways they were achieved,

1. Efficient object code,

All constant arithmetic (e.g.  $5+7$ ) is done at compile time. Global variables are accessed (at run-time) with no overhead. The intermediate language specifies nearly optimal use of the registers, resulting in a minimum of temporary saves. Optimization which involves rearrangement of the source program (for instance, removing computations from for loops) is not done.

2. Code generation only for syntactically and semantically correct programs.

A complete syntactic check and a search for all errors detectable at compile time are completed before any code is generated. Pass Three is called only if no errors are found.

3. Useful tools for numerical computation.

Complex arithmetic in standard mathematical notation and double-precision (long) arithmetic are implemented features of the language. Facilities to detect overflow and make appropriate recovery are provided, as is a set of standard functions of analysis.

4. Fail-safe reliability.

Run-time checks on such things as array subscript bounds, substring operations and formal procedure parameters prevent loss of control (i.e. wild transfers) by the object program.

5. Good diagnostics.

Specific error messages are generated at compile-time and at run-time. All messages give an indication of where in the source program the error occurred. A listing of the parsing stack at the time of a syntactic error can be obtained as a programmer option.

## B. Run-Time Program and Data Segmentation

Program segments and data segments are both logically and physically separate, Program segments correspond to the structural unit "procedure" in ALGOL W. The scope of a data segment is an ALGOL W block containing declarations. Program segments are allocated statically (i.e. once only at compile-time); data segments are created dynamically (i.e. each time the block is entered at run-time).

### C. Pass One

Pass One receives the source program as input in 80 character records, Its functions are to

1. list the character string and assign it line numbers;
2. recognize basic entities of the language and replace them in an output string with byte (8'bit) codes;
3. convert constants to internal form;
4. make a table of identifiers arranged by blocks and containing type and simple type information specified in declarations,,

The input is scanned until a symbol is recognized - i.e. a delimiter, an identifier, or a literal,, In response to this symbol a code representing the symbol is placed in the output string. New blocks are noted, and declared variables are placed in the NAMETABLE which is organized by blocks, A new block is entered at each begin, at the beginning of the formal parameter list in a procedure declaration, and at each for statement, A BLOCKLIST table containing one entry for each block in order of entrance points to the entries in the NAMETABLE corresponding to the identifiers declared in a given block. A table of identifier character strings is also filled for use in Pass One and Pass Two.

## Do Pass Two

### 1. Description of Principles and Main Tasks

The function of Pass Two is to do a complete syntax check of the source program, to do a thorough error analysis and generate all compile-time error messages, to complete the NAMETABLE, to build the constant tables, and to convert the program to an intermediate language to be used by Pass Three for code generation. The syntax analysis is done by means of a simple precedence analyzer. The interpretation rules of the grammar specify the other Pass Two actions.

### 2. Parsing Algorithm

The algorithm for syntactic analysis is essentially that used by Wirth in EULER.<sup>1)</sup> Some program modifications have been made. First, the look-up to determine whether a string is the right part of a production has been changed to include a check on the length of the string and the length of the right part, Second, the full precedence matrix is used rather than the precedence functions. This is done in order to detect errors sooner and to provide better error recovery than is possible with functions. Third, the relations found when scanning to the right looking for > are stacked, Therefore, they can be easily retrieved when in the process of scanning to the left for < rather than having to be fetched again from the matrix. The matrix is packed four elements to a byte in order to conserve space. Consequently, a fetch

---

1) Wirth, Niklaus and Weber, Helmut. "EULER: A Generalization of ALGOL and its Formal Definition: Part I." Comm. ACM 9 (January 1966), pp. 13-23, 25.

from the matrix is slower than retrieval from a stack, However, every time a reduction is made, the relation of the new symbol to the symbol below it on the parsing stack must 'be fetched from the matrix and stacked. If most of the rules that are applied have right parts of length one or two, there is no significant gain in speed by stacking the relations since few unnecessary matrix fetches would have to be done. However, there is a gain in efficiency with longer right parts.

For each syntax rule there is a corresponding interpretation rule which is executed when the reduction is made, For efficiency, Interpretation rules are written directly in PL360 rather than in some metalanguage. Associated with the parsing stack is a parallel value stack containing information used by the interpretation rules.

### 3. Error Recovery

When simple precedence analysis is used, there are two situations in which a syntactic error can be detected - when a reducible substring (i.e. one delimited by  $\langle \text{and} \rangle$ ) is not the right part of any production and when the top of the parsing stack has no relation ( $\langle, =, \rangle$ ) to the incoming symbol. .

In the first situation, the statement in which the error occurred is deleted from the program To accomplish this in ALGOL W, the stack is backed up to  $\langle \text{BLOCK BODY} \rangle$ ,  $\langle \text{BLOCK HEAD} \rangle$ ,  $\langle \text{CASESEQ HEAD} \rangle$ , or the file delimiter and the input string is advanced to end, ";", begin, or the file delimiter. If end is erased from the stack, it becomes the incoming symbol, otherwise the next symbol on the input string is taken, If a nonterminal which affects the value of the block number is removed

from the stack, the block number is adjusted accordingly,

Special care is taken with begin's, end's and the block number so that the block numbers conform to those assigned by Pass One. If the block structure were to be destroyed, many spurious errors would be generated. If Pass One had been done by syntactic analysis, these special fix-ups would be unnecessary provided that Pass One and Pass Two recovered in the same way.

If the top of the stack has no relation to the incoming symbol, a variety of recovery actions are possible. A symbol can be inserted, the top of the stack can be deleted, another symbol can replace the top of the stack, a reduction of the stack can be forced., or the incoming symbol can simply be stacked. The action to be taken is determined by the symbol at the top of the stack. For each symbol in the grammar, there is an entry in table EMTB pointing to a list of recovery actions in table ERTB.

In order for a symbol to be inserted, it must have a relation to the incoming symbol and the top of the stack must have a relation to it. If the inserted symbol is 4 the incoming symbol, the input string is backed up and the inserted symbol becomes the incoming symbol. Similarly a symbol replacing the top of the stack must have a relation on either side,

An inserted or replacing symbol may generate another error message. For instance, an undefined identifier is assumed to be integer although it may be intended as another simple type. If the trace flag is set:, the error recovery action is always printed out unless the incoming symbol was stacked. A flag is set so that the same action will not be

tried the next time through. (e.g. If the top of the stack is <BLOCK-BODY> and it has no relation to the incoming symbol, a ";" may be inserted, "<BLOCKBODY> ;" reduces to <BLOCKBODY> . If the error routine is called again before the input string has advanced, it must not again insert a ";".)

#### 4. Register Analysis

Two register counts are kept for each relevant position in the stack - a count of the integer registers and a count of the floating registers up to that point, The simple type of the operation determines the 'active' set of registers. The active count resulting from a binary operation is determined as follows:

Suppose the active counts for the two arguments are equal - both have value  $k$ . Then  $k$  registers will be needed to calculate the first argument. At the end of that calculation, one register will be in use, containing the value of the first argument. That register remains in use during the calculation of the second argument, Since the binary operation uses only the register containing the first argument, the resulting count is  $k+1$ .

Example  $k_i =$  active count for  $i, k_i \geq 0$ )

integer  $a, b; \dots a + b \dots$

$k_a = k_b = 0$ . To compute the sum it is necessary to load a register with  $a$  and add  $b$  into the register containing  $a$ . Thus

$$k_{a+b} = 1 .$$



Example 2

integer a, b, c, d; . . . (a+b) - (c+d) . . .

$k_{a+b} = k_{c+d} = 1$ , The result (a+b) occupies one register, This register holds the value of a+b while c+d is computed, using another register. Then the register for a+b is subtracted from the register for c+d, leaving the result in the register previously occupied by a+b. Thus ,

$$k_{(a+b) - (c+d)} = 2 .$$

Suppose the active counts for the two arguments are unequal - the counts are  $k_1$  and  $k_2$  where  $k_1 > k_2$ . Then if the argument using  $k_1$  registers is computed first, that result occupies one register leaving  $k_1-1$  registers to compute the second argument. Since  $k_1 > k_2$ ,  $k_1 \geq k_2+1$ , hence  $k_1 - 1 \geq k_2$ . Therefore there are enough registers left to compute the second argument. Hence  $\max(k_1, k_2)$  is the resulting count. (If the other argument were computed first,  $k_1+1$  registers would be necessary.)

Notice that the above reasoning assumes that the operators are commutative (or that appropriate reverse operators exist). Adjustments must be made for some noncommutative operators, For instance DIV and REM require a minimum of two registers if the second argument has count 0 and three integer registers if it has a non-zero count,

The resulting count of the number of 'inactive' registers is the maximum of the counts for the arguments. The counts for an if expression or a case expression are the maxima of the counts of the constituent expressions. Register counts for function calls are set arbitrarily to a large number since all registers in use before a function call are saved,

## 5. Tables

Pass Two completes NAMETABLE, assigning hierarchy numbers, program segment numbers and addresses for variables and descriptors, and inserting array dimensions, local stack origins and record information. A bit string is inserted for every reference variable, indicating positionally to which record classes it may refer. A run-time constant table and a compile-time constant pointer table are constructed for each program. Information local to Pass Two is kept in the interpretation stack rather than in tables,

## 6. Output=

The output of Pass Two is a string called TREE representing the linearization of a modified structural tree of the program being parsed. Each nonterminal node has either one or two subtrees.

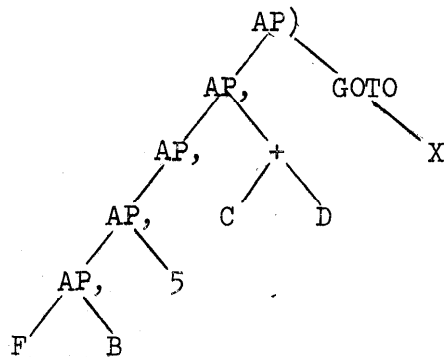
An n-ary construction is represented as a binary tree by making the n components terminal nodes joined by a binary list operator.

### Example

program fragment: F(B, 5, C + D, GOTO X)

where F is a procedure, C is integer, D is real

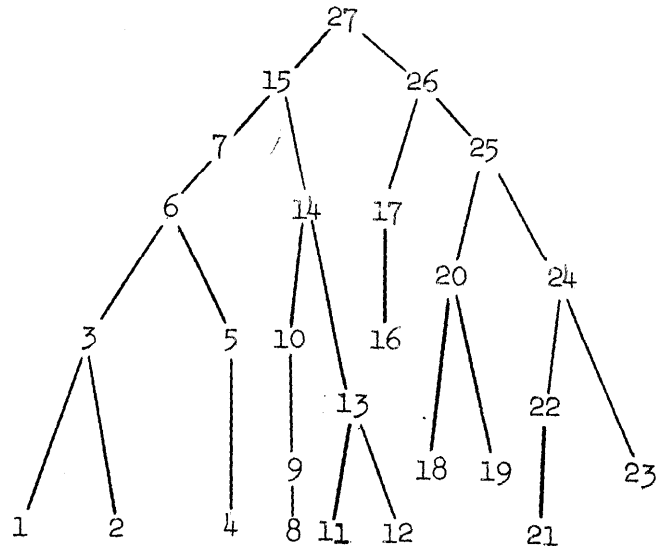
Tree:



where AP, is an actual parameter list operator and AP) indicates the end of the list.

Semantic information is not included in the tree because it is contained in NAMETABLE.

The order in which the nodes occur in the string is shown in the following diagram:



It can be seen that the subtrees of a node precede the node. A nonterminal binary node contains a pointer to its left subtree; its right subtree will directly precede it. Each binary node has a switch indicating which of its subtrees is to be processed first. Nodes are not processed until their subtrees (in most cases arguments) have been processed. The normal mode is to process the left subtree first, thereby preserving the order in which the structures occurred in the source program. The exceptions are binary arithmetic operators and the assignment operators. For these operators, the subtrees represent two operands. In order to minimize register usage, the operand using the larger number of registers is compiled first. (Such optimization

is permissible according to the language definition,<sup>1)</sup> which states that:

"If an operator operates on two operands, then these operands may be evaluated in any order, or even in parallel, with the exception of the case mentioned in 6.4.2.2."

Another motivation for using the tree rather than reverse polish was the hope that it would be a natural way to represent parallelism in the language. This use of the tree was investigated but was not fully developed because it was decided not to implement the parallel features of the language.

A separate tree is generated for each program segment. In theory the program segments (procedures) could be processed by Pass Three in any order; in practice they are processed in the order they occur.

---

<sup>1)</sup> Wirth, N. and Hoare, C.A.R. "A Contribution to the Development of ALGOL", Corn. ACM. 9 (June 1966), 413-432.

### E. Pass Three

The essence of Pass Three is the algorithm for scanning the linearized trees, beginning at the root node. The switch with each binary operator indicates which branch the scan should follow. The operator nodes are not otherwise examined at this stage; code generation begins with the first terminal node encountered,

Pointers to the nonterminal nodes are stacked in STACK as they are encountered in the scan, STACK also contains a field in which information about the first subtree is kept while the second subtree is compiled.

For each binary node there are two phases of code generation. In the first phase the operator is considered together with its first operand; in the second phase the operator and its second operand are considered. Hence there are two compilation (output-generation) rules associated with each binary node, Each unary nonterminal node has one associated rule.

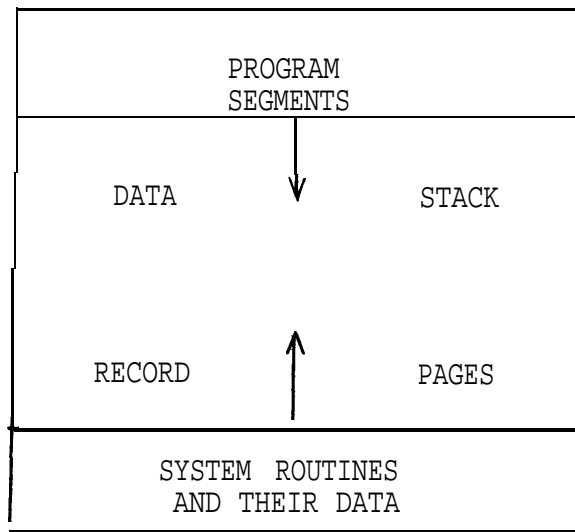
## IV COMPILER DETAILS

### A. Run-Time Organization

#### 1. Program and Data Segmentation

Since no compiled code is modifiable at run-time, all program segments are re-entrant. Data segments are created at block and procedure entry and deleted (by resetting the stack pointer) at block and procedure exit.

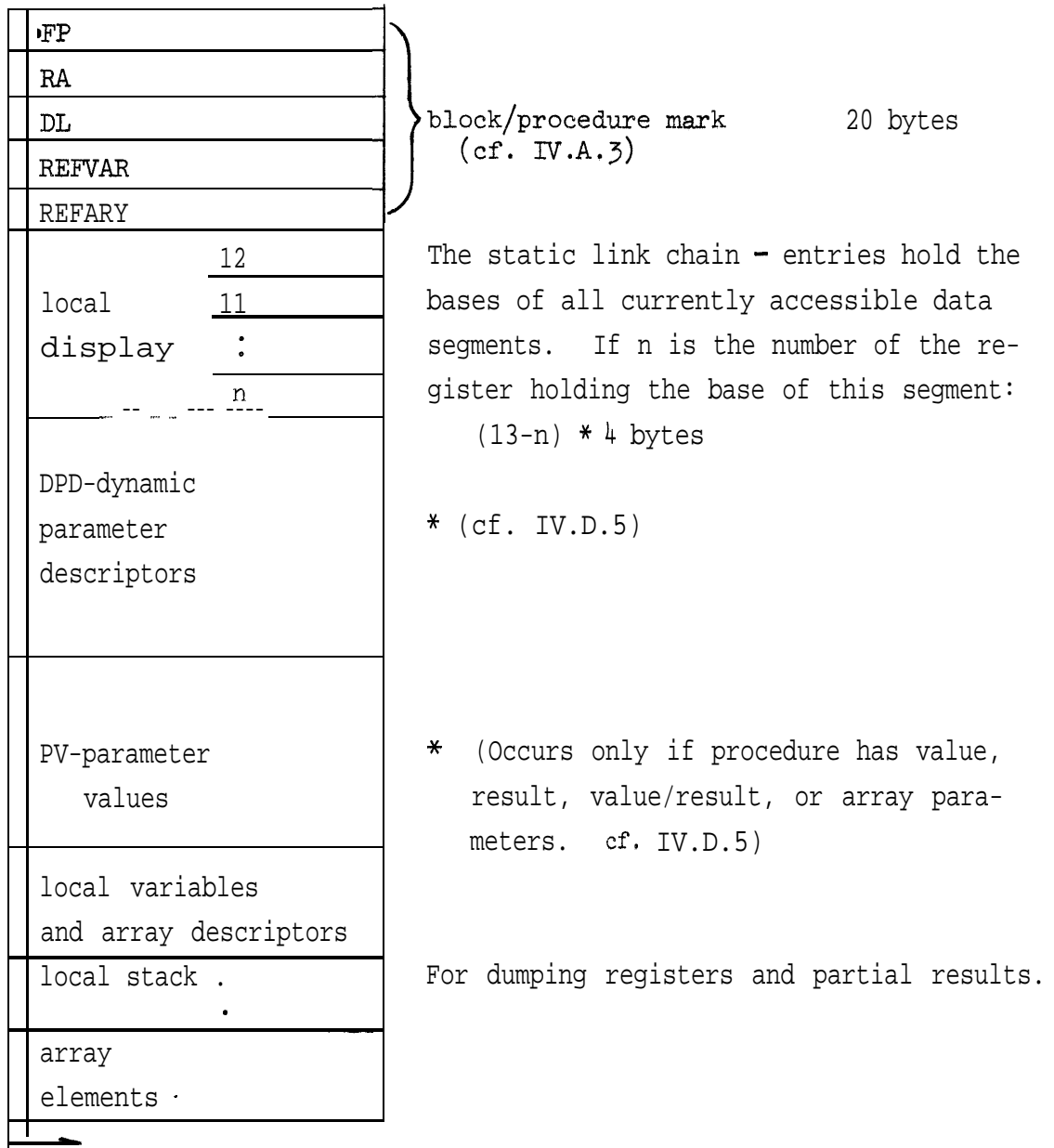
Program segments are allocated statically at the low end of available core. Data segments are then allocated dynamically, beginning just after the program segments and proceeding toward upper core, **Segments** for system routines and their data are allocated statically at the high end of available core. Record pages are allocated dynamically downward beginning immediately before the system routines and system data. If the data stack and the record pages meet, the run is terminated.



AVAILABLE CORE

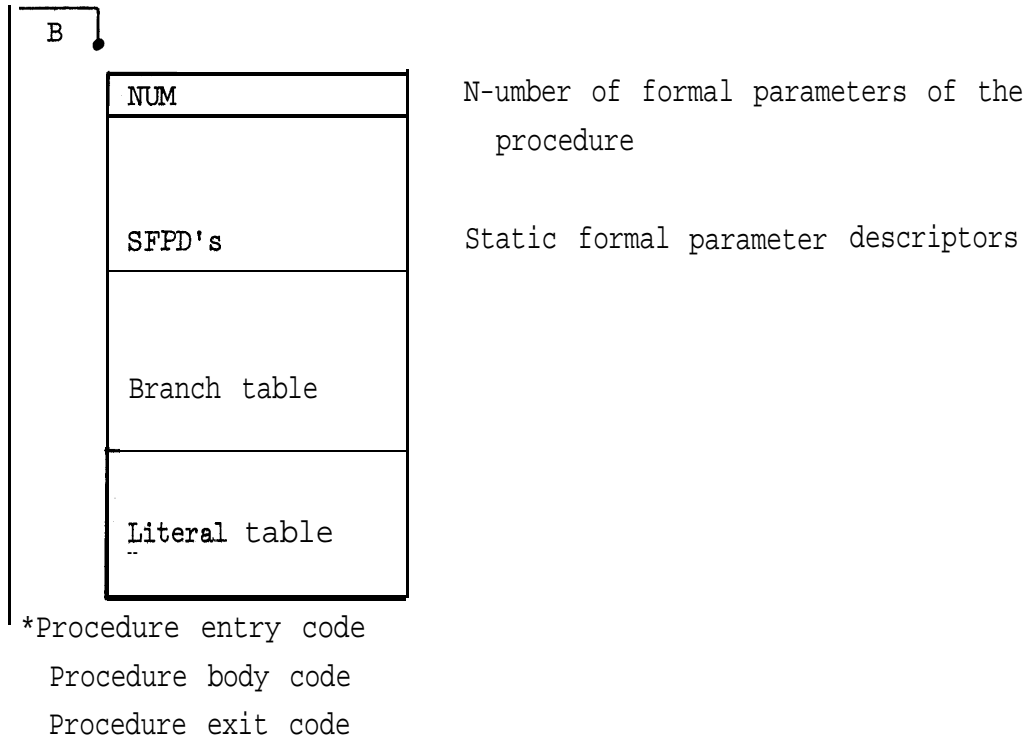
Each block and procedure requires a data segment. When a block occurs as the 'body of a procedure, its data segment is merged with the procedure data segment.

A diagram of a typical data segment is shown below.



\* Occurs only for block which is the procedure body of a procedure with parameters.

Each program segment has the following form:



The static formal parameter descriptors (SFPD's) are one-word descriptors, one for each formal parameter, giving all information needed by the system subroutine CHECK to check the formal-actual parameter correspondence at run-time. This type of checking is done at compile-time by Pass Two for non-formal procedure calls, but must be done at run-time for formal procedure calls.

A branch table exists in the heading of each procedure and contains one branch instruction for each label in the procedure. When a goto statement is executed, a branch is made to the appropriate instruction in the branch table which then branches to the labeled location.

The literal table is a table of all literals (constants) used in the procedure. During execution, each literal is addressed by a displacement relative to the base of the program segment given by R15.



Only one copy of each literal is given.

The literal table is obtained from Pass Two and is placed into the program segment at compile-time by Pass Three.

## 2. Addressing Conventions

Because of the structure of the addressing mechanism in the IBM System 360 Computer, program segments and the statically allocated portion of data segments may not exceed 4096 bytes.

During the execution of a procedure or run-time system subroutine, R15 is a pointer to the base of the procedure or system subroutine. All branching internal to a procedure is accomplished with a displacement relative to the base in R15. Branches between procedures are accomplished by first setting R15 to the base address of the procedure being branched to and then branching.

Upon entering each procedure and block, a data segment is allocated and a general register is assigned to hold the base of that data segment. All local variables, descriptors, and value and result parameters are then addressed relative to the base of the data segment via the general register. Because the base addresses of all accessible data segments are held in registers, all accessible variables are immediately addressable.

## 3. Block and Procedure Marks

At the base of a data segment, a \$-word procedure or block mark is created and filled with all administrative data necessary for the proper usage of reference quantities in the data segment, for the

creation of new data segments while this data segment is active, and for the deletion of the data segment when its corresponding block or procedure is exited.

A mark consists of five full-word fields, as shown in the following diagram.

FP
RA
DL
REFVAR
REFARY

FP: The free pointer field points to the first free byte in the data stack. When a new array or a new data segment is allocated, this pointer indicates its base.

RA: The return address field holds the return address for procedures. This field is not used in block marks but is allocated ~~nonetheless~~ for consistency.

DL: The dynamic link field contains the base of the data segment which was the most recently allocated data segment before the current one. When the current data segment is deleted at an exit from the corresponding block or procedure, the stack pointer is reset to the contents of DL. By tracing backward through the chain of dynamic links, one may obtain the bases of all data segments which have been allocated and not yet deleted. These correspond to all blocks or procedures which have been entered and not yet exited.

REFVAR: The upper two bytes of the field REFVAR contain the number of reference variables local to this block. (Reference value/result parameters are treated as local variables.)  
All reference variables and reference value/result parameters

are grouped together so that the garbage collector may process them. The lower two bytes of the field REFVAR point to the first reference variable or value/result parameter, relative to the base of the data segment. If no reference variables are declared in the block, the REFVAR field is zero.

REFARY : The upper two bytes of the field REFARY contain the number of reference arrays declared in the block. The lower two bytes point to the first reference array descriptor, relative to the base of the data segment. All reference array descriptors are contiguous in the data segment. From the array dimension contained in the first byte of each reference array descriptor, the garbage collector is able to locate all reference array descriptors and hence all the elements in all reference arrays. If no reference arrays are declared in the block, the REFARY field is zero.

#### 4. Array Indexing Conventions

A data segment corresponding to a block in which arrays are declared contains an array descriptor for each array. The descriptor specifies the upper and lower bounds of the indices of the array, and a pointer to the first array element. The size of the descriptor is dependent only upon the number of dimensions of the array; therefore the portion of the data segment used by the descriptor is allocated by Pass Two. At run-time, the bounds are stored into the descriptor, the total number of bytes required for the array elements is calculated, storage is allocated in the data stack, and a pointer to the first array element is placed into the descriptor.

When an array element is referenced, the descriptor is used to calculate the actual address of the array element.

## 5. Base Address Table and Linkage to System Routines

During the execution of a program, a table giving the base addresses of all the user's program segments and the base addresses of all run-time system routines resides at a fixed displacement from R14. The displacement for each segment base is known at compile-time, allowing the compilation of instructions to load R15 with a segment base before branching to that segment,

The standard calling sequence from a user procedure to another procedure or system routine is

```
L      15, d1 (14)
BALR   15, 1
L      15, d2 (14)
```

where  $d_1$  is the displacement of the entry in the base address table giving the base address of the called procedure or system routine and  $d_2$  is the displacement of the entry giving the base address of the calling procedure.

Because of addressability problems, the above code sequence is modified when calling certain system routines. The first load instruction above may be preceded by

```
MVI runtime flag, byte
```

and the second load instruction may be preceded by a halfword of information. The relative origin within the system routine is then established using the value of the run-time flag or the halfword of data.

The instruction BALR 15,1 is replaced by BALR 15,0 for some system routines so that the routines may use their parameters more effectively,

## 6. Special Constants and Error Code

Certain special constants needed at run-time, as well as some run-time error check code, are placed at specified locations based off R14. The inclusion of the constants makes it unnecessary to insert these constants in the literal tables thus saving room in the program segment.

The precise locations relative to R14 of the constants and various run-time entry points into the error checking code are known at compile-time so that the proper addresses may be compiled.

	<u>Constants</u>	
SEVEN	7 }	used to make an address fall on a double word boundary
DUBLMASK	#FFFFFFF8 }	
THREE	3 }	used to make an address fall on a single word boundary
SINGLMASK	#FFFFFFFC }	
ALLONES	#FFFFFFF	used in bit-not operations
NULLREF	#00FF0000	the null reference
ALLCERR	C 0, LIM BCR ≤, 4	used for data allocation; return to point of call (BAL 4, ALLOCERR) if LIM = (beginning of record pages) has not been reached
	IR 1, 4	
	LA 0, 5(0)	error condition
ARRAYERR	BCR ≤, 1	used for run time array bounds checking
MAINERR	L 15, base of ERROR	error routine prints location of error = R1. R0 is parameter to error routine, giving the type of error so that appropriate termination messages may be given.

	BCR 15, 15	
UBLBERR	BCR $\leq$ , 1	used in array declarations to be
	LA 0, 13(0)	sure that upper bound > lower bound.
	BC 15, MAINERR	Error condition.

## 7. Register Usage

At run-time the following uses are made of registers:

R0 and R1 are used by the system as save and link registers for system subroutines. They are otherwise available for local use.

R2 - R6 and F0 - F6 are used in evaluating arithmetic expressions.

R7 - R13 hold the run-time display pointers to all data segments which at any given time are accessible to the block being executed.

R13 always holds the base of the data segment of the main program block.

R7 - R12 are allocated statically downwards from R13. word "statically" is emphasized since data segments are created dynamically and the size of the data stack is limited only by the physical size of available memory. Any two or more parallel blocks (or procedures) will have the same display register pointing to their data segments, since only one of those data segments may exist at any one time.

It should be remembered that the data segments for a procedure and its outermost block (if there is one) are merged into one data segment.

In the following diagram the numbers represent data segment base

registers. Each begin is assumed to be followed by one or more declarations.

```
13 begin
    procedure P
      12 begin
          11 begin
              end
          end
      procedure Q
          12 begin
              procedure P
                  11 begin
                      end
                  11 begin
                      10 begin L:
                          end
                      end
                  end
              end
          12 begin
              11 begin
                  10 begin
                      procedure S
                          9 begin
                              8 begin
                                  end
                              end
                          end
                      end
                  end
              end
          end
      end
  end
```

Those registers not in use as display registers are available for arithmetic evaluation, For example, at label L in the preceding dia-

gram, R10 - R13 are in use as display registers, and R2 - R9 are available for arithmetic evaluation.

R14 always points to an area in memory which contains:

1. the base address table,
2. special constants,
3. error codes, and
4. local data for system subroutines.

R15 always holds the base of the program segment currently being executed.

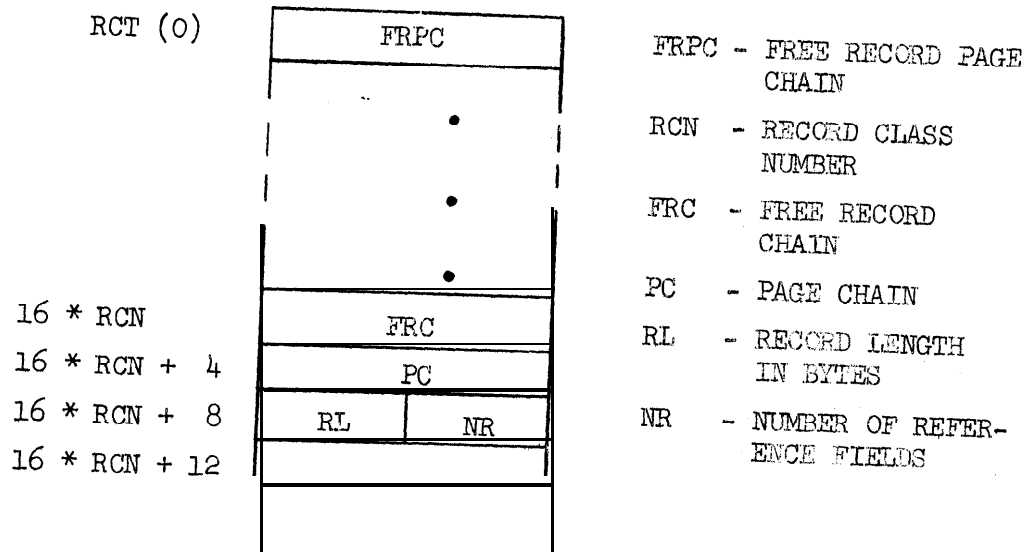
At particular points in the execution of a program when it is known that none of the arithmetic evaluation registers are in use (such as at procedure entry and exit, block entry and exit, and in a procedure call), they may be used by the run-time administration.

## 8. Record Allocation and Storage Reclamation

Space for records is allocated by pages beginning at the end of core working downward. Size of the pages is a parameter of the run-time routines. As each page is allocated, the pages are formatted so that each record on the page is pointed to by a previous record or by the FRC (see below). Each page is dedicated to one record class.

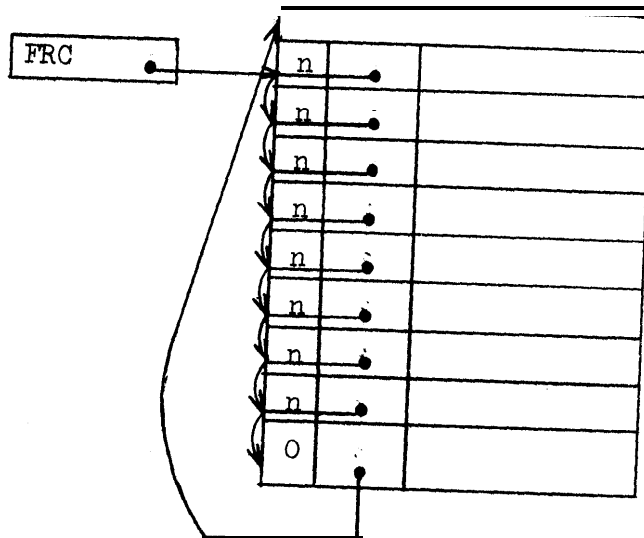
Table RCT is prepared by Pass Three and loaded along with the compiled program. It contains a 16 byte entry for each record class declared and is indexed by record class number. No record class 0 exists. This allows RCT(0) to be used for a free record page chain. RCT contains the following information about each record class:





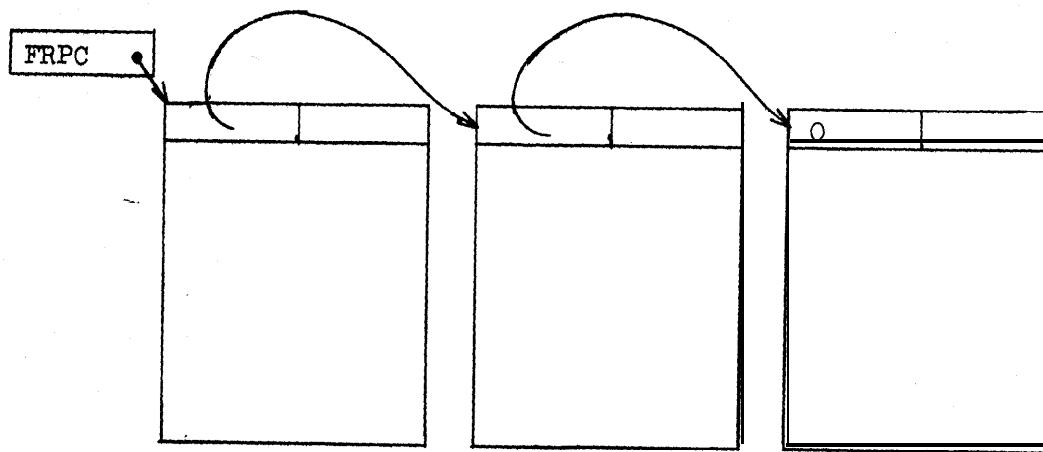
FRC, FRPC, and PC are initialized to 0. The last entry in the table is set to #FFFFFFFF when fewer than 15 record classes exist.

FRC is the origin of the Free Record Chain for the given record class.



where n is the record class number and each list element is a record of class n.

FRPC is the origin of the free record page chain. Each page on the chain is a page whose origin address is greater than at least one of the pages in use. This chain always releases as many pages as possible to free storage so that free storage may be used by either data segments or record pages as needed. A record page which was allocated and later released may then be used for data segments.



A new reference to a record class is always obtained from the FRC corresponding to that class. If the upper byte of FRC is 0, the garbage collector is called. If the garbage collector cannot free enough storage for a new reference, execution is terminated.

Storage reclamation (i.e. garbage collection) consists of three phases: marking used records, collecting unused records, returning unused pages. For each call of the garbage collector all record classes are searched and the FRC of each record class is updated.

Records are marked in two steps. First, each reference variable and each reference array element is tested; for each non-null reference, the first bit of the record referenced is set to 1. The first byte of each record is not allocated for fields and is available.

When a record is marked which had not been previously marked, a check is made of the NR field corresponding to the record class. If this field is zero, nothing more needs to be done. If this field is non-zero, each reference field of the record must be checked. The reference fields are checked starting with the last reference field and ending with the first reference field. Each reference field in turn is treated recursively as a reference variable. The last reference field has been processed when the marking bit of the record is encountered. This test restricts the number of record classes to at most 127.

Since the reference fields of a record are checked when the record is marked, a backward chain must be kept so that the path may be retraced and all reference fields of each record inspected. This chain consists only of the three low order bytes of the reference. The high order byte remains unchanged. Before proceeding to inspect the fields of a new record B designated by a field of record A, the address of the record inspected previous to A replaces the reference field in A designating the new record. If the record A had been designated by a simple reference variable or a reference array element, zero replaces the reference field in A.

e.g. record sample (reference (sample) one, two)  
reference (sample) R;

Let A, B, C, D be symbolic names for record addresses of class sample and let N be the null reference. Suppose Example 1 represents the situation when the garbage collector begins. Reference R is inspected and points to record A of class n (i.e., sample). Record A is

marked (first bit on). The last reference field of A (two(A)) is checked first. Two(A) points to a previously marked record, namely A. Then one(A) is tested and points to record B which is still unmarked. A zero is placed in the 24 bit address field of the reference. Record B is marked. Two(B) points to the record C which is unmarked. The address of A replaces the address of C in two(B). The process is repeated until record D is marked and its fields tested. Example 2 represents this state. A return is made up the chain until each field of each record involved is checked and until the zero field in record A is encountered and changed, At this point, the result is similar to Example 1 except the first bit of records A, B, C and D is on.

All references in a block are scanned before following the dynamic links to a previous data segment., When the dynamic link is zero, the process is completed.

Phase One of the garbage collection is completed by looking at each record. The second bit of each record is used to protect records which have been created but not yet assigned to a reference location or used in some other manner. Therefore, each record must be scanned to inquire if this bit is on; if so, the record is marked and its reference fields scanned as previously described.

In Phase Two, any record whose first bit is not 1 is put on the free list for its record class. Phase Three is integrated with Phase Two. If any record page has no used records, it is returned to the free record page chain. Furthermore, if the page adjoins the free

space for data segments, the page is returned instead to the free space for data segments. In this case, the free record chain is checked for record pages adjoining the free space for data segments. Those found are removed from the FRPC and given to the free space.

After all the storage reclamation is complete, the garbage collector must supply a record of the class desired. If no free record of the class desired exists, a new page is allocated for this record class and placed on the class's page chain. If no space for a new page is available, execution is terminated.

Example 1

n	A
---	---

A	0	n	B	n	A
B	0	n	D	n	C
C	0	0	N	n	D
D	0	0	N	0	N

Example 2

n	A
---	---

A	80	0n	0	n	A
B	80	0n	D	n	A
C	80	00	N	n	B
D	80	00	N	0	N

## B. Pass One

The output of the compiler's first pass is

- 1) a listing of the source program with each line numbered beginning at 1,
- 2) a character string representing in detail the original source code,
- 3) a nametable, having an entry for each identifier, arranged by blocks,
- 4) a blocklist table which indexes the nametable by blocks,
- 5) a table listing the record classes to which the declared references are bound.

Other tables are passed on by Pass One but have significance only in producing trace output in Pass Two.

Pass One makes decisions as to the size of the tables based on the size of the core available, The algorithm used is

```
CB = commonbase
LC = last core location available
cs = common size
```

```
cs := LC - CB;
If cs >= #30000 then CS := #18000 else CS := CS DIV 2;
NAMETABLE := CB + NTORIGIN;
IDDLISTBASE := ((cs DIV 3 + CB + NTORIGIN) DIV 8) * 8;
REFRECBASE := IDDLISTBASE + ((CS DIV 24) DIV 8) * 8;
IDDIRBASE := 2 * REFRECBASE - IDDLISTBASE;
INPOINT := IDDIRBASE + 3 * ((CS DIV 24) DIV 8) * 8;
PASSTWOOUTPUTBASE := (ADDRESS OF END OF PASS ONE OUTPUT) DIV 8 * 8;
```

If the Pass Two output area is not at least twice as long as the Pass One output area, a flag is set so that Pass Two output will be on tape.

## 1. Table Formats Internal to Pass One

Four main tables direct the work of Pass One. Two are initialized at entrance. They are the table RESERVED of the EBCDIC representations of the delimiters or reserved symbols and the table CODE containing an entry corresponding to each reserved symbol. Two other tables are partially initialized at entry to Pass One and added to during its execution. They are the identifier directory IDDIR which has the EBCDIC representation of each identifier, and IDLIST which indexes IDDIR.

The table RESERVED is divided into segments which accommodate the ALGOL W symbols grouped (alphabetically) by length. Hence RESERVED1 contains all the symbols of length 1 such as :, =, (. RESERVED2 contains all symbols of length 2 such as do, go, if. This arrangement continues through RESERVED9 containing -procedure, reference. Once a match is found in the RESERVED table, a 2-byte entry corresponding to the reserved symbol is found in CODE. For example in Figure 3, the corresponding CODE entry for if is hexadecimal 6401.

In most cases, the first byte of the CODE entry represents the one-byte output code for the ALGOL W symbol. This code corresponds to the symbol number of the ALGOL W symbol in the syntactic productions of Pass Two. The exception to this rule occurs with the RESERVED entries representing the simple types such as integer, real, logical. These symbols are represented in the output string by the same character. Instead, the first byte of the CODE entry gives the simple type number (see Figure 1). In the example of if, 64 is its output string representation.

The second byte of %he CODE entry is used as an index to a case statement. The hexadecimal value 01 means no special processing takes place. Such is the case in the example of if. Any other value means that some special note must be made of this symbol such as to enter declaration mode or to declare a control variable. These special situations are described in the following pages.

IDDIR is a character array of all identifiers predefined or occurring in the program being compiled. The list is arranged so that if only the identifiers SQRT, A, TILDA appeared, the IDDIR table would appear as SQRTATILDA and the index to %he table would have a value equal to the number of characters relevant - in this case, 10.

IDLIST indexes IDDIR by an array of full words with one entry corresponding to each identifier. The first half word of each entry is the length of the identifier minus 1. The second half of the entry is a pointer to the first character of the identifier. Hence, in Figure 4, the entry (4) (5) corresponds to TILDA with the length specification of 4 and pointer value of 5. Also in Figure 2, note that IDLIST INDEX is a pointer to IDLIST = 8.

Figure 1

Reserved Word Tables

<u>RESERVED (in EBCDIC)</u>		<u>CODE (in hexadecimal)</u>		
RESERVED1	( + *	CODE1	5506	4F01 5005
RESERVED2	DØ IF	CODE2	6301	6401
	⋮			
RESERVED9	PRØCEDURE	CODE9	8515	



Figure2  
Identifier Tables .

IDDIR	:	SQRTATILDA	IDDIRINDEX = 10
IDLIST	:	(3) (0)	IDLISTINDEX = 8
		(0) (4)	
		(4) (5)	

2. The Output String Representing an ALGOL W Program

The characters of the output string representing an ALGOL W source program are the numbers which correspond to the syntactic elements in Pass Two. For most cases, there is a one-one correspondence between the ALGOL W symbols and their codes. As an example, Figure 3 shows that do is represented by hexadecimal 93. Some codes represent two ALGOL W symbols. These are exponentiation, '\*\*', and assignment, ':=', and the bound pair colons, '::'. The following list itemizes the other special situations requiring modification of the normal correspondence between ALGOL W symbols and string representation.

1. The reserved words and reserved word pairs, integer, real, long real, complex, long complex, logical and bits receive the code for <simple type>.
2. Each identifier is replaced by a 3 byte code. The first byte is a code for <identifier>. The following two bytes contain the unique identifier number, (Starting from 0). In Figure 4, the identifier number of A would be 1.
3. Each number is represented by a 1 byte code for <number>. followed 'by a 1 byte indication of the type of the number, followed by the number.

4. Each bit sequence (e.g., #FA12C (in hexadecimal)), results in a 1 byte code representing <bit sequence, followed by the 4 byte literal,
5. A comma appearing in the identifier list of a declaration or in the record class specification of a reference declaration receives the code designated SPECComma.
6. In a reference declaration, the left parenthesis preceding the record class specification is omitted from the output string .
7. In a string declaration, if the length is specified explicitly, the entire length specification, (number), is omitted from the output string,
8. Each new card is indicated in the output string by a 3 byte code . The first byte specifies 'new card' and the following 2 bytes give the card number.
9. The reserved word comment and all characters up to and including the next semicolon are omitted from the output string.
10. An identifier following the reserved symbol end is omitted from the output string.
11. A period (.) following the reserved word end is recognized as the end of program.

Figure 3  
Output Codes

;	70	ABS	8D	RECORD	75
(	6A	AND	86	RESULT	73
)	67	DIV	84		
:	99	END	6F	PROCEDURE	71
=	90	FOR	9B	REFERENCE	68
+	7E	REM	85		
-	7F	SHL	88	SPECCOLON	6D
*	74	SHR	89	SPFCOMMA	9A
/	83			ASSIGNMENT	9A
,	69	CASE	7B		
<	8F	ELSE	7A	END OF FILE	92
>	91	FILE	6C	EXPONENT!	88
█	76	GOTO	94	LINE MARK	FE
#	8E	LONG	8C		
"	81	NULL	82	NUMBER	77
┘	87	STEP	9C	IDENTIFIER	65
		THEN	79	STRING SEQ	81
DO	93	TRUE	8A	BITS SEQ	8E
IF	78			SIMPLETYPE	0D
IS	7D	ARRAY	6E		
OF	7C	BEGIN	97		
OR	80	FALSE	8B		
		SHORT	9F		
		UNTIL	9D		
		VALUE	72		
		WHILE	9E		

### 3. The Table Output of Pass One

**Three tables are part of the necessary output of Pass One:**

NAMETABLE, BLOCKLIST (which indexes NAMETABLE), and RCCLIST,

The BLOCKLIST table has a one-word entry for each block in the program in the order encountered. (Each program has a predefined outer block numbered 0 containing predefined symbols such as WRITE and SQRT.) This full-word entry is divided into two half-word fields. The second **field** points to the first **'byte of** the entries in NAMETABLE corresponding to identifiers declared in the block. The **first** field is equal to 12 times the number of identifiers **declared in** the block (i.e., the length of the NAMETABLE entry for the block). If **no** identifiers are declared, both fields are zero. In Figure 4, the first BLOCKLIST entry points to WRITE and encompasses both WRITE and SQRT which are predefined. The second BLOCKLIST entry points to i, and encompasses i, j declared in the outer block of the program. The third entry corresponds to the control variable i.

The entrance and exit to blocks are defined by the following rules.

- a) Each 'begin signifies the entrance to a block and the corresponding end signifies the close of the block,
- b) Each statement following a <for clause> is surrounded by a block in which the control variable is implicitly declared.
- c) Each procedure body is surrounded by a block in which its formal parameters, if any, are declared.

In the NAMETABLE all identifiers declared in a block are grouped together, Therefore the permanent entries in the NAMETABLE cannot **'be** made until the block closes. If viewed **'by** blocks, the identifiers in

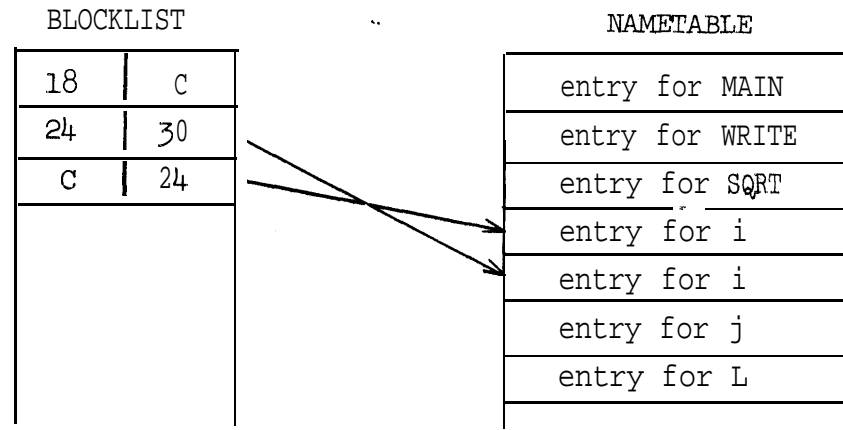
the NAMETABLE are listed in order of the closing of the blocks. In Figure 4, the control variable block closes before the outer block and, hence, appears in the NAMETABLE first.

The layout and field contents of NAMETABLE are shown in Figure 5. Pass One puts in only that information required by Pass Two to check the semantic correctness of the program. Many fields are filled by Pass Two. The information entered during Pass One consists of the following attributes appropriate to the variable.

- |             |   |
|-------------|---|
| IDNO        | - The number assigned to the identifier.<br>This number is equal to the number of the IDLIST entry.   |
| SIMPLETYPE  |   |
| TYPE        |   |
| TYPEINFO    | - block number of the formal parameters of a procedure. Simple type of the argument of a standard function.<br>a) Value-result<br>for formal parameter<br>1. if value<br>2. if result<br>3. if value-result<br>b) Record class number<br>for record class identifiers, the record class number<br>for record fields, the record class number. |
| SIMTYPEINFO | - a) for string, length -1<br>b) for a reference, a pointer to the RCCLIST.   |

Figure4

Example of BLOCKLIST and NAMETABLE



```

begin integer ;
    j := 0;
    for i := 1. do j := j + 1;
L: end.

```

Each entry of RCCLIST is a half-word which gives the IDNO of a record to which the reference is bound. A zero entry signifies the end of the group. The NAMETABLE entry for a reference variable contains a pointer to the first entry of RCCLIST for that variable,

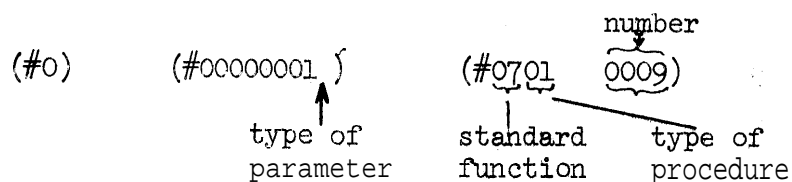
#### 4. Introducing Predefined Identifiers

To introduce in the compiler new predefined identifiers such as standard functions or standard procedures, a series of changes must be made in Pass One.

1. The EBCDIC code of the identifier and its length must be added to array IDLISTFILL.

2. Two half-word entries corresponding to the identifier must be added to IDDIRFILL. The first half-word is the (number of characters -1) in the identifier. The second half-word is the (sum of the preceding pairs of entries +1).
3. IDDIRINDEX must be initialized.
4. IDLISTINDEX must be initialized to be equal to the (sum of the last pair of IDDIRFILL entries +1).
5. A 12 byte entry (3 integers) must be added to NAMETFILL as described in the description of the NAMETABLE entries (cf. IV.C.5).

For example the entry for ROUND is:



6. BLFILL must be changed to be initialized to (#aaaa000C) where aaaa is the hexadecimal representation of the (number of integers -3) declared for NAMETFILL) \* 4.
7. SYMBOLINDEX must be initialized to the (number of integers declared for NAMETFILL) \* 4.
8. In the initialization section of the algorithm, the initialization of IDDIR, IDLIST, and NAMETABLE must be corrected to represent the length changes.

## C. Pass Two

### 1. Storage Allocation

All static storage allocation for variables and constants is done by Pass Two. For this purpose a number of counters and link tables are necessary,

BNC contains the current block number (cf. IV.B). BN contains the highest block number assigned so far (necessary in order to set BNC when a new block is entered). BLOCKLIST2 contains static links for blocks. These are necessary to restore BNC to the current block,

Program segment numbers are assigned by Pass Two. Each procedure constitutes a separate program segment and is assigned a unique number. SNC contains the current segment number; SN contains the largest segment number already assigned, SNLIST contains static links for program segments.

The hierarchy number *represents* the level of nesting of data and in actuality is the number of the base register used to access the data segment. HN contains the current data hierarchy number.

DRELAD contains the address of the first free byte relative to the beginning of the current data segment. DRELSAVE is a stack used to save values of DRELAD while parsing actual parameter lists. DRELPOINT contains a pointer to DRELSAVE. While a record class declaration is being parsed, RELAD contains the current address relative to the beginning of the record class layout,

All addresses of variables, array descriptors, and other data are indicated in NAMETABLE. An address consists of the hierarchy number



(base register number) plus the address relative to the beginning of the data segment (displacement). Reference variables are grouped together at the head of the data segment; other variables occur in the order in which they are declared in a block. A location is allocated for each control identifier as well,

Fields of records are given addresses relative to the origin of the record. Field addresses are first assigned to reference fields, then to logical and string fields, then to other fields. The first byte of the record or the two high-order bits of the first reference (if there is one) are reserved for the garbage collector.

The length in bytes of any record in a record class is indicated in the `NAMETABLE` entry for the record class. The length is always a multiple of 8.

Labels are given an address relative to the beginning of the program segment in which they occur. The location is used for indirect transfers.

The dimension of an array is inserted in `NAMETABLE` when the first array designator or the declaration is encountered (whichever occurs first). This information is subsequently used to compute the length of the descriptor (and to check the number of dimensions each time that array identifier occurs).

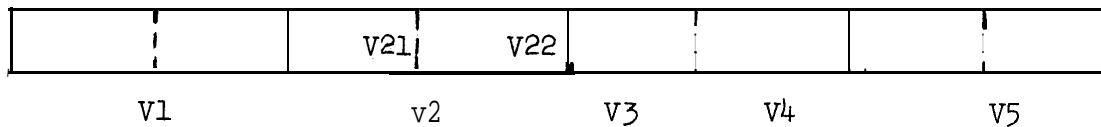
Storage is allocated in the program segment of a procedure for descriptors of its formal parameters. Descriptors of actual name parameters are assigned addresses relative to the beginning of the data segment of the procedure. Space is allocated in the data segment for values of the actual value and result parameters, since they are treated like local variables while control is within the procedure

body. Value and result parameters of simple type "reference"! follow all others so as to be adjacent to the local reference variables.

The first free location following the variables in each data segment is the origin of the local stack (temporary storage) for the data segment. Its address is indicated in NAMETABLE for the outermost data segment of a procedure and in the associated begin output node otherwise.

## 2. Value Stack

The value or interpretation stack consists of 8-byte elements. This stack works in parallel with the parsing stack.



The standard uses for the fields are described below, although the actual uses vary with the construction being parsed.

- V1 Simple type information
- V21 Type
- V22 Simple type
- V3 Integer register count
- V4 Floating register count
- V5 Output pointer

When an identifier is looked up in NAMETABLE, a pointer to NAMETABLE is inserted in V1, V2 is filled, and V3 and V4 are set to zero. When a node is put in the output array TREE, the tree pointer is put in V5.

## 3. Interpretation Rules

Associated with each syntax rule is a body of code, the interpretation rule, which performs the semantic actions appropriate to the

syntactic construction, The interpretation rules are contained in procedures EXECUTE1, MECUTE2, and EXECUTE3 and are accessed via a case ~~statement~~ indexed by the rule number. (Three procedures rather than one are necessary because of the addressing structure of PL360.)

The interpretation rules use the value stack for working storage, Semantic actions and value stack layouts for major constructions of the language follow:

1. Simple variable declaration
  - a. Layout is standard
  - b. Each identifier is located in NAMETABLE, checked for multiple declaration, and allocated storage, No output is generated,
2. Array declaration
  - a. Layout
    - V1 pointer to NAMETABLE entry of first identifier
    - V2 current block number of block containing declaration
    - V3 number of identifiers
    - v4 dimension
    - V5 output pointer
  - b. The identifiers are counted, the simple types of the bound pair expressions are checked, the bound pairs are counted, storage is allocated for the descriptors, the array dimension is inserted in NAMETABLE for all the identifiers, and output is generated for the structure.
3. Procedure declaration
  - a.1 Layout of procedure head
    - V1 simple type information (if typed procedure)
    - V21 type (i.e. code for procedure)
    - V22 simple type (if typed procedure)
    - v3 & v4 current DRELAD of procedure head (mark, descriptors, etc.)
    - V5 output pointer

a.2 Layout of procedure body

V1 simple type information of expression (if typed procedure)

V2 0

v3 & v4 DRELAD of procedure body

V5 output pointer

- b. The counters and pointers are stacked, storage is allocated for the descriptors of the formal parameters, record class masks are constructed for reference parameters (cf. IV.C.4), the relative origin of the label transfer table is computed, the simple types (for a typed procedure) are compared, the output for the procedure and the literal table are generated, the counters and pointers are restored, and the output is (optionally) listed.

4. Record class declaration

a. Layout

V1 pointer to NAMETABLE for current field

V2 current RELAD

v3 & v4 not used

V5 pointer to NAMETABLE entry of record class identifier

- b. The identifiers are located in NAMETABLE and checked for multiple declaration, storage is allocated for the record class identifier, relative addresses are assigned to the fields and the number of fields is inserted in the NAMETABLE entry for the record class.

5. Substring designator

a. Layout is standard

- b. The simple types of the simple variable, the index expression, and the length are checked, the length is checked against the length of the simple variable, and output is generated for the structure.

6. Field designator
  - a. Layout is standard
  - b. The simple type of the reference is checked, a check is made that the reference expression can point to a record of the record class containing the field, and output is generated for the structure.
  
7. Array designator
  - a. Layout (replaced by standard layout after structure is parsed),,
    - V1 pointer to NAMETABLE
    - V21 number of \*'s
    - V22 number of subscripts remaining, #FF if dimension unknown
    - V3,V4,V5 standard
  - b. The subscripts are counted (in NAMETABLE) if dimension is not already known; otherwise the number of subscripts is checked against the dimension The simple type of each subscript is checked, register counts are computed, and output is generated for the structure,
  
8. Function designator and Procedure statement
  - a. Layout (replaced by standard layout after structure is parsed),
    - V1 simple type information (if typed procedure)
    - V21 contains #FF if too many actual parameters, number of parameters yet to come otherwise,
    - V22 simple type (if typed procedure)
    - v3 & v4 pointer to NAMETABLE entry of current formal parameter if it is actual procedure, 0 if it is formal procedure
    - V5 output pointer
  - b. If the procedure is not formal the number of parameters and their types are checked, output for the structure is generated.

9. If expression
  - a. Layout is standard
  - b. Simple types of then expression and else expression are checked for type compatibility, type conversion is indicated if necessary, simple type of expression in if clause is checked, output is generated.
  
10. Case expression
  - a. Layout
    - V1            simple type information
    - v21          number of cases
    - V22          simple type
    - V3,V4,V5 standard
  - b. Simple type of expression in case clause is checked, cases are counted and simple types are checked for compatibility, register counts are adjusted, output is generated.
  
11. argument1 [=, >=, <, <=, >, and, or, +, -, \*, /, shr, shl, div, rem, \*\*] argument2
  - a. Layout is standard
  - b. Simple types of arguments are checked, type conversion is indicated where necessary, register counts are adjusted, order of compilation is indicated, and output is generated.
  
12. [ -, ¬, long, short, abs] argument1
  - a. Layout is standard
  - b. Simple type of argument is checked, output is generated.
  
13. Record designator
  - a. Layout (replaced by standard layout after structure is parsed).
    - V1            pointer to NAMETABLE entry for current field
    - v21          number of fields
    - V22          record class number
    - V3,V4,V5 standard
  - b. The number of fields is checked, the simple type of each field is checked, conversion is indicated if necessary, register counts are adjusted, and output is generated.

14. Blockbody
  - a. Layout
    - V1 not used
    - V2 0 if no declarations, #F if enclosing block of procedure body (with declarations), #FF otherwise
    - v3 & v4 DRELAD of surrounding 'block
    - V5 output pointer
  - b. At begin BN, BNC, and HN are stepped, V2 and DRELAD are set, storage is allocated for reference variables, and record class masks are constructed (cf. IV.B.4). At end, DRELAD and HN are restored. Output is generated for **structure**.
15. Label definition
  - a. Layout is standard
  - b. Storage is allocated for transfer, SNC and HN are inserted in **NAMETABLE**, output is generated.
16. Assignment statement
  - a. Layout is standard
  - b. Simple types are checked for compatibility, register counts are adjusted, order of compilation is indicated, output is generated,
17. Case statement
  - a. Layout is same as for case expression.
  - b. Cases are counted, output is generated.
18. For statement
  - a. Layout is standard
  - b. Simple types of expressions are checked, storage is allocated for control identifier, output is generated.
19. While statement
  - a. Layout is standard
  - b. Simple type of expression in while clause is checked, output is generated.

#### 4. Pass Two Tables

Pass Two completes NAMETABLE and creates literal tables.

The information entered in NAMETABLE consists of those of the following fields appropriate to the variable, For field contents and table format, see Figure 5.

1. IDLOC1
2. IDLOC2
3. SIMTYPEINFO
  - a. for a record class identifier, the record length is inserted
  - b. for a reference, the pointer to RCCLIST (a list of record classes to which the reference may point) is replaced by a 16 bit mask in which each bit position represents a record class and is a 1 if the reference may point to records of that class.
4. TYPEINFO
  - a. for a label, the hierarchy number is inserted
  - b. for an array, the dimension is inserted
  - c. for a record class identifier, the number of fields is inserted.
5. TYPE
  - a. for a formal value/result parameter, the TYPE code is replaced by the code plus 16.

Two tables to handle literals are constructed for each program segment. The literal table contains all literals (numbers, literal strings and bit sequences) occurring in the program segment. At run-time it is located before the program segment code. The literal pointer



table is used by Pass Three and contains the simple type, the length (if the literal is a string), and a pointer to the literal table for each literal. The integer 1 and the logical values occur in every literal table. Pass Two uses the stack CONSPINTERSTACK to save the pointers to these tables when a nested program segment is parsed.

Figure5

FORMAT OF NAMETABLE AND FIELD CONTENTS AFTER PASS TWO

12 bytes/entry

	IDLOC1		IDLOC2
		hierarchy	prog seg
	SIMTYIEINFO		TYPINFO dimen
		vr	rccl number
TYPE	SIMPLETYPE		IDNO

<u>FIELD</u>	<u>KIND OF ENTRY</u>	<u>CONTENTS</u>	
IDLOC1	simple variable	hierarchy number	
	label	program segment number	
	array	hierarchy number	
	procedure	origin of local stack	
	record class identifier	hierarchy number	
	record field	hierarchy number	
	control identifier	hierarchy number	
	standard function	simtypeinfo of argument	
	IDLOC2	formal parameter	hierarchy number
		simple variable	relative address
label		relative address	
array		relative address of de- scriptor	

<u>FIELD</u>	<u>KIND OF ENTRY</u>	<u>CONTENTS</u>
	record class identifier	relative address
	record field	address relative to origin of record
	control identifier	relative address
	formal parameter	relative address of descriptor or value/result
hierarchy	procedure	hierarchy number
prog seg	procedure	program segment number
SIMTYPEINFO	string	length -1
	reference	record class mask
	record class identifier	record length
TYPEINFO	label	hierarchy number
	procedure (not formal)	block number of formal parameters
dimen	array	dimension
rcclnumber	record class identifier	record class number
vr	record class identifier	number of fields
	formal parameter	1 if value, 2 if result, 3 if value/result
	standard procedure	vr for parameters
TYPE	simple variable	0
	label	1
	array	2
	procedure	3
	record class	4
	record field	5
	control identifier	6
	standard function	7
	standard procedure	9
	formal name parameter	16 + TYPE number
SIMPLE TYPE	integer	1
	real	2
	long real	3
	complex	4

<u>FIELD</u>	<u>KIND OF ENTRY</u>	<u>CONTENTS</u>
	long complex	5
	logical	6
	string	7
	bits	8
	reference	9

NOTE: The SIMTYPEINFO entry for a reference variable and the TYPE entry for a formal value/result parameter are changed from their contents at the end of Pass One.

The tables PRTB, MTB, and MATRIX: are used by the syntactic analyzer and are initialized upon entry to Pass Two. MATRIX contains the simple precedence relations of the ALGOL W (simple precedence) grammar (cf. Appendix 2). The array is packed two bits per entry. PRTB contains the productions of the simple precedence grammar grouped so that all productions having the same leftmost symbol of the right part are together. The format for a production is the following

production:  $L ::= R_1 R_2 \dots R_n \quad 1 \leq n \leq 5$

representation in PRTB (one byte per entry):

n-1

$R_1$

$R_2$

$R_n$

L

production number

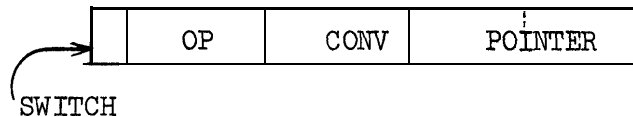
The symbol #FF indicates the end of a production group, MTB is an index to PRTB. The entry for a given symbol indicates the beginning

of the group of productions of which that symbol is the leftmost symbol of the right part.

**METATABLE** contains the EBCDIC representation of the symbols of the simple precedence grammar and is used for printing out the parsing stack. **OPTABL** contains the EBCDIC representation of the Pass Two output nodes and is used for printing out the tree. Both tables are initialized upon entry to Pass Two

### 5. Output of Pass Two

Each element of the output string **TREE** consists of a four-byte word with the following format:



**SWITCH** is on (1) if the right **subtree** is to be compiled first and off (0) if the left **subtree** is taken first. Conversion of arithmetic type may be indicated in the source program implicitly, by mixed-type expressions, or explicitly, by the operators long or short. In either case, the simple type to which the expression is to be converted is indicated in **CONV**. For a terminal node **POINTER** points to **NAMETABLE** or the literal pointer table; for a nonterminal it points to the last node of the first **subtree**.

Example

program fragment and tree - previous example (cf. III.D.6)

output substring:

SWITCH	OP	CONV	POINTER
	FUNCID		points to table entry for F
	VARID		points to table entry for B
0	AP,		•
	NUMBER		points to table entry for 5
0	AP,		•
	VARID	2	points to table entry for C
	VARID		points to table entry for D
0	+		•
0	AP,		•
	LABELID		pointer to table entry for X
	GOTO		
0	AP)		•

A separate tree is generated for each program segment, with output pointers relative to that tree, The output for each program segment is of the following form:

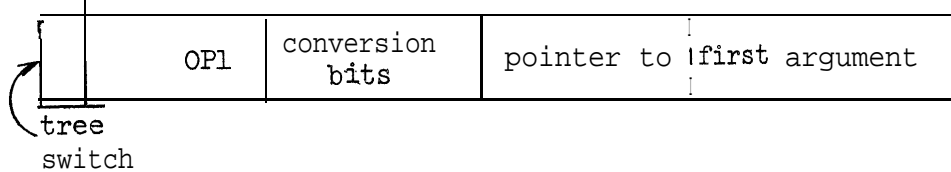
	pointer to end of tree
PROCDC	pointer to NAMETABLE
	⋮ (tree for procedure body)
PCL	pointer to PROCDC

Origin of literal table  
 Length of literal pointer table  
 Literal pointer table  
 Length of literal table  
 Literal table

Figure 6

OUTPUT VOCABULARY

I. Binary Operators



Where OPl can be one of the following binary operators;

<u>OPERATOR</u>	<u>CODE</u>	<u>REMARKS</u>
+	1	
	2	
*	3	
/	4	
**	5	exponentiation
L :=	6	logical assignment
A :=	7	arithmetic assignment
S :=	8	string assignment - conversion field contains string length
R :=	9	reference assignment - no conversion
STEPUNTIL	12	
DIV	13	
REM	14	
<	15	} conversion bits indicate length for string comparison
≤	16	
>	17	
>	18	
=	19	
≠	20	
L := 2	22	multiple assignment
A := 2	23	
s := 2	24	
R := 2	25	

OP2	conversion bits	pointer to first argument
-----	-----------------	---------------------------

(left branch always processed first)

(conversion field may contain-string length for string arguments)

<u>OPERATOR</u>	<u>CODE</u>	<u>REMARKS</u>
AP)	29	Indicates end of actual parameter list. Conversion bits indicate conversion of result of function call.
INDX	30	Indicates subscripting operation. Conversion bits can occur only with last such operator and indicate that resulting array element must be converted.
REFX	31	Indicates computation of field (1st arg.) of record reference (2nd arg.).
IFEXP	32	Indicates that label should be issued for end of if exp. and unconditional jump patched. Conversion bits indicate that resulting expression must be converted,
PCL	39	Indicates end of procedure declaration.
SUBSTRING	40	

	OP3	pointer to first argument
--	-----	---------------------------

tree (no conversion)  
switch

<u>OPERATOR</u>	<u>CODE</u>	<u>REMARKS</u>
SHL	35	left shift
SHR	36	right shift

OP4		pointer to first argument
-----	--	---------------------------

(no conversion: left branch always processed first)

<u>OPERATOR</u>	<u>CODE</u>	<u>REMARKS</u>
BB	37	indicates end of declarations, beginning of blockbody.
END	38	
I	41	
AP,	42	for actual parameters
R,	43	for record designators
AR,	44	for array declarations
AR)	45	indicates end of array declaration
R)	46	indicates end of record designator
<del>LOG</del>	47	indicates <del>OR</del> of logical arguments
<del>BIT</del>	48	indicates <del>OR</del> of bit sequences
<del>LOGAND</del>	49	indicates AND of logical arguments
BITAND	50	indicates AND of bit sequences
ITERST	51	indicates generation of transfer to iteration test (for WHILE st and simple <del>FOR</del> st)
ITERST2	52	indicates generation of transfer to iteration test (for <del>FOR</del> st with <del>FOR</del> list)
<del>FOR</del> LIST	53	
<del>FOR</del> CL	54	links control assignment and STEPUNTIL
END <del>FOR</del> LIST	55	
UJIFEXP	56	indicates unconditional jump in IF exp
UJ	57	indicates issue jump to end of case list or IF st. (to be patched)
CL	58	indicates label should be issued for end of CASE st and jump addresses patched
IFST	59	indicates label should be issued for end of IF statements and jump addresses patched
::	60	array bounds COLON
IS	61	
,	63	indicates <del>NOOP</del> (statement separator)
WHILEOP	64	
WHELEST	65	
IFJ	66	indicates issue jump on condition false to end of IF exp. or IF st.



II. Unary Operators

OP5	conversion bits		
-----	-----------------	--	--

Where OP5 can be one of:

<u>OPERATOR</u>	<u>CODE</u>	<u>REMARKS</u>
UMINUS	67	unary minus
ABS	68	absolute value

OP6			
-----	--	--	--

Where OP6 can be one of:

<u>OPERATOR</u>	<u>CODE</u>	<u>REMARKS</u>
LØG ¬	71	negation of logical value
BIT ¬	72	negation of bit sequence
ØN	73	
ØFF	74	
GØTØ	75	
:	76	label COLON
STACKADDR	77	argument is local stack origin for implicit subroutine (statement parameter)

CARD(79)		source card number
CASE(80)	simple type (if expr.)	number of cases

unary operator for  
BEGIN, PROCDC, ARRAYDC,  
", "

III. Terminal Nodes

BEGIN(83)	block no.	local stack origin
-----------	-----------	--------------------

block no. and local stack origin occur only if begins data segment

INUMBER (85)		integer value
-----------------	--	---------------

NUMBER (86)	conversion bits	pointer to constant table
----------------	-----------------	---------------------------

X1	conversion bits	pointer to NAMETABLE
----	-----------------	----------------------

Where X1 can be:

<u>TERMINAL</u>	<u>CODE</u>	<u>REMARKS</u>
ID	87	
LABELID	88	no conversion
ARRAYID	89	no conversion
FUNCID	90	no conversion if proper procedure
RCCLID	91	no conversion
FIELDID	92	no conversion
CONID	93	
PROCDC	95	no conversion (procedure declaration)
RCCLDC	96	no conversion (record class declaration)

SEG(97)		program segment number
---------	--	------------------------

indicates program segment occurring in outer segment.

X2		pointer to constant table
----	--	---------------------------

Where X2 can be:

<u>TERMINAL</u>	<u>CODE</u>	<u>REMARKS</u>
BIT	98	
STRING	99	
TRUE	100	
FALSE	101	

X3			
----	--	--	--

Where X3 can be:

<u>TERMINAL</u>	<u>CODE</u>	<u>REMARKS</u>
IF	111	
WHILE	102	
NULL	103	indicates undefined reference
NULLST	104	indicates empty statement
ARRAYDC	105	array declaration
AR*	106	indicates dummy array subscript

X4	conversion bits	pointer to NAMETABLE
----	-----------------	----------------------

Where X4 can be:

<u>TERMINAL</u>	<u>CODE</u>	<u>REMARKS</u>
STFUNCID	107	
STPROCID	108	

## D. Pass Three

### 1. Register Allocation

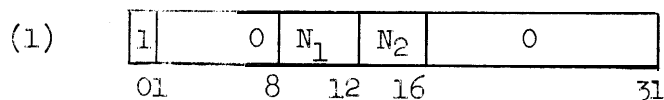
Code generation for arithmetic operations involves the knowledge of which registers are occupied and where each partial result is held. Temporary storage must be provided for dumping partial results from registers into main memory when either too few registers are available or a subroutine call is made. An even-odd pair of general registers is required for integer multiplication and division.

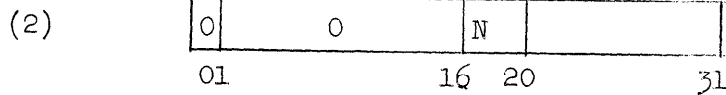
All the floating registers are available for arithmetic. Some of the general registers are reserved for special purposes. The compiler variable CLN always contains the number of the lowest-numbered base register in the current program segment. All lower-numbered general registers are available for arithmetic with the exception of R0 and R1, and R2 in iterative statements.

The compiler uses two half-word arrays R and F to indicate which registers are occupied. To each general register which is free corresponds a flag equal to 0 in the array R. A non-zero flag indicates the register is occupied. The array F serves the same function for the floating registers.

Partial results are located by referring to LSTACK. Each current partial result, whether value or address, has an entry in LSTACK.

These entries have the following formats:





In (1),  $N_2$  is zero except for one case: a complex value is in the floating registers  $N_1$  and  $N_2$ .  $N_1$  is the number of either a general or floating register, and bits 16-31 are interpreted as a base with displacement address.

In general, a procedure call **involves** dumping all partial results, Also, one or more partial results will be moved from registers to main memory when a shortage of registers occurs. Each quantity dumped must have its LSTACK entry changed to indicate the new location. Thus pointers to the LSTACK entries indicating registers are required. These pointers are in two arrays, FSTACK for general registers and RSTACK for floating registers. Each RSTACK entry consists of only the displacement field, for indexing LSTACK. Each FSTACK entry has this index and two other bits of information: bit 0 is on for type real and off for type complex, and bit 1 is on only if the quantity is not long. Complex values are never split between a register and a memory call; either both real and imaginary parts are in registers or both are in memory.

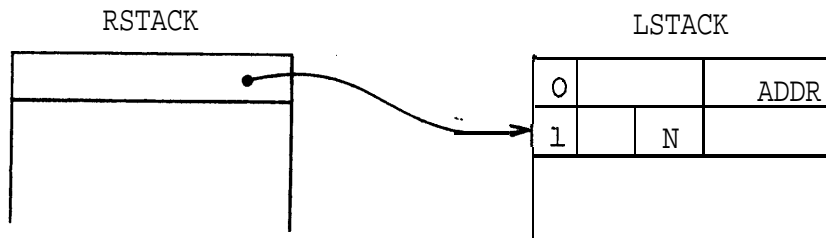
A procedure call requiring the **saving of registers** causes the necessary store instructions to be generated, all corresponding LSTACK entries referenced via RSTACK and FSTACK to be updated, and RSTACK and FSTACK to be emptied. During Pass Three R2 always points to the next available word in RSTACK and R4 similarly for FSTACK. The pro-

cedures DUMPALLGENREG and DUMPALLFLREG carry out these functions.

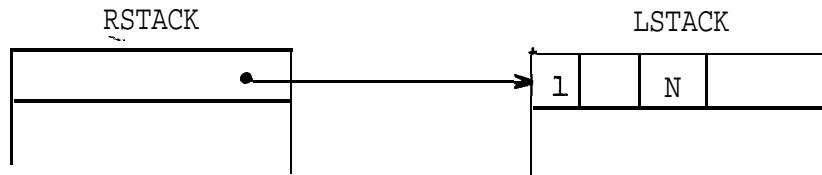
When one or two registers are needed or partial results are not available, one or two registers holding the currently oldest partial results are stored. This involves updating at most two LSTACK entries. The relevant RSTACK or FSTACK element(s) are eliminated, and all elements above are moved down. The currently oldest partial results in registers are thus always referenced via the bottom entries of RSTACK and FSTACK. The procedures DUMPGENREG, DUMPFLREG, and DUMPPRFLREG generate the store instruction(s) and do the necessary updating,

When a register or pair of registers is needed, the appropriate register request routine is called and is one of the following: GENREG, PRGENREG, FLREG, or PRFLREG. This routine scans the R or F array to find, if possible, the required single register or pair. If necessary, it will call the appropriate save procedure as described above. Having determined or created the requested register(s), the procedure will flag the appropriate element(s) of R or F, set up the LSTACK entry at the top of the stack, and create the appropriate RSTACK or FSTACK entry. A register release is performed by either RELEASE or ZRELEASE.

In certain cases of inputs to binary operations, an adjustment must be made in the top pointer value of either RSTACK or FSTACK. Consider the situation below just before code is to be generated for an add operation,,



It is only necessary to generate one ADD instruction to add the contents of memory location ADDR to register N. Afterwards, the situation must be the following.



The pointer at the top of RSTACK must be decremented to point to the new top of LSTACK, Whenever this is necessary, procedure ADJSTACKS is called.

Procedure ASSEMBLE, though used in many parts of Pass Three, was designed primarily with arithmetic instruction generation in mind, It accepts as inputs registers holding two LSTACK-format entries, one of them also holding the second half-byte of the instruction code in bits 4-7. The third input contains the type, From these the routine can determine the first half-byte of the instruction code and build each field of the instruction.

## 2. Block Entry

There are four purposes of block-entry code: First, the data stack pointer, a system cell called MP, must be updated. At any given

time, MP contains the base address of the most recently created data segment.

Secondly, space must be allocated in the data stack for the data segment to be created.

Thirdly, the block mark must be built and placed at the base of the data segment.

Finally, the local display must be set to reflect the accessibility of all variables which can be referenced within the block.

The total amount of storage to be allocated for the data segment is not known when Pass Three encounters a block. Pass Two calculates the static amount of storage required for the block mark, local display, and local variables and array descriptors. This information is given to Pass Three. However, during compilation of the block body., registers with partial **results** may need to be dumped due to procedure calls, etc., and the amount of storage required for this purpose, called the local stack, is not known until the block is compiled. Hence at the end of compilation of the block the instruction which specifies the total amount of data storage required for the data segment is fixed up, and at execution time the total amount of data storage needed is correctly given.

Since the display registers are allocated statically downwards from R13, the base register to be used for the data in the block being entered is numbered one less than for the enclosing block. The display for the block is then identical with the display for the enclosing block with the addition of the display entry for this block.

The code for block entry is given below: n is the number of the register which will be the base of the data segment for this block.



LR	2,n+1	R2 = base of data segment of enclosing block
L	6,FP(2)	R6 = free pointer in enclosing data segment
A	6,=7	= base of new data segment
N	6,X'FFFFFFF8'	set data segment on a double word boundary
LA	0,length(,6)	length is the total amount of static storage needed for this data segment - fixed up at block exit, RO = new FP
BAL	4,ALLOCERR	see discussion of error code (Sec. IV.A.6)
LA	3,X	see discussion below
LA	4,Y	see discussion below
STM	0,4,0(6)	RO = FP R1 = not used in block mark R2 = dynamic link R3 = REFVAR R4 = REFARY
ST	6,MP	update stack pointer
LR	n,6	R6 = Rn = base of this data segment
STM	n,12, 20 (,6)	store local display (if n=12, then ST 12, 20(,6))

In the instructions

LA	3,X
LA	4,Y

X is the relative address of the first reference variable declared in the block, and Y is the relative address of the base of the first reference array descriptor declared in the block.

After all code producing declarations (e.g. array declarations) have been processed, MVI instructions are used to insert the number of reference variables and number of reference arrays in their appropriate

fields in the block mark.

Note that if there are no reference variables declared in the block, the instruction

IA 3,X is replaced by SR 3,3

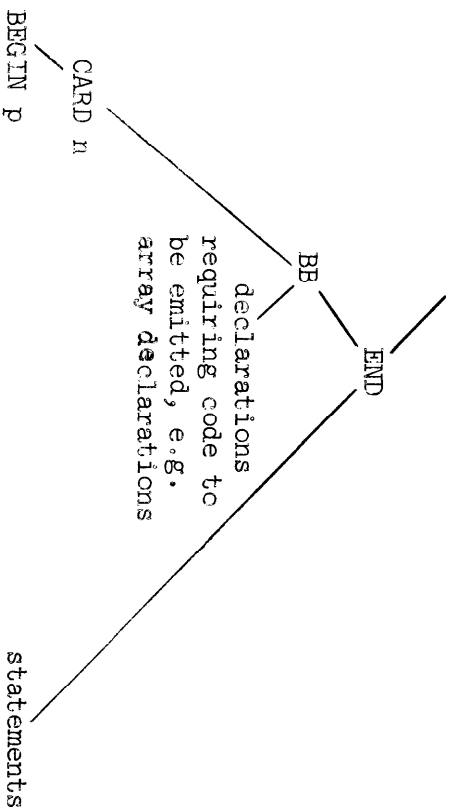
and no MWI REFVAR+1, N<sub>1</sub> is compiled.

Likewise, if there are no reference arrays declared in the block, the instruction

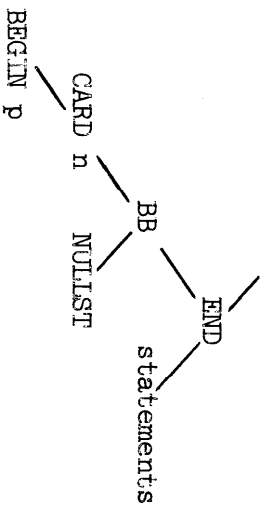
IA 3,Y is replaced by SR 4,4

and no MWI REFARRY+1, N<sub>2</sub> is compiled.

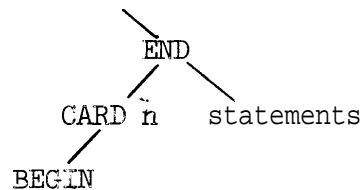
The tree output of Pass Two for a block with declarations is



The tree node BB is present even if there are no declarations requiring code to be emitted, in which case the tree is as follows:



Blocks without declarations have the following tree:



The pointer field  $p$  in the node BEGIN is the amount of data storage required for the block, with the exclusion of the local stack, except for the outermost block of a procedure whose data segment is merged with the procedure data segment. In this case, the  $p$ -field in the node BEGIN is 0 and the amount of storage required for the combined procedure-block data segment is given in the NAMETABLE entry for the procedure,

The second byte in the node BEGIN is a pointer (by 1's) to the BLOCKLIST table. Hence, the NAMETABLE entries for the variables and arrays declared in the block can be scanned, and the count and starting addresses of the reference variables and array-s can be obtained for the inclusion in the block mark.

The node CARD  $n$  is explained in a following section (cf. IV.D.23).

### 3. Block Exit

The purpose of the code emitted for block exit is to reset MP to the base of the data segment for the block to which control is being returned.

The tree output of Pass Two for block exit is the same part of the tree used for block entry. It is encountered again after all statements in the block have been processed. Compound statement exit and

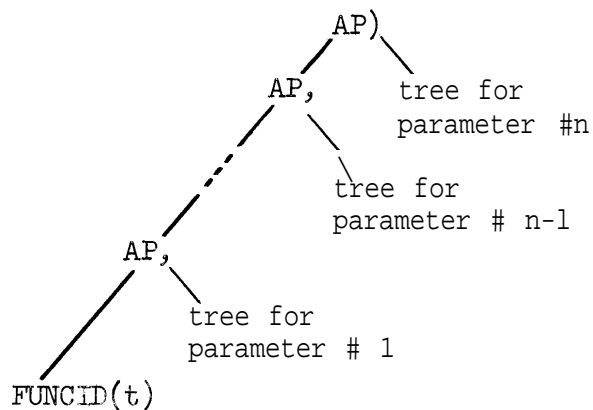
block exit are distinguishable, as 'before, by the presence or absence of the tree node BB.

Code emitted for a block exit is as follows: n is the number of the register which holds the base of the data segment corresponding to the block being exited.

L	l,DL(,n)	RI = dynamic link (field mark block) = base of data segment of block re- turning to
ST	l,MP	Reset data pointer stack

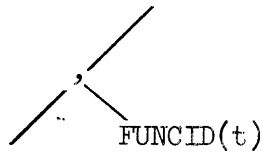
#### 4. Procedure Statements and Typed Procedure Designators

The tree output for procedure statement and function designator parameters (n > 0) is as follows:



The pointer field t of FUNCID is a pointer to the NAMETABLE.

The tree for a proper procedure without parameters is:



The tree for a typed procedure without parameters looks just like an identifier except that the terminal node is `FUNCID(t)` instead of `ID(t)`.

The code generated for a proper or typed procedure call, with or without parameters, is as follows where `m` is the number of the register which holds the base of the data segment corresponding to the block in which the called procedure was declared:

```

LR    5,m          R5 = base of data segment from which
                    display will be updated in pro-
                    cedure entry (after parameters
                    are established)
  
```

```

L     15, base of procedure
  
```

```

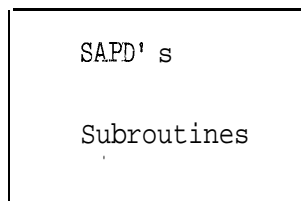
BALR  1, 15
  
```

```

L     15, base of current
      procedure
  
```

```

B     SETDIS
  
```



(cf. IV.D.5)

```

SETDIS LM    n, 12, 20(2)
  
```

```

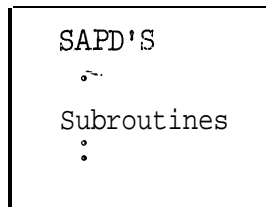
Reset the display --
R2 = dynamic link loaded at procedure
    exit
    = base of current data segment
  
```

`n` is the number of the general register holding the base of the data segment for the current block. If `n=13`, the `LM` instruction is omitted.

### Call of a Formal Procedure

The following code is emitted for the call of a formal procedure:

```
LM      4,5,DPD          R4 = address of subroutine (cf. IV.D.5)
LA      0, number of actual
        parameters
L       15, CHECK
BALR   1, 15
L       15, base of current
        procedure
B       SETDIS
```



```
SETDIS LM      n, 12, 20(2)
```

The CHECK routine checks actual-formal correspondence, since this checking cannot be done at compile-time, Actual parameter descriptors are obtainable via R1 (the 2nd-4th byte of each SAPD). Formal parameter descriptors are in the head of the called procedure (SFPD'S). R4 contains the address of the subroutine which will call the procedure; therefore there is an instruction in the subroutine of the form

```
L      4, base of called procedure .
```

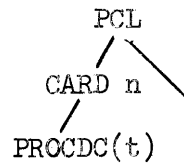
The CHECK routine locates this instruction (via R4), executes it and then checks actual-formal correspondence.

The CHECK routine saves R4 and R5, and ends with

```
BCR   15, 4 .
```

## 5. Procedure Entry

The tree produced by Pass Two for procedure entry is:



The purposes of procedure entry code are almost those of block entry code, and for this reason the codes will be quite similar.

The additional requirements of procedure entry are those of setting up dynamic formal parameter descriptors, evaluating value parameters, and the more complicated manner of setting up the display.

At procedure call (cf. IV.D.4), R5 holds the base of the data segment surrounding the declaration of the called procedure. This data environment is precisely that which should be valid while the procedure is being executed. \*Therefore the display of this surrounding block plus the display entry for the called procedure constitute the display while executing the procedure.

Procedure entry code is as follows: Rn will hold the base of the data segment to be created.

L	2,MP	base of calling data segment
L	6,FP( ,2)	R6 = base of new data segment
LA	0,length( ,6)	add in required storage. R0 = new FP.
BAL	4,ALLOCERR	check to see that allocation is valid
LA	3,X	(Note 1) (cf.IV.A.6)
LA	4,X	
STM	0,4,0(6)	store procedure mark
ST	6,MP	update stack pointer

SAPD → DPD operations

LM	n+1,12, 20 (5)	(Note 2) update the display
LR	n,6	
STM	n,12, 20 (6)	(Note 3)

DPD → PV operations

Note 1: X is the relative address of the first reference value/result parameter; or if there are no value/result parameters, X is the relative address of the first reference variable local to the block whose data segment is merged with this procedure's data segment; or if there are no reference value/result parameters and no local reference variables or no block, then X is 0.

Y is the relative address of the first reference array descriptor in the block whose data segment is merged with the procedure's data segment. If there are no reference arrays or no block, then Y is 0.

MVI instructions are used to place the number of reference value/result parameters and local reference variables, and the



number of local reference arrays:, into the fields REFVAR and REFARY,  
respectively, in the procedure mark.

Note 2: This instruction is omitted if  $n = 12$ .

If  $n = 11$ , the instruction becomes `L 12, 20 (,5)`

Note 3: If  $n = 12$ , then this instruction becomes `ST 12, 20 (,6)`

Notice that  $6 < n < 12$ .

### SAPD's - Static Actual Parameter Descriptors and Subroutines

The calls of procedures without parameters have no SAPD's or subroutines corresponding to them, and the reloading of R15 to the base of the current program segment is immediately followed by the resetting of the display at procedure call (cf. IV.D.4).

For procedures with parameters, each parameter has associated with it one SAPD of 8 bytes. According to different forms of actual parameters, different SAPD's are established. In general, an actual parameter is represented by a subroutine, and the SAPD gives the address of that subroutine. If the parameter is an identifier, the SAPD contains the address of the identifier. Note that addresses of subroutines are given relative to the instruction

`L 15, base of current program segment`

immediately following the instruction `BALR 1,15` in procedure call.

The Pq bits in the SAPD define the character of the actual parameter. P specifies whether a subroutine exists or not:

P=1 : accessto parameter involves a subroutine call

P=0 : no subroutine call

Q specifies whether the parameter may occur in the left part of an assignment statement:

Q=P : assignment is possible

Q≠P : assignment not possible

The type information field of three bytes is used only by the CHECK routine when a formal procedure is called.

ACTUAL PARAMETER IS

SAPD IS

DPD IS

<u>ACTUAL PARAMETER IS</u>		<u>SAPD IS</u>		<u>DPD IS</u>								
identifier	I	<div style="text-align: center;">           PQ            ↓ ↓         </div> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td> <td>type</td> </tr> <tr> <td>LA</td> <td>3, id(n)</td> </tr> </table>	00	type	LA	3, id(n)		<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td> <td>address of id</td> </tr> <tr> <td>ST</td> <td>data base</td> </tr> </table>	00	address of id	ST	data base
00	type											
LA	3, id(n)											
00	address of id											
ST	data base											
constant, expression or statement	II	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>type</td> </tr> <tr> <td>LA</td> <td>3, subr(1)</td> </tr> </table>	10	type	LA	3, subr(1)		<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>address of subr.</td> </tr> <tr> <td>ST</td> <td>data base</td> </tr> </table>	10	address of subr.	ST	data base
10	type											
LA	3, subr(1)											
10	address of subr.											
ST	data base											
procedure	III	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>type</td> </tr> <tr> <td>LA</td> <td>3, subr(1)</td> </tr> </table>	10	type	LA	3, subr(1)		<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>address of subr</td> </tr> <tr> <td>ST</td> <td>data base</td> </tr> </table>	10	address of subr	ST	data base
10	type											
LA	3, subr(1)											
10	address of subr											
ST	data base											
subscripted variable or field designator	IV	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>11</td> <td>type</td> </tr> <tr> <td>LA</td> <td>3, subr(1)</td> </tr> </table>	11	type	LA	3, subr(1)		<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>11</td> <td>address of subr.</td> </tr> <tr> <td>ST</td> <td>data base</td> </tr> </table>	11	address of subr.	ST	data base
11	type											
LA	3, subr(1)											
11	address of subr.											
ST	data base											
formal parameter	V	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td> <td>type</td> </tr> <tr> <td>LM</td> <td>3,4,DPD(n)</td> </tr> </table>	00	type	LM	3,4,DPD(n)		<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td colspan="2" style="text-align: center;">Copy of DPD</td> </tr> </table>	Copy of DPD			
00	type											
LM	3,4,DPD(n)											
Copy of DPD												

The implicit subroutines corresponding to parameter types II (expressions and statements) and IV create data segments of hierarchy level one less than at the point of procedure call. The format of these data segments is like those created by blocks except that for implicit subroutines, there are no local variables.

Implicit subroutines corresponding to constants are as follows:

L	15, base of segment in which constant table lies	
L	2, MP	set R2 for return
LA	3, address of constant (15)	
BCR	15, 1	this subroutine branched to via R1

Implicit subroutines corresponding to proper procedures and all typed procedures are as follows:

L	4, base of called procedure	
LR	15, 4	
L	5, =F'(X-CLN+1)*4' (5)	where X = hierarchy # of called procedure CLN = current hierarchy number
BCR	15, 15	

The purpose of this subroutine is to set R5 correctly. Recall that R5 will be used as the base to update the display in the entry code of the called procedure. R5 cannot be set correctly at the point of mention of the formal name parameter corresponding to the procedure for which this subroutine is set up in certain recursive procedure call situations.

Notice that the subroutines given above do not set up a data segment of their own.

All string routines (i.e. string procedures and implicit subroutines returning the results of string procedures) are exited with the address of the resulting string in R3. For some string routines

the string itself may 'be in the data segment of the string routine. When the routine is exited, the data segment is released, and the resulting string may thus be destroyed if another data segment is allocated before the string (whose address is in R3) is used.

This situation arises for typed procedures of types other than string, but the manner of compiling expressions of these types insures that the result of the typed procedure will be used (i.e. either placed in a register, added to an accumulating sum, compared, etc.) before any new data segment could be created.

This is not the case for strings.

Hence, to insure that the string which is the result of a string routine is not lost, the string must be moved to a data segment which cannot possibly be 'released, until the string is used. In the case under discussion, the string must be moved into the local stack of the data segment at the point of call of the string routine.

In the description of the DPD's (to be discussed presently), the address and data base fields are absolute core addresses. The data base field is the base of the data segment of the block in which the procedure call occurs. This field is used as the base from which to update the display when executing implicit subroutines or procedures corresponding to the mention of the corresponding formal parameters.

The byte ST is the simple type of the actual parameter (0 for proper procedures and statements) and is used for type conversion for value/result parameters. Recall that all name parameters must match exactly in type.

Implicit subroutines which 'have values are so constructed. that the

address of the result is returned.

SAPD → DPD Operations

SAPD : Static Actual Parameter Descriptor

DPD : Dynamic Parameter Descriptor

The SAPD → DPD operation consists of an evaluation of the static addresses given in each SAPD at procedure call, and the transmission of the type information about the actual parameter including the two-bit code (PQ).

If the actual parameter is a formal parameter? the DPD must be copied. Each DPD is eight bytes wide and there is a 1-1 correspondence between SAPD and DPD. The possible formats for the DPD's are given in the section discussing the SAPD's.

The code for producing the DPD's is as follows:

Let a = address of DPD to be created (using R6 as base - see, procedure entry code)

b = address of SAPD (using R1 - see procedure call code)

LR	4,2	dynamic link = data base for DPD
EX	0,b+4	executes instruction in SAPD. For all types except V, this loads R3 with address of procedure or implicit subroutine. for type V, (actual parameter is formal parameter), this loads DPD of formal parameter into R3 and R4.
STM	3,4,a	store DPD
OC	a(1),b	establish PQ bits
MVC	a+4(1),b+3	establish ST field

## DPD → PV Operations

As stated in the report, each value parameter is evaluated and its value is stored in the procedure's data segment. Any further occurrence of the parameter uses the parameter value (PV).

Since, by definition, arrays are always passed by name, the DPD is used to obtain the address of the actual descriptor, which is then copied into the data segment of the procedure. The DPD may or may not require a subroutine call to obtain the address of the descriptor, depending on whether or not a sub-array is being passed. Any further occurrence of the array parameter uses the copied descriptor, the parameter value (PV), to compute the addresses of the array elements.

## 6. Procedure Exit

Because of the tree scanning mechanism in Pass Three of the compiler, typed procedures with parameters and typed procedures without parameters are detected as requiring a procedure call at different places in Pass Three. For this reason, the mode of returning the result is different,

For typed procedures with parameters, the result of the procedure is returned in a register, depending on the type, as follows:

integer	R3
real	F0
Long real	F0F
complex	F0-F2
long complex	F01-F23
bits	R3
reference	R3
logical	R3 (address of result)
string	R3 (address of result)

For typed procedures without parameters (which include implicit subroutines which return values), the address of the result is returned in R3.

The addresses of the actual parameters corresponding to result parameters are evaluated and a validity check is made to be sure that the actual parameter can be stored into. The type of the result is converted if necessary and the result is stored.

The code emitted for procedure exit is as follows:

```
LM      1,2,RA(n)      R1 = return address
                        R2 = dynamic link
ST      2,MP
BCR     15,1
```

Notice that upon return, the display is updated from R2, set correctly here in procedure exit.

## 7. Formal Parameters in Expressions and Assignments

Reference to a formal n&e parameter requires testing whether a subroutine call is necessary, or whether the descriptor (DPD) already contains the absolute address of a variable. Furthermore, a validity test is performed if an assignment is to be made to the formal parameter.

The code emitted for a formal parameter in an expression is:

	TM	DPD(n),X'02'	test P-bit
Y	BC	1,X	branch if P=1, i.e. must call subr.
	L	3,DPD(n)	"no subroutine, R3 = address of id
	BC	15,Z	
x	L	5,DPD+4(n)	R5 = data base = base to update display inside subroutine or procedure
	L	15,DPD	R15 = base of subr. or procedure
	BCR	15,15	
	L	15, base of current program segment	
	LM	n,12, 20 (2)	reset display
Z	:	:	
	:	:	

At Z, R3 has the address of the formal parameter, and its value is easily obtained.

Value parameters are referred to only once as shown above, in the DPD → PV operations. If the type of the value parameter is arithmetic, a call to a system routine which converts the actual parameter if necessary and stores the result in the formal value location is placed at the label Z. If the type is non-arithmetic no conversion is possible and an instruction to store the value is placed at Z. If the type is string, instructions to insure that non-significant characters of the formal parameter are set to blank are inserted before the store instruction.

For a formal name parameter occurring on the left of an assignment statement, the code is as before except for the first instruction, which is replaced by:



TM	DPD(n),X'03'	test P and Q bits
BC	B,Y	branch if PQ bits not mixed, i.e. can store into
BAL	1,MAINERR	branch to error routine, R1 = location of error

Result parameters are referred to only once in this manner in procedure exit.

### 8. Array Declaration

Corresponding to the array declaration of n dimensions

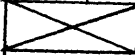

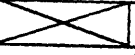
<simple type> array X ( $l_0 :: \mu_0, l_1 :: \mu_1 \dots l_{n-1} :: \mu_{n-1}$ )

in the head of a block, an array descriptor of length  $12n+8$  bytes is built in the data segment of the block.

SIMPLE TYPE	NUMBER OF BYTES PER ARRAY ELEMENT
1. integer	4
2. real	4
3. long real	8
4. complex	8
5. long complex	16
6. logical	1
7. string	declared string length
8. bits	4
9. reference	4

The size of the descriptor depends only upon the number of dimensions of the array and hence the storage for the descriptor is allocated statically. The storage for the array elements themselves must, of course, be allocated dynamically. The descriptor has the

following format:

r	$\alpha_0$
	$\Delta_0$
	$l_0$
	$\mu_0$
	$\Delta_1$
	$l_1$
	$\mu_1$
	$\Delta_2$
	$\vdots$
	$l$
	$\mu_{n-1}$
	$A_n$

where  $\alpha_0$  - is the base address of the array elements  
 $\Delta_0$  - is as given in the table above and is the number of bytes per array element  
 $l_i$  - the lower bound of the  $i^{\text{th}}$  dimension  
 $\mu_i$  - the upper bound of the  $i^{\text{th}}$  dimension  
 $\Delta_i = (\mu_{i-1} - l_{i-1} + 1) \times \Delta_{i-1} \quad i = 1, 2, \dots, n$

We require that  $\Delta_i, i=0, 1, \dots, n-1$  fit into 15 bits so that the more convenient multiply halfword (MH) instruction may be used for the multiplication. Note that no such restriction is required for  $\Delta_n$ , which represents the total number of bytes required for the array.

The value of  $\Delta_i, i=1, 2, \dots, n$  is the number of bytes required for the first  $i$  dimensions of the array. The restriction that  $A_{j, j=0, \dots, n-1}$  fit into 15 bits results in the restriction that  $A_{n-1}^{j'}$  fit into 15 bits, for if any  $\Delta_j, j=0, \dots, n-2$  does not fit into 15 bits, then  $A_{n-1}$  will not fit into 15 bits. Therefore, the value of  $A_{n-1}$

must be less than or equal to  $32767_{10}$ . Observe that for a 1-dimensional array, this restriction is automatically satisfied,

The following table gives the maximum number of elements for the first n-1 dimensions of an array of the indicated simple type,

<u>simple type of array</u>	<u>maximum number of elements in first n-1 dimensions</u>
logical	32767
integer, real, bits, reference	8191
long real, complex	4095
long complex	2047
string	32767 DIV q

where q is the  
declared string  
length

For storage of the array itself upon block entry,  $A_n$  bytes are requested and the free pointer (FP) of the data segment in which the descriptor resides becomes the base of the array, after which FP is incremented by  $\Delta_n$ .

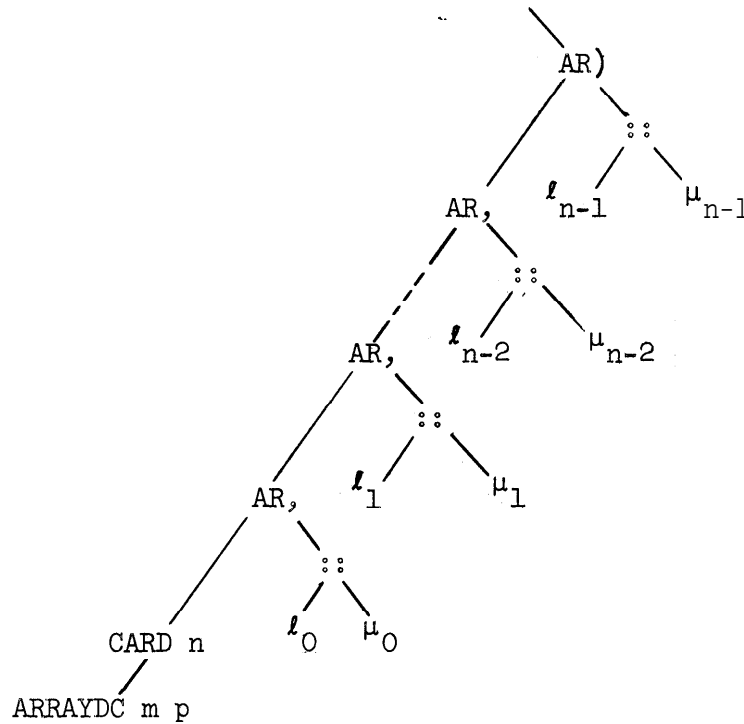
In Algol notation:

$$\alpha_0 := \text{FP}$$

$$\text{FP} := \text{FP} + \Delta_n$$

In the case of reference arrays, the upper byte of the first word of the descriptor, the r-field, gives the number of dimensions so that the garbage collector can find the next reference array descriptor.

The tree format for the array declaration <simple type> array X1, X2, ..., X<sub>m</sub> ( $l_0 :: \mu_0, l_1 :: \mu_1, \dots, l_{n-1} :: \mu_{n-1}$ ) is as follows:



The pointer field  $p$  in ARRAYDC is a pointer to the NAMETABLE entry for X1;  $m$  is the number of identifiers. The nodes  $l_i$  and  $\mu_i$  can be subtrees for any integer arithmetic expression.

All left subtrees are processed first. The descriptor is built into the descriptor location of the last identifier, in this case X<sub>m</sub>, and finally at AR) the completely built descriptor is copied into the descriptor locations for the other arrays. As each descriptor is copied, storage for that array is allocated and the base address is placed in the  $\alpha_0$  field of the descriptor,

Example:            integer a y X,Y(0::10,A::A+B)

L	2,=F'0'	lower bound of first dimension	}	First dimension
ST	2, $l_0$			
LA	2,4	number of bytes per array element		
ST	2, $\Delta_0$			
L	2,=F'10'	upper bound of first dimension		
ST	2, $\mu_0$			
S	2, $l_0$			
BAL	1,UBLBERR	see error code discussion in section IV.A.6		
LA	2,1(2)	$(\mu_0 - l_0 + 1)$		
SLL	2,2	$(\mu_0 - l_0 + 1) \times \Delta_0$		
ST	2, $\Delta_1$		}	Second dimension
SLA	2,16	check if $\Delta_1$ can fit into a halfword		
L	2,A	lower bound of second dimension		
ST	2, $l_1$ --			
L	2,A			
A	2,B	upper bound of second dimension		
ST	2, $\mu_1$			
S	2, $l_1$			
BAL	1,UBLBERR			
MH	2, ( $\Delta_1 + 2$ )			
ST	2, $\Delta_2$			
L	0,FP	free pointer	}	set base of array to word boundary *
A	0,THREE	see discussion of special constants based off R14 (cf. IV.A.6)		
N	0,SINGLMASK	see discussion of special constants		
ST	0, $\alpha_0$	store base Y in descriptor Y		
A	0, $\Delta_2$	RO = new FP = 'base of next array		
BAL	4,ALLCERR	see error code discussion		
MVC	X(29),Y	move descriptor (30 bytes) from Y to X		
ST	0, $\alpha_0$	store base X in descriptor X		
A	0, $\Delta_0$	RO = new FP		
BAL	4,ALLCERR			
ST	0,FP	store new free pointer		

\* For arrays of type logical and string, the free pointer is not adjusted. For arrays of type long real and long complex, the free pointer is adjusted to a double word boundary. For all other types, the free pointer is adjusted to a word boundary.

At each node ":", the lower bound is placed in the descriptor when the left sub-tree has been processed. After the right sub-tree has been processed, the upper bound is placed in the descriptor,

$$\Delta_{i+1} = (\mu_i - l_i + 1) \times \Delta_i \quad i=0, \dots, n-2$$

is calculated, and  $\Delta_{i+1}$  is placed into the descriptor. For  $i=0, \dots, n-3$ , a test is performed to assure that  $\Delta_{i+1}$  will fit into a half-word. For  $i=0$ , the multiplication by  $\Delta_0$  is performed by a shift for all types except <string>, since  $\Delta_0$  will be a power of two for these types. Arrays are stored by columns. At the completion of the execution of this code, the descriptors in the stack would look like the following, assuming  $A=3, B=4$  (all numbers in base 10).

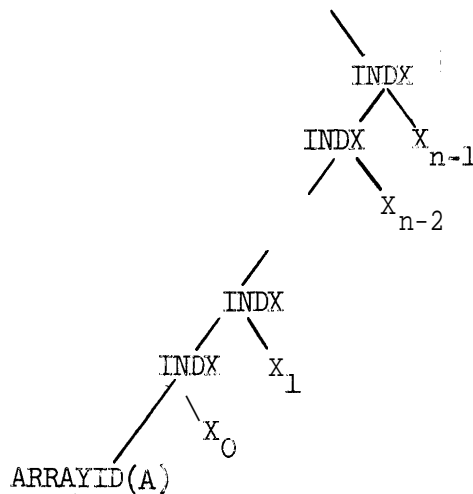


## 9. Subscripted Variables

Consider the following reference to a subscripted variable from an array A of n dimensions:

$$A (X_0, X_1, X_2, \dots, X_{n-1})$$

where  $X_i$  may be any integer arithmetic expression. In tree form, the above construction is represented as:



The address  $\alpha$  of the array element is given by

$$\alpha = \alpha_0 + \sum_{i=0}^{n-1} (X_i - l_i) \times \Delta_i$$

where the left sub-trees are always processed first. The pointer field of the node ARRAYID is a pointer to the NAMETABLE.

Each node  $X_i$  may be a subtree for an arithmetic expression. The indices are evaluated in order from  $X_0$  to  $X_{n-1}$ .

After the value of  $X_i$  has been computed, it is checked against  $l_i$  and  $\mu_i$  (the upper and lower bounds for the  $i^{\text{th}}$  dimension). If either bounds test fails, the run is terminated with an appropriate error message. If the bounds tests are successful, the lower bound is subtracted from the subscript and this quantity is multiplied by the current  $\Delta_i$  and added into the accumulating address.

As an example, consider a reference  $Y(3, T-27)$  to an array declared integer array  $Y(0::10, A::A+B)$ , where  $T=32$ ,  $A=3$ ,  $B=4$ .

The address of the array element is given by

$$\alpha = \alpha_0 + (3-0) \times 4 + (5-3) \times 44 = \alpha_0 + 100$$

where  $\alpha_0$  is the base of array and is obtainable from the first word of the descriptor. (See descriptor given in section on array declarations.)

The following code is generated for this array reference:

```

L    3,  $\alpha_0$           R2 will be accumulating address register
L    3, =F' 3'          first subscript
C    3,  $\mu_0$ 
LA   0, 0(3)           sets R0 to type of error if bounds check
                        fails (see discussion of error checking
                        code [section IV.A.6])
BAL  1, ARRAYERR      (cf. IV.A.6)
S    3,  $l_0$ 
BC   <, MAINERR      (cf. I-V.A.6)
SLL  3, 2              $(X_0 - l_0) \times \Delta_0$ 
AR   2, 3             add into accumulating register
L    3, T
S    3, =F' 27'       second subscript
C    3,  $\mu_1$ 
BAL  1, ARRAYERR
S    3,  $l_1$ 
BC   <, MAINERR
MH   3,  $(\Delta_1 + 2)$ 
AR   2, 3

```

At this point, R2 has the address of  $Y(3, T-27)$ .



## 10. Passing Sub-Arrays as Parameters

The user may pass any generalized row or column, i.e. any sub-array of dimension  $1,2,\dots,n-1$  of an  $n$ -dimensional array as a parameter to a procedure. Since all array parameters are passed by name, all that is needed is to copy certain parts or all of the array descriptor.

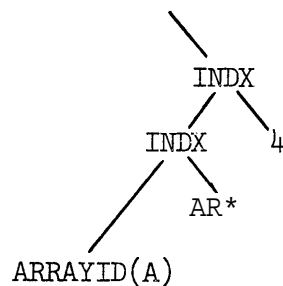
At this point, the reader should familiarize himself with the details concerning the building and format of array descriptors, and the calculation of the address of an array element when the element is referenced,

According to the syntax, **an asterisk (\*)** is placed in those positions of the actual sub-array parameter to indicate which dimensions are to be included in the formal array.

In those positions in which \* occurs in the source code, the Pass Two tree output is the node **AR\***. For example, the tree corresponding to the actual parameter

$A(*,4)$

is



indicating that the first dimension of the two-dimensional array  $A$  is to be unspecified and that the fourth column corresponds to the one-dimensional formal array.

It should be recalled that an array descriptor consists of a series of triples  $\{\Delta_i, l_i, \mu_i\}$ , where  $l_i$  and  $\mu_i$  are the lower and upper bounds of the  $i^{\text{th}}$  dimension,  $\Delta_i = (\mu_{i-1} - l_{i-1}) \times \Delta_{i-1}$  (except for  $\Delta_0$ ), and that the first entry in the descriptor is  $\alpha_0$ , the absolute address of the first array element. Therefore, to compose the sub-array descriptor, rules must be given on how to build the triples  $\{\Delta_i, l_i, \mu_i\}$  and how to calculate  $\alpha_0$ . These rules are as follows:

If  $X_i$  is the  $i^{\text{th}}$  index, then for each position with

$X_i = *$  : copy the descriptor triple  $\{\Delta_i, l_i, \mu_i\}$

$X_i \neq *$  : omit the descriptor triple

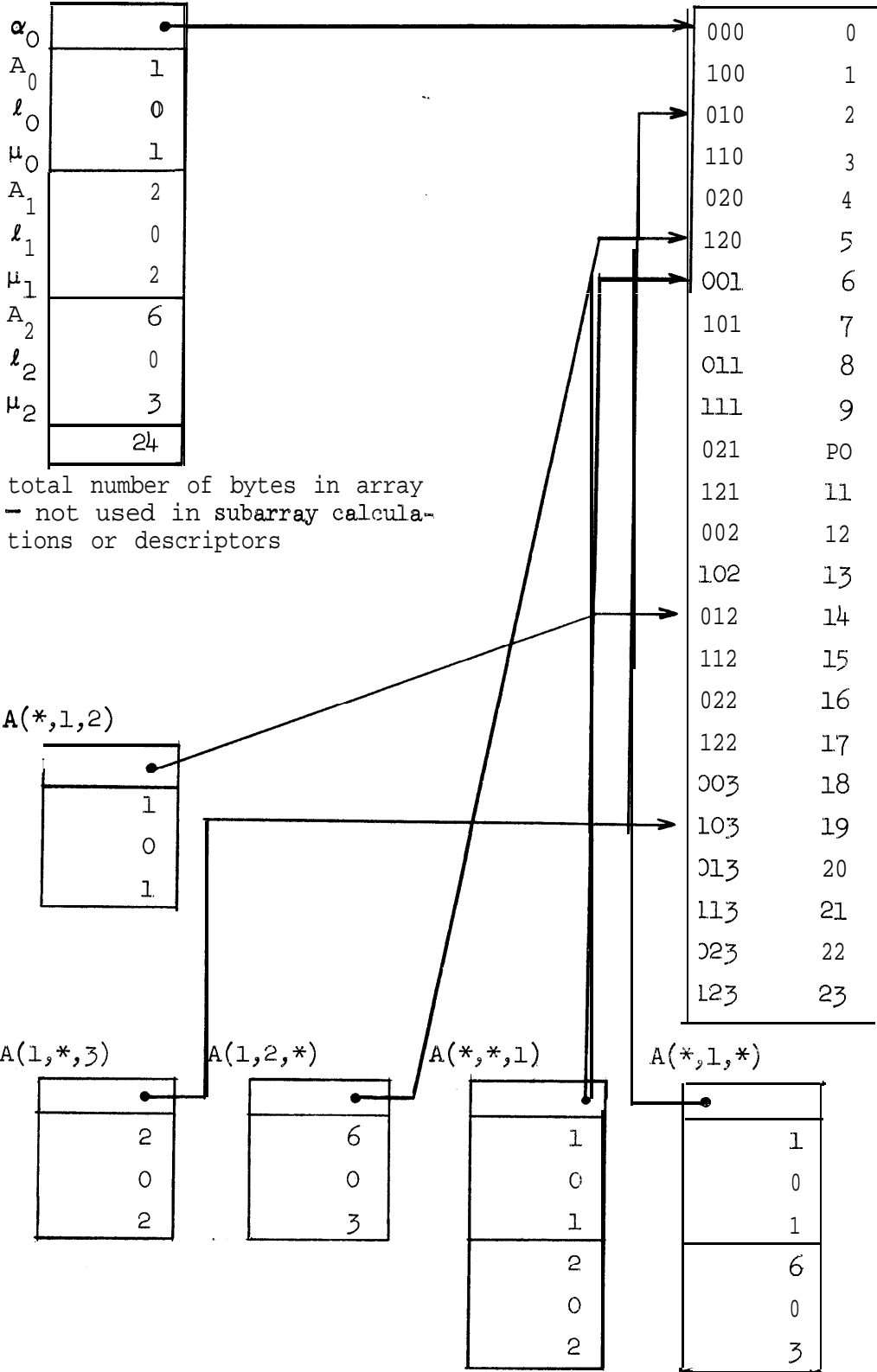
To calculate  $\alpha_f$ , the absolute address of the first formal array element:

$$\alpha_f = \alpha_0 + \sum_{i=1}^{n-1} (Z_i - l_i) \times \Delta_i,$$

$$\text{where } Z_i = \begin{cases} l_i & \text{if } X_i = * \\ X_i & \text{if } X_i \neq * \end{cases}$$

As an example of the use of these rules, consider the following array declaration and the layout of the array elements in core:

logical array A(0::1,0::2,0::3)



The calculation of the addresses of sub-array elements is the same as for ordinary array elements.

The implicit subroutine corresponding to an actual sub-array parameter builds the sub-array descriptor in the local stack of its data segment and returns the address of this descriptor. During the DPD → PV operations, this descriptor is copied into the procedure's data segment,

#### 110 Arithmetic Conversion

Type conversion in ALGOL W is implicit in a number of cases. However, real to integer, or complex or long complex to real or integer must be specified by transfer functions,

##### I. Integer to real or long real

A quantity of type integer is converted to long real by means of a subroutine. The linkage code is:

LA	1,X'rii'
L	15, base of segment 57
BALR	0,15
L	15, current segment base

The quantity placed in register 1 is a parameter to the conversion routine. i specifies the register which contains the quantity to be converted and r specifies the destination floating point register, Therefore, the same conversion routine is called for integer to real conversion as for integer to long real conversion. Likewise, the same routine is used to obtain the real part in conversion from integer to complex and long complex, The imaginary part is attained by the in-

struction

SDR             $r_2, r_2$

The routine to do the conversion stores the absolute value of register  $i$  in the lower half of a double word whose upper half is #4E000000. This quantity is loaded into register  $r$  to which zero is added to normalize the number. Register  $r$  is negated if register  $i$  contained a negative number. The execute instruction is used to manipulate register  $i$  and register  $r$ .

II. Real to long real, complex or long complex

A quantity of type real is converted to long real by two methods.

a) If the value  $V$  is not in a floating-point register, the sequence of instructions used to load  $V$  into register  $r$  is

SDR             $r_1, r_1$   
LE              $r_1, V$

b) If the value is in register  $r$ , the sequence of instructions used to convert  $V$  is

STE             $r, TEMP$   
SDR             $r, r$   
LE              $r, TEMP$

A quantity of type real is converted to complex by subtracting the second of the pair of floating-point registers from itself.. If the conversion is to long complex, the real value is first converted to long real. and then the subtract register instruction is emitted.

### III. Conversion from long real

No instructions are used to convert to real. A conversion to either complex or long complex is done by subtracting the register representing the ~~imaginary~~ part from itself.

### IV. Conversion from complex

A complex value is converted to long complex by applying the rules for converting from real to long real to both the real and imaginary parts of the complex value.

### V. Conversion from long complex

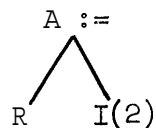
No instructions are emitted to convert long complex values to complex values.

The indication for conversion is made in Pass Two by placing the destination type in the conversion bits (8-15) of the node to which the conversion is applied. (cf. IV.C.5) If the node is a terminal node, (i.e. variable, constant), the conversion takes place before the value is used. If the node is a non-terminal node, the Conversion takes place after the operation the node specifies is completed,

#### Example

INTEGER I; REAL R;

R ← I



L 2,I

L 1,=X'022'

L, 15, base of seg 52

BAL 0,15

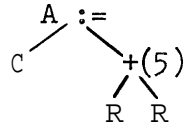
L 15, curreg base

STE 0,R

Example 2

LONG COMPLEX C; REAL R;  
 C := R + R;

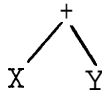
LE 0,R  
 AE 0,R  
 STE 0,TEMP  
 SDR 0,0  
 LE 0,TEMP  
 SDR 2,2  
 STD 0,C  
 STD 2,C+8



12. Arithmetic Expressions

ADDITION

The tree produced by Pass Two for addition is



Since the addition operator is commutative, the code produced does not depend on the order in which the subtrees are processed. Let X be the first subtree and Y the second.

Case I. The result of processing X is not dumped while processing Y.

If Y is in core:

	<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
Register(s) holding the result of first subtree:	R2	F0	F01	F0,F2	F01,F23
Code generated:	A 2,Y	AE 0,Y	AD 0,Y	AE 0,Y AE 2,Y+4	AD 0,Y AD 2,Y+8

If the processing of Y is in a register(s) then the following code

sequence is emitted. Assume the register(s) holding the result of processing X is as shown above.

	<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
Register(s) holding result of second subtree:	R3	F2	F23	F4,F6	F45,F67
Code generated:	AR 293	AER 0,2	ADR 092	AER 0,4 AER 2,6	ADR 0,4 ADR 2,6

Case II. The result of processing X is stored in TEMP while processing Y.

Then the result of the second subtree must be in a register(s).

	<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
Register(s) holding result of second subtree:	R2	F0	F01	F0,F2	F01,F23
Code generated:	A 2,TEMP	AE 0,TEMP	AD 0,TEMP	AE 0,TEMP AE 2,TEMP+4	AD 0,TEMP AD 0,TEMP+8

#### MULTIPLICATION

The tree produced by Pass Two for multiplication is



Since the code needed for complex and long complex multiplication is lengthy, a run-time subroutine is called for multiplication of these types. A discussion of the linkage and parameter conventions is found elsewhere in this section.

For integer, real, and long\_real, the situations and corresponding



codes are identical with those for addition except for the following substitutions in the code sequences:

<u>Addition</u>	<u>Multiplication</u>
A	M
AR	MR
AE	ME
<b>AER</b>	<b>MER</b>
AD	MD
ADR	MDR

All integer multiplications are followed by SLDA r,32 where r specifies the even register of the result. This instruction detects an overflow if it occurred during the multiplication.

#### SUBTRACTION

The tree produced by Pass Two for subtraction is



There are four situations which can arise while processing the tree<sup>9</sup> as in the case of arithmetic assignment (cf. IV.D.22).

Case I. Process X first.

A. The register(s) holding the result of the left subtree X is not dumped while processing Y.

	<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
Register(s) holding X: R2	F0	F0	F01	F0,F2	F01, F23
Code generated:	s 2,Y	SE 0,Y	SD 0,Y	SE 0,Y	SD 0,Y
				SE 0,Y+4	SD 0,Y+8

- B. The register(s) holding X is dumped at TEMP while processing Y.

The result of processing Y must then be in a register(s).

	<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
Register(s) holding X:	R2	F0	F01	F0,F2	F01,F23
Code generated:	L 3,TEMP SR 3,2	LE 2,TEMP SER 2,0	LD 2,TEMP SDR 2,0	LE 4,TEMP LE 6,TEMP+4 SER 4 9 0 SER 6,2	LD 4,TEMP LD 6,TEMP+8 SDR 4,0 SDR 6,2

Case II. Process Y first.

- A. The register(s) holding Y is not dumped while processing X. X is then loaded into a register(s) and the appropriate register-to-register instruction is generated.
- B. The register(s) holding Y is stored in TEMP while processing X. The result of X is then loaded into a register and the appropriate subtract from storage (TEMP) is generated.

#### DIVISION

The tree produced by Pass Two for division is



As in multiplication, complex and long complex division is performed in a run-time subroutine and is discussed elsewhere in this section.

Integer division is accomplished using DIV and REM and is also discussed elsewhere in this section. For real and long real, the situations and corresponding code sequences are identical with those for subtraction except for the following substitutions in the code sequences.

<u>Subtraction</u>	<u>Division</u>
SE	DE
SER	DER
SD	DD
SDR	DDR

#### DIV AND REM

The trees produced by Pass Two for DIV and REM are



The code sequences for both are identical. After the division, the result of DIV is in the odd register of the even-odd pair required for integer division, and the result of REM is in the even register.

No matter which subtree is processed first, the dividend is eventually placed in the even register of an even-odd register pair. This register pair is then shifted right-double-arithmetic  $32_{10}$  bit positions in order to place the dividend in the odd register. The division is then performed with the divisor in a register if it has been placed there or from storage if the divisor is simply a single variable or if it has been dumped into storage while processing the dividend subtree.

As an example, consider

A DIV A1(1)

where A1 is a 1-dimensional integer array. Assume the subscripting has been accomplished leaving A1(1) in R2. Then

L	4,A
SRDA	4,32
DR	4,2

The result is then in R5.

If an even-odd register pair is not available, then the fewest number of registers are dumped (maximum of two) in order to secure the even-odd pair.

As another example, consider

A1(1) DIV A .

As before, A1(1) will be processed first - assume A1(1) is left in R2 with R3 already occupied.

LR	4,2
SRDA	4,32
D	4,A

#### COMPLEX MULTIPLICATION AND COMPLEX DIVISION

Complex multiplication and division are carried out by means of a subroutine.

For multiplication, one multiplier must be in the pair of floating point registers F01 and F23, and the second in storage. If necessary, one multiplier will be stored in a temporary location. Separate

routines exist for complex and long complex multiplication. The calling sequence when one multiplier is in location `TEMP` is:

```

LA      1,TEMP
L       15, base of segment 62
MVI     FLAG,X'02'
BALR    0,15
X'0001'
L       15, base of current segment

```

For division, the numerator must be in the pair of floating point registers `F01` and `F23`; the denominator must be in storage. If necessary, the denominator will be stored in a temporary location. Separate routines exist for complex and long complex division. The calling sequence when the denominator is in location `TEMP` is:

```

LA      1,TEMP
L       15, base of segment 62
MVI     FLAG,X'02'
BALR    0,15
X'0003'
L       15, base of current segment

```

The algorithm used for complex multiplication  $X := A*B$  is

$$e+if := (v + iw) * (x + iy)$$

```

r := y * w           s := y * v
e := v * x - r;     f := w * x + s;

```

The algorithm used for complex division  $X := A/B$  is:

$$e + if := (v + iw) / (x + iy)$$

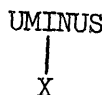
```

r := abs x;    s := abs y;
if r >= s then
  begin r := y/x;    s := y * r + x;
        t := (r * w + v)/s;  e := v * r;
        f := (w - e)/s;    e := t;
  end else
  begin r := x/y;    s := r * x + y;
        t := (r * v + w)/s;  f := (w * r - v)/s;
  end;

```

#### UNARY MINUS

The tree produced by Pass Two for unary minus is



The result of processing the subtree is loaded into a register(s).

	<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
Register(s) holding result of processing subtree:	R2	F0	F01	F0,F2	F01,F23
Code generated:	LCR 2,2	LCER 0,0	LCDR 0,0	LCER 0,0 LCER 2,2	LCDR 0,0 LCDR 2,2

## EXPONENTIATION

The tree produced by Pass Two for exponentiation is



Since the code needed for exponentiation is lengthy, exponentiation for all types of bases is accomplished with run-time routines. Recall that all powers must be of simple type integer.

One run-time routine, `EXPON`, handles bases of simple type integer, real and long real, converting the base to long real before exponentiating. Input to the routine is the type of the base, the register holding the base, and the register holding the power. The result of the exponentiation is left in the register of the base if the base is of simple type real or long real. If the base is of simple type integer, the result is left in `F01`.

Another run-time routine, `CEXPON`, handles the bases of simple type complex and long complex, converting the base to long complex before exponentiating. Input to the routine is the simple type of the base, the base in `F0`, `F2` (or `F01`, `F23`), and the register holding the power. The result of the exponentiation is left in `F01`, `F23`.

Consider  $X ** Y$ , where  $X$  is real and in  $F4$  and  $Y$  is in  $R3$ . Then the calling sequence for `EXPON` is

<code>LA</code>	<code>0,X'243'</code>	simple type of base, reg. of base, reg. of power
<code>MVI</code>	<code>FLAG,X'01'</code>	
<code>L</code>	<code>15</code> , base of standard functions	
<code>BALR</code>	<code>1,15</code>	
<code>X'0001'</code>		
<code>L</code>	<code>15</code> , base of current segment	

Now consider  $X ** Y$  where  $X$  is long complex (in F01, F23) and  $Y$  is in R2. Then the calling sequence for CEXPON is

```

LA      0,X'502'
MVI     FLAG,X'01'
L       15, base of standard functions
BALR    1,15
X'0002'
L       15, base of current segment

```

The algorithm for real exponentiation is given in the form of an Algol W procedure,

```

LONG REAL PROCEDURE EXPON (LONG REAL VALUE BASE; INTEGER VALUE POWER);
BEGIN
    LONG REAL X; BITS A; LOGICAL NEGATIVE,
    NEGATIVE := FALSE;
    IF POWER < 0 THEN
        BEGIN
            POWER := -POWER; NEGATIVE := TRUE
        END;
    A := BITSTRING(POWER); X := 1L;
L:    B := A; A := A SHR 1;
    IF (B AND #1) = #1 THEN X := X * BASE;
    IF A AND #0 THEN
        BEGIN
            BASE := BASE * BASE; GOTO L
        END;
    IF NEGATIVE THEN 1L/X ELSEX
END EXPON;

```

The algorithm for CEXPON is the same as for EXPON except all long real's above become long complex%,



## ABSOLUTE VALUE

The abs operator has an argument of any arithmetic simple type. For the simple types integer, real and long real, the quantity must first be placed in a register r corresponding to its type, if it is not already there, and one of the following instructions executed:

```
LPR      r,r      for integer
LPER      r,r      for real
LPDR      r,r      for long real
```

For the types complex and long complex, a subroutine is called to obtain the absolute value, which is a real or long real number. The argument of the operator must be placed in the floating point register pair F01,F23. The result is returned in register F01. Separate routines exist within the subroutine for complex absolute value and long complex absolute value. The calling sequence for the routine is:

```
L          15, base of segment 62
MVI      FLAG,X'01'
BALR     1,15
X'0004'
L          15, base of current segment
```

The algorithm for the complex absolute value is:

```
a := | x + iy |
x := abs x; y := abs y
a := if x = 0 then y else if y = 0 then x else
      if x > y then x * sqrt (1 + (y/x) ** 2)
      else y * sqrt (1 + (x/y) ** 2)
```

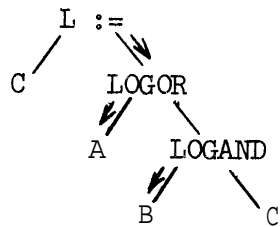
### 13. Logical Expressions

The philosophy of implementation of logical expressions was guided by two principles. First, only those parts of the expression needed to determine the truth value of the whole expression need be evaluated. For instance, in the expression A or (B and C), if A is true the whole expression is true. Therefore, neither B nor C requires evaluation if A is true. Analogously, if A evaluates to be false, B must be evaluated. If B is false, C need not be evaluated since the whole expression is false. A, B, and C are all evaluated only if A is false and B is true,

The second principle followed in implementation required that an explicit logical result be created in a register only when necessary. For example, the logical expression of the conditional statement, if A or B then S, need not have a logical value created for the expression A or B. Only a 'branch is required 'based on the condition code set by the evaluated expression. As succeeding examples will illustrate, the principle involving explicit evaluation is carried to its ultimate in logical conditional expressions and conditional ease expressions with at most one extraneous branch instruction being emitted after the expression.

1. logical A,B,C

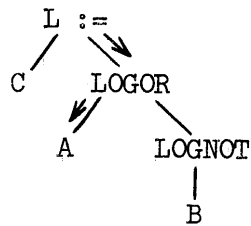
C := A or B and C



	CLI	A,X'01'
	BC	=,T
	CLI	B,X'01'
	BC	≠,F
	CLI	C,X'01'
	BC	≠,F
T	LA	2,1
	B	STORE
F	LA	2,0
STORE	STC	2,C

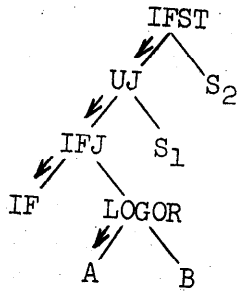
2. Arithmetic

C := A or ¬ B



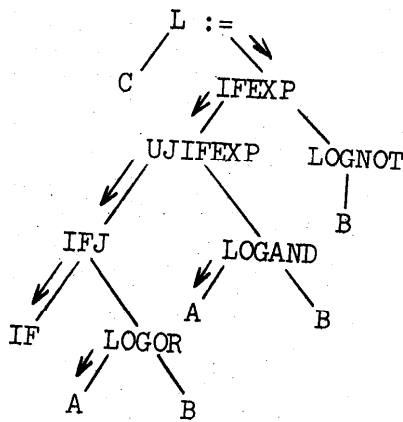
	CLI	A,X'01'
	BC	=,T
	CLI	B,X'01'
	BC	≠,T
	LA	2,0
	B	STORE
T	LA	2,1
STORE	STC	2,C

3. 1A,B,a1 C  
 if A or B then S else S



	CLI	A,X'01'
	BC	=,T
	CL1	B,X'01'
	BC	≠,F
T	S <sub>1</sub>	
	B	NEXT
NEXT	S <sub>2</sub>	

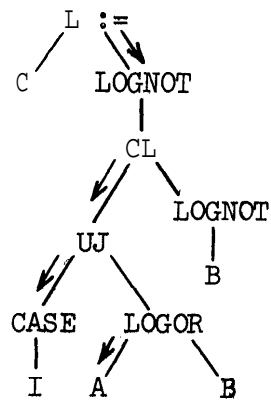
4. logical ~ A, &, C  
 C := if A or B then A and B else ¬ B;



	CLI	A,X'01'
	BC	=,T1
	CLI	B,X'01'
	BC	≠,F1
T1	CLI	A,X'01'
	BC	≠,F2
	CLI	B,X'01'
	BC	≠,F2
	B	T2
F1	CLI	B,X'01'
	BC	=,F2
T2	LA	2,1
	B	STORE
F2	LA	2,0
STORE	STC	2,C

5. A, B, C, I

$C := \neg (\text{case } I \text{ of } (AVB, \neg B))$



	L	2,1
	LA	1,2
	CR	2,1
	BAL	1,ARRAYERR
	LTR	2,2
	Bc	≤,MAINERR
	SLA	2,2
	B	LAST(2)
L1	CLI	A,X'01'
	BC	=,T
	CL1	B,X'01'
	BC	≠,F
	B	T
L2	CLI	B,X'01'
	B	=,F
LAST	B	T
	B	L1
	B	L2
T	B	0,F
	LA	2,0
	B	STORE
F	LA	2,1
STORE	STC	2,C

## RELATIONAL OPERATORS

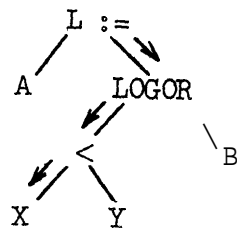
Relational expressions give logical results and hence are treated the same as logical expressions in that an explicit value is not created unless necessary. In the case of the equivalence or nonequivalence of logical expressions a truth value for one side of the expression must be explicitly generated and the address of the resulting truth value placed in a register.

In the case of string expressions, efforts have been made to use the CLC instruction as efficiently as possible in analogy to the use of MVC instructions in string assignments.

### 1. Arithmetic relations

logical A,B; 1 X,Y

A := (X < Y) or B

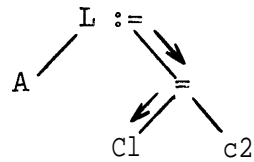


	LE	2,X
	CE	2,Y
	BC	<,T
	CL1	B,X'01'
	BC	≠,F
F	LA	2,0
	B	STORE
T	LA	2,1
STORE	ATC	2,A

2. Complex relation

complex C1,C2; logical A;

A := C1 = C2

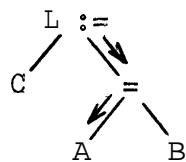


	LE	0,REAL(C1)
	LE	2,IMAG(C1)
	LE	4,REAL(C2)
	LE	6,IMAG(C2)
	CER	4,0
	BC	≠,F
	CER	6,2
	BC	=,T
F	LA	2,0
	B	STORE
T	LA	2,1
STORE	STC	2,A

3. Logical relations

a. logical A,B,C

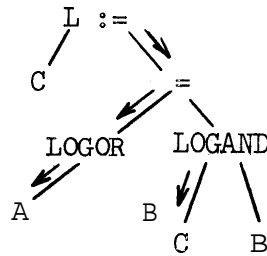
C := A = B



	IC	2,A
	N	2,=F'1'
	IC	3,B
	N	3,=F'1'
	CR	3,2
	BC	=,T
	LA	2,0
	B	STORE
T	LA	2,1
STORE	STC	2,C

b. logical A,B,C

C := (A or B) = (C AND B)

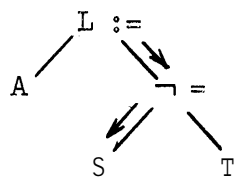


	CLI	C,X'01'
	BC	≠, F1
	CLI	B,X'01'
	BC	≠, F1
T1	LA	2,1
	B	NEXT
F1	LA	2,0
NEXT	CLI	C,X'01'
	BC	=,T2
	CLI	B,X'01'
	BC	=,T2
	LA	3,0
	B	COMP
	LA	3,1
COMP	CR	3,92
	BC	=,T3
T2	LA	2,0
	B	STORE
T3	LA	2,1
STORE	STC	2,C

4. String relation

string (5) S,T; logical A;

A := S  $\neg$  = T



	CLC	S(4),T
	B	≠,T
	LA	2,0
	B	STORE
T	LA	2,1
STORE	STC	2,A

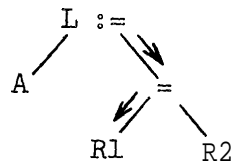


5. Reference relation

logical A;

reference (R) R1,R2;

A := R1 = R2



L	2,R1
C	2,R2
BC	=,T
LA	2,0
B	STORE
T	LA
STORE	STC
	2,A

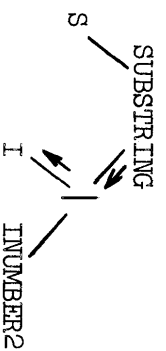
14. String Expressions

The substring operator forms a string valued expression of the form  $V(E|N)$  where  $V$  is a simple variable, an array variable or record field,  $E$  is an integer expression and  $N$  is an integer number. The result of the expression is an address of the string in a general register. The restriction that  $0 \leq E \leq (\text{length of } V) - N$  is checked. If  $E$  is an integer constant, the restriction may be checked at compile-time and the run-time code shortened.

Example 1

STRING(5) S; INTEGER I;

S(5|2)

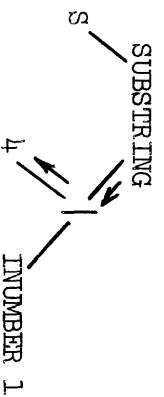


L r,I  
LTR r,r  
BAL 1,ARRAYERR  
IA 0,3  
CR 0,r  
BC ≤,MAINERR  
IA r,s(r)

Example 2

STRING(5) S;

... S(4|1)



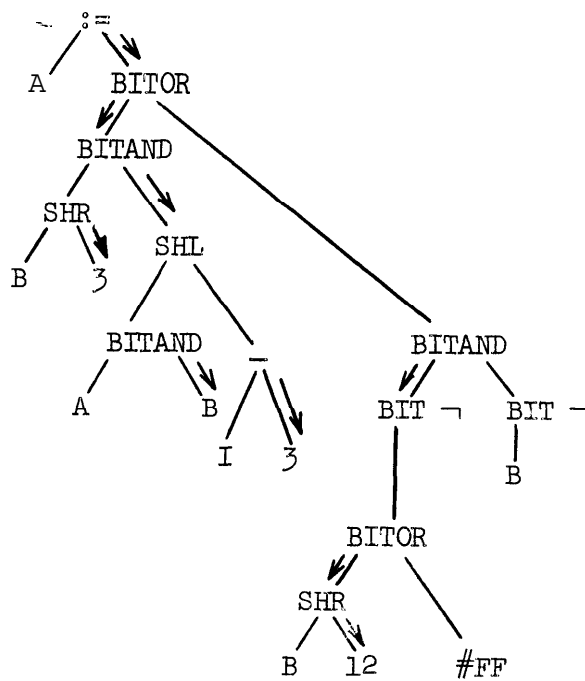
L r,=F'4'  
IA r,s(r)

15. Bit Expressions

Bit sequences may be ANDed, ORed or shifted, For the shift operations, the absolute value of the shift expression is loaded, No distinction is made between constant and nonconstant shift expressions. The compile-time procedures involved are SHIFAMOUNT, BITSSHIFTARG2, and BITSANDORARG2.

As an example, consider the following:

A:=B shr 3 and (A and B) shl (I-3) or  $\neg$  (B shr 12 or #FF) and  $\neg$  B;



- L 2,=3
- L 3,I
- SR 3,2
- LPR 3,3
- L 2,B
- N 2,A
- ALL 2,0(3)
- L 3,=3
- LPR 3,3
- L 4,B
- SRL 4,0(3)
- NR 2,4
- L 3,12
- LPR 3,3
- L 4,B
- SRL 4,0(4)
- OR. 4,=X'FF'
- XOR 4,=X'FFFFFFFF'
- L 3.B
- XOR 3,=X'FFFFFFFF'
- NR 4,3
- OR 2,4
- ST 2,A

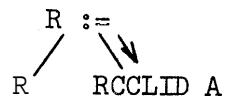
16. Record gnators

ALGOL W permits records to be created in two ways, First, the name of the record class may stand alone. Second, the name of the record class may be followed by a list of the initial values of the fields. Both record creations are reference expressions.

RECORD A(INTEGER I,J);

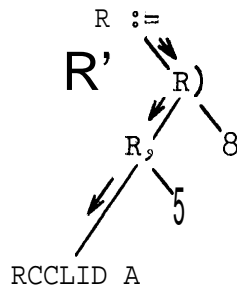
REFERENCE(A) R;

R :=A;



LA	3, address of A's free record chain (FRC)
L	15, base of record creator
BALR	1,15
L	15, current segment base
ST	3,R

R := A(5,8);



LA	3, address of A's FRC
L	15, base of record creator
BALR	1,15
L	15, current segment base
L	4,=F'5'
ST	4,0(,3)
L	4,=F'8'
ST	4,4(,3)
ST	3,R

17. Field Designators

Since a reference points to a record with fields of any of the nine simple types, field designators of the form

$$F(R)$$

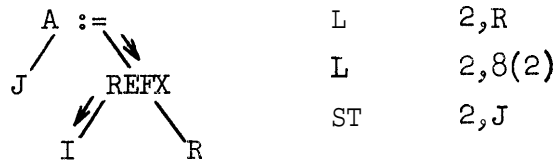
where F is a field name and R a reference expression **select** the desired field of the **simple type declared for F**. Throughout the compiler, the loading of the reference value into a register is analogous to the address resulting from a subscript calculation. This address is then used as a base to index the proper element of the record while the displacement is the relative displacement of field F within the record,

```

record At(reference (A) X, Y; I ) ;
Integer reference (A) R;

J := I(R);

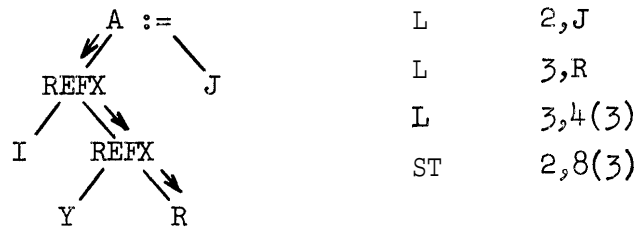
```



```

I(Y(R)) := J;

```



## 18. Case Statements and Case Expressions

The purpose of the case construction is to select the statement or expression given by the value of the expression following case. When beginning case expressions all registers except the for-variable register are stored. This occurs immediately before the unconditional branch selecting the appropriate expression.

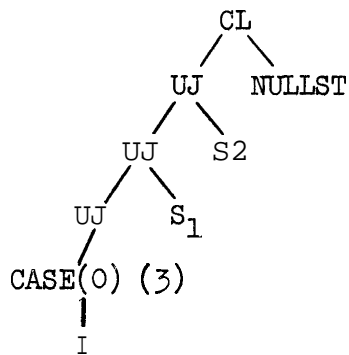
1. case I of

begin

S<sub>1</sub>;

S<sub>2</sub>;

end;



L	2,I
LA	1,3
CR	2,1
BAL	1,ARRAYERR
LTR	2,2
BC	≤,MAINERR
SLA	2,2
B	LAST(2)

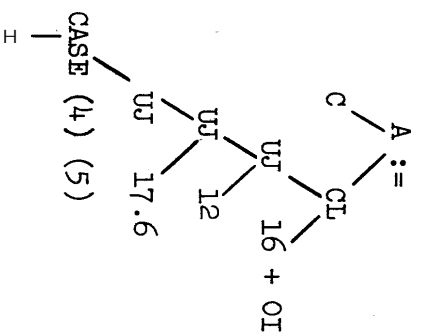
L1:	S <sub>1</sub>	
	B	NEXT

L2:	S <sub>2</sub>	
	B	NEXT

LAST	L3	B	NEXT
		B	L1
		B	L2
		B	L3

NEXT

2. C := case I of (17.6, 12, 16 + 0I)



L	2, I
IA	1, 5
CR	2, 1
BAL	1, ARRAYERRØR
LTR	2, 2
BC	≤, MAINERR
SIA	2, 2
BC	15, LAST(2)
IE	0, =R'17.6'
SER	2, 2
B	NEXT
L	2, =F'12'
IA	1, X'022'
L	15, INTREAL
BALR	0, 15
L	15, current seg base
SDR	2, 2
B	NEXT
L3	0, =R'16.0'
LE	2, =R'0.0'
B	NEXT
B	L1
B	L2
B	L3
NEXT	0, C
STE	2, C+4

19. If Statement, If Expression, While Statement

The while statement has the following interpretation,

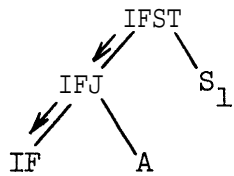
```

WHILE C DO S1 ≡ L: IF C THEN
                   BEGIN S1; GO TO L
                   END
    
```

All registers except the control variable register must be dumped before entering the if expression. They are dumped before the evaluation of the conditional expression,

1. Logical ;

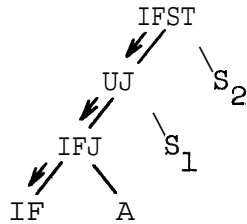
if A then S<sub>1</sub>



```

CLI   A,X'01'
BC    ≠,NEXT
S1
NEXT
    
```

2. If A then S<sub>1</sub> else S<sub>2</sub>

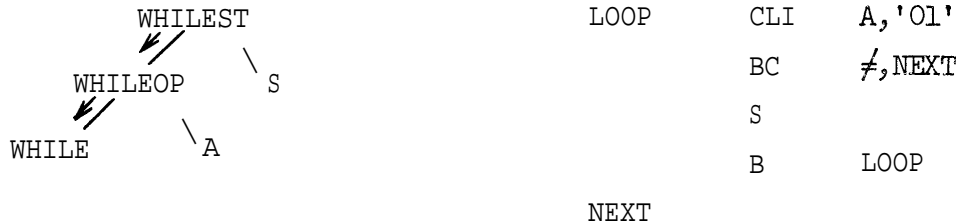


```

CLI   A,X'01'
BC    ≠,L
S1
B     NEXT
S2
L
NEXT
    
```



3. while A do S



20. For Statement

The two kinds of for statements will be designated here -- the step-until statement and the for-list statement

A, The control identifier

Both the step-until and for-list statements have control identifiers, The implementation treats this identifier essentially the same in both cases. R2, designated symbolically as FORREG, is generally used to hold its value. Each control identifier is also assigned by Pass Two a relative location in a data segment, into which the value is stored when a transfer of control to a closed subroutine is to occur or R2 is needed for some other purpose, At compile-time GETADDRESS will deliver the correct register or location for a reference to a control identifier, The occurrence of the control identifier immediately after for causes the initial processing of this identifier; this is done by NUMERICALASSIGN.

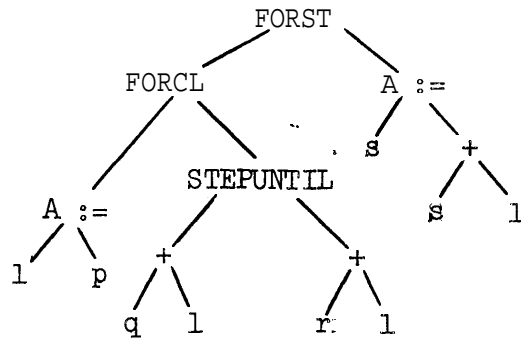
At compile-time a 20-word stack CSTACK and a location LASTFORLOC are used to keep track of the locations of the various control identifiers that may be active at a given time. At any time LASTFORLOC holds the address assigned by Pass Two to the innermost control identifier

for the text being compiled. CSTACK is a stack of pointers to the entries in LSTACK which are control identifier locations. The pointer for CSTACK itself is a memory location called CPOINTER.

The routines DUMPFORREG and RESTOREFORREG generate instructions to move the value of a control identifier to and from memory as required.

#### B. Step-until statement

In addition to the memory location for the control identifier, three other locations are used for each statement of this type. These are assigned by Pass Three and are called "incr", "mask", and "lim"; they hold the increment value, the mask used by an execute instruction in the test, and the limit value, respectively. The example below illustrates their use.



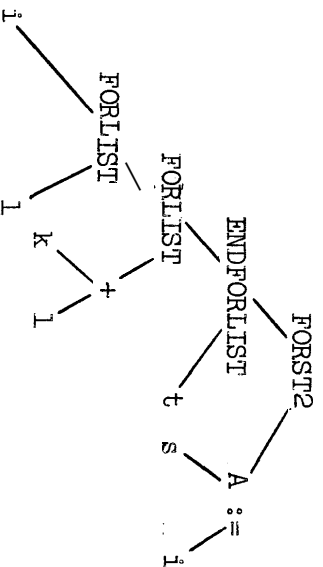
for i := p step q+1 until r+1 do s := s+1

	L	2,p
	L	3,q
	A	3,one (one contains 1)
	LTR	3,3
	ST	3,incr
	LA	3,const (const contains 20 <sub>16</sub> )
	BC	≥,*+8
	SLL	3,one
	ST	3,mask (=0010 0000 or 0100 0000),
	L	3,r
	A	3,one
	ST	3,lim
	B	*+8
L	A	2,incr
	C	2,lim
	L	3,mask
	EX	3,M
	L	3,s
	AR	3,2
	ST	3,s
	B	L
M	BC	0,*+4

C. For-list statement

In the case of a for-list statement, the statement following the for clause is compiled as a closed subroutine. R1 is used for branch-and-link instructions. The following example illustrates the compiled code.

for i := 1, k+1, t do s := i



L	2,=F'1'
BAL	1,L
L	2,k
A	2,=F'1'
BAL	1,L
L	2,t
BAL	1,L
B	N
L	
ST	1,TEMP
ST	2,s
L	1,TEMP
BR	1
N	

The addresses in the BAL instructions are fixed up by a simple chaining.

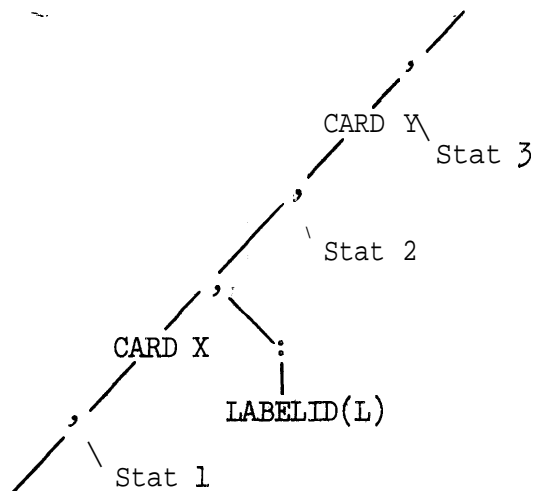
## 21. Goto tement

A branch table is built in the head of each program segment, and each label in the procedure is represented by a branch instruction in the branch table,

The Pass Two tree format for a labeled statement

```
Stat 1;  
L: Stat 2;  
Stat 3;
```

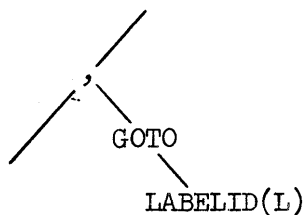
is as follows:



where L is a pointer to the **NAMETABLE**. Since the left sub-trees are always processed first, the label declaration is encountered just before the compilation of Stat 2.

When the node **LABELID(L)** is encountered, as above, the **NAMETABLE** entry for L enables Pass Three to calculate the address of the branch instruction corresponding to the label L in the branch table in the head of the procedure. The current value of the instruction counter is then placed in the displacement field of the branch instruction.

The Pass Two tree format for the statement goto L is as follows:



where L is a pointer to the NAMETABLE. With the NAMETABLE entry for L, Pass Three looks up the address of the branch instruction in the branch table corresponding to the label L. If this address (relative to the base of the program segment is  $\alpha$ , then the code

B      $\alpha(15)$

is emitted,

By the end of compilation of the procedure, all labels have been encountered and all branch instructions in the branch table have their correct form,

If the label occurs in a different program segment, code is emitted for procedure exit, for loading R15 with the base of the program segment being branched to, and for a branch to the appropriate instruction in the branch table of the target program segment,

The following is the code generated for the statement goto L where n is the number of the register which gives the base of the data segment where the label L is defined, and  $\alpha$  is the displacement of the instruction in the branch table corresponding to the label L. The label L is in a procedure different from the procedure where the goto statement occurs.

	ST	n,MP	reset data stack pointer
X	L	15, base of program segment in which label resides	
	B	$\alpha(15)$	

Notice that precisely the same code is emitted for a **branch** out of a block, e.g.

```

begin integer A;
  :
  begin integer B;
    :
    Lyto
    .
  end;
  :
  ...L
end;

```

In this case, the load instruction at X above is superfluous and is not compiled.

#### GOTO STATEMENTS AND LABELS INSIDE FOR-LOOPS

Because of the manner in which the control identifier is manipulated inside a for-loop and the desire to keep the innermost control identifier in a register whenever possible, special code is emitted for goto statements and labels which are inside the scope of a for-loop,

As explained more fully in the section on for-loops (cf. IV.D.20), Pass Two allocates one word in the data stack for each control identifier. In the event that a control identifier must be dumped, it is dumped into its special location rather than into the local stack.

Since only the innermost control identifier is kept in a register, the compiler always has a variable LASTFORLOC which contains the relative address of the word in the data stack into which the control identifier is dumped when necessary and from which it is reloaded.

- 1) For a goto statement inside the scope of a for-loop,, the control identifier is first dumped into LASTFORLOC:

```

      ST      2,LASTFORLOC(n)
      B      α(15)           branch to branch table

```

- 2) At the definition of a label L, a branch is made around the instruction to which transfer is controlled by the branch instruction in the branch table, At the label, the control identifier is reloaded, i.e.:

```

      BC      NEXT
L      L      2,LASTFORLOC(n)
      NEXT

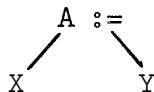
```

This allows transfers within a for-loop and from an inner for-loop into an outer for-loop.

## 22. Assignment Statements

### ARITHMETIC ASSIGNMENTS

The tree produced by Pass Two for arithmetic assignments is



Since the discussion concerning implicit conversion between the arithmetic types occurs elsewhere in this report (cf. IV.D.11), this



section will deal only with arithmetic assignments of identical type,

Four situations may occur in processing an arithmetic assignment since either the right or left subtree may be processed first, and for each of these cases, the register(s) holding the result of the subtree processed first may be dumped while processing the second subtree.

I. Process right subtree first

- A. The register(s) holding Y is not dumped while processing the left subtree.

	<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
Register(s) holding X:	R2	F0	F0P	F0,F2	F01,F23
Code generated:	ST 2,X	STE 0,X	STD 0,X	STE 0,X STE 2,X+4	STD 0,X STD 2,x+8

- B. The register(s) holding Y is dumped while processing the left subtree.

This situation may occur when the left subtree contains a procedure call. For example

$$X(P) := Y$$

where X is a 1-dimensional array and P is an integer procedure with no arguments.

Assume the register(s) holding the results of the right subtree have been dumped at TEMP, and that general register.2 holds the address of X(P).

Code generated:

<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
L 3,TEMP	LE 0,TEMP	LD 0,TEMP	LE 0,TEMP	LD 0,TEMP
ST 3,0(2)	STE 0,0(2)	STD 0,0(2)	LE 2,TEMP+4	LD 2,TEMP+8
			STE 0,0(2)	STD 0,0(2)
			STE 2,4(2)	STD 2,8(2)

II. Process left subtree first.

Assume the processing of the left subtree results in an address in general register 2.

A. R2 is not dumped while processing the right subtree.

	<u>Integer</u>	<u>Real</u>	<u>Long Real</u>	<u>Complex</u>	<u>Long Complex</u>
Register holding Y: R3		F0	F01	F0,F2	F01,F23
Code generated:	ST 3,0(2)	STE 0,0(2)	STD 0,0(2)	STE 0,0(2)	STD 0,0(2)
				STE 2,4(2)	STD 2,8(2)

B. R2 is dumped at TEMP while processing the right subtree.

The code sequences are then identical to those given in II.A except that each code sequence is prefixed by

L 2,TEMP

#### LOGICAL ASSIGNMENTS

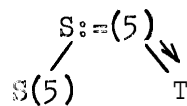
For logical assignments, a truth value must be generated, 1 represents true and 0 represents false. This value is placed in an integer register and stored by an STC instruction. Examples of this assignment may be seen in the section concerning logical expressions, (IV.D.13).

STRING ASSIGNMENTS

The assignment of string variables is defined so that the assignment takes place left to right, character by character. If the assigned string is shorter than the destination string, the remaining characters are filled with blanks. The MVC instruction is used for the assignment and some combination of MVI and MVC instructions used for the insertion of blanks. The length of the assignment appears in the conversion bits of the S:= operator and the length of the string appears in the node immediately to the left of the S:= node.

Example 1

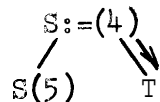
STRING(5) S,T; S:=T



MVC S(5),T

Example 2

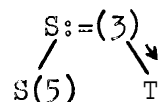
STRING(5) S; STRING(k) T; S:=T



MVC S(4),T  
MVI S+4,X'40'

Example 3

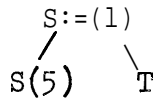
STRING(5) S; STRING(3) T; S:=T



MVC S(3),T  
MVI S+3,X'40'  
MVI S+4,X'40'

Example 4

STRING(5) S; STRING(1) T; S:=T



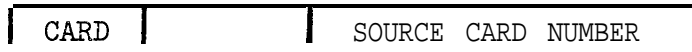
MVC S(1),T  
MVI S+1,X'40'  
MVC S+2(3),S+1

REFERENCE ASSIGNMENTS

Reference assignments are handled just as integer assignments are handled in the integer registers. Examples of reference assignments may be seen in the section on field designators (cf. IV.D.16).

23. Card Numbers

In order to give the user a meaningful message if an error occurs during Pass Three or at run-time, a unary card node having the form



is placed in various places in the tree, as described in the documentation of Pass Two. With this information, Pass Three always has available the current (or almost current) user card number. If an error occurs during Pass Three, the current card number is printed out along with an appropriate message.

In addition, to prepare for possible errors at run-time, Pass Three builds one table for each user procedure (including the main block) associating a card number with a relative location in the user's procedure.

If no errors are detected during Pass Three, the card tables are

written out onto the same device used to hold the user's compiled procedures prior to their loading and execution. The card tables are written out only after all the user ' s procedures have been written out, and associating each card table with a procedure, the card tables are written out in order of ascending (procedure) number, beginning at 1.

If an error is detected at run-time, the absolute location of the error is available to a run-time error routine. This routine determines the number of the user procedure in which the error occurred by scanning the program reference table which contains the base addresses of all user procedures . In addition, the relative location of the error within that procedure is determined. The appropriate card table is then read in, and with the relative location available, the card number is retrieved.

### E. Trace Facilities

An optional trace card of the form \$TRACE<sub>xy</sub> beginning in column 1 of the card allows the user to trace certain features of the compilation and execution of his job,

x and y are integers which may take on the following values, with the associated results:

x	<u>Action</u>
2 or greater	Complete map of all compiler passes is printed.
6	All actions of garbage collector are printed.
4 or greater	In case of run error, dump of absolute location of error, contents of general registers, data area, and record and run-time data area are printed.
0 or blank	None of the above.

Different values of y will cause printing of different parts of the output of Pass Two and Pass Three of the compiler, The following abbreviations will be used:

NT	nametable
BL	blocklist
TREE	tree
lst	compiled code before certain addresses are fixed up -listed as procedure is being compiled.
final	final version of compiled code which will be executed - listed at end of procedure compilation,
reg	contents of general registers at end of compiling a procedure.

<u>y</u>	<u>Actions</u>
1	reg, final
2	lst, reg, final
3	NT, BL
4	NT, BL, reg, final
5	NT, BL, lst, reg, final
6	TREE, NT, BL
7	TREE, NT, BL, reg, final
8	TREE, NT, BL, lst, reg, final
0	no action

The trace card ~~\$STACK~~ has the same effect as ~~\$TRACE03~~.

APPENDIX I

EXAMPLE OF ALGOL W COMPILER OUTPUT

SOURCE LISTING

XALGOL

```

0001      BEGIN
0002      REAL X,SUMX,MEANX;
0003      INTEGER N,I;
0004      I := 0;
0005      SUMX := MEANX := 0 ;
0006      READ(N);
0007      WRITE(N);
0008      L:READON(X);
0009      I := I + 1;
0010      sum x := SUMX + X ;
0011      MEANX := SUMX / I;
0012      WRITE(I,X,SUMX,MEANX);
0013      IF I = N THEN WRITE("FINISHED") ELSE GO TO L;
0014      END.
    
```

PASS ONE OUTPUT

65002A70	FE00030D	65002866	65002C70	FE000197	FE00020D	65002866	65002966
C5650029	9A65002A	9A770100	00000070	FE000463	002C9A77	01000000	0070FE00
00016A65	002B6770	FE000865	002D9965	FE000665	00106A65	00286770	FE000765
002C7E77	01000000	0170FE00	0A650029	00186A65	00286770	FE000965	002C9A65
650C2983	65002C70	FE000C65	00016A65	9A650029	76650028	70FE000B	65002A9A
FE00D78	65002C90	65002879	6500016A	002C6965	00286965	00296965	002A6770
70FE000E	6F920000	00000000	00000000	8107C6C9	D5C9E2C8	C5C4677A	94650020
				00000000	00000000	00000000	00000000



PASS TWO OUTPUT TREE

PROGRAM	SEGMENT	LOC	FLAG	OPCODE	CONV	POINTER
		0000				0159
		0004	J	PROCDC		0000
		0008	J	CARD		0001
		000C	O	BEGIN	1	0000
		0010	O	CARD		0002
		0014	O	NULLST		0000
		0018	3	BB		0010
		001C	3	NULCST		0000
		0020	O	CARD		0004
		0024	O	ID		0210
		0028	O	NUMBER		000C
		002C	1	A:=		G024
		0030	O	,		0020
		0034	O	CARD		0005
		0038	3	IO		01EC
		003C	O	ID		01F8
		0040	O	NUMBER	2	000C
		0044	1	A:=2		003C
		0048	1	A:=		0038
		004C	O	,		G034
		0050	O	CARD		0006
cc 54		0054	O	STPROCID		0000
		0058	O	ID		0204
		005C	J	AP)		0054
		0060	3	,		0050
		0064	O	CARD		0007
		0068	3	STPROCID		000C
		006C	O	ID		0204
00 70		0070	3	AP)		0068
co 74		0074	O	,		0064
		0078	O	CARD		0008
		007C	O	ID		021C
		0080	O	:		0000
		0084	3	,		0078
		0088	O	STPROCID		0120
		008C	3	ID		01E0
		0090	O	AP)		0088
		0094	O	,		0084
		0098	O	CARD		0009
		009C	O	ID		0210
		00A0	O	ID		0210
		00A4	O	NUMBER		0000
		00A8	1	+		00A0
		00AC	1	A:=		009C
		00B0	O	,		0098
		00B4	3	CARD		000A
		00B8	G	ID		01EC
		00BC	J	ID		01EC
		00C0	O	ID		01E0
		00C4	1	+		00BC
		00C8	1	A:=		00B8
		00CC	3	,		00B4
		00D0	3	CARD		000B
00D4		00D4	3	ID		01F8
G0D8		00D8	3	ID		01EC
00DC		00DC	O	ID	2	0210
G0E0		00E0	1	/		00D8

00E4	1	b:=	00D4
00E8	0	,	00D0
00EC	0	CARD	00C0
00FC	0	STPROCID	00C0
00F4	0	ID	0210
00F8	0	AP,	00F0
00FC	0	ID	01E0
0100	0	AP,	00F8
0104	3	ID	01EC
0108	3	AP,	0100
010C	0	10	01F8
0110	3	AP)	0108
0114	3	,	0GEC
0118	3	CARD	00D0
011C	0	ID	3210
	0	ID	0204
	1	=	011C
3 120124128	3	IF	0C00
012C	1	IFJ	0124
0130	0	STPROCID	000C
0134	0	STRING	0010
0138	0	AP)	0130
013C	0	UJ	012C
0140	0	LABELID	021C
0144	3	GOTO	0000
0148	0	I FST	013C
014C	0	,	0118
0150	3	CARD	000E
0154	0	END	0C18
0158	0	PCL	0008

LITERAL ORIGIN - GOOC  
LITERAL POINTER TABLE

LOC	LENGTH	TYPE	POINTER
0000		1	0000
00C4		6	0000
0008		6	0003
GGOC		1	0 0 0 4
0010	7	7	0008

LITERAL TABLE  
050108 00000001 OG0000QO C6C9D5C9 E2C8C5C4

ELAPSED TIME IS 00:01:58  
TOTAL TREE LENGTH IS 015C  
TOTAL OUTPUT LENGTH IS 018C

NAMETABLE	IDLOC1	IDLOC2	SIMTYPE	INFO	TYPEINFO	TYPE	SIMTYPE	I D
LDC	(HEX)	HN SFG			RCCLNO	(HEX)		
0000	0028	0D 1			0	03		.MAIN
000C	0000	0GGO		1		00		WRITE
0018	0000	0000				00		ADUMP
0024	0000	0COG			1	07	6	ODD
0030	0000	0000			1	07	8	BITSTRING
003C	0000	0000			8	07	1	NUMBER
0048	0000	0000			7	07	1	DECODE
0054	0000	0000	0		1	07	7	CODE
0060	0000	0000			2	07	2	TRUNCATE
006C	0000	0000			2	07	1	ROUND
0078	0000	0000			2	07	1	ENTIER
0084	0000	0oco			4	07	2	REALPART
0090	0000	0 00			4	07	2	IMAGPART
009C	0000	0 a 00			5	07	3	LONGREALPART
00A8	0000	0000			5	07	3	LUNGIMAGPART
00B4	0000	0000			3	07	3	LONGSQRT
00C0	0000	0000		2		00		READ
00CC	0000	0000			2	07	2	SQRT
00D8	0000	0000			2	07	2	EXP
00E4	0000	0000			2	07	2	LN
00FC	0000	0000			2	07	2	LOG
00FC	0000	0000			2	07	2	SIN
0108	0000	0000			2	07	2	COS
0114	0000	0000			2	07	2	ARCTAN
0120	0000	0000		2		00		READON
012C	0000	0000			3	07	3	LONGEXP
0138	0000	0000			3	07	3	LONGLN
0144	0000	0000			3	07	3	LONGLOG
0150	0000	0000			3	07	3	LONGSIN
015C	0000	0000			3	07	3	LONGCOS
0168	0000	0000			3	07	3	LONGARCTAN
0174	0000	0000			2	07	4	IMAG
0180	0000	0000			3	07	5	LONGIMAG
018C	0000	0000			4	07	4	COMPLEXSQRT
0198	0000	0000			5	37	5	LONGCOMPLEXSORT
01A4	0000	0000			1	07	1	MSGLEVEL
01B0	0000	0000			1	07	1	TIME
01BC	000E	0280				00	1	INTFIELDSIZE
01C8	000E	02AD				00	6	UNOERFLOW
01D4	000E	02AE				00	6	OVERFLOW
01E0	0000	0014				00	2	x
01EC	0000	0018				00	2	SUMX
01F8	0000	001C				00	2	MEANX
0204	0000	0020				00	1	N
0210	0000	0024				00	1	I
021C	0001	0008			13	01		L

LOCKLIST	LENGTH	POINTER
LOCKNB	01D4	0C0C
4	0048	01E0

PASS THREE OUTPUT

0001	0000	RC	47F0FC2C
	0004		000000
0001	0008	BC	47F0FC96
0001	000c	****	0003
0001	000E	****	0000
	0010		00000301
	0014		00000000
	0018		C6C9D5C9
	001C		E2C8C5C4
0001	0020	L	5820E17C
0001	0024	L	58602000
0001	0028	A	5A60E194
0301	002C	N	5460E198
3001	0030	LA	41006028
0001	0034	BALR	4540E17A
0301	0038	LA	41300000
0001	003C	LA	41400000
0301	0040	STM	90046000
0001	0044	ST	5060E170
0001	0048	LR	1806
0004	004A	L	5820F014
0004	004E	ST	5020D024
0005	0052	L	5820F014
0005	0056	LA	41100022
0005	005A	L	58F0E0E4
0005	005E	BALR	050F
0005	0060	L	58F0E004
0005	0064	STE	7000D01C
0005	0068	STE	70030018
0306	006C	MVI	92FFE179
0006	0070	LA	41203100
0006	0074	LA	41300020
0006	0078	L	58F0E0DC
0006	007C	BALR	051F
0006	007E	L	58F0E004
0307	0082	LA	41200001
0007	0086	L	5830D020
0307	008A	L	58F0E0F4
3007	008E	BALR	051F
0037	0090	****	0001
0007	0092	L	58F0E004
0308	0096	LA	41200200
0008	0094	LA	4130D014
0008	009E	L	58F0E0DC
0108	00A2	BALR	051F
0008	00A4	L	58F0E034
0009	00A8	L	5820F010
0009	00AC	A	5A20D024
0009	00B0	ST	50200024
0310	00B4	LE	78000014
0010	00B8	AE	7A00D018
0310	00BC	STE	7000D018
0311	00C0	L	58200024
0311	00C4	LA	41100022
0311	00C8	L	58F0E0E4
0011	00CC	BALR	050F

0011	00C E	L	58F0E004
0011	00D2	LE	78200018
0011	00D6	DER	3D20
0011	00D8	STE	7020D01C
0012	00DC	LA	41200001
0012	00E0	L	5830D024
0012	00E4	L	58F0E0E8
0012	00E8	BALR	051F
0012	00EA	L	58F0E004
0012	00EE	L A	41200002
0012	00F2	LE	7800D014
0012	00F6	L	58F0E0E8
0012	GOFA	BALR	051F
0012	00FC	L	58F0E004
0012	0100	LA	41200002
0012	0104	LE	7800D018
0012	0108	L	58F0E0E8
0012	010C	BALR	051F
0012	010E	L	58F0E004
0012	0112	LA	41200902
0012	0116	LE	7800D01C
0012	011A	L	58F0E0F4
0012	011E	BALR	051F
0012	0120	****	0001
0012	0122	L	58F0E004
0013	0126	L	58200020
0013	012A	C	59200024
0013	012E	BC	4770F152
0013	0132	LA	41200007
0013	0136	SLA	88200010
0013	013A	LA	41202007
0013	013E	LA	4130F018
0013	0142	L	58F0E0F4
0013	0146	BALR	051F
0013	0148	****	0001
0013	014A	L	58F0E004
0013	014E	BC	47F0F156
0013	0152	BC	47F0F008
0014	0156	LM	98120004
0014	015A	ST	5020E170
0014	015E	BCR	07F1
0014	0160	****	0000
0014	0162	****	0000

OUTPUT FROM EXECUTION OF COMPILED PROGRAM

3				
1	1.000000'+00	1.000000'+00	1.000000'+00	
2	2.000000'+00	3.000000'+00	1.500000'+00	
3	3.000000'+00	6.000000'+00	2.000000'+00	

FINISHED

APPENDIX II

SIMPLE PRECEDENCE GRAMMAR FOR ALGOL W

```

1  <T V A R I D> ::= <ID>
2  <LABEL ID> ::= <ID>
3  <T A R R A Y I D> ::= <CID>
4  <T F U N C I D> ::= <ID>
5  <R C C L I D> ::= <ID>
6  <T F L D I D 7> ::= <ID>
7  <C O N I D> ::= <ID>
8  <S T F U N C I D> ::= <ID>
9  <S T P H O C I D> ::= <ID>
10 <S I V A R D C> ::= <S I V A R D C * >
11 <S I V A R D C * > ::= <S I T Y P E > <ID>
12     <S I V A R D C * 7 > , <ID>
13 <S I T Y P E > ::= <R E F T Y P E > )
14 <R E F T Y P E > ::= R E F E R E N C E <I D >
15     <R E F T Y P E > , , <I D >
16 <A R R A Y D C 7 > ::= <B N D L S T H D > <T E X P > :: C T E X P > )
17 <A R R A Y H D > ::= <S I T Y P E > A R R A Y <I D >
18     <A R R A Y H D > , , <I D >
19 <B N D L S T H D > ::= <A R R A Y H D > (
20     <B N D L S T H D > <T E X P > :: <T E X P > ,
21 <P R O C D E C L > ::= <T P R H E A D 7 <S T A T E M E N T * >
22     <T P R H E A D >
23     C T P R H E A D > <T P R B O D Y >
24 <T P R B O D Y > ::= <T E X P >
25     <B L O C K B O D Y > <T E X P > E N D
26 <T P R H E A D > ::= <T P R H E A D * > ;
27 <T P R H E A D * > ::= <P R O C E D U R E >
28     <P R O C E D U R E > <F P A R H E A D > )
29 <P R O C E D U R E > ::= P R O C E D U R E <I D >
30     <S I T Y P E > P R O C E D U R E <I D >
31 <F P A R H E A D > ::= <F P A R H E A D * >
32     <F B N D L I S T 7 >
33 <F P A R H E A D * > ::= ( <S I T Y P E 7 <I D >
34     ( <S I T Y P E > V A L U E <I D >
35     ( <S I T Y P E > R E S U L T <I D >
36     ( <S I T Y P E > V A L U E R E S U L T <I D >
37     ( <S I T Y P E > P R O C E D U R E <I D >
38     ( P R O C E D U R E <I D >
39     <F P A R H E A D - > <S I T Y P E 7 <I D >
40     <F P A R H E A D - 7 <S I T Y P E > V A L U E <I D >
41     <F P A R H E A D - > <S I T Y P E > R E S U L T <I D >
42     <F P A R H E A D - > <S I T Y P E 7 P R O C E D U R E <I D >
43     <F P A R H E A D - > <S I T Y P E > V A L U E R E S U L T <I D >
44     <F P A R H E A D - > P R O C E D U R E <I D > ,
45     <F P A R H E A D * > , , <I D >
46 <F P A R H E A D - > ::= <F P A R H E A D * > ;
47 <F B N D L I S T > ::= <F B N D H E A D 7 * >
48 <F B N D H E A D > ::= <F A R R A Y H D > (
49     <F B N D H E A D > * ,
50 <F A R R A Y H D > ::= ( <S I T Y P E > A R R A Y <I D >
51     <F P A R H E A D - 7 <S I T Y P E > A R R A Y <I D >
52     <F A R R A Y H D > , , <I D >
53 <R C C L D C > ::= <R C H E A D > )
54 <R C H E A D > ::= <R E C O R D > ( <S I T Y P E > <I D >
55     <R C H E A D 7 > , , <I D >
56     <R C H E A D * > <S I T Y P E > <I D >
57 <R C H E A D * > ::= <R C H E A D > ;

```

```

59 <RECORD> ::= RECORD <ID9
60 <T VAR> ::= <SIT VAR>
61 <T ARRAY ID9
62 <STR SEL HO9 <T EXP> <LENGTH9 )
63 <STR SEL HD> ::= <SIT VAR> (
64 <LENGTH> ::= | <T NUMBER>
65 <SIT VAR> ::= <TVAR ID>
66 <T FLD HD> <T EXP> )
67 <T ARRAY HD> <T EXP> )
68 <T ARRAY HD> * )
69 <T FLD HD> ::= CT FLD ID9 (
70 <T ARRAY HO> ::= <T ARRAY ID> (
71 <T ARRAY HD> <T EXP> ,
72 <T ARRAY HD> * ,
73 <T FUNC DES> ::= <T FUNC ID>
74 <APAR HEAD> <T EXP> )
75 <APAR HEAD> <STATEMENT> )
76 <APAR HEAD> )
77 <APAR HEAD> ::= CT FUNC ID> (
78 <APAR HEAD> <T EXP> ,
79 <APAR HEAD> <STATEMENT> ,
80 <APAR HEAD> ,
81 <T EXP> ::= <T EXP*>
82 <T EXP*> ::= <SIT EXP>
83 <IF CL9 <TRUE EXP> <T EXP*>
84 <CASE HEAD9 <T EXP> )
85 <IF CL9 ::= IF <T EXP> THEN
86 <TRUE EXP> ::= <T EXP> ELSE
87 <CASE HEAD> ::= <CASE CL> (
88 <CASE HEAD> <T EXP> ,
89 <CASE CL9 ::= CASE <T EXP> OF
90 <SIT EXP> ::= <SIT EXP**>
91 <SIT EXP**> <EQL OP> <SIT EXP**>
92 <SIT EXP**> <REL OP> <SIT EXP**>
93 <SIT EXP**> IS <RC CL ID>
94 <SIT EXP**> ::= <SIT EXP*>
95 <SIT EXP*> ::= <T TERM>
96 + <T TERM9
97 - <T TERM9
98 <SIT EXP*> + <T TERM9
99 <SIT EXP*> - <T TERM9
100 <SIT EXP*> OR <T TERM9
101 <RC CL 109
102 <RC DES HD> <T EXP> )
103 <STRING>
104 NULL
105 <T TERM> ::= <T TERM*>
106 <T TERM*> ::= <T FACT9
107 <T TERM*> * <T FACT>
108 <T TERM*> / <T FACT>
109 <T TERM*> DIV <T FACT>
110 <T TERM*> REM <T FACT9
111 <T TERM*> AND <T FACT>
112 <T FACT> ::= CT SECON>
113 -> <T FACT9
114 <T SECON> ::= <T PRIM9
115 <T SECON> <SHL O R **> <T PRIM>
116 <T SECON> SYR <T PRIM9
117 CT PRIM> ::= <T VAR>
118 <T FUNC DES>

```

```

119         <ST FUNC ID>
120         <LEFT PAR> <T EXP> )
121         TRUE
122         FALSE
123         <CON ID>
124         LONG <T PRIM>
125         SHORT <T PRIM>
126         ABS <T PRIM>
127         <T NUMBER>
128         <BIT SEQ>
129 <REL OP> ::= <
130         < =
131
132         > =
133 <EQL OP> ::= =
134         ~ =
135 <LEFT PAR> ::= (
136
137 <RC DES HD> ::= <RC FUNC ID> (
138         <RC CL ID> (
139         <RC DES HD> <T EXP> ,
140 <PROGRAM 9> ::= . <BLOCK> .
141 <STATEMENT> ::= <STATEMENT*>
142 <STATEMENT*> ::= <ST ST>
143         <FOR CL 9 00>
144         <FOR CL> DO <STATEMENT*>
145         <WHILE CL> DO
146         <WHILE CL> DO <STATEMENT*>
147         <IF CL>
148         <IF CL> <STATEMENT*>
149         <IF CL 9 <TRUE PART>
150         <IF CL> <TRUE PART> <STATEMENT* 9>
151         <CASE SEQ> END
152         <CASE SEQ> <STATEMENT> END
153 <SI ST> ::= <BLOCK>
154         <T ASS ST 9>
155         <T FUNC DES>
156         GOTO <LABEL ID>
157 <BLOCK> ::= <BLOCK BODY> END
158         <BLOCK BODY> <STATEMENT> END
159 <BLOCK BODY> ::= <BLOCK HEAD>
160         <BLOCK BODY> ;
161         <BLOCK BODY> <STATEMENT> ;
162         <BLOCK BODY> <LABEL DEF>
163 <BLOCK HEAD> ::= BEGIN
164         <BLOCK HEAD> <SI VAR DC> ;
165         <BLOCK HEAD> <ARRAY DC> ;
166         <BLOCK HEAD> <PROC DECL> ;
167         <BLOCK HEAD> <KC CL DC> ;
168 <LABEL DEF> ::= <ID> :
169 <T ASS ST> ::= <T VAR> ::= <T EXP*>
170         <T VAR> ::= <T ASS ST>
171 <TRUE PART> ::= <SI ST 9 ELSE
172         ELSE
173 <CASE SEQ> ::= <CASE CL> BEGIN
174         <CASE SEQ> <STATEMENT> ;
175         <CASE SEQ>;
176 <FOR CL> ::= <FOR HEAD> <STEP UNTIL> <T EXP>
177         <FOR HEAD>
178         <FOR LIST> <T EXP>

```



```

179 <FOR HEAD>      ::= <FOR> := <T EXP*>
180 <FOR LIST>     ::= <FOR HEAD> ,
181               ::= <FOR LIST> <T EXP> ,
182 <FOR>          ::= FOR <ID>
183 <STEPUNTIL>   ::= STEP <T EXP> UNTIL
184               UNTIL
185 <WHILE CL>     ::= WHILE <T EXP>
186 <ST PROC HD>  ::= <ST PROC ID> (
187               <ST PROC HD> <T EXP>

```