THE IMPLEMENTATION OF ALGOL W ON A BURROUGHS B6700 COMPUTER

by

Henry D. Meekin

A thesis presented in partial fulfilment

of the requirement for the degree of

Master of Science in Computer Science

at

Massey University

May 1976

ABSTRACT

This thesis describes an implementation of a revised version of ALGOL W on a Burroughs B6700 computer, and was written so that excerpts can be made to produce a user manual and a system documentation manual. The first part is a brief discussion of the language as implemented and discusses the main features of the language and the differences from ALGOL 60. The remainder of the thesis gives a detailed description of the compiler.

ACKNOWLEDGEMENTS

In presenting this thesis I would like to take this opportunity to express my thanks to the following people:

To my supervisor, Lloyd Thomas, whose guidance and encouragement helped at all times.

To Neil James for helping with those niggling program errors.

To the Computer Unit operators, especially Colin Read, for performing their job so well in helping me run my program.

Finally to my family and the group, for their persistence in prodding me to finish.

Massey University Harry Meekin

May 1976

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 0

INTRODUCTION

The language ALGOL W was first described in a report drafted by Niklaus Wirth (hence the 'W'), asked for by IFIP Working Group 2.1 at its May meeting at Princeton, 1965. The report was distributed to members of the group as a "Proposal for a Report on a Successor of ALGOL 60" [1]. However, at its October meeting, 1965, at Grenoble, the group decided the report did not represent a significant advance on ALGOL 60 so it was dropped as an official Working Document of the Group. Wirth then collaborated with C.A.R. Hoare and revised and supplemented the draft. This revised report can be found in Wirth and Hoare [2].

Why then was it felt to be of sufficient interest to implement ALGOL W? ALGOL W is similar in many aspects to ALGOL 60 but some concepts have been simplified and some extensions have been introduced. The most important extension is the introduction in the language of the concept of generalised data structures. To supplement the array concept, which is virtually unchanged from ALGOL 60, there is a new data structure, the <u>record</u> ([3] and [4]). This makes ALGOL W a more powerful language than ALGOL 60 in its ability to handle data structures and therefore a more suitable language for use in the commercial field.

The language described by Wirth and Hoare, with a few changes [5], was first implemented on an IBM 360 at Stanford University [6] in 1968.

In 1971 further revisions were made to the language at Stanford [7] and an improved implementation was developed. ALGOL W has since gained only slowly in popularity being used mainly at University sites but also at a few commercial installations.

This thesis describes an implementation of ALGOL W as revised in 1971, with some further modifications to be described in subsequent chapters, on a Burroughs B6700 computer. Chapter 1 gives a brief description of the language as implemented. Subsequent chapters describe the organization in detail of the compiler.

This thesis has been written so that excerpts may be made to produce both a user manual and a system documentation manual.

Chapter 1

BRIEF DESCRIPTION OF THE LANGUAGE

As the name suggests, a large part of the language is taken directly from ALGOL 60 [8]. As ALGOL 60 has been in extensive use for some time and is therefore fairly well known, this chapter will only discuss the major changes to ALGOL 60. A full description of the Extended ALGOL W implemented by the author may be found in Appendix B.

1.1 Data Types

As is the case in most modern languages there has been an increase in the number of primitive data types from the three in ALGOL 60. The types included in the language are integer, real, long real (double precision real), complex, long complex (double precision complex), logical (equivalent to ALGOL 60's boolean), bits, string, and reference. The ALGOL 60 concept of own variables has been dropped as it doesn't add any power to the language and leads to semantic ambiguities in many cases.

The type complex is internally represented as two real words, one for the real part of the complex value, and one for the imaginary part. The type long complex is internally represented as two long real words with the same meanings as in the case of complex.

The type bits is one word containing a sequence (i.e. an ordered number of elements) of binary digits. Operations defined for bit sequences include the logical operations ¬ , ∧ (and), and ∨ (or), and those of shifting left (shl) and shifting right (shr).

To represent an ordered sequence of characters the type string has been included. When this type is declared, the maximum number of characters able to be held in the sequence must be explicitly stated, e.g.

string (10) A

declares a variable A which represents a character sequence of up to 10 characters. The string type is internally represented in an analogous way to the Burroughs Extended ALGOL EBCDIC arrays. Originally the operations for string sequences included the catenation operator cat, but with the addition of the concept of substrings the cat operation was found to be redundant and so was abandoned.

The type reference will be discussed in section 1.4.

An interesting and very useful aspect of the design of the language is that the type and length of the result of evaluating every expression or subexpression can be uniquely determined at compile-time, so no type testing, except possibly on procedure entry, is needed at run-time, thereby not wasting execution process time for non-compatible type testing.

With the increase in the number of data types there is a greater number of possible type conversions. Automatic type conversion (i.e. conversion performed by the compiler) is confined only to cases where no confusion about the required conversion is possible, i.e. from integer to long real, and real to complex, and from shorter to longer variants of the types. All other conversions must be programmed explicitly by the programmer with the use of standard functions. This is so the programmer knows exactly what type of result he is getting rather than relying on a default conversion which he may only have vague or even mistaken ideas about.

1.2 Statement Sequencing

The concept for control of statement sequencing has been simplified. The switch declaration, switch designator, designational expressions, integer labels, and label parameters have all been abolished.

The switch declaration and switch designator have been replaced by the case statement and case expression. These allow the selection and execution (or evaluation) of one of a list of statements (or expressions) due to the value of an integer expression. As the case construction is in use in a number of modern languages and is fairly well known, the reader is referred to Appendix B for more detail if required.

A goto statement can not lead from outside into any if statement, case statement, or iterative statement.

There are three types of iterative statements:

(i) for<id>:=<for list>do<statement>,

(ii) for<id>:=<int.exp.>step<int.exp.>until<int.exp.>do<statement>,

(iii) while<log.exp.>do<statement>.

These are the simple and most common cases of iterative statements and more complex cases can be easily dealt with by explicit program instructions using labels. There are a few points to notice. The <for list> and step-until parts can no longer be mixed. The "step <int.exp.>" is optional and if missing a default step of 1 is used. The <id>, called the control identifier, is implicitly declared at the start of the for statement and is undefined outside the scope of the for statement. No explicit assignments are allowed to be made to the control identifier.

1.3 Procedures and Parameters

There are a few changes towards clarification and efficiency of implementation, to the ALGOL 60 concept of procedures.

The meaning of parameters is unchanged, i.e. they can be explained in terms of the "copy rule". In addition to the "name parameter" and "value parameter" there has been added the concept of a "result parameter". This formal parameter, like the value parameter, can be thought of as a local variable. Upon the termination of the procedure the actual parameter, which must always be a variable, is assigned the value of the pseudo formal parameter.

Array parameters can only be called by name.

An actual parameter may be a statement (or expression) providing the corresponding formal is procedure (or <simple type> procedure). This statement (expression) is considered as a proper procedure body (function procedure body) without parameters. This enables a procedure (function) to be specified in the actual place it is to be used rather than in the head of an embracing block.

As mentioned in 1.2, the label parameter has been abolished. This results in no loss of power because the result of the old label parameter can be achieved by writing a goto statement in an actual parameter position as outlined in the preceding paragraph.

In this implementation the concept of virtual parameters has been included. A programmer may optionally specify parameters of formal procedures and thus enable compile-time formal procedure parameter checking. In the cases that this facility is used, no run-time parameter checking is needed on procedure entry. This concept is used extensively in ALGOL 68.

The specification of all formal parameters (except parameters of formal procedures) and the correct matching of actuals to formals has been made compulsory. The number of dimensions of an array parameter must also be shown. The specifications are included in the formal parameter list rather than in a separate specification part as in ALGOL 60. This is a much tidier form of specification as it has the attributes of the parameters in their actual position.

1.4 Data Structures

The only changes to the ALGOL 60 array concept are notational. The type of the array must always be specified and only arrays of the same type and dimension may be contained in the same array declaration.

There has been a major addition of another type of data structure, the record [4]. Like the array, records consist of one or more elements (called fields), but unlike the array the fields do not have to be of the same simple type, so that each field may occupy a different amount of storage. Because of this to select a particular field a computed ordinal number cannot be used. Each particular field is given a name (identifier) which is used in the program whenever that field is referred to. Also, unlike arrays, records are created dynamically by statements in the program rather than by declarations (see Appendix B, section 6.6).

The normal data types (see 1.1) are sufficient to represent the properties of the fields of records, but a new type, reference, is required to represent relationships between the records. For example, if a record which represents a person has a field named "father", then this is likely to be used to contain a reference to the record which represents that person's father.

References are also used to provide programmers access to records. For this purpose, variables of type reference should be declared in the head of an embracing block, e.g.

reference(<record class list>)<id list>.

The <record class list> is a list of the record classes to which the identifiers in the <id list> may refer. Thus reference variables are somewhat analogous to a restricted form of pointer variables.

A record class is a group of records which are similar, i.e. records which have the same number of fields and corresponding fields have the same names and types. Each record class is introduced in a program by a record class declaration which associates a name with the class and specifies the names and types of the fields of that class.

So that any particular field of a particular record of a record class can be referred to, a field designator must have associated with it a reference expression which is a reference to the required record (see Appendix B, section 6.8). This is checked for compatibility at compile-time.

Because a reference variable may refer to more than one record class, it is sometimes necessary to know at a particular part of a program to which class the reference is then referring. To achieve this knowledge there is a logical expression,

<reference primary>is<record class identifier>,

which is true if the reference primary is referring to a record of that record class and false otherwise.

There is also a null reference, null, which points to no record, i.e. if a reference has the value null it is undefined.

1.5 Block Expressions

There has been the introduction of a typed block which is a block that has a value (see Appendix B, section 6). The block acts like a function procedure body with no parameters and is a useful notational convenience because like the statement parameter, it allows the function to be specified actually in the place where used, rather than disjointly in the head of an embracing block.

1.6 Assert Statement

During the running of many programs it is useful to terminate any further execution of the program if some condition arises.

This is achieved by the use of the assert statement,

assert <logical expression>,

(see Appendix B, section 5.1) which will terminate the program if the <logical expression> is not true.

1.7 Input/Output

In the original implementation of the language input/output was achieved by the use of primitive standard procedures READ, READON, READCARD, WRITE, WRITEON, and IOCONTROL [7]. These procedures did not allow for programmer formatting. In 1971, the University of Manitoba developed format-directed input/output facilities for ALGOL W [9]. Upon study of these facilities it was found that they were not as versatile as those employed by Burroughs Extended ALGOL [10 and 11]. Because of this and because of the desire to make the implementation compatible with the existing system on the Burroughs B6700, a slightly simplified version of the input/output facilities used by Burroughs Extended ALGOL [12] was adopted. These facilities include the file statement, read statement, write statement, space statement, rewind statement, seek statement, close statement, and lock statement. It is planned to add the standard ALGOL W input/output procedures so that this implementation is compatible with other ALGOL W implementations.

Chapter 2

GENERAL ORGANIZATION

In writing this ALGOL W compiler it was decided to follow the original implementation and have three passes with the syntax being checked by the use of a simple precedence matrix. This decision was made because of the following reasons:

1. It was felt that to be most useful to programmers using ALGOL W, better and clearer diagnostics were needed than those able to be given by a one-pass compiler using recursive descent.

2. ALGOL W was designed as a precedence grammar (one of the effects of this being the use of a double colon in array declarations

   <u>array</u> A(0::6,0::6)).

3. It is planned at a later time to add debugging aids similar to those developed by Satterthwaite [13], and the three pass organization is best suited for this.

The compiler is written in Burroughs Extended ALGOL and each pass is represented by an internal procedure.

Pass One is the scanner of the source program. It performs the following:

(i) Reads the source program
(ii) Converts the symbols to an internal code to be passed to Pass Two with blanks and comments deleted
(iii) Converts numeric constants to internal machine form
(iv) Builds a block-structured nametable
(v) Lists the source program if a listing is required.

Pass Two does the syntactic analysis of the source program as passed to it from Pass One. In most cases the analysis is performed by means of a simple precedence analyzer thus allowing extensive error checking. However, in the case of file declarations and read/write statements the analysis is performed by recursive descent. This was done since these features are extensions to the language, and the conversion of their syntax to a simple precedence form was found to be difficult when these extensions were embedded in an already simple precedence ALGOL W grammar (see Appendix A). For these statements a simple precedence form would not add any great error checking ability anyway. Pass Two also completes the nametable and forms a binary tree representing the parts of the source program for which code needs to be generated.

Pass Three scans the tree produced by Pass Two and generates the object program in Burroughs B6700 machine code. The Pass Three procedure is only called if no non-warning errors were found in Pass One or Pass Two.

Chapter 3

PASS ONE

Pass One reads the source program in 80 character records. It assigns coordinate numbers beginning at 1 as follows. The coordinate number is incremented once for each ";" (except end-of-comment), begin or else. This number is used in error messages to pinpoint the part of the program where the error occurred and is more useful than line or card numbers. The source program is listed if desired. The basic entities of the language are recognized and replaced, minus blanks and comments, in an output string with byte (8 bit) internal codes. Constants are converted into internal machine form and a number of tables are either initialized or filled for use in Pass Two and/or Pass Three.

3.1 Internal Pass One Tables

There are separate "reserved" tables, initialized at entry to Pass One, to hold the ALGOL W symbols (alphabetically) by length. RESERVED1 contains all of the legal symbols of length 1 such as [, <, (, +, *. RESERVED2 contains all of the legal symbols of length 2 such as do, go, if, and so on until RESERVED9 which contains all of the legal symbols of length 9 such as procedure and reference. For each symbol the entry consists of the EBCDIC representation of the symbol, followed by a 1 byte output code to be passed to Pass Two, followed by an index to a case statement which performs any other action required for example the processing of declarations.

Figure 1

Reserved Tables

Table	Entry (in hexadecimal)	Symbols represented
RESERVED1	4A87004BC0014CA1204D841B	[. < (
RESERVED2	C4D66D00C7D66F02C9C6A300	do go if
...		
RESERVED9	D7D9D6C3C5C4E4D9C5A808	procedure

As the source program is scanned and a basic entity is recognized the reserved tables are scanned to see if this basic entity is a reserved entity. If found the output code is put into the output string and any action needed to be taken is selected by a case clause using the last byte of the entry in the table. An index of hexadecimal 00 means no action needs to be taken. In figure 1, the first entry in RESERVED2 is the entry for do. The first 2 bytes contain the hexadecimal representation of do. The next byte, hexadecimal 6D, is the output code for do, and the last byte of the entry, hexadecimal 00, means no further action needs to be taken.

Two other tables are partially initialized at entry to Pass One and filled during the execution of the pass. They are available to Pass Two for use in error routines. They are the identifier directory table, IDDIR, and IDLIST which indexes IDDIR. IDDIR is a character array containing all identifiers both predefined and those occurring in the source program. For example, if the only identifiers appearing were SQRT,A,SEC, then IDDIR would look like SQRTASEC and the index to the table would be 8 (i.e. the number of relevant characters). IDLIST is a list of entries, one for each identifier, that indexes IDDIR. Each entry consists of the length of the identifier minus 1 and a pointer to the first character of the identifier in IDDIR. Hence in Figure 2, the entry (2)(5) corresponds to SEC with the length part of 2 and pointer (i.e. index) value of 5.

Figure 2

Identifier Tables

IDDIR:	SQRTASEC	IDDIRINDX	= 8
IDLIST:	(3) (0)	IDLISTINDX	= 3
	(0) (4)		
	(2) (5)		

3.2 Pass One Output String Representing the Source Program

The source program minus blanks and comments, is passed to Pass Two in a coded form by way of a character array, BETWPASS. The codes correspond to the syntactic elements to be parsed in Pass Two, for example in Figure 3 <u>if</u> is represented by hexadecimal A3.

Figure 3

Pass One Output Codes (in hexadecimal)

[	87	ABS	9D	ARRAY	89	SPECCOLON(::)	85
.	C0	AND	94	BEGIN	78	SPECCOMMA	79
<	A1	DIV	92	CLOSE	B8	ASSIGNMENT(:=)	B1
(	84	END	6E	FALSE	9A	END-OF-FILE	00
+	8B	FOR	BB	PURGE	74	EXPONENTIATION(**)	96
|	9F	REM	93	SHORT	9C	COORDINATE	BF
]	86	SHL	97	SPACE	B3	NUMBER	83
*	90	SHR	98	UNTIL	BD	IDENTIFIER	6C
)	71			VALUE	A9	SIMPLE TYPE	13
;	77	CASE	A6	WHILE	BE	INITIAL ATTRIBUTES	AF
¬	95	ELSE	A5	WRITE	73	READ/WRITE STATE	B2
/	91	FILE	AE				
,	88	GOTO	6F	ASSERT	70		
-	8C	LOCK	B9	CRUNCH	76		
>	A2	LONG	9B	RECORD	AD		
:	B0	NULL	8F	RESULT	AA		
#	9E	OPEN	BA	REWIND	B4		
=	A0	READ	72				
"	8E	REEL	75	PROCEDURE	A8		

		SEEK	B5	REFERENCE	82
DO	6D	STEP	BC		
IF	A3	THEN	A4		
IS	8A	TRUE	99		
OF	A7				
OR	8D				

There are some special cases where a modification of the source program is required and these are the following:

1. The reserved words and word pairs, <u>integer</u>, <u>real</u>, <u>long</u> <u>real</u>, <u>complex</u>, <u>long</u> <u>complex</u>, <u>logical</u>, <u>bits</u>, and <u>string</u>, receive the code for SIMPLE TYPE.
2. In a reference declaration, the left parenthesis preceding the record class list specification part is omitted.
3. In a string declaration, if the length is explicitly specified, the entire length specification part is omitted.
4. A comma appearing in the identifier list of a declaration or in the record class specification part of a reference declaration, receives the code for SPECCOMMA.
5. 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 2, the identifier A has an identifier number of 1 corresponding to its index in IDLIST.
6. 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 (not split across word boundaries).
7. Each bit sequence (for example #FA12C02D31E6 (in hexadecimal)) is represented by a 1 byte code for #, followed by the 12 byte literal.

8. Each string sequence is represented by a 1 byte code for " followed by a 1 byte indication of the length of the sequence minus 1, followed by the string sequence.
9. In a file declaration, if there are any attributes explicitly declared, a code for INITIAL ATTRIBUTES is inserted after the left parenthesis and before any attributes.
10. In a read/write statement a code for READ/WRITE STATE is inserted after the reserved words read or write.
11. Each new coordinate is indicated in the output string by a 3 byte code. The first byte specifies COORDINATE and the following 2 bytes give the coordinate number.
12. The reserved word comment and all characters up to and including the next semicolon are omitted.
13. An identifier following the reserved word end is omitted.

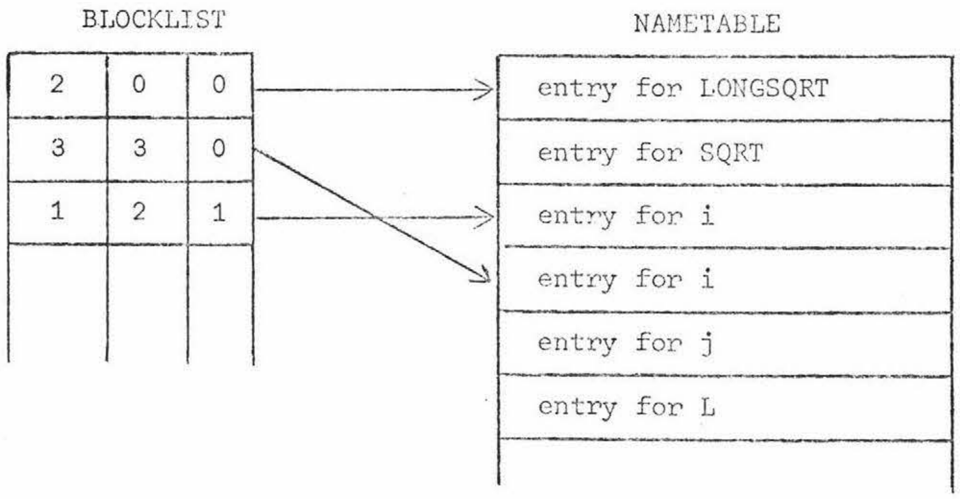
3.3 Pass One Table Output

As well as the coded string of the source program, there are three tables that Pass One partially fills to output to Pass Two. They are the NAMETABLE, BLOCKLIST, and RCCLIST.

The BLOCKLIST table has a word entry for each block in the source program in the order of block opening. Each program has a predefined outer block numbered 0, containing predefined identifiers, for example SQRT. Each full-word entry is divided into thirds (i.e. 16 bit parts). The first 16 bits contain the number of identifiers declared in that block. The second 16 bits contains a pointer (i.e. index) to the entry in NAMETABLE which contains the first identifier declared in the block. If no identifiers are declared these first two parts will contain zero. The remaining 16 bits contains the block number of the immediate surrounding block of the current block. In figure 4, the first BLOCKLIST entry points to LONGSQRT and takes in both LONGSQRT and SQRT which are both predefined. The second BLOCKLIST entry points to i, and takes in i, j and L which are declared in the outer block of the source program and the third entry points to the control variable i.

Figure 4

Example of BLOCKLIST and NAMETABLE



```
 begin
   integer i,j;
   j:=0;
   for i:=1 until 7
      do j:=j+1;
L:
 end.
```

Blocks are entered and closed by the following rules:

1. Each <u>begin</u> is the entrance to a block and the matching <u>end</u> is the close of the block.

2. Each statement in a for statement is surrounded by a block in which the control variable is implicitly declared.
3. Each procedure body is surrounded by a block in which any formal parameters are declared.

The NAMETABLE has all entries of identifiers declared in a block grouped together. Thus permanent entries are not made until the block has been closed. For example in Figure 4 the entry for the control variable is before the group of entries of the identifiers declared in the scope of the outer block.

The full layout and field contents of NAMETABLE are given in Figure 5 (see next chapter). Pass One only puts in enough information so that Pass Two can check for any semantic errors (for example type compatibility in expressions) in the source program. The information entered by Pass One consists of the following attributes appropriate to the variable

IDNO	-	The number assigned to the identifier. This number is equal to the number of the IDLIST entry. Virtual parameters have idno 0.
SIMPLETYPE		
TYPE		
TYPEINFO	-	If a procedure (not formal or virtual) then the block number of the formal parameters.
VR	-	If formal or virtual parameter 1 if value 2 if result 3 if value-result
RCCLNUM	-	If record class identifier or record field identifier then the record class number.

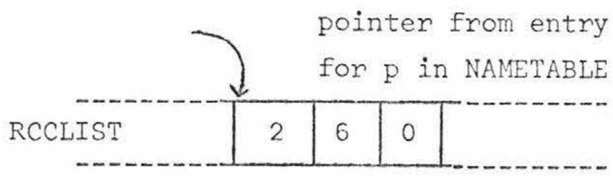
SIMTYPEINFO	-	If string then length-1. If reference then pointer to RCCLIST

All predefined identifiers, for example SQRT, are entered permanently into NAMETABLE with all fields filled, upon entering Pass One.

Each entry of RCCLIST is 1/3 word (i.e. 16 bits) and contains the IDNO of a record class (or classes) to which the reference is bound. A zero entry shows the end of a group. The NAMETABLE entry for a reference contains a pointer (i.e. index) to the first entry of RCCLIST for that reference. For example,

<u>reference</u> (node1, node2)p

would cause the entry in NAMETABLE for p to have a pointer in SIMTYPEINFO to RCCLIST as follows:



where node1 has been assigned an IDNO of 2 and node2 has been assigned an IDNO of 6.

Chapter 4

PASS TWO

Pass Two performs the following tasks:

1. A complete syntax analysis of the source program as passed to it from Pass One.
2. A thorough error analysis.
3. Completes the NAMETABLE entries.
4. Builds constant tables.
5. Converts the program to an intermediate language to be passed to Pass Three for the generation of code.

The syntax analysis is mainly done by means of a simple precedence analyzer, but in the case of file declarations and read/write statements the analysis is done by recursive descent.

The interpretation rules which are associated with the syntax rules of the grammar specify the other actions of Pass Two.

4.1 The Parsing Algorithm

The parsing algorithm used in the compiler is a bottom up simple precedence method. The ALGOL W grammar was rewritten in a simple precedence form (see Appendix A) and a separate program was developed to check the precedence relations ([14] and [15]) of the rewritten grammar and that it was, in fact, simple precedence. This program for checking for simple precedence had a large process-time but after some manipulation a precedence matrix was produced. This matrix is already initialized on entry to Pass Two.

The algorithm for using this matrix for the syntactic analysis is, with a couple of modifications, that used by Wirth in Euler [16]. When looking up to see if a string is the right part of a production, the lengths of the string and right part are checked before the matching is checked. This makes it easier to search through the production array. Also, the full precedence matrix is used as opposed to using precedence functions so that errors are detected sooner and thus providing for better error recovery. The third change is that the relations found when scanning to the right looking for ·> are stacked. This makes them easily retrieved when scanning to the left for <· rather than having to be refetched from the precedence matrix. The precedence matrix is packed four elements to a byte in order to save space, so a fetch from the matrix is slower than a 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 has to be fetched from the precedence matrix and stacked. This gives a gain in efficiency with right parts of length greater than two, but no significant gain with right parts of length one or two.

Every syntax rule has a corresponding interpretation rule which is executed when the reduction is made. This interpretation rule checks semantics, for example type compatibility in expressions. Associated with the parsing stack is a parallel value stack (see 4.4) which contains information used by the interpretation rules.

4.2 Error Recovery

There are two ways in which syntactic errors are detected when using simple precedence analysis:

1. A reducible string (i.e. one delimited by <· and ·>) is not the right part of any production.
2. The top of the parsing stack has no relation ($<\cdot, \doteq, \cdot>$) to the incoming symbol.

To recover from the first case, the statement in which the error occurred is deleted from the program. This is achieved by backing up the parsing stack until <block body>, <block hd>,

<caseseq>, or <endfile>, and the input string is advanced to end, ";", begin, or <endfile>. If end is deleted from the parsing stack, it becomes the next incoming symbol, else the next symbol in the input string is taken. Special care is taken to keep the block numbers the same as those assigned in Pass One, so if a nonterminal which affects the value of the block number is removed from the parsing stack, the block number is correspondingly adjusted.

In the second error case, a number of recovery actions are possible:

1. A symbol can be inserted.
2. The top of the parsing stack can be deleted.
3. Another symbol can replace the top of the parsing stack.
4. The incoming symbol can simply be stacked on the parsing stack (this is done if the other three can not be done).

If a symbol is to be inserted, it must have a relation to the incoming symbol, and the top of the parsing stack must have a relation to it. If the inserted symbol is ·> the incoming symbol, the input string is backed up and the inserted symbol becomes the incoming symbol, otherwise the inserted symbol is stacked on the parsing stack.

If a symbol is to replace the top of the parsing stack it must have a relation to the incoming symbol, and the next-to-top of the parsing stack must have a relation to it. If the replacing symbol is ·> the incoming symbol, then the top of the parsing stack is deleted, the input string is backed up and the replacing symbol becomes the incoming symbol, otherwise the replacing symbol just replaces the top of the parsing stack.

If the top of the parsing stack is to be deleted, then the next-to-top of the parsing stack must have one of the relations

$<\cdot$ or $\doteq$ the incoming symbol.

An inserted or replacing symbol can be the cause of other error messages especially in type compatibility, for example an undefined identifier is always assumed to be of type <u>integer</u>.

Special care has to be taken so that the same action is not attempted the next time through. For example, if the top of the parsing stack is <block body> and it has no relation to the incoming symbol, a ";" may be inserted. "<block body>;" reduces to <block body>, so if the error routine is called again before the input string has advanced, another ";" must not be inserted. This is achieved by the use of a flag which indicates the last symbol inserted.

4.3 Storage Allocation

Program segment numbers are assigned by Pass Two. Each program, block with declarations, or procedure with a body that is a block with declarations, is a separate program segment and is assigned a unique segment number. SEGNO contains the current segment number. SEGINDX contains the largest segment number currently assigned. SEGLIST is an array with entries indexed by SEGNO and SEGINDX and holds the immediate surrounding segment number.

All addresses of variables, array descriptors, files, and other data are also assigned by Pass Two and are indicated in NAMETABLE. An address consists of the hierarchy number (i.e. the lexical level) plus the address relative to the beginning of the data segment (the displacement).

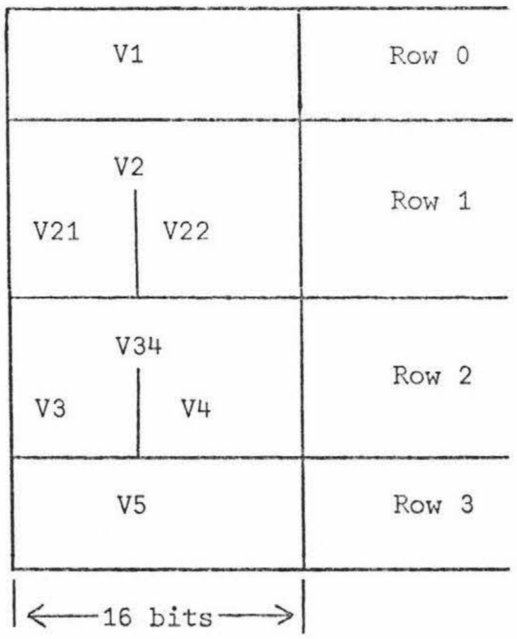
Fields of records are given addresses relative to the origin of the record. The length in words of any record in a record class is indicated in the NAMETABLE entry for that record class.

The dimension of an array is inserted in NAMETABLE when the array declaration is encountered. This information is used to compute the descriptor and to check the number of dimensions each time that array designator occurs.

Addresses are 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. Addresses are allocated in the data segment for values of the actual value and result parameters, since they are treated as local variables while control is within the procedure body.

4.4 Value Stack

The value or interpretation stack consists of 4 row by 2 byte elements, and works in parallel with the parsing stack.



The standard uses of the fields are given below, but the actual use depends on the construction being parsed.

V1	Simple type information (see SIMTYPEINFO field in NAMETABLE, Figure 5).
V21	Type
V22	Simple type
V34	0 (used in certain special cases detailed in 4.5)
V5	Output pointer

When an identifier is looked up in NAMETABLE, a pointer (i.e. index) to its entry in NAMETABLE is put in V1 and V2 is filled. When any node is put in the output array TREE (see 4.7), the tree pointer (i.e. index to the TREE array for that node) is put in V5.

4.5 Interpretation Rules

For every syntax rule of the grammar there is a corresponding interpretation rule which performs the semantic actions for that syntactic construction. These interpretation rules are contained in a procedure INTERPRET and are accessed via a case statement which is indexed by the production rule number. The interpretation rules use the value stack (see 4.4) for working storage.

The semantic actions and value stack layouts for the major syntactic constructions are:

1. Simple variable declaration
   a. Value stack layout is standard.
   b. Each identifier is found in NAMETABLE, checked for multiple declaration and allocated an address. No output is generated.
2. Array declaration
   a. Value stack layout

V1	Pointer to NAMETABLE entry of first identifier.
V2	0
V3	Number of identifiers
V4	Dimension
V5	Output pointer

b. Identifiers in the list are counted, the simple types of the bound pair expressions are checked, the bound pairs are counted, addresses for descriptors are allocated, the array dimension is inserted in NAMETABLE for all the identifiers, and output is generated.

3. Procedure declaration

   a.1 Value stack layout for procedure head

      V1 Simple type information (if typed procedure)

      V21 Type

      V22 Simple type (if typed procedure)

      V3 and V4 Used when scanning virtual parameters

      V5 Output pointer

   a.2 Value stack layout for procedure body

      V1 Simple type information of expression (if typed procedure)

      V2 0

      V34 0

      V5 Output pointer

   b. Addresses are allocated for the descriptors of the formal parameters, the simple types (for a typed procedure) are compared, output is generated.

4. Record class declaration.

   a. Value Stack layout

      V1 Pointer to NAMETABLE for current field

      V2 0

      V34 0

      V5 Pointer to NAMETABLE entry of record class identifier

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

5. File declaration
   a. Value stack layout
      V1 Pointer to NAMETABLE entry of first identifier
      V2 0
      V3 Number of identifiers
      V4 0
      V5 Output pointer
   b. The identifiers are located in NAMETABLE and checked for multiple declaration, addresses are allocated, attributes checked, and output generated.

6. Substring designator
   a. Value stack 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.

7. Array designator
   a. Value stack layout
      V1 Simple type information
      V21 Type
      V22 Simple type
      V3 Number of dimensions
      V4 Number of dummy subscripts
      V5 Output pointer
   b. The number of dimensions and simple type of subscripts are checked, output is generated.

8. Field designator
   a. Value stack layout is standard
   b. Simple type of the expression is checked, output is generated.

9. Procedure designator
   a. Value stack layout
      V1 Simple type information (if typed procedure)
      V21 Type

V22 Simple type (if typed procedure)
V34 Pointer to NAMETABLE entry for current parameter
V5 Output pointer

10. If expression
    a. Value stack layout is standard
    b. Simple types of then expression and else expression are checked for type compatibility, simple type of expression in if clause is checked, output is generated.

11. Case expression
    a. Value stack layout
        V1 Simple type information
        V21 Number of cases
        V22 Simple type
        V34 0
        V5 Output pointer
    b. The simple type of the expression in the case clause is checked, cases are counted, simple types are checked for compatibility, and output is generated.

12. argument1 {=, >=, <, <=, >, and, or, +, -, *, /, shr, shl, div, rem, **, is} argument2
    a. Value stack layout is standard
    b. Simple types of arguments are checked for type compatibility, output is generated.

13. {-, ¬ , long, short, abs} argument
    a. Value stack layout is standard
    b. Simple type of argument is checked for type compatibility, output is generated.

14. Record designator
    a. Value stack 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 0
        V4 Number of fields already parsed
        V5 Output pointer

   b. The number of fields is checked, the simple type of each field is checked for compatibility, output is generated.

15. Blockbody
   a. Value stack layout
      V1 0
      V2 0 if no declarations, #F if enclosing block of procedure body (with declarations), #FF otherwise
      V34 0
      V5 Output pointer
   b. At begin the block number and hierarchy number are stepped, V2 and displacement are set. At end the displacement and hierarchy number are restored. Output is generated.

16. Label definition
   a. Value stack layout is standard
   b. The segment number and hierarchy number are inserted in NAMETABLE, output is generated.

17. Assignment statement
   a. Value stack layout is standard
   b. Simple types are checked for type compatibility, output is generated.

18. Case statement
   a. Value stack layout is the same as for case expression
   b. Cases are counted, output is generated.

19. For statement
   a. Value stack layout is standard
   b. Simple types of expressions are checked, an address is allocated for the control identifier, output is generated.

20. While statement
   a. Value stack layout is standard
   b. The simple type of the expression in the while clause is checked, output is generated.

21. Assert statement
    a. Value stack layout is standard
    b. The simple type of the expression is checked, output is generated.

4.6 Pass Two Tables

Pass Two completes the NAMETABLE and creates a literal table. The information entered in NAMETABLE is that in Figure 5 that was not entered in Pass One. Note that the TYPE entry for a formal or virtual parameter is changed from its contents at the end of Pass One.

Figure 5

Format of NAMETABLE and Field Contents After Pass Two

idloc1		idloc2		simtypeinfo	Row 0
hierarchy	prog.seg.				
typeinfo dimen		type	simpletype	idno	Row 1
vr	rccl.number				
← 16 bits →		← 16 bits →		← 16 bits →	

Field	Kind of Entry	Contents
IDLOC1	simple variable	hierarchy number
	label	program segment number
	array	hierarchy number
	file	hierarchy number
	record class identifier	hierarchy number
	record field	hierarchy number
	control identifier	hierarchy number
	standard function	simtypeinfo of argument
	formal parameter	hierarchy number
HIERARCHY	procedure	hierarchy number
PROGSEG	procedure	program segment number
IDLOC2	simple variable	relative address
	array	relative address of descriptor
	file	relative address
	label	relative address in label table
	record class identifier	relative address
	record field	address relative to start of record
	control identifier	relative address
	procedure	relative address
	formal parameter	relative address
SIMTYPEINFO	string	length-1
	reference	pointer to RCCLIST
	record class identifier	record length
TYPEINFO	label	hierarchy number
	procedure (not formal or virtual)	block number of formal parameters
	standard function	simpletype of parameter
VR	record class identifier	number of fields
	formal procedure	number of virtual parameters
	virtual procedure	number of virtual parameters
	standard procedure	vr for parameters
	formal parameter	1 if value, 2 if result, 3 if value-result
	virtual parameter	1 if value, 2 if result, 3 if value-result

DIMEN	array	dimension
	formal procedure	1 if has virtual parameters
	virtual procedure	1 if has virtual parameters
RCCLNUM	record class identifier	record class number
	record field identifier	record class number
	standard function	1 if inline
TYPE	simple variable	0
	label	1
	array	2
	procedure	3
	record class	4
	record field	5
	control identifier	6
	standard function	7
	file	8
	standard procedure	9
	formal parameter	16 + TYPE
	virtual parameter	32 + TYPE
SIMPLETYPE	integer	1
	real	2
	long real	3
	complex	4
	long complex	5
	logical	6
	string	7
	bits	8
	reference	9
IDNO		the unique identifier number

Two tables for literals are constructed by Pass Two. The literal table (LITTABLE) contains all literals (numbers, character strings, and bit sequences). The literal pointer table (LITPOINT-TABLE) contains the simple type, the length (if a character string), and a pointer (i.e. index) to the literal table for each literal.

The tables PRODUCTIONS, PRODINDX, and MATRIX are used by the syntactic analyzer and are initialized upon entry to Pass Two.

MATRIX contains the simple precedence relations of the Extended ALGOL W (simple precedence) grammar (see Appendix A). The entries are packed four/byte.

PRODUCTIONS contain 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:

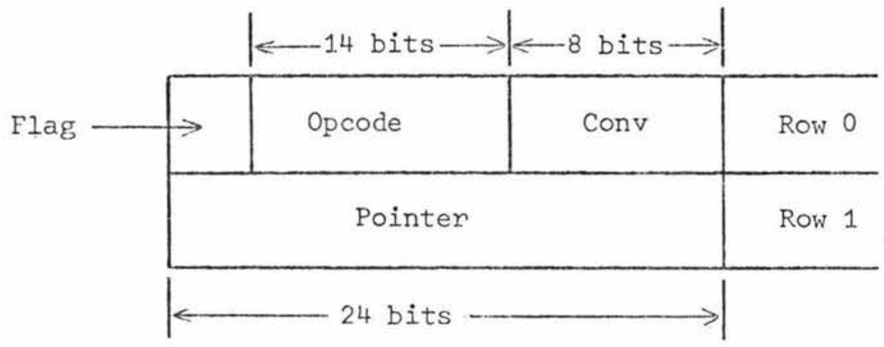
```
    production L ::= R(1) R(2) ... R(N)  0 < N < 6
    representation in PRODUCTIONS (12 bits/entry)
                  N-1
                  R(1)
                  R(2)
                  ...
                  R(N)
                  L
                  production number
```

The symbol #FFF indicates the end of a production group.

PRODINDX is an index to PRODUCTIONS. 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.

4.7 Pass Two Output

The output of Pass Two is an array called TREE which represents a linearization of a modified structural tree of the program being parsed. Each nonterminal node has either one or two subtrees. Each nonterminal binary node contains a pointer to its left subtree; its right subtree directly precedes it. In the array TREE, the subtrees of a node precede that node. Each element of TREE consists of two 3-byte entries with the format:

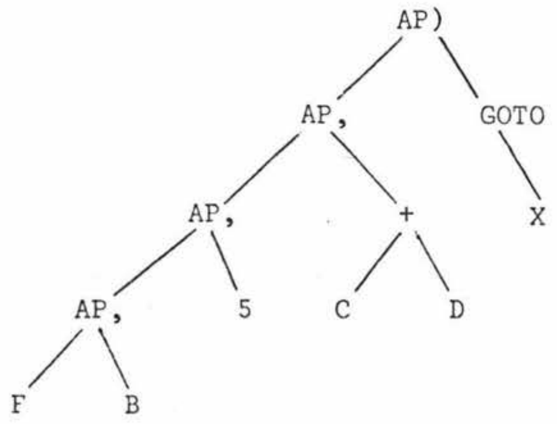


FLAG is on (i.e. 1) if the right subtree is to be compiled first and off (i.e. 0) if the left subtree is compiled first. Conversion of arithmetic type is indicated in the source program implicitly, by mixed-type expressions, or explicitly, by long or short. In either case, the simple type to which the expression is to be converted is given in CONV. For a terminal node POINTER points to NAMETABLE or the literal pointer table (LITPOINTTABLE). For a nonterminal node POINTER points to the last node of the first subtree. The first node in TREE only uses the POINTER field which points to the end of TREE.

Example from [2].

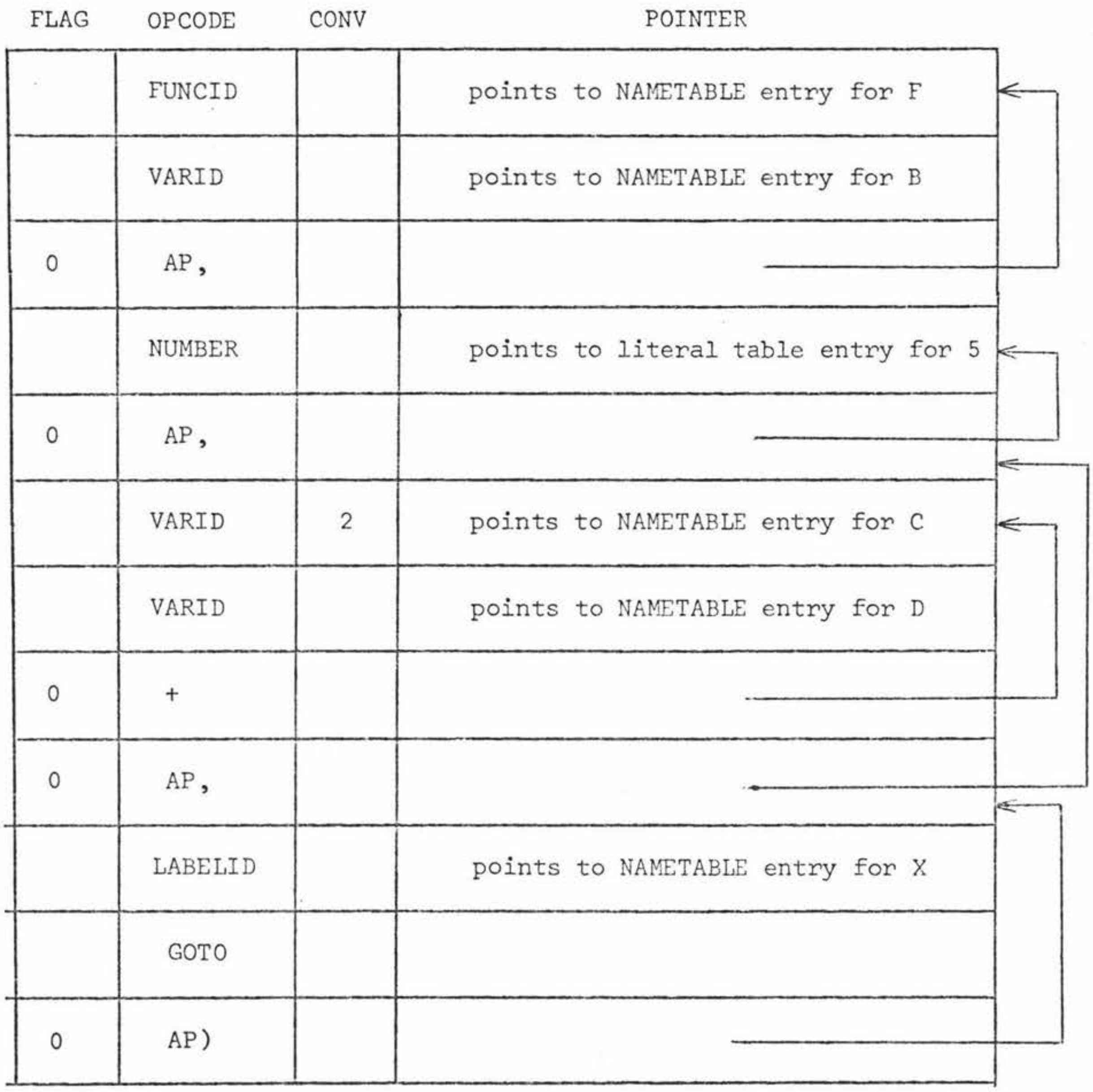
```
   program fragment:F(B,5,C+D,GOTO X)
                     -F is a procedure
                      C is integer
                      D is real
```

tree:



For meanings of nodes see Figure 6.

TREE:



FLAG	OPCODE	CONV	POINTER
	FUNCID		points to NAMETABLE entry for F
	VARID		points to NAMETABLE entry for B
0	AP,		
	NUMBER		points to literal table entry for 5
0	AP,		
	VARID	2	points to NAMETABLE entry for C
	VARID		points to NAMETABLE entry for D
0	+		
0	AP,		
	LABELID		points to NAMETABLE entry for X
	GOTO		
0	AP)		

Figure 6

Pass Two Output Vocabulary

Operator	Code	Remarks
A:Binary Operators		
+	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
STEPUNTIL	10	
DIV	11	
REM	12	
<	13	conversion bits indicate length for string comparisons
<=	14	
>	15	
>=	16	
=	17	
¬=	18	
L:=2	19	multiple assignment
A:=2	20	
S:=2	21	
R:=2	22	
(Conversion field may contain string length for string arguments)		
AP)	26	Indicates end of actual parameter list. Conversion bits indicate conversion of result of function call.
INDX	27	Indicates subscripting operation. Conversion bits can occur only with last such operator and indicate that resulting array element must be converted.

REFX	28	Indicates computation of field (first argument) of record reference (second argument)
IFEXP	29	Indicates that label should be issued for end of if expression and unconditional jump patched. Conversion bits indicate that resulting expression must be converted.
PCL	30	Indicates end of procedure declaration
SUBSTRING	31	
SHL	32	left shift
SHR	33	right shift
BB	34	Indicates end of declarations, beginning of blockbody
END	35	
|	36	
AP,	37	for actual parameters
R,	38	for record designators
AR,	39	for array declarations
AR)	40	indicates end of array declaration
R)	41	indicates end of record designator
LOGOR	42	OR of logical arguments
BITOR	43	OR of bit sequences
LOGAND	44	AND of logical arguments
BITAND	45	AND of bit sequences
ITERST	46	indicates generation of transfer to iteration test for simple for statement
ITERST2	47	indicates generation of transfer to iteration test for for statement with for list
FORLIST	48	
FORCL	49	links control assignment and step-until
ENDFORLIST	50	

UJIFEXP	51	indicates unconditional jump in if expression
UJ	52	indicates issue jump to end of case list or if statement to be patched
CL	53	indicates label should be issued for end of case statement and jump addresses patched
IFST	54	indicates label should be issued for end of if statement and addresses patched
::	55	array bounds colon
IS	56	
,	57	indicates NOOP (statement separator)
WHILEOP	58	
WHILEST	59	
IFJ	60	indicates issue jump on condition false to end of if expression or if statement
SPACE	61	
SEEK	62	
CLOSE	63	
LOCK	64	
INIT)	65	indicates end of initial attributes
INIT,	66	for file declarations
LIST,	67	I/O list separator
EDITSPECS	68	
FORMLIST	69	
FILEPT	70	
RWHEAD	71	
CARCONT	72	
B:Unary Operators		
UMINUS	73	unary minus
ABS	74	absolute value

LOG ¬	77	negation of logical value
BIT ¬	78	negation of bit sequence
GOTO	79	
:	80	label colon
STACKADDR	81	For implicit subroutine. If block expression conversion bits indicate if value of block is to be converted
ASSERT	82	
READ	83	
WRITE	84	
CLOSEST	87	
LOCKST	88	
OPEN	89	
REWINDST	90	
LISTPART	91	
COORD	93	Pointer is the coordinate number. Unary operator for BEGIN, PROCDC, ARRAYDC, ",", FILEDC nodes
CASE	94	Conv field is the simple type if expression. Pointer is the number of cases.
SEG	95	Indicates program segment. Pointer contains segment number
C:Terminal Nodes		
BEGIN	97	Conv contains block number if begins data segment
NUMBER	98	Pointer points to literal pointer table
VARID	99	
LABELID	100	
ARRAYID	101	
FUNCID	102	
RCCLID	103	
FIELDID	104	
CONID	105	
PROCDC	106	procedure declaration

FILEID	107	
ATTRIB	108	
ATTRIBMNEMON	109	
BIT	111	Pointer points to literal pointer table
STRING	112	Pointer points to literal pointer table
TRUE	113	
FALSE	114	
IF	115	
WHILE	116	
NULL	117	indicates undefined reference
NULLST	118	indicates empty statement
ARRAYDC	119	Array declaration. Pointer points to first identifier. Conv is the number of identifiers
AR*	120	indicates dummy array subscript
FILEDC	121	File declaration. Pointer points to first identifier. Conv is the number of identifiers
PURGE	122	
REEL	123	
CRUNCH	124	
SPACEV	125	used in carriage control
LINE	126	
SKIP	127	
NO	128	
STFUNCID	129	
STPROCID	130	

Chapter 5

PASS THREE

The essence of Pass Three is the algorithm for scanning the linearized tree, beginning at the root node. The flag with each binary node indicates which branch the scan should follow. The operator nodes are not otherwise examined at this stage. Pointers to the nodes are stacked in STACK as they are encountered in the scan for easy retrieval. Code generation begins with the first terminal node encountered and the tree is traversed by the generating routines.

The code produced is Burroughs B6700 machine code [17] and is put into standard B6700 code files [18]. Before a discussion on the code produced can be meaningful, some understanding of the operating system and the stack (not to be confused with the array STACK) operation is required. The first part of this chapter gives a brief insight to these features, and if a more detailed description is required the reader is referred to [17] and [19].

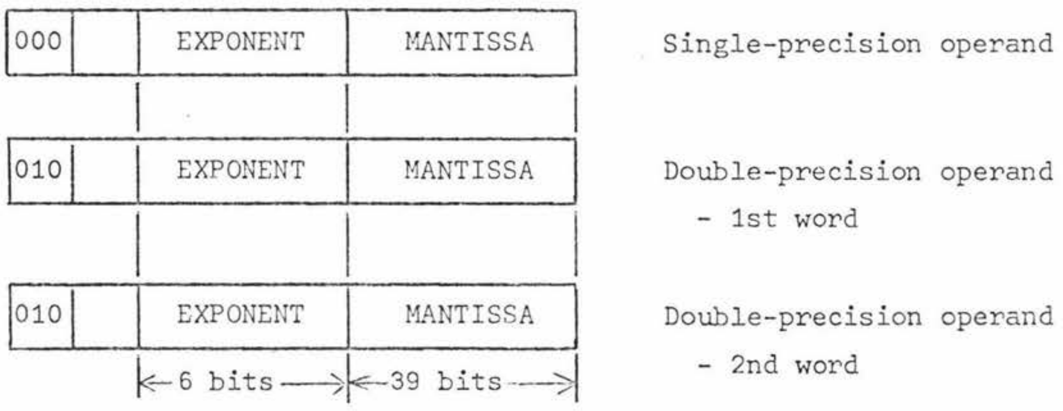
5.1 B6700 Architecture

The B6700 follows the design of the simulated machine of Randell and Russell [20], and has a typed memory (i.e. there are a few bits of each word which are used as a tag indicating what type of information the word holds) consisting of 51 bit words. The bits 50, 49, 48 are the tag bits, bit 48 is a memory protection bit which if on indicates the word can not be written into by the normal store operators, and the remaining 48 bits (47 to 0) contain the information (see Figure 7).

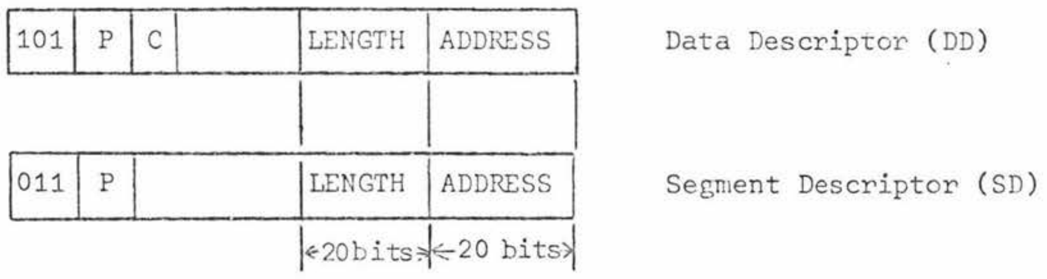
Figure 7

B6700 Word Formats With Tag Mnemonics

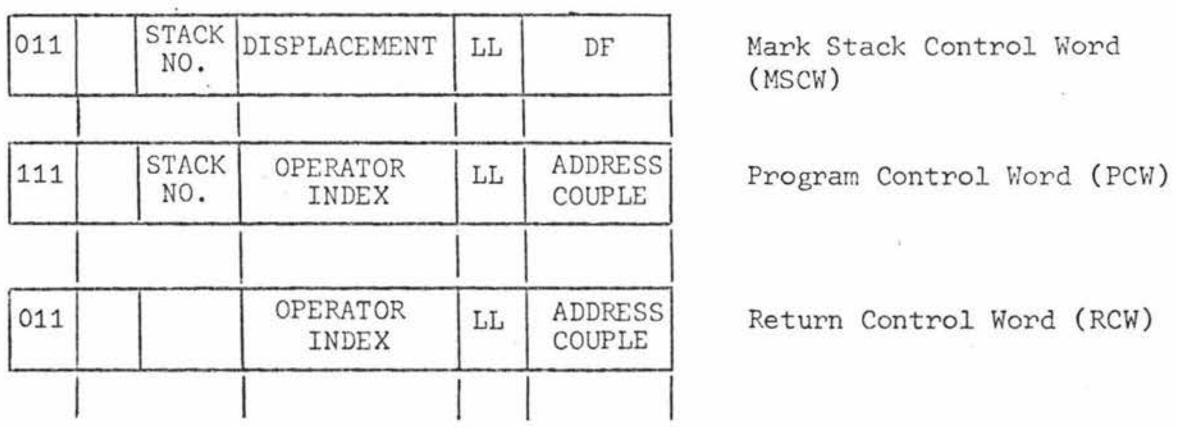
DATA WORDS

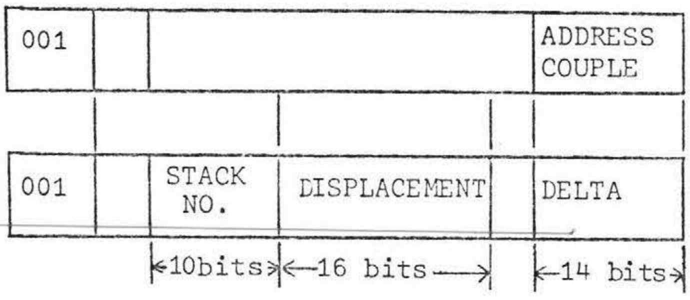


DESCRIPTOR WORDS



SPECIAL CONTROL WORDS





Indirect Reference Word (IRW)

Stuffed Indirect Reference Word (IRWS)

The memory is of the segmented virtual type. The user may use a number of linear memory segments of varying lengths so there may be more main memory required than is available. Although the user assumes all his segments are in high speed memory, it is likely some are being held on secondary storage such as a disk. The Master Control Program (MCP) brings the required segment into the main memory when it is needed. So that it can do this, each segment is described by at least one tag 5 word (descriptor) and any reference to a segment must be made through a descriptor. As seen in Figure 7, descriptors contain the main memory or disk address of the segment described plus a presence bit which if on indicates the segment is in main memory. If a segment is referred to that is not in main memory, the MCP fetches it and changes the descriptor to show that the segment is then in main memory. If the MCP removes a segment from main memory it turns off the presence bit in all descriptors of that segment, and replaces the memory address by a disk address. Program segments are read-only so they are not removed to disk, just removed from main memory.

A program segment has only one descriptor. A program (Burroughs literature calls these processes) may have more than 1 segment and all their descriptors are kept in a stack pointed to by the level 1 display register. The stack proper for a process starts at level 2. MCP programs have descriptors in the level 0 stack.

Data segments are more complicated because as they are arrays there can be many references to them held in the stack. When a data segment is removed from main memory all stacks in the system are searched for references into the data segment and all presence bits are turned off. One descriptor is chosen as the master descriptor and holds the disk address of the data segment and all the other descriptors are copies and contain the stack address of the master. If a descriptor is a copy the copy bit (C in Figure 7) is turned on. When a reference is made to an absent data segment the MCP fetches it back from disk and all other references use the copies to obtain the main memory address from the master descriptor.

Two other methods for addressing data or program code is provided. They are the Indirect Reference Word (IRW) and the Stuffed Indirect Reference Word (IRWS). These address data located within the process's stack and their address fields both hold relative addresses. The IRW addresses information which is global or local to the particular active procedure. The IRWS is used for addressing across stacks and for handling parameters where the actual parameters are not necessarily within the addressing environment of the procedure to which they are passed and can not be accessed by an IRW. The IRW has in its Address Couple field a Display Register number and a Displacement. The IRWS holds three bits of information: (a) a Stack Number, (b) the start of the addressing space of the process within that stack, and (c) the displacement of the information within that addressing space. An IRW can be changed to an IRWS by the use of the operator stuff environment-STFF.

5.2 Program Structure in Memory

A program in memory occupies separately allocated areas, i.e. each part of the program can be anywhere in memory with the actual address determined by the MCP.

The separately allocated areas of a program are (see Figure 8):

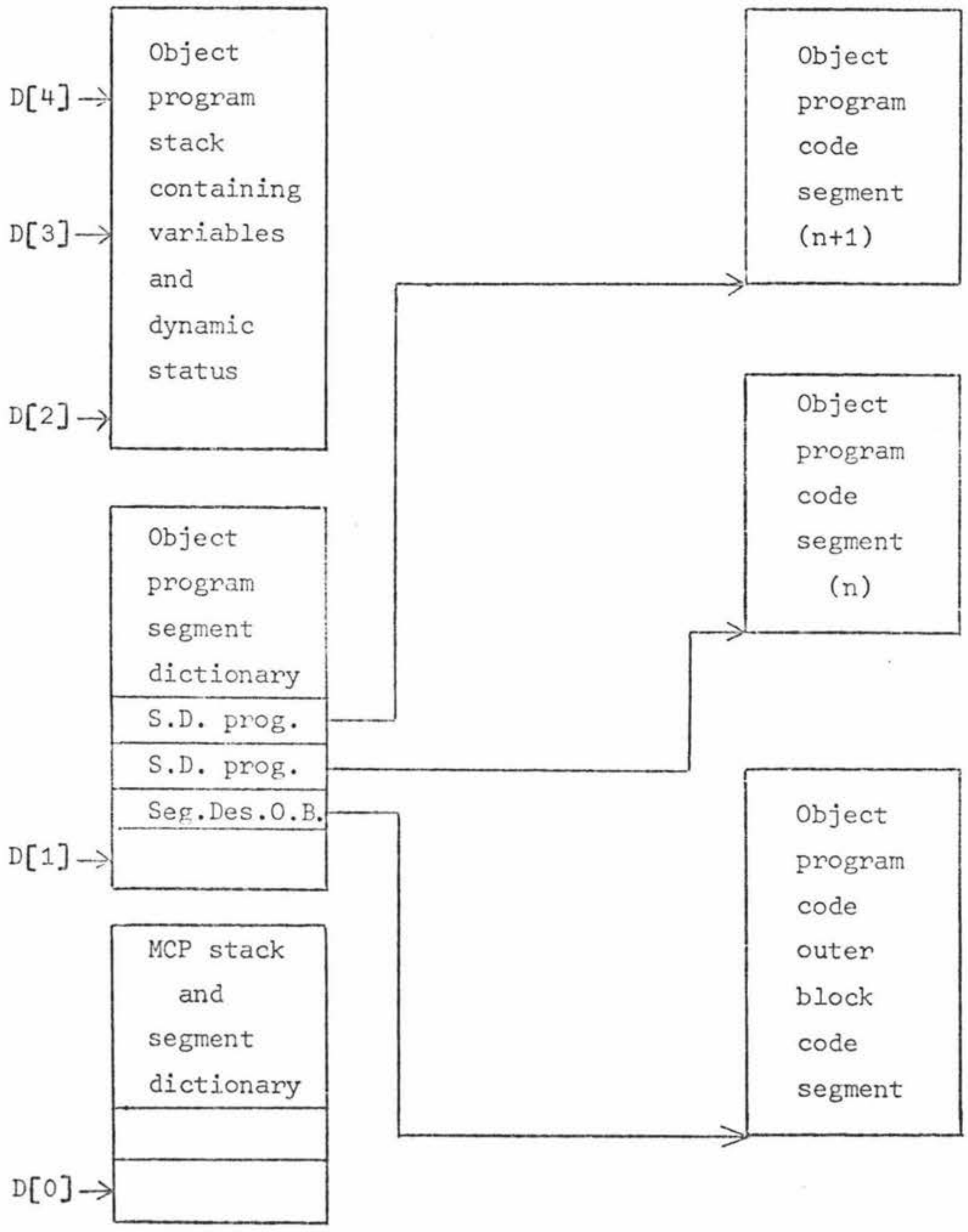
1. Program Segments. These hold a sequence of instructions (syllables) which the processor executes. The program segments hold no data and are never modified.

2. Segment Dictionary. This is a table containing the descriptors of the program segments.

3. Stack Area. This is the pushdown stack storage, which contains the variables associated with a program and the control words which indicate the dynamic status of the program as it is being executed.

Figure 8

Object Program in Memory

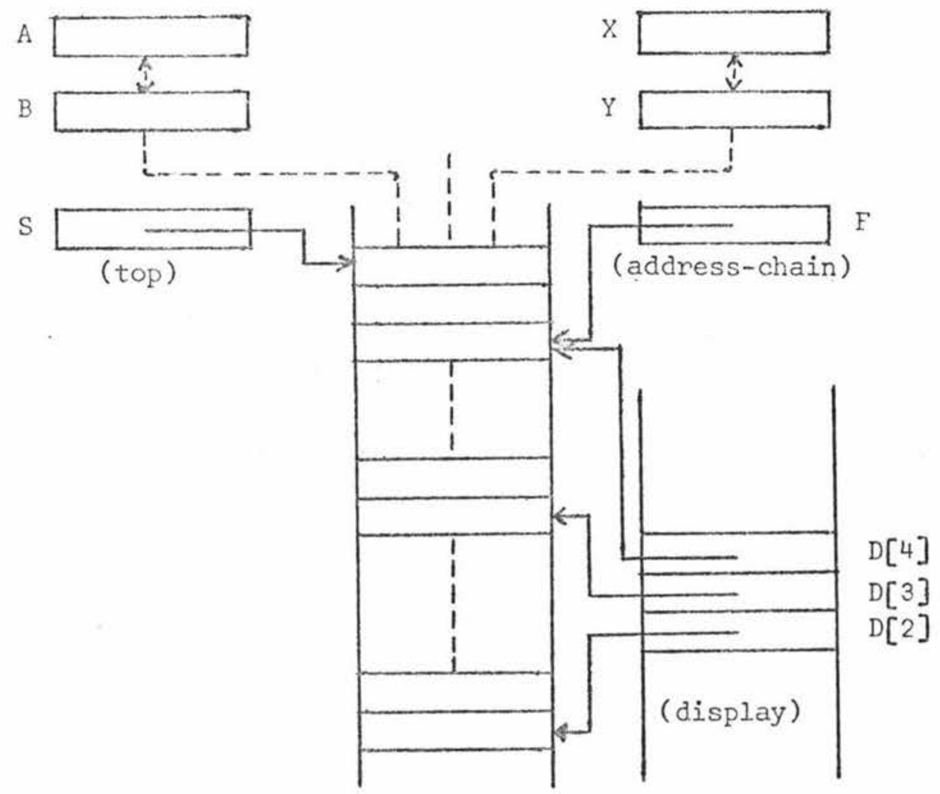

D[4]
D[3]
D[2]
D[1]
D[0]
Object program stack containing variables and dynamic status
Object program segment dictionary
S.D. prog.
S.D. prog.
Seg.Des.O.B.
MCP stack and segment dictionary
Object program code segment (n+1)
Object program code segment (n)
Object program code outer block code segment

5.3 Stack Operation

The stack arrangement (see Figure 9) has two top-of-stack registers (A and B) with associated validity bits. With each top-of-stack register there is a companion register (X and Y) which is used to hold the second half of a double-precision operand. When held in the memory stack a double-precision operand is held in two adjacent stack words. For simplicity in this discussion it is assumed all operands are single-precision. The necessary changes for double-precision operands will be fairly obvious, for example when an operand is moved from the stack into the top-of-stack register the tag bits are checked and if a double-precision operand then two stack words are moved into A and X which are then concatenated. The stack top is pointed to by the S register and the address-chain is given by the F register. The machine also has a check to see that the stack bounds are not violated.

Figure 9

Stack Arrangement



The stack operates as a last in, first out storage area. An operand is stored into register A with consequent push-downs into register B and into the memory location pointed at by register S. Extraction of data is from register A with consequent pop-ups from B and the location pointed at by S. The contents of S are incremented by 1 on a push-down and decremented on a pop-up. These actions are performed automatically by the processor to the requirements of the operator currently being executed.

5.4 Example of Simple Stack Operation

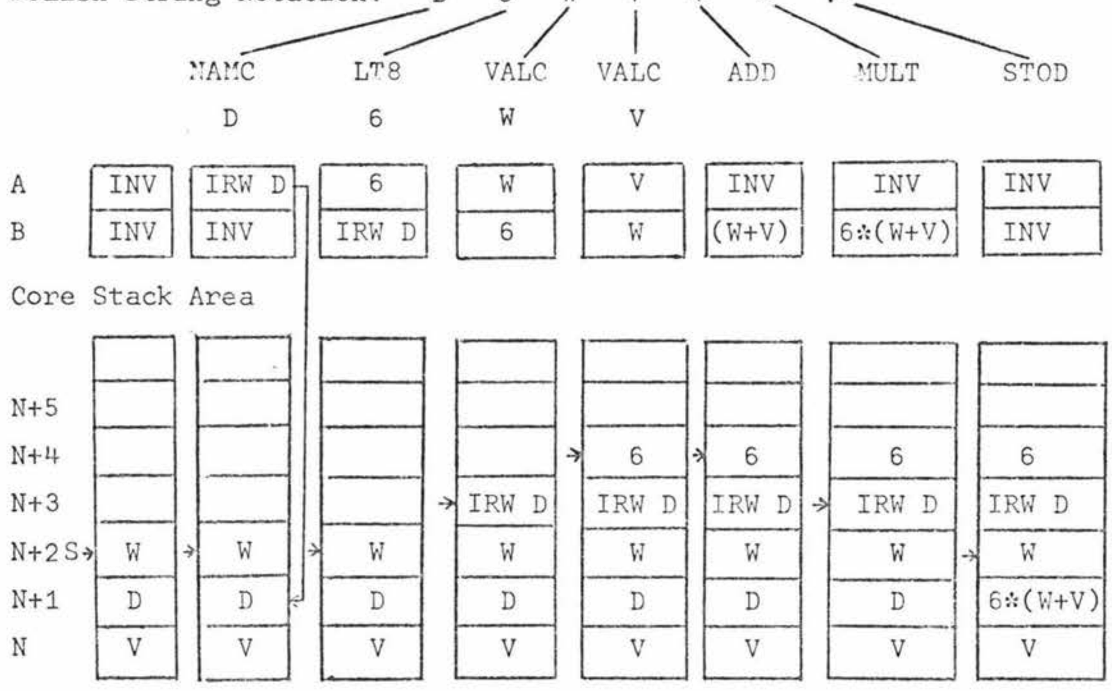
In the program segments the instructions are kept in the order of executing the source program in reverse Polish order. In this section, a simple example of the stack operation when executing the statement D:=6*(W+V) will be discussed but the explanation of the syllable types will be left to later sections.

Figure 10

Stack Operation

ALGOL W Statement: D:=6*(W+V)

Polish String Notation: D 6 W V + * :=



Syllable Types:		
	NAMC	Name Call
	LT8	Literal (8 bit)
	VALC	Value Call
	STOD	Store Destructive

When D:=6*(W+V) is changed to Polish notation the result is D6WV+*:=. Each element causes a syllable type to be placed in the machine language program during Pass Three (see Figure 10). D is to receive a value so the address of D must be put in the stack before the store command. This is done by a name call syllable (NAMC) which puts an IRW in the stack. The IRW contains the address of the stack location of the variable D. The value 6 is then put in the stack by using an eight-bit literal syllable

(LT8). Since W and V are to be added, the variables are put in the stack by Value Call syllables. The ADD operator adds the two top operands and places the sum in the top of the stack (in this case register B). The multiply operator (MULT) then multiplies the two top stack operands and places the result in the top of the stack. The store syllable (STOD) examines the two top of stack operands looking for an IRW or Data Descriptor. In this example it finds an IRW which addresses the location where the computed result is to be stored and stores it.

5.5 Syllable Format and Types

A machine language program is a string of syllables which are usually executed sequentially. Each word in the memory contains six 8-bit syllables with the first labelled syllable 0 and is contained by bits 47 to 40 inclusive.

There are three types of syllables (see Figure 11):
(a) Name Call, (b) Value Call, and (c) operators. The two high-order bits (bits 7 and 6) determine which type a syllable is.

Figure 11

Syllable Table

(Bits 7 and 6) Identification	Syllable Type	No. of syllables	Function
00	Value Call	2	Brings an operand into the stack
01	Name Call	2	Brings an IRW into the stack
1X	Operators	1 to 7	Performs the specified operation

Name Call builds an Indirect Reference Word in the stack. Stack adjustment takes place so that the A register is empty. The six low-order bits of the first syllable are concatenated with the eight bits of the following syllable to form a 14-bit address-couple. The address-couple is placed right-justified in the A register, with the remainder of the A register filled with zeros. The TAG field (bits 50, 49, 48) of the A register is set to 001 and the register is marked full.

Value Call loads into the top of the stack the operand referenced by the address-couple. The operator is formed in the same way as the Name Call operator. If the referenced memory location is an Indirect Reference Word or a Data Descriptor, memory accesses are made until the operand is found. The operand is then placed in the top-of-stack registers. The operand may be either single- or double-precision, causing either one or two words to be loaded into the stack.

Operators vary from one to seven syllables long. The first syllable determines the number of following syllables which, with the first syllable, forms the operator. Operators work on data as either full words (48 data bits plus tag bits), or as strings of data characters. Word operators work with the operands in the top of the stack. String operators are used for transferring, comparing, or translating strings of characters. There is also a set of micro-operators providing a means of formatting data for input/output.

5.6 Addressing

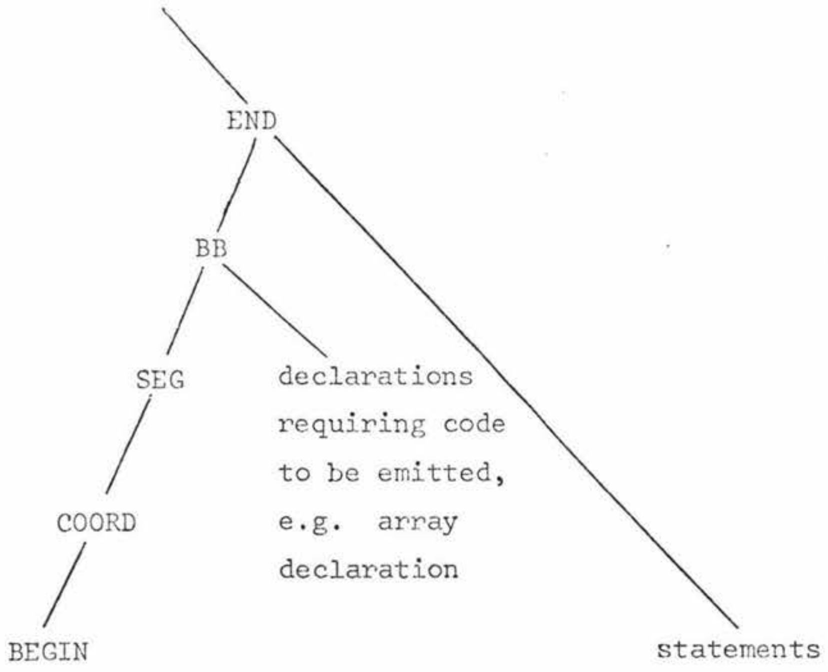
An address-couple consists of two parts: (a) the addressing level (LL) of the variable, and (b) an index value (delta) used to locate the variable within its addressing level.

The B6700 processor contains an array of display registers

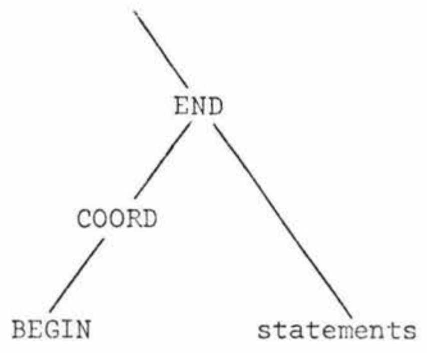
(D0 to D31) and these registers address the base of each addressing level segment. The local variables of all procedures are addressed relative to the D registers (thus delta is a relative displacement value). The address-couple is converted to an absolute memory address when a variable is referenced. The addressing level field of the address-couple selects the D register. The index value field of the address-couple is then added to the contents of the specified D register to get the absolute memory address.

5.7 Block and Procedure Entry

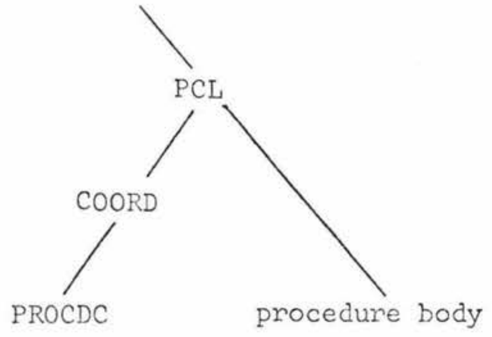
The tree output of Pass Two for a block with declarations is


END
BB
SEG
declarations requiring code to be emitted, e.g. array declaration
COORD
BEGIN
statements

If there are only declarations which do not need code emitted the declarations branch of node BB is node NULLST. Blocks without declarations have the following tree:

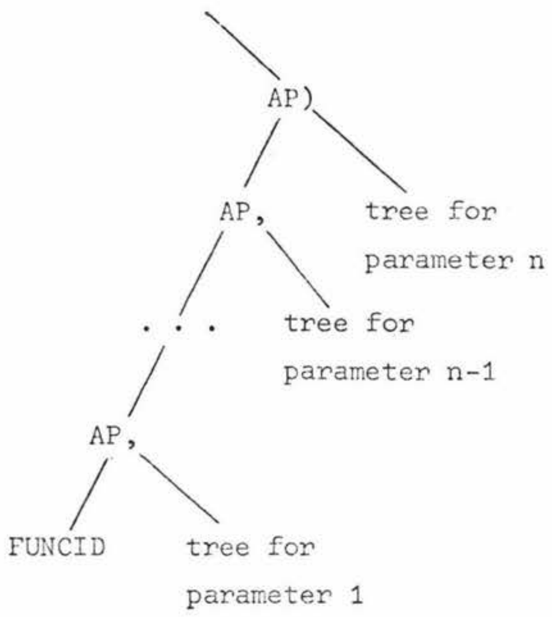


and the tree for procedure or function declaration is:

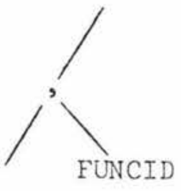


The discussion on block and procedure entry can be combined because on the B6700 the mechanism is the same in both cases. A block is treated as a procedure which is called where it appears and a procedure is always considered to contain a block. This section will refer only to procedure but remember this is synonymous with block. In ALGOL a procedure as a value and a procedure as a thunk [21] can not occur in the same context, therefore the B6700 has only one type of program address word called a PCW-Program Control Word (see Figure 7). A PCW is created by the instruction MPCW which must be followed in the instruction stream by a 48 bit literal which has the stack number and tag inserted to make the PCW which is pushed on the stack.

The tree output of Pass Two for a call on a procedure or function is:



The tree for a proper procedure without parameters is:



When a procedure is called the operator MKST (Mark Stack) pushes a skeleton MSCW (see Figure 7) onto the stack. The MSCW contains the DF field, i.e. the environment pointer F. Then an IRW or IRWS pointing to the PCW of the procedure is pushed onto the stack followed by any actual parameters.

The Enter operator (ENTR) is then pushed onto the stack. The following occurs due to ENTR:

1. The F register is made to point to the new MSCW.
2. The lexical level at which the procedure's PCW appears is found by:
   (a) If the PCW is referenced by an IRW it is obtained directly from the address-couple of that IRW.

(b) If the PCW is referenced by an IRWS it is obtained from the LL field of the MSCW pointed at by the Delta field of the IRWS.

Note that a procedure declared at level n must run at level n+1. Display Register D_{n+1} is set to point to the new MSCW, i.e. is given the same value as the F register. The number n+1 is inserted in the LL field of the MSCW.

3. The Stack Number and Displacement fields of the new MSCW are set to point to the MSCW pointed at by D_n. Hence there is a static link (Burroughs calls this a Displacement link) which expresses the lexical structure of a program.
4. If necessary, the static link in the MSCW pointed at by D_n is examined and Display Register D_{n-1} is reset. The static link is followed and all required registers are reset. (Note D_o is never reset).
5. The new IRW or IRWS is changed to a RCW (see Figure 7). Note that the RCW is similar to the PCW except it has a tag of 3. The RCW references the program code of the calling procedure one operator past the point of call via a Segment Descriptor. The LL field of the RCW contains the lexical level of the calling procedure.

The called procedure is now active.

The calling sequence for a procedure such as P(1,1) is:

```
MKST
NAMC    to P's PCW
ONE     ⎧ initialize
ONE     ⎩ parameters
ENTR
```

Standard functions (called intrinsics by Burroughs) are treated as if declared within procedures that execute at display level 0. Hence they always execute at display level 1.

5.8 Block and Procedure Exit

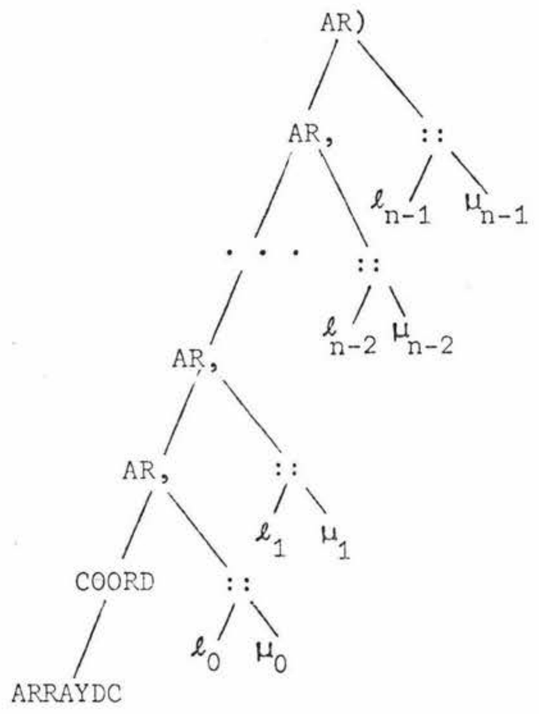
There are two instructions for returning from procedures (blocks), EXIT and RETN. They both operate in the same way except that RETN leaves the top of the stack as a value and EXIT does not. Also, if the value bit of the MSCW pointed to by F is 1, RETN operates a VALC instruction sequence because the value bit is turned on when a VALC causes a thunk.

Each MSCW is linked to the prior MSCW through the DF field so that the point in the stack where the prior procedure began can be found. When a procedure is exited, its part of the stack is discarded. This is done by the S register being set to address the memory location preceding the last MSCW. This topmost MSCW is deleted from the stack history list by changing the F register to point to the prior MSCW.

Finally, the code segment and the next operator for the procedure exited to are accessed via the RCW. Operation resumes at the point following the procedure call.

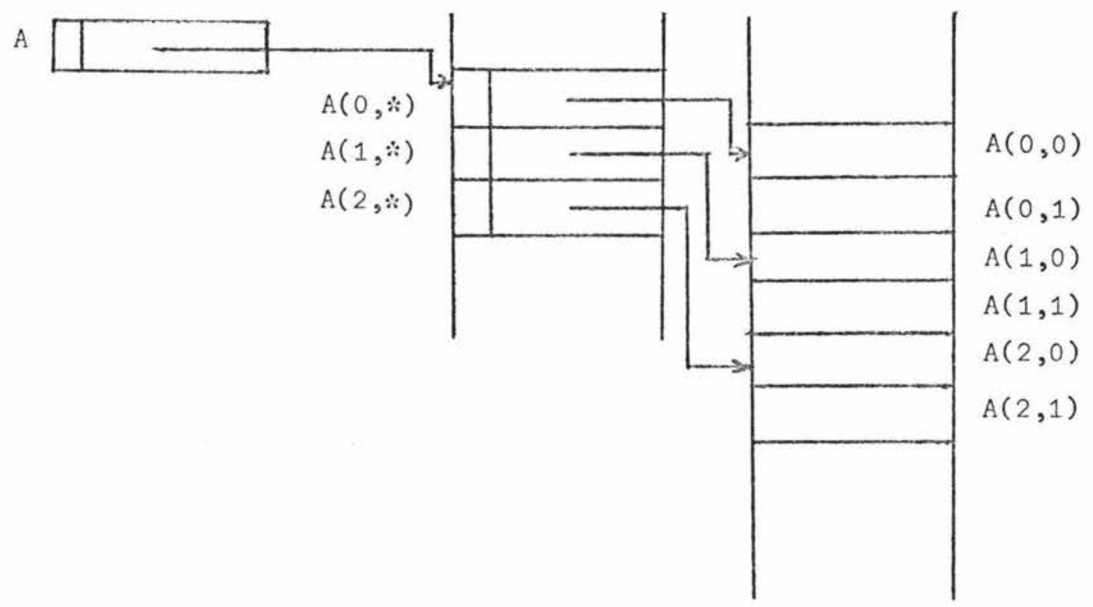
5.9 Array Declaration

The tree format for the array declaration <simple type> array $X_1,X_2,\ldots,X_m(\ell_0::\mu_0,\ell_1::\mu_1,\ldots,\ell_{n-1}::\mu_{n-1})$ is:



A B6700 array is a segment and therefore has a segment descriptor. This data descriptor (DD) contains the base address, maximum index (from zero), size of elements (double-precision, single-precision, 8 bit characters), etc. When the descriptor is indexed, the indexing integer is checked against the size and if allowable replaces the size field, a bit being set in the descriptor indicating that it has been indexed. An indexed data descriptor may be used in most places where an address is required by other instructions.

Two dimensional arrays may be handled by defining arrays of descriptors, for example A(0::2,0::1) is set up as:

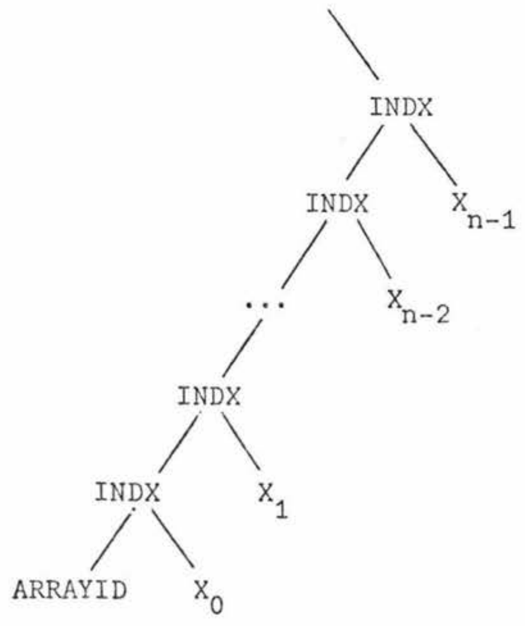


5.10 Subscripted Variables

The tree format of a subscripted variable from an array A of n dimensions:

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

where X_i is an integer expression is:



The B6700 indexing instruction is INDX (not to be confused with tree node INDX). It handles an operand and an array descriptor in any order on top of the stack. It will fetch a descriptor pointed to by an IRW. INDX interprets the length field of a DD as being in units of the correct size and the indexed DD contains the pointer to the appropriate character or word, for example if the DD points to a double-precision array the indexing operand is doubled.

If location (4,6) contains the unindexed descriptor of array A and (5,3) is i, then

```
          A(i) is  NAMC  (4,6)
                   VALC  (5,3)  i.e.  NAMC(5,3),LOAD
                   INDX

               or

                   VALC  (5,3)
                   NAMC  (4,6)
                   INDX
```

If j is (5,4), then

```
        A(i,j) is  NAMC  (4,6)
                   VALC  (5,3)
                   INDX
                   LOAD
                   VALC  (5,4)
                   INDX
```

The pair INDX, LOAD can be replaced by NXLN if another descriptor is expected (as above), or NXLV if an operand is required.

Because of the virtual memory it is advisable to keep segments small. Therefore usually large arrays are segmented and treated as two dimensional although the programmer sees it as linear. In this case the main descriptor of the array has a special bit set.

On indexing, the index value is divided by a constant depending on the data item's length (double-precision 128, single 256, 8 bit characters 1536) to give a row number and index within the row. The array is treated as 2 dimensional and indexed twice, for example a single-precision array descriptor indexed with 1040 would actually fetch word 15 of row 4 (counting from zero).

5.11 Passing Sub-Arrays as Parameters

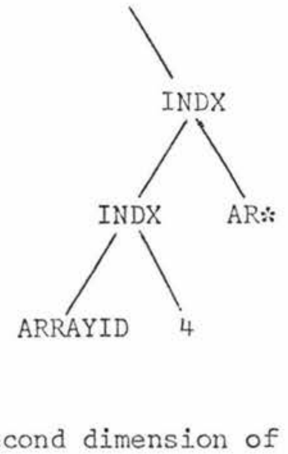
The user may pass any generalized row or column, i.e. any subarray of dimension 1,2,..,n-1 of an n-dimensional array, as a parameter to a procedure. Since all array parameters are passed by name, all that is needed is to copy certain parts or all of the array descriptor.

According to the syntax of subarrays, an asterisk (*) is put in the positions of the actual subarray parameter to indicate which dimensions are to be included in the formal array.

In the positions in which * occurs in subarrays 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 second dimension of the two dimensional array A is unspecified and the 4th row corresponds to the one dimensional formal array.

5.12 Operands

Arithmetic operands are regarded as floating point numbers - integers have zero exponents. There are two instructions, NTGR and NTIA, for rounding or truncating the top of the stack to an integer. There are a few arithmetic operators (for example ADD) which gives a double-precision result if one of the top two stack words is double-precision.

For logical values, an operand with a 1 in bit zero is regarded as true, an operand with a 0 in bit zero is regarded as false. Relational operators GRTR, GREQ, EQUL, LSEQ, LESS, NEQL operate on the top two words in the stack and produce a logical value result. Logical operators LAND, LOR, LNOT operate bitwise on the top of the stack words extending a single-precision word with zeros to double-precision if one operand is double-precision.

Constants are put in the stack by one of 5 instructions:

LT8	"Literal Calls" followed by an 8, 16, or
LT16	48 bit constant which is put in the
LT48	stack as a 48 bit operand
ZERO	puts on the stack an integer 0 constant
ONE	puts on the stack an integer 1 constant

String operands are treated as arrays.

The NAMC (Name Call) instruction creates an IRW. The contents of any location addressed by the IRW on the top of the stack may be put in to the top of stack by using the LOAD instruction, i.e. a value is put on the top of the stack by using

```
NAMC   (lev,disp)
LOAD
```

The main operand fetching instruction is VALC (lev,disp). It follows indirect references, enters thunk procedures, etc. It also indexes a non indexed array descriptor with the top of stack operand. Thus the A(i,j) of section 5.10 also translates as:

```
VALC  (5,4)  puts j in stack
VALC  (5,3)  puts i in stack
VALC  (4,6)  fetches A and indexes it twice
```

The EVAL instruction examines the address on top of the stack and follows any references and performs thunks until it gets an address of an operand or an unindexed array descriptor.

The store instructions are STON and STOD (store "nondestructive" and store "destructive"). They do an EVAL on the address on the top of the stack. Actually the top two elements of the stack may be an operand and an address in any order. The store instructions put the address on top, EVAL it, and then performs the store.

5.13 Branching

The "instruction counter" of the B6700 is a collection of registers which keep the base address (and limit) of the program segment being executed and of the syllable address within that segment. The simple branch instructions - branch on false, BRFL, branch on true, BRTR, and branch unconditionally, BRUN - are followed by two syllables giving the destination address within the current segment. If a destination in another segment is required, a PCW is put on top of the stack and the dynamic branch instructions DBFL, DBTR and DBUN are used.

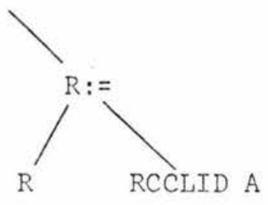
5.14 Record and Field Designators

ALGOL W permits records to be created in two ways. The name of the record class may stand alone or the name of the record class may be followed by a list of initial values of the fields.

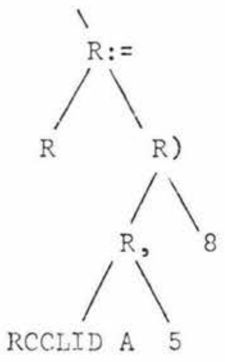
Both creations are reference expressions.

<u>Example</u>

```
RECORD A (INTEGER I,J);
REFERENCE(A)R;
R:=A;
```



```
R:=A(5,8);
```

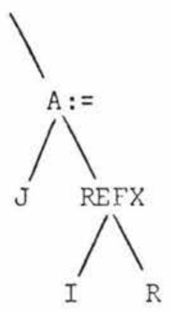


In this implementation, each record class is held in a separate array. When a new record of a record class is created, the length of the array is increased by the length of the new record. This is achieved in an analogous way to the resize statement in Burroughs Extended ALGOL [12] and entails only changing the length field in the array's descriptor. Thus no memory space is allocated before it is required. No garbage collection has been implemented as it is thought that with this method of allocating storage for record classes in arrays, the B6700 memory management will remove arrays that are not referenced. Also when the descriptor disappears on block exit the array is deallocated by the system.

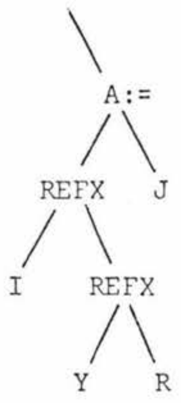
Since a reference points to a record with fields of any of the nine simple types, field designators of the form F(R), where F is a field name and R a reference expression, select the desired field of the simple type of F. The loading of the reference value is analogous to getting an address from a subscript 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.

<u>Example</u>

```
RECORD A(REFERENCE(A)X,Y;INTEGER I);
INTEGER J;
REFERENCE(A)R;
J:=I(R);
```

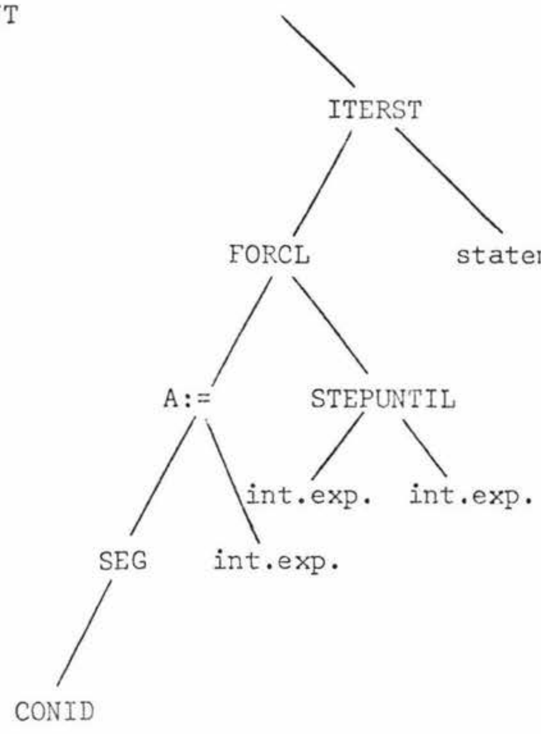


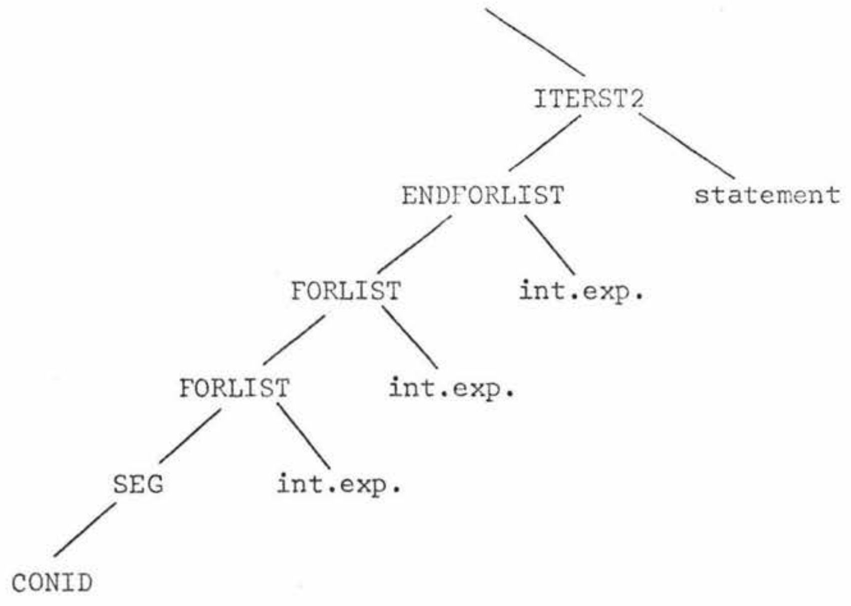
```
I(Y(R)):=J;
```



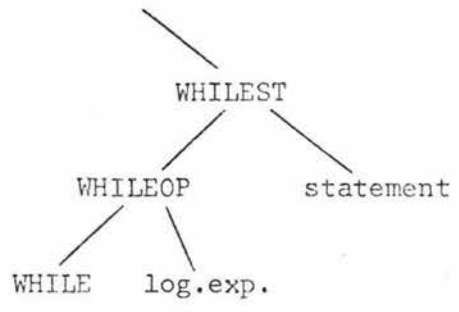
5.15 Further Examples of Pass Two Tree Output as Received by Pass Three

FOR STATEMENT

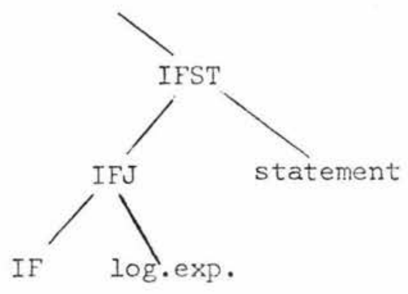


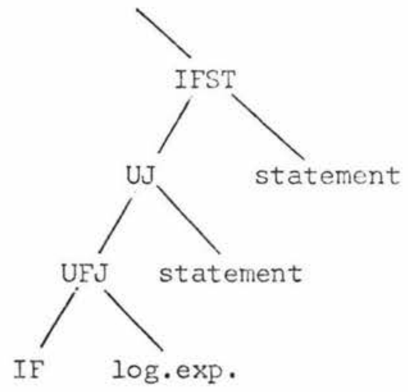


WHILE STATEMENT

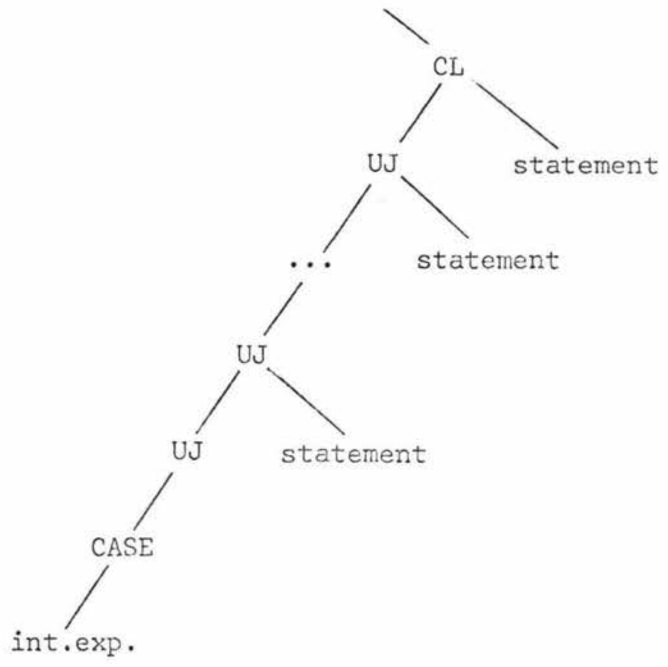


IF STATEMENT

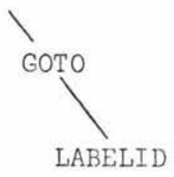




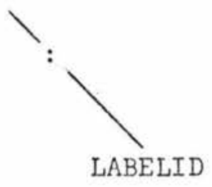
CASE STATEMENT



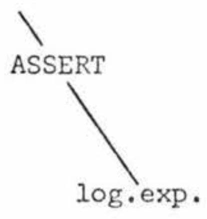
GOTO STATEMENT



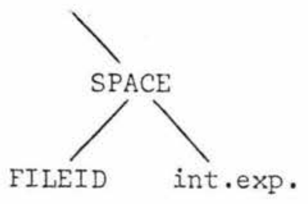
LABEL DECLARATION



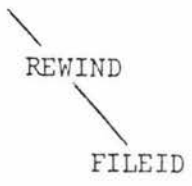
ASSERT STATEMENT



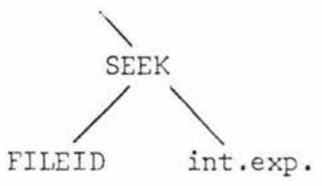
SPACE STATEMENT



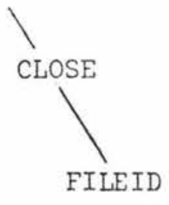
REWIND STATEMENT

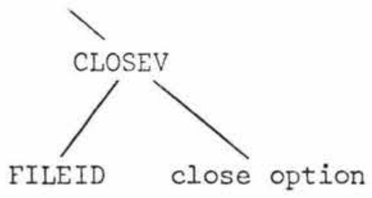


SEEK STATEMENT

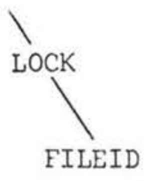


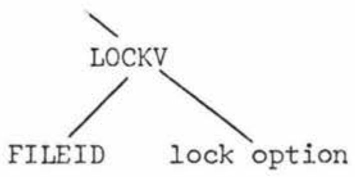
CLOSE STATEMENT



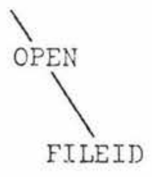


LOCK STATEMENT

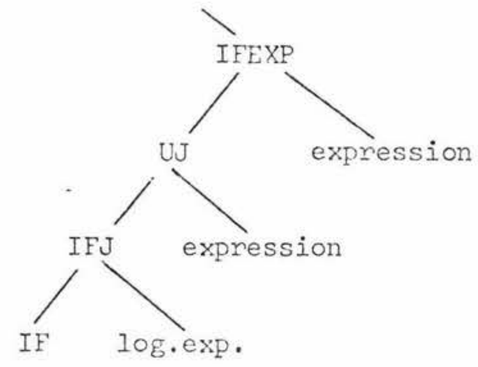




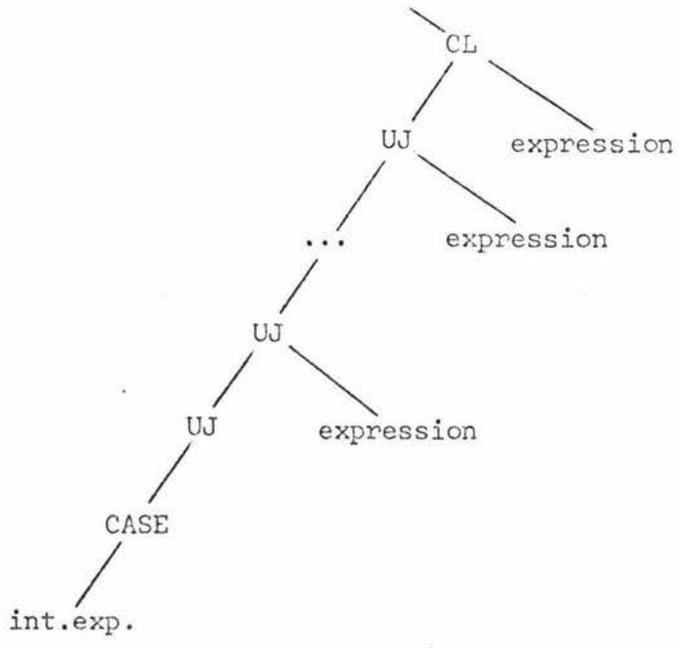
OPEN STATEMENT



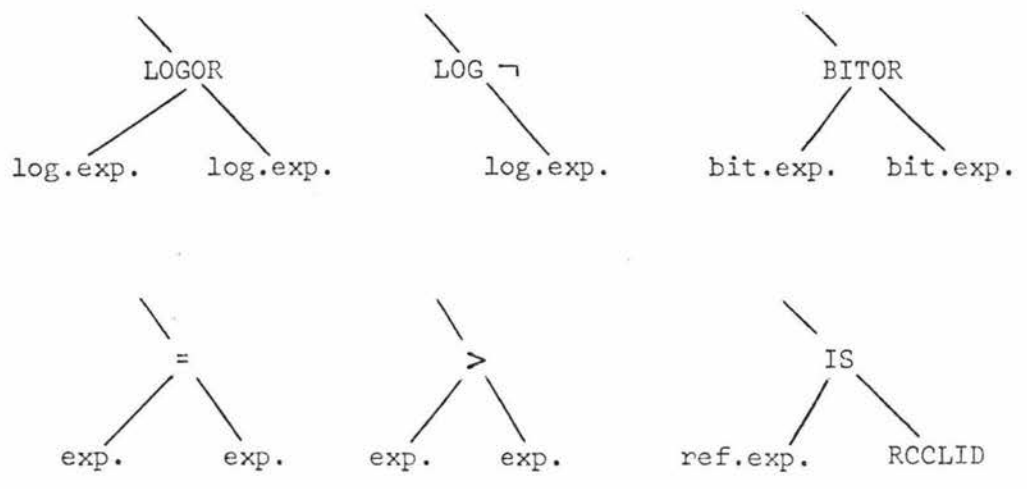
IF EXPRESSION



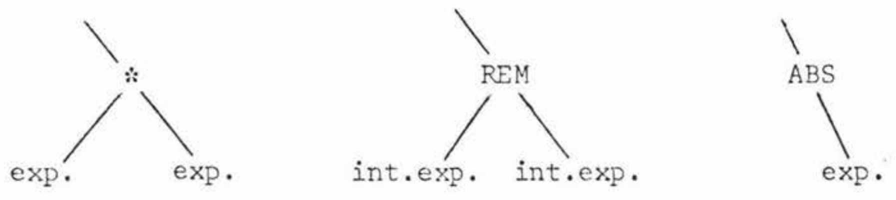
CASE EXPRESSION



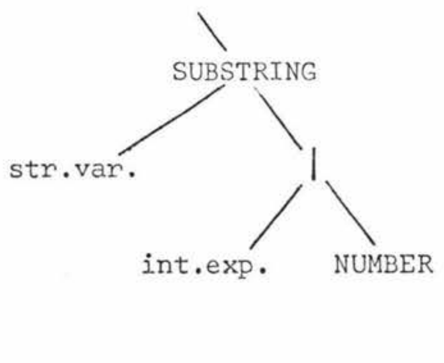
LOGICAL EXPRESSIONS



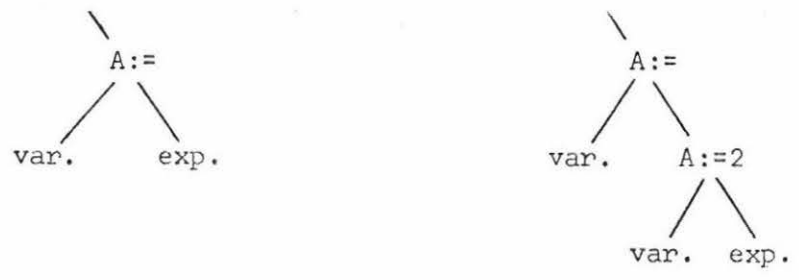
ARITHMETIC EXPRESSIONS



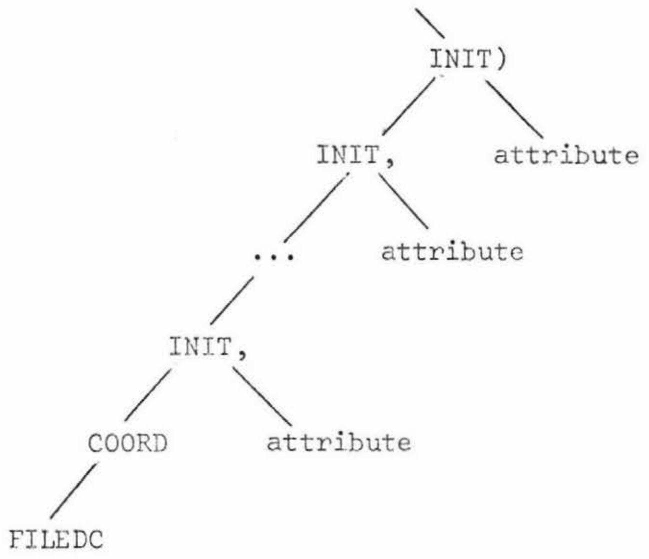
SUBSTRING



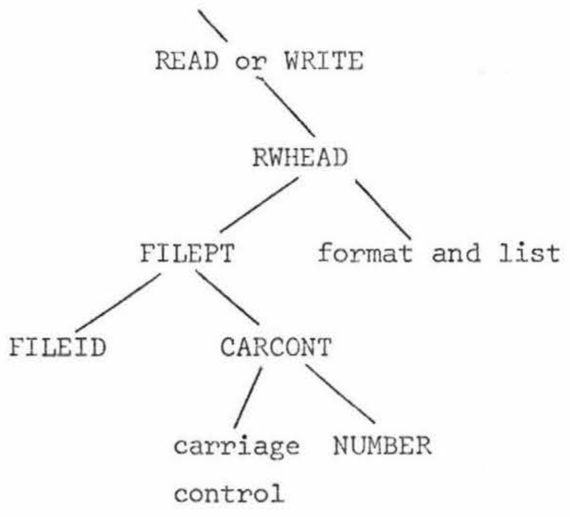
ASSIGNMENT STATEMENT



FILE DECLARATION



READ/WRITE STATEMENT



Chapter 6

SUMMARY

At the time of writing this thesis Pass One had been extensively checked and was working very well although with more use by programmers errors may be found. Pass Two was working well although had not been extensively checked. Pass Three was still causing some trouble but was expected to be in good working order in a very short time.

There was times when it was felt that it might have been far easier to write a recursive descent compiler but with the inclusion of debugging features along the lines of those devised by Satterthwaite [13], the three pass compiler will have proved its worth.

Apart from the inclusion of debugging features it is hoped to have the standard ALGOL W I/O procedures included so making it compatible with other ALGOL W implementations.

REFERENCES

1. WIRTH, Niklaus
   Proposal for a Report on a Successor of ALGOL 60. MR75, Mathematical Centre, Amsterdam, August 1965.

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

3. ROSS, D.T.
   A Generalised Technique for Symbol Manipulation and Numerical Calculation. Comm. ACM 4, (March 1961), pp 147-150.

4. HOARE, C.A.R.
   Record Handling. pp 291-347. In Genuys, F. ed. Programming Languages. Academic Press, 1968.

5. BAUR, H.R. et al.
   ALGOL W Language Description. Technical Report CS 89, Computer Science Department, Stanford University, March 1968.

6. BAUR, H.R. et al.
   ALGOL W Implementation. Technical Report CS 98, Computer Science Department, Stanford University, May 1968.

7. SITES, Richard L.
   ALGOL W Reference Manual. Computer Science Department, Stanford University, August 1971.

8. NAUR, P. (ed.)
   Revised Report on the Algorithmic Language ALGOL 60, Comm. ACM 6, (January 1963), pp 1-17.

9. MOIR, D.A.K. and J.M. Wells
   Format-Directed Input/Output For Algol W. Scientific Reports No. 23, Department of Computer Science, The University of Manitoba, April 1971.

10. PATEL, Rajini M.
Basic I/O Handling on Burroughs B6500.
Proceedings of 2nd ACM Symposium on Operating
Systems Principles, (October 1969), pp 120-129.

11. THE BURROUGHS CORPORATION
Burroughs B6700 Input/Output Subsystem Information
Manual. No. 5000185, 1974

12. THE BURROUGHS CORPORATION
Burroughs B6700/B7700 ALGOL Language Reference
Manual, 1974.

13. SATTERTHWAITE, E.
Debugging Tools for High Level Languages.
Technical Report No. 29, University of Newcastle
upon Tyne, December 1971.

14. FLOYD, R.W.
Syntactic Analysis and Operator Precedence.
JACM 10, (July 1963), pp 316-333.

15. MARTIN, D.F.
Boolean Matrix Methods for the Detection of Simple
Precedence Grammars. CACM 11, (October 1968),
pp 685-687.

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

17. THE BURROUGHS CORPORATION
Burroughs B6700 Information Processing Systems Reference
Manual, 1972.

18. THE BURROUGHS CORPORATION
Burroughs B6700 Code File Formats

19. ORGANICK, Elliott I.
Computer System Organization. The B5700/B6700 Series. Academic Press, 1973.

20. RANDELL, B. and L.J. Russell
ALGOL 60 Implementation. Academic Press, 1974.

21. INGERMANN, P.Z.
Thunks. CACM 4, (January 1961), pp 55-58.

APPENDICES

Appendix A

SIMPLE PRECEDENCE GRAMMAR FOR ALGOL W

```
1.    <T var id> ::= <id>
2.    <label id> ::= <id>
3.    <T array id> ::= <id>
4.    <proc id> ::= <id>
5.    <rec cl id> ::= <id>
6.    <T fld id> ::= <id>
7.    <cont id> ::= <id>
8.    <T func id> ::= <id>
9.    <file id> ::= <id>
10.   <prog> ::= <statement>
11.          ::= <proc dec>
12.   <statement> ::= <state>
13.   <state> ::= <si st>
14.           ::= <for cl> DO
15.           ::= <for cl> DO <state>
16.           ::= <while cl> DO
17.           ::= <while cl> DO <state>
18.           ::= <if cl>
19.           ::= <if cl> <state>
20.           ::= <if cl> <true pt>
21.           ::= <if cl> <true pt> <state>
22.           ::= <case seq> END
23.           ::= <case seq> <statement> END
24.   <si st> ::= <blck>
25.           ::= <T ass st>
26.           ::= GOTO <label id>
27.           ::= ASSERT <T exp*>
28.           ::= <proc id>
29.           ::= <proc hd> <T exp>)
30.           ::= <proc hd> <statement>)
31.           ::= <proc hd>)
32.           ::= READ <rw hd>
```

```
33.             ::= WRITE <rw hd>
34.             ::= <space hd> <T exp>)
35.             ::= <rewind hd> <file id>)
36.             ::= <seek hd>)
38.             ::= <close hd> <file id>)
39.             ::= <close hd*> <ast>)
40.             ::= <close hd*> PURGE)
41.             ::= <close hd*> REEL)
42.             ::= <close hd*> CRUNCH)
43.             ::= <lock hd> <file id>)
44.             ::= <lock hd*> <ast>)
45.             ::= <lock hd*> CRUNCH)
46.             ::= <open hd> <file id>)
47.   <blck> ::= <blockbody> END
48.           ::= <blockbody> <statement> END
49.   <blockbody> ::= <block hd>
50.               ::= <blockbody>;
51.               ::= <blockbody> <statement>;
52.               ::= <blockbody> <label def>
53.   <block hd> ::= BEGIN
54.              ::= <block hd> <si var dc>;
55.              ::= <block hd> <array dec>;
56.              ::= <block hd> <proc dec>;
57.              ::= <block hd> <re cl dec>;
58.              ::= <block hd> <file dec>;
59.   <si var dc> ::= <si var dc*>
60.   <si var dc*> ::= <simp type> <id>
61.                ::= <si var dc*> ,, <id>
62.   <simp type> ::= <ref type>)
72.   <ref type> ::= REFERENCE <id>
73.              ::= <ref type> ,, <id>
76.   <array dec> ::= <bnd list hd> <T exp> :: <T exp>)
77.               ::= <bnd list hd> <T exp> :: <T exp>]
78.   <bnd list hd> ::= <array hd>(
79.                 ::= <array hd>[
80.                 ::= <bnd list hd> <T exp> :: <T exp>,
81.   <array hd> ::= <simp type> ARRAY <id>
82.              ::= <array hd> ,, <id>
83.   <T exp> ::= <T exp*>
84.   <T exp*> ::= <si T exp>
85.            ::= <if cl> <true exp> <T exp*>
86.            ::= <case hd> <T exp>)
87.   <si T exp> ::= <si T exp*>
88.              ::= <si T exp**> <eql op> <si T exp*>
89.              ::= <si T exp**> <rel op> <si T exp*>
90.              ::= <si T exp**> IS <rec cl id>
91.   <si T exp*> ::= <si T exp**>
92.   <si T exp**> ::= <T term>
93.                ::= + <T term>
94.                ::= - <T term>
95.                ::= <si T exp**> + <T term>
96.                ::= <si T exp**> - <T term>
97.                ::= <si T exp**> OR <T term>
98.                ::= <rec cl id>
99.                ::= <rec des hd>)
100.               ::= <rec des hd> <T exp>)
101.               ::= <string>
102.               ::= NULL
103.  <T term> ::= <T term*>
104.  <T term*> ::= <T fact>
105.            ::= <T term*> * <T fact>
106.            ::= <T term*> / <T fact>
107.            ::= <T term*> DIV <T fact>
108.            ::= <T term*> REM <T fact>
109.            ::= <T term*> AND <T fact>
110.  <T fact> ::= <T second>
111.           ::= ¬ <T fact>
112.  <T second> ::= <T prim>
113.             ::= <T second> ** <T prim>
114.             ::= <T second> SHL <T prim>
115.             ::= <T second> SHR <T prim>
116.  <T prim> ::= <T var>
117.           ::= <T func id>
```

```
118.                ::= <T func hd> <T exp>)
119.                ::= <T func hd> <statement>)
120.                ::= <T func hd>)
121.                ::= <left paren> <T exp>)
122.                ::= TRUE
123.                ::= FALSE
124.                ::= <cont id>
125.                ::= LONG <T prim>
126.                ::= SHORT <T prim>
127.                ::= ABS <T prim>
128.                ::= <T numb>
129.                ::= <bit seq>
130.                ::= <blockbody> <T exp> END
131.    <T var> ::= <si T var>
132.            ::= <sub strng hd> <T exp> <lngth>)
133.            ::= <sub strng hd> <T exp> <lngth>]
134.    <si T var> ::= <si T var*>
135.               ::= <T array id>
136.    <si T var*> ::= <T var id>
137.                ::= <T fld hd> <T exp>)
138.                ::= <T sub des>
139.    <T fld hd> ::= <T fld id>(
140.    <T sub des> ::= <T sub hd> <T exp>)
141.                ::= <T sub hd> <T exp>]
142.                ::= <T sub hd> <ast>)
143.                ::= <T sub hd> <ast>]
144.    <T sub hd> ::= <T array id>(
145.               ::= <T array id>[
146.               ::= <T sub hd> <T exp>,
147.               ::= <T sub hd> <ast>,
148.    <ast> ::= *
149.    <sub strng hd> ::= <si T var>(
150.                   ::= <si T var>[
151.    <lngth> ::= | <T numb>
152.    <T func hd> ::= <T func id>(
153.                ::= <T func hd> <T exp>,
154.                ::= <T func hd> <statement>,
155.                   ::= <T func hd>,
156.   <left paren> ::= (
157.   <rec des hd> ::= <rec cl id>(
158.                   ::= <rec des hd> <T exp>,
159.                   ::= <rec des hd>,
160.   <eql op> ::= =
161.            ::= ¬=
162.   <rel op> ::= <
163.            ::= <=
164.            ::= >
165.            ::= >=
166.   <if cl> ::= IF <T exp> THEN
167.   <true exp> ::= <T exp> ELSE
168.   <case hd> ::= <case cl>(
169.               ::= <case hd> <T exp>,
170.   <case cl> ::= CASE <T exp> OF
171.   <proc dec> ::= <proc head> <state>
173.              ::= <proc head>
174.              ::= <proc head> <T exp>
175.   <proc head> ::= <proc head*>;
176.   <proc head*> ::= <proc>
177.                  ::= <proc> <f par hd>)
178.   <proc> ::= PROCEDURE <id>
179.          ::= <simp type> PROCEDURE <id>
180.   <f par hd> ::= <f par hd*>
181.              ::= <f array dec>
182.              ::= <f proc dec>
183.   <f par hd*> ::= (<simp type> <id>
184.                ::= (<simp type> VALUE <id>
185.                ::= (<simp type> RESULT <id>
186.                ::= (<simp type> VALUE RESULT <id>
187.                ::= <f par hd*> ,, <id>
188.                ::= <f par hd**> <simp type> <id>
189.                ::= <f par hd**> <simp type> VALUE <id>
190.                ::= <f par hd**> <simp type> RESULT <id>
191.                ::= <f par hd**> <simp type> VALUE RESULT <id>
192.   <f par hd**> ::= <f par hd*>;
```

```
193.                ::= <f array dec>;
194.                ::= <f proc dec>;
195.  <f array dec> ::= <f bnd hd> <ast>)
196.                ::= <f bnd hd> <ast>]
197.  <f bnd hd> ::= <f array hd>(
198.             ::= <f array hd>[
199.             ::= <f bnd hd> <ast>,
200.  <f array hd> ::= (<simp type> ARRAY <id>
201.               ::= <f par hd**> <simp type> ARRAY <id>
202.               ::= <f array hd> ,, <id>
203.  <f proc dec> ::= <f proc hd>
204.               ::= <f proc hd> <v par hd>)
205.  <f proc hd> ::= (<simp type> PROCEDURE <id>
206.              ::= (PROCEDURE <id>
207.              ::= <f proc hd> ,, <id>
208.              ::= <f par hd**> <simp type> PROCEDURE <id>
209.              ::= <f par hd**> PROCEDURE <id>
210.  <v par hd> ::= <v par hd*>
211.             ::= <v array dec>
212.             ::= <v proc dec>
213.  <v par hd*> ::= <left paren>
214.              ::= (<simp type>
215.              ::= (<simp type> VALUE
216.              ::= (<simp type> RESULT
217.              ::= (<simp type> VALUE RESULT
218.              ::= <v par hd**> <simp type>
219.              ::= <v par hd**> <simp type> VALUE
220.              ::= <v par hd**> <simp type> RESULT
221.              ::= <v par hd**> <simp type> VALUE RESULT
222.  <v par hd**> ::= <v par hd*>;
223.               ::= <v array dec>;
224.               ::= <v proc dec>;
225.  <v array dec> ::= <v bnd hd> <ast>)
226.                ::= <v bnd hd> <ast>]
227.  <v bnd hd> ::= <v array hd>(
228.             ::= <v array hd>[
229.             ::= <v bnd hd> <ast>,
230.  <v array hd> ::= (<simp type> ARRAY
231.                 ::= <v par hd**> <simp type> ARRAY
232.  <v proc dec> ::= <v proc hd>
233.                 ::= <v proc hd> <v par hd>)
234.  <v proc hd> ::= (<simp type> PROCEDURE
235.                ::= (PROCEDURE
236.                ::= <v par hd**> <simp type> PROCEDURE
237.                ::= <v par hd**> PROCEDURE
240.  <re cl dec> ::= <rec hd>)
241.  <rec hd> ::= <record*> (<simp type> <id>
242.           ::= <rec hd> ,, <id>
243.           ::= <rec hd*> <simp type> <id>
244.  <record*> ::= RECORD <id>
245.  <rec hd*> ::= <rec hd>;
246.  <file dec> ::= <file part>
247.             ::= <file part hd> <init>
248.  <file part> ::= FILE <id>
249.              ::= <file part> ,, <id>
250.  <file part hd> ::= <file part>(
251.  <init> ::= <init attrib>
252.  <label def> ::= <id>:
253.  <T ass st> ::= <T var> := <T exp*>
254.             ::= <T var> := <T ass st>
255.  <proc hd> ::= <proc id>(
256.            ::= <proc hd> <T exp>,
257.            ::= <proc hd> <statement>,
258.            ::= <proc hd>,
259.  <rw hd> ::= <rw state>
260.  <space hd> ::= SPACE <space hd*>
261.  <space hd*> ::= <left paren> <file id>,
262.  <rewind hd> ::= REWIND(
263.  <seek hd> ::= <seek hd*> <T exp>]
264.  <seek hd*> ::= <seek hd**> <file id>[
265.  <seek hd**> ::= SEEK(
269.  <close hd> ::= CLOSE(
270.  <close hd*> ::= <close hd> <file id>,
271.  <lock hd> ::= LOCK(
272.   <lock hd*> ::= <lock hd> <file id>,
273.   <open hd> ::= OPEN(
274.   <for cl> ::= <for hd> <step until> <T exp>
275.               ::= <for hd>
276.               ::= <for list> <T exp>
277.   <for hd> ::= <for*> ::= <T exp*>
278.   <for*> ::= FOR <id>
279.   <step until> ::= STEP <T exp> UNTIL
280.                  ::= UNTIL
281.   <for list> ::= <for hd>,
282.                 ::= <for list> <T exp>,
283.   <while cl> ::= WHILE <T exp>
284.   <true pt> ::= <si st> ELSE
285.              ::= ELSE
286.   <case seq> ::= <case cl> BEGIN
287.                 ::= <case seq> <statement>;
288.                 ::= <case seq>;
```

Appendix B

FULL DESCRIPTION OF EXTENDED ALGOL W

1. METALANGUAGE DEFINITION

The Reference Language is a phrase structure language, defined by a formal metalanguage. This metalanguage makes use of the notation and definitions explained below. The structure of the language ALGOL W is determined by:

(1) V, the set of basic constituents of the language,
(2) U, the set of syntactic entities, and
(3) P, the set of syntactic rules, or productions.

1.1 NOTATION

A syntactic entity is denoted by its name (a sequence of letters) enclosed in the brackets < and >. A syntactic rule has the form

<A> ::= x

where <A> is a member of U, x is any possible sequence of basic constituents and syntactic entities, simply to be called a "sequence". The form

<A> ::= x|y|...|z

is used as an abbreviation for the set of syntactic rules

```
<A> ::= x
<A> ::= y
  . . .
<A> ::= z
```

1.2 DEFINITIONS

1. A sequence x is said to directly produce a sequence y if and only if there exist (possibly empty) sequences u and w, so that either (i) for some <A> in U, x = u<A>w, y = uvw, and <A> ::= v is a rule in P; or (ii) x = uw, y = uvw and v is a "comment" (cf. 2.4)

2. A sequence x is said to produce a sequence y if and only if there exists an ordered set of sequences s[0], s[1], ..., s[n], so that x = s[0], s[n] = y, and s[i-1] directly produces s[i] for all i=1,...,n.

3. A sequence x is said to be an ALGOL W program if and only if its constituents are members of the set V, and x can be produced from the syntactic entity <program>.

To provide explanations for the meaning of ALGOL W programs, the letter sequences denoting syntactic entities have been chosen to be English words describing approximately the nature of that syntactic entity or construct. Where words which have appeared in this manner are used elsewhere in the text, they refer to the corresponding syntactic definition. Along with these letter sequences the symbol τ may occur. It is understood that this symbol must be replaced by any one of a finite set of English words (or word pairs). Unless otherwise specified in the particular section, all occurrences of the symbol τ within one syntactic rule must be replaced consistently, and the replacing words are

integer	logical
real	bit
long real	string
complex	reference
long complex	

For example, the production

```
      <τ term> ::= <τ factor>              (cf. 6.1)
```

corresponds to

```
      <integer term>      ::= <integer factor>
      <real term>         ::= <real factor>
      <long real term>    ::= <long real factor>
      <complex term>      ::= <complex factor>
      <long complex term> ::= <long complex factor>
```

The production

$<\tau_0$ primary$>$ ::= LONG $<\tau_1$ primary$>$ (cf. 6.1 and table for <u>long</u>)

corresponds to

```
      <long real primary>    ::= LONG <real primary>
      <long real primary>    ::= LONG <integer primary>
      <long complex primary> ::= LONG <complex primary>
```

It is recognized that typographical entities exist of lower order than basic symbols, called characters. The accepted characters are those of the Burroughs B6700 EBCDIC code.

2. LANGUAGE COMPONENTS

2.1 BASIC SYMBOL

<u>Syntax</u>

```
<basic symbol> ::= <space>|
                   <letter>|
                   <digit>|
                   <reserved words>|
                   <special characters>
<space> ::= <single space>|
            <space> <single space>
<single space> ::= <one blank position>
<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>  ::= 0|1|2|3|4|5|6|7|8|9.
```

```
<reserved words> ::= TRUE|FALSE|NULL|INTEGER|REAL|COMPLEX|
                     LOGICAL|BITS|STRING|REFERENCE|LONG REAL|
                     LONG COMPLEX|ARRAY|PROCEDURE|RECORD|BEGIN|
                     END|IF|THEN|ELSE|CASE|OF|DIV|REM|SHR|SHL|
                     IS|ABS|LONG|SHORT|AND|OR|GOTO|GO TO|FOR|
                     STEP|UNTIL|DO|WHILE|COMMENT|VALUE|RESULT|
                     ASSERT|READ|WRITE|PURGE|REEL|CRUNCH|FILE|
                     SPACE|REWIND|SEEK|CLOSE|LOCK|OPEN
<special characters> ::= "|#|'|,|;|:|.|(|)|+|-|*|/|**|¬|▮|=|¬=|
                        <|<=|>|>=|::|:=|[|]|_
```

(Note: the ▮ stands for the vertical bar |)

Semantics

<space>

Adjacent reserved words, reserved word pairs, identifiers and numbers must include no blanks and must be separated by at least one blank space. The multicharacter <special characters> (e.g. **,¬=,:=) must include no blanks.

Other than these restrictions, a <space> can appear, if desired, between any two <basic symbol>s to improve the readability of the program.

<letter>s

Only uppercase <letter>s are permitted. Individual letters have no special meanings except in the format part of a read/write statement (cf. 5.11).

<digit>s

<digit>s are used for forming <number>s, <identifier>s and <string>s.

<reserved words>

The <reserved words> are reserved for specific use in the language and may not be used for any other use.

<special characters>

The purpose of the <special characters> is explained elsewhere in the text in the syntax of the appropriate constructs.

2.2 IDENTIFIERS

Syntax

```
<identifier> ::= <letter>|
                 <identifier> <letter>|
                 <identifier> <digit>|
                 <identifier>_
<τ variable identifier> ::= <identifier>
<τ array identifier> ::= <identifier>
<procedure identifier> ::= <identifier>
<τ function identifier> ::= <identifier>
<record class identifier> ::= <identifier>
<τ field identifier> ::= <identifier>
<file identifier> ::= <identifier>
<label identifier> ::= <identifier>
<control identifier> ::= <identifier>
<identifier list> ::= <identifier>|
                      <identifier list> , <identifier>
```

Semantics

An <identifier> can be no more than 63 <character>s long and cannot include <space>s or <special characters>

Variables, arrays, procedures, record classes and record fields are said to be quantities. Identifiers serve to identify quantities, or they stand as files, labels, formal parameters or control identifiers.

Identifiers have no inherent meaning and can be chosen freely except for the restriction that reserved words can't be used.

Every <identifier> used in a program must be defined in one of the following ways:

(a) a declaration, (cf. section 4) if the <identifier> identifies a quantity or file. It is then said to denote that quantity or file and to be a τ variable identifier, τ array identifier, procedure identifier, τ function identifier, record class identifier, τ field identifier or file identifier, where τ stands for the appropriate type of the declared quantity;

(b) a label definition, (cf. 5.3) if the <identifier> stands as a label. It is then said to be a label identifier;

(c) its occurrence in a formal parameter list (cf. 4.3). It is then said to be a formal parameter;

(d) its occurrence following the symbol FOR in a for clause (cf. 5.8). It is then said to be a control identifier;

(e) its implicit declaration in the language (cf. 6.9). Standard procedures, standard functions, and pre-defined variables are considered to be declared in a block containing the program.

The recognition of the definition of an <identifier> is determined by the following rules:

Step 1. If the <identifier> is defined by a declaration of a quantity or a file or by its standing as a label within the smallest block embracing a given occurrence of that <identifier>, then it denotes that quantity, file, or label. A statement following a procedure heading or a for clause is considered to be a block.

Step 2. Otherwise, if that block is a procedure body and if the given <identifier> is identical with a formal parameter in the associated procedure heading, then it stands as that formal parameter.

Step 3. Otherwise, if that block is preceded by a for clause and the identifier is identical to the control identifier of that for clause, then it stands as that control identifier.

Otherwise, these rules are applied considering the smallest block embracing the block which has previously been considered.

If either step 1 or step 2 could lead to more than one definition then the identification is undefined.

The scope of a quantity, file, label, formal parameter, or control identifier is the set of statements in which occurrences of an identifier may refer by the above rules to the definition of that quantity, file, label, formal parameter, or control identifier.

examples

legal identifiers	illegal identifiers
I	BEGIN
PERSON	49
X15	5AD
D2P964LZ	*
A_2	NUM.
A_	_B3

2.3 NUMBERS

Syntax

```
<number> ::= <long complex number>|
             <complex number>|
             <long real number>|
             <real number>|
             <integer number>
<long complex number> ::= <complex number>L
<complex number> ::= <imaginary number>
<imaginary number> ::= <real number>I|
                       <integer number>I
<long real number> ::= <real number>L|
                       <integer number>L
<real number> ::= <unscaled real>|
                  <unscaled real> <scale factor>|
                  <integer number> <scale factor>|
                  <scale factor>
<unscaled real> ::= <integer number> . <integer number>|
                    .<integer number>|
                    <integer number>.
<scale factor> ::= '<integer number>|
                   '<sign> <integer number>
```

```
<integer number> ::= <digit>|
                     <integer number> <digit>
<sign> ::= +|-
```

Note: a long complex constant may have the I and L in any order.

Semantics

Numbers are interpreted according to the conventional decimal notation. A scale factor denotes an integral power of 10 which is multiplied by the unscaled real or integer number preceding it. Each number has a uniquely defined type. (Note that all <T number>s are unsigned). No <space> can appear within an <integer number>. All numbers that do not contain the letter L are considered to be single-precision.

NUMBER RANGES

The maximum and minimum integers and numbers that can be represented are as follows (decimal versions are only approximate).

(a) Any integer between and including 0 and 549755813887 = 8**13-1 = #007FFFFFFFFF can be represented in integer form.

(b) The maximum normalized single-precision number is 4.31359146674'68 = (8**13-1)*8*8863 = #1FFFFFFFFFFF.

(c) The minimum normalized single-precision number is 8.75811540203'-47 = 8**(-51) = #3F9000000000. The number zero and numbers with values between the maximum and minimum values given above may be represented in real form.

(d) The maximum normalized double-precision number is 1.94882938205028079124469'29603L = (8**26-1)*8**32767 = #1FFFFFFFFFFFFFFFFFFFFFFF.

(e) The minimum normalized double-precision number is 1.93854585713758583355564' - 29581L = 8**(-32742) = #3F9000000000FF8000000000. The number zero and numbers with values between the maximum and minimum values given above may be represented in long form.

COMPILER NUMBER CONVERSION

The ALGOL W compiler can convert a maximum of 24 significant decimal digits of mantissa in double-precision. The "effective exponent", which is the explicit exponent value following the ' sign minus the number of digits to the right of the decimal point, must be less than 29604 in absolute value.

examples

```
              1.5               1I
             0100               1'3
             0.67I              3.1416
         6.02486'+23            1IL
   2.7182818284590452353602L    2.3'-6
```

2.4 REMARKS

Syntax

```
<remark> ::= <end remark>|
             <comment remark>
<end remark> ::= <any unreserved identifier>
<comment remark> ::= COMMENT <any sequence of EBCDIC characters
                              not containing a semicolon>;
```

Semantics

Two methods are provided in the language to insert program documentation at various locations throughout the source file. The <end remark> is only allowed immediately following the reserved word END.

The <comment remark> is allowed between any two <basic component>s. The compiler considers the first semicolon encountered after the reserved word COMMENT as the end of the <comment remark>. All characters in the <comment remark> plus the word COMMENT and the semicolon are ignored during compilation and execution of the program.

2.5 STRINGS

Syntax

```
<string> ::= "<sequence of characters>"
```

Semantics

Strings consist of any sequence of at most 256 characters enclosed by ", the string quote. If the string quote appears in the sequence of characters it must be immediately followed by a second string quote which is then ignored. The number of characters in a string is said to be the length of the string.

examples

```
"THIS IS A STRING"
"SO IS THIS;#/,])"
"""" is the string of length 1 consisting of the string
quote.
```

3. PROGRAM STRUCTURE

Syntax

```
<program> ::= <statement>.|
              <proper procedure declaration>.|
              <τ function procedure declaration>.
```

Semantics

If the <statement> is not a <block> or <for statement> then it is treated as though it was in a block with no declarations, i.e. it is implicitly enclosed by the reserved words BEGIN and END.

4. DECLARATIONS

Syntax

```
<declaration> ::= <simple variable declaration>|
                  <array declaration>|
                  <procedure declaration>|
                  <record class declaration>|
                  <file declaration>
```

Semantics

Declarations serve to associate <identifier>s with the quantities used in the program, to attribute certain permanent properties to these quantities (e.g. type, structure), and to determine their scope. Every <identifier> must be declared prior to using it in an ALGOL W program and upon exit from a block, all quantities declared within that block lose their value and significance.

4.1 ARRAY DECLARATIONS

Syntax

```
<array declaration> ::= <τ array declaration>
<τ array declaration> ::= <simple type> ARRAY <identifier list>
                                        (<bound pair list>)|
                          <simple type> ARRAY <identifier list>
                                        [<bound pair list>]
<bound pair list> ::= <bound pair>|
                      <bound pair list> , <bound pair>
<bound pair> ::= <lower bound> :: <upper bound>
<lower bound> ::= <integer expression>
<upper bound> ::= <integer expression>
```

Semantics

Each <identifier> of the <identifier list> of an <array declaration> is associated with a variable which is declared to be of type array. A variable of type array is an ordered set of variables whose type is the <simple type> preceding the symbol ARRAY. The dimension of the array is the number of entries in the bound pair list.

Every element of an array is identified by a list of indices. The indices are the integers between and including the values of the <lower bound> and the <upper bound>. Every expression in the <bound pair list> is evaluated exactly once upon entry to the block in which the declaration occurs. The <bound pair> expressions can depend only on variables and procedures global to the block in which the declaration occurs. In order to be valid, for every <bound pair>, the value of the <upper bound> must not be less than the value of the <lower bound>. The maximum value of a <lower bound> is 131,071.

examples

```
INTEGER ARRAY H(1::100)
INTEGER ARRAY J[1::IF B THEN M+N ELSE M]
REAL ARRAY A,B(1::M,1::N)
STRING(12)ARRAY STREET,TOWN,CITY(J::K+1)
```

4.2 FILE DECLARATIONS

Syntax

```
<file declaration> ::= FILE <file list>
<file list> ::= <file list part>|
                <file identifier> , <file list part>
<file list part> ::= <file identifier>|
                     <file identifier> (<initial attribute list>)
<initial attribute list> ::= <initial attribute>|
                             <initial attribute list> , <initial attribute>
```

Semantics

A <file declaration> associates a <file identifier> with a file. The attributes for that particular file may or may not be specified in the <file declaration>. For information regarding the file attributes, refer to the B6700 Input/Output Subsystem Reference Manual, form 5000185, and B6700/B7700 System Software Handbook, form 5000722.

examples

```
FILE A
FILE COM(KIND=DISK,FILETYPE=8,BUFFERS=2,INTMODE=EBCDIC)
FILE GOT(KIND=PRINTER,BUFFERS=3,OPEN=TRUE,TITLE="GONE")
```

4.3 PROCEDURE DECLARATIONS

<u>Syntax</u>

<procedure declaration> ::= <proper procedure declaration>|
<τ function procedure declaration>

<proper procedure declaration> ::= PROCEDURE <procedure heading>;
<proper procedure body>

<τ function procedure declaration> ::= <simple type> PROCEDURE
<procedure heading>;<τ function procedure body>

<proper procedure body> ::= <statement>

<τ_0 function procedure body> ::= <τ_1 expression>

<procedure heading> ::= <identifier>|
<identifier> (<formal parameter list>)

<formal parameter list> ::= <formal parameter segment>|
<formal parameter list>;<formal parameter
segment>

<formal parameter segment> ::= <formal type> <identifier list>|
<formal array parameter>|
<formal procedure parameter>

<formal type> ::= <simple type>|
<simple type> VALUE|
<simple type> RESULT|
<simple type> VALUE RESULT

<formal array parameter> ::= <simple type> ARRAY <identifier list>
(<dimension specification>)|
<simple type> ARRAY <identifier list>
[<dimension specification>]

<dimension specification> ::= *|
<dimension specification>,*

<formal procedure parameter> ::= <simple type> PROCEDURE <identifier
list> (<virtual parameter list>)|
<simple type> PROCEDURE <identifier
list> ()|
<simple type> PROCEDURE <identifier
list>|
PROCEDURE <identifier list>
(<virtual parameter list>)|
PROCEDURE <identifier list> ()|
PROCEDURE <identifier list>

```
<virtual parameter list> ::= <virtual parameter segment>|
                             <virtual parameter list>;<virtual
                              parameter segment>
<virtual parameter segment> ::= <virtual type>|
                                <virtual array parameter>|
                                <virtual procedure parameter>
<virtual type> ::= <formal type>
<virtual array parameter> ::= <simple type> ARRAY (<dimension
                               specification>)|
                              <simple type> ARRAY [<dimension
                               specification>]
<virtual procedure parameter> ::= <simple type> PROCEDURE (<virtual
                                   parameter list> )|
                                  <simple type> PROCEDURE ()|
                                  <simple type> PROCEDURE |
                                  PROCEDURE (<virtual parameter list> )|
                                  PROCEDURE ()
                                  PROCEDURE
```

Semantics

τ_1 must be assignment compatible (c.f. 5.2) with τ_0.

A procedure declaration associates the procedure body with the identifier immediately following the symbol PROCEDURE. The principal part of the procedure declaration is the procedure body. Other parts of the block in whose heading the procedure is declared can then cause this procedure body to be executed or evaluated. A proper procedure is activated by a procedure statement (c.f. 5.10), a function procedure by a function designator (c.f. 6.3). Associated with the procedure body is a heading containing the procedure identifier and possibly a list of formal parameters.

TYPE SPECIFICATION OF FORMAL PARAMETERS

All formal parameters of a formal parameter segment are of the same indicated type. The type must be such that the replacement of the formal parameter by the actual parameter of this specified type leads to correct ALGOL W expressions and statements.

The effect of the symbols VALUE and RESULT appearing in a formal type is explained by the following rule, which is applied to the procedure body before the procedure is invoked:

(1) The procedure body is enclosed by the symbols BEGIN and END if it is not already enclosed by these symbols;

(2) For every formal parameter whose formal type contains the symbol VALUE or RESULT (or both),

 (a) a declaration followed by a semicolon is inserted after the first BEGIN of the procedure body, with a simple type as indicated in the formal type, and with an identifier different from any identifier valid at the place of declaration.

 (b) throughout the procedure body, every occurrence of the formal parameter identifier is replaced by the identifier defined in step 2a;

(3) If the formal type contains the symbol VALUE, an assignment statement (cf. 5.2) followed by a semicolon is inserted after the declarations of the procedure body. Its left part contains the identifier defined in step 2a, and its expression consists of the formal parameter identifier. The symbol VALUE is then deleted;

(4) If the formal type contains the symbol RESULT, an assignment statement preceded by a semicolon is inserted before the symbol END which terminates a proper procedure body. In the case of a function procedure, an assignment statement preceded by a semicolon is inserted after the final expression of the function procedure body. Its left part contains the formal parameter identifier, and its expression consists of the identifier defined in step 2a. The symbol RESULT is then deleted.

SPECIFICATION OF ARRAY DIMENSIONS

The number of "*"'s appearing in the formal array specification is the dimension of the array parameter.

SPECIFICATION OF VIRTUAL PARAMETERS

The optional facility of specifying virtual parameters allows compile time checking of procedure parameters. When the virtual parameter list is empty, i.e. there is nothing between the left and right parentheses, then the procedure is specified to have no parameters.

examples

```
PROCEDURE INCREMENT;
    X := X+1
REAL PROCEDURE MAX(REAL VALUE X,Y);
    IF X < Y
    THEN Y
    ELSE X
PROCEDURE COPY(REAL ARRAY U,V(*,*);INTEGER A,B);
    FOR I := 1 UNTIL A
     DO FOR J := 1 UNTIL B
          DO U(I,J) := V(I,J)
LONG REAL PROCEDURE SUM(INTEGER K,N;LONG REAL X);
   BEGIN
     LONG REAL Y;
     Y := 0;
     K := N;
     WHILE K > = 1
     DO BEGIN
          Y := Y + X;
          K := K - 1
        END;
     Y
   END
```

```
REAL PROCEDURE SERIES(INTEGER VALUE K; REAL PROCEDURE(INTEGER
                                                  VALUE)TERM);
   BEGIN REAL SUM;
         SUM := 0;
         FOR J := 1 UNTIL K DO
              SUM := SUM + TERM(J):
         SUM
   END

REFERENCE(PERSON)PROCEDURE YOUNGESTUNCLE(REFERENCE(PERSON)R);
   BEGIN
     REFERENCE(PERSON)P,M;
     P:=YOUNGESTOFFSPRING(FATHER(FATHER(R)));
     WHILE(P¬ =NULL)AND(¬ MALE(P))OR(P=FATHER(R))
     DO P:=ELDERSIBLING(P);
     M:=YOUNGESTOFFSPRING(MOTHER(MOTHER(R)));
     WHILE(M¬ =NULL)AND(¬ MALE(M))
     DO M:=ELDERSIBLING(M);
     IF P=NULL
     THEN M
     ELSE IF M=NULL
          THEN P
          ELSE IF AGE(P) < AGE(M)
               THEN P
               ELSE M
   END
```

4.4 RECORD CLASS DECLARATION

Syntax

```
<record class declaration> ::= RECORD <identifier> (<field list>)
<field list> ::= <simple variable declaration>|
                 <field list> ; <simple variable declaration>
```

Semantics

A record class declaration serves to define the structural properties of records belonging to the class. The principal constituent of a record class declaration is a sequence of simple variable declarations which define the fields and their simple types for the records of this class and associate identifiers with the individual fields. A record class identifier can be used in a record designator (cf. 6.6) to construct a new record of the given class.

examples

```
    RECORD NODE(REFERENCE(NODE)LEFT,RIGHT)
    RECORD PERSON(STRING NAME;INTEGER AGE;LOGICAL
                 MALE;REFERENCE(PERSON)FATHER,
                 MOTHER,YOUNGESTOFFSPRING,
                 ELDERSIBLING)
```

4.5 SIMPLE VARIABLE DECLARATION

Syntax

```
<simple variable declaration> ::= <simple type> <identifier list>
<simple type> ::= INTEGER|REAL|LONG REAL|COMPLEX|
                  LONG COMPLEX|LOGICAL|BITS|STRING|
                  STRING (<integer number>)|
                  REFERENCE (<record class identifier list>)
<record class identifier list> ::= <record class identifier>|
                                   <record class identifier list> ,
                                   <record class identifier>
```

Semantics

Each identifier of the identifier list is associated with a simple variable which is declared to be of the indicated type. If a variable is declared to be of a certain type, then this implies that only values which are assignment compatible with this type (cf. 5.2) can be assigned to it. It is understood that the value of a variable is equal to the value of the expression most recently assigned to it.

The value of each simple variable is as follows:

INTEGER:	the value is a 48 bit integer,
REAL:	the value is a 48 bit floating point number,
LONG REAL:	the value is a 96 bit floating point number,

COMPLEX:	the value is a complex number composed of two numbers of type real,
LONG COMPLEX:	the value is a complex number composed of two long real numbers,
LOGICAL:	the value is a logical value,
BITS:	the value is a linear sequence of 48 bits,
STRING:	the value is a linear sequence of characters of number equal to the specified number (default length of 24 characters and maximum length is 256 characters),
REFERENCE:	the value is a reference to a record and may refer only to records of the record classes whose identifiers appear in the record class identifier list of the reference declaration specification.

examples

```
   INTEGER I,J,K,M,N
   REAL X,Y,Z
   LONG COMPLEX C
   LOGICAL L
   BITS G,H
   STRING(10)S,T
   REFERENCE(PERSON)JACK,JILL
```

5. STATEMENTS

Syntax

```
<statement> ::= <simple statement>|
                <iterative statement>|
                <if statement>|
                <case statement>
<simple statement> ::= <block>|
                       <τ assignment statement>|
                       <empty>|
                       <procedure statement>|
<goto statement>|
<assert statement>|
<close statement>|
<lock statement>|
<read statement>|
<rewind statement>|
<seek statement>|
<space statement>|
<write statement>
```

Semantics

<statement>s are the active elements of an ALGOL W program. By the execution of a <statement> is meant the performance of this action, which may consist of smaller units of action such as the evaluation of expressions or the execution of other statements.

5.1 ASSERT STATEMENT

Syntax

<assert statement> ::= ASSERT <logical expression>

Semantics

The <assert statement> is equivalent to the <if statement> (cf. 5.7):

IF ¬ (<logical expression>) THEN endexecution

where "endexecution" signifies a procedure which terminates the execution of an ALGOL W program. The <assert statement> can be used as a debugging aid asserting conditions which should be true, but may not be if a bug exists.

5.2 ASSIGNMENT STATEMENT

Syntax

<τ_0 assignment statement> ::= <τ_0 left part> <τ_1 expression>|
<τ_0 left part> <τ_1 assignment statement>

<τ left part> ::= <τ variable> :=

Semantics

In the above rules the symbols τ_0 and τ_1 must be replaced by words as indicated in Section 1, subject to the restriction that the type τ_1 is assignment compatible with the type τ_0 as defined below.

The execution of a simple assignment statement

<τ_0 assignment statement> ::= <τ_0 left part> <τ_1 expression>

causes the assignment of the value of the expression to the variable. If a shorter string is to be assigned to a longer one, the shorter string is first extended to the right with blanks until the lengths are equal. In a multiple assignment statement

<τ_0 assignment statement> ::= <τ_0 left part> <τ_1 assignment statement>

the assignments are performed from right to left. For each left part variable, the simple type of the expression or assignment variable immediately to the right must be assignment compatible with the simple type of that variable.

ASSIGNMENT COMPATIBILITY

A simple type τ_1 is said to be assignment compatible with a simple type τ_0 if either

(1) the two types are identical (except that if τ_0 and τ_1 are string, the length of the τ_0 variable must be greater than or equal to the length of the τ_1 expression or assignment), or

(2) τ_0 is real or long real, and τ_1 is integer, real or long real, or

(3) τ_0 is complex or long complex, and τ_1 is integer, real, long real, complex or long complex.

In the case of a reference, the reference to be assigned must refer to a record of one of the classes specified by the record class identifiers associated with the reference variable in its declaration.

<u>examples</u>

```
Z:=AGE(JACK):=28
X:=Y+ABS Z
C:=I+X+C
P:=X¬=Y
```

5.3 BLOCKS

Syntax

```
<block> ::= <blockbody> <statement> END
<blockbody> ::= <block head>|
                <blockbody> <statement>;|
                <blockbody> <label definition>
<block head> ::= BEGIN|
                <block head> <declaration>;
<label definition> ::= <identifier>:
```

Semantics

Every <block> introduces a new level of nomenclature. This is realized by execution of the block in the following steps:

Step 1. If an <identifier>, say A, defined in the <block head> or in a <label definition> of the <blockbody> is already defined at the place from which the <block> is entered, then every occurrence of that <identifier>, A, within the <block> except for occurrence in array bound expressions is systematically replaced by another <identifier>, say APRIME, which is defined neither within the <block> nor at the place from which the <block> is entered.

Step 2. If the <declaration>s of the <block> contain array bound expressions, then these expressions are evaluated.

Step 3. Execution of the <statement>s contained in the <blockbody> begins with the execution of the first <statement> following the <block head>.

After execution of the last <statement> of the <blockbody> (unless it is a <goto statement>) a block exit occurs, and the <statement> following the entire <block> is executed.

example

```
BEGIN
  REAL U;
  U:=X;
  X:=Y;
  Y:=Z;
  Z:=U
END
```

5.4 CASE STATEMENT

Syntax

```
<case statement> ::= <case clause> BEGIN <statement list> END
<statement list> ::= <statement>|
                     <statement list> ; <statement>
<case clause> ::= CASE <integer expression> OF
```

Semantics

The execution of a <case statement> proceeds in the following steps:

Step 1. The expression of the <case clause> is evaluated

Step 2. The <statement> whose ordinal number in the <statement list> is equal to the value obtained in Step 1 is executed. In order that the <case statement> be defined, the current value of the expression in the <case clause> must be the ordinal number of some <statement> of the <statement list>. The n <statement>s are numbered from 1 to n.

examples

```
CASE I OF
   BEGIN
     X:=X+Y;
     Y:=Y+Z;
     Z:=Z+X
   END
```

```
CASE J OF
   BEGIN
     H(I):=-H(I);
     BEGIN
       H(I-1):=H(I-1)+H(I);
       I:=I-1
     END;
     BEGIN
       H(I-1):=H(I-1)*H(I);
       I:=I-1
     END;
     BEGIN
       H(H(I-1)):=H(I);
       I:=I-2
     END
```

5.5 CLOSE STATEMENT

Syntax

```
<close statement> ::= CLOSE(<file identifier>)|
                      CLOSE(<file identifier> , <close option>)
<close option> ::= *|
                   PURGE|
                   REEL|
                   CRUNCH
```

Semantics

For semantics see the B6700/B7700 Algol Language Reference Manual.

5.6 GO TO STATEMENT

Syntax

```
<goto statement> ::= GOTO <label identifier>|
                     GO TO <label identifier>
```

Semantics

An <identifier> is called a <label identifier> if it stands as a label.

A <goto statement> determines that execution of the text be continued after the <label definition> (cf. 5.3) of the <label identifier>. The identification of that label definition is accomplished in the following steps:

Step 1. If some <label definition> within the most recently activated, but not yet terminated, block contains the <label identifier>, then this is the designated <label definition>. Otherwise,

Step 2. The execution of that block is considered as terminated and Step 1 is taken as specified above.

5.7 IF STATEMENT

Syntax

```
<if statement> ::= <if clause> <statement>|
                   <if clause> <simple statement> ELSE <statement>
<if clause> ::= IF <logical expression> THEN
```

Semantics

The execution of <if statement>s causes certain <statement>s to be executed or skipped depending on the values of specified <logical expression>s. An <if statement> of the form

<if clause> <statement>

is executed in the following steps:

Step 1. The <logical expression> in the <if clause> is evaluated.

Step 2. If the result of Step 1 is true, then the <statement> following the <if clause> is executed. Otherwise Step 2 causes no action to be taken at all.

An <if statement> of the form

<if clause> <simple statement> ELSE <statement>

is executed in the following steps:

Step 1. The <logical expression> in the <if clause> is evaluated.

Step 2. If the result of Step 1 is true, then the <simple statement> following the <if clause> is executed. Otherwise the <statement> following ELSE is executed.

examples

```
IF X=Y
THEN GOTO L
IF X < Y
THEN U:=X
ELSE IF Y < Z
     THEN U:=Y
     ELSE V:=Z
```

5.8 ITERATIVE STATEMENT

Syntax

```
<iterative statement> ::= <for clause> <statement>|
                          <while clause> <statement>
<for clause> ::= FOR <identifier> := <initial value> STEP <increment> UNTIL
                                                  <limit> DO|
                 FOR <identifier> := <initial value> UNTIL <limit> DO|
                 FOR <identifier> := <for list> DO
<for list> ::= <integer expression>|
               <for list> , <integer expression>
<initial value> ::= <integer expression>
<increment> ::= <integer expression>
<limit> ::= <integer expression>
<while clause> ::= WHILE <logical expression> DO
```

Semantics

The <iterative statement> serves to express that a <statement> be executed repeatedly depending on certain conditions specified by a <for clause> or a <while clause>. The <statement> following the <for clause> or the <while clause> always acts as a <block>, whether it has the form of a <block> or not. The value of the <control

identifier> (the <identifier> following FOR) cannot be changed by assignment within the controlled <statement>. The <control identifier> doesn't need to be declared by way of a <declaration> and is invalid outside the <iterative statement>.

(a) An <iterative statement> of the form

```
   FOR <identifier>:=<E1>STEP<E2>UNTIL<E3>DO <statement>
```

is exactly equivalent to the <block>

```
   BEGIN
     <statement-0>;
     <statement-1>;
         ...       ;
     <statement-I>;
         ...       ;
     <statement-N>
   END
```

In the I'th <statement> every occurrence of the <control identifier> is replaced by the value of the expression (E1+I*E2).

The index N of the last <statement> is determined by $N \leq (E3-E1)/E2 < N+1$. If $N < 0$, then it is understood that the sequence is empty. The expressions E1, E2, and E3 are evaluated exactly once, namely before execution of <statement-0>, therefore, they can not depend on the <control identifier>.

(b) An <iterative statement> of the form

```
      FOR <identifier>:=<E1>UNTIL<E3>DO <statement>
```

is exactly equivalent to the <iterative statement>

```
      FOR <identifier>:=<E1>STEP 1 UNTIL<E3>DO <statement>.
```

(c) An <iterative statement> of the form

```
        FOR <identifier>:=<E1>,<E2>,...,<EN>DO <statement>
```

is exactly equivalent to the <block>

```
        BEGIN
          <statement-1>;
          <statement-2>;
              ...      ;
          <statement-I>;
              ...      ;
          <statement-N>
        END
```

where in the I'th <statement> every occurrence of the <control identifier> is replaced by the value of the expression EI.

(d) An <iterative statement> of the form

```
        WHILE <E> DO <statement>
```

is exactly equivalent to

```
        BEGIN
     L:   IF <E>
          THEN BEGIN
                 <statement>;
                 GOTO L
               END
        END
```

where it is understood that L represents an <identifier> which is not defined at the place from which the while statement is entered.

examples

```
    FOR V:=1 STEP 1 UNTIL N-1
      DO S:=S+A(U,V)
    WHILE(J > 0)AND(CITY(J)¬ =S)
    DO J:=J-1
    FOR I:=X,X+1,X+3,X+7
      DO P(I)
```

5.9 LOCK STATEMENT

Syntax

```
<lock statement> ::= LOCK(<file identifier>)|
                     LOCK(<file identifier> , <lock option>)
<lock option> ::= *|
                  CRUNCH
```

Semantics

For semantics see the B6700/B7700 Algol Language Reference Manual.

5.10 PROCEDURE STATEMENT

Syntax

```
<procedure statement> ::= <procedure identifier>|
                          <procedure identifier>(<actual parameter list>)
<actual parameter list> ::= <actual parameter>|
                            <actual parameter list> , <actual parameter>
<actual parameter> ::= <τ expression>|
                       <statement>|
                       <τ subarray designator>|
                       <procedure identifier>|
                       <τ function identifier>
<τ subarray designator> ::= <τ array identifier>|
                            <τ array identifier>(<subarray designator
                                                   list>)|
                            <τ array identifier>[<subarray designator
                                                   list>]
<subarray designator list> ::= <subscript>|
                               *|
                               <subarray designator list> , <subscript>|
                               <subarray designator list>,*
```

Semantics

The execution of a <procedure statement> is equivalent to a process performed in the following steps:

Step 1. A copy is made of the body of the proper procedure whose <procedure identifier> is given by the <procedure statement>, and of the <actual parameter>s of the latter. The <procedure statement> is replaced by the copy of the procedure body.

Step 2. If the procedure body is a <block>, then a systematic change of <identifier>s in its copy is performed as specified by Step 1 of 5.3.

Step 3. The copies of the <actual parameter>s are treated in an undefined order as follows: If the copy is an expression different from a variable, then it is enclosed by a pair of parentheses, or if it is a <statement> it is enclosed by the symbols BEGIN and END.

Step 4. In the copy of the procedure body every occurrence of an <identifier> identifying a formal parameter is replaced by the copy of the corresponding <actual parameter>. In order for the process to be defined, these replacements must lead to correct ALGOL W expressions and <statement>s.

Step 5. The copy of the procedure body, modified as indicated in Steps 2-4, is executed.

ACTUAL-FORMAL CORRESPONDENCE
The correspondence between the <actual parameter>s and the formal parameters is established as follows: The <actual parameter list> of the <procedure statement> (or of the <τ function designator>) must have the same number of entries as the <formal parameter list> of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

SUBARRAY DESIGNATORS
A complete array may be passed to a procedure by specifying the name of the array if the number of <subscript>s of the <actual

parameter> equals the number of <subscript>s of the corresponding formal parameter. If the actual array parameter has more <subscript>s than the corresponding formal parameter, enough <subscript>s must be specified by <integer expression>s so that the number of *'s appearing in the subarray designator equals the number of <subscript>s of the corresponding formal parameter. The <subscript> positions of the formal array designator are matched with the positions with *'s in the subarray designator in the order they appear.

PARAMETER CORRESPONDENCE

Formal Type	Actual Parameter
<simple τ type>	<τ expression>
<simple τ_0 type> VALUE	<τ_1 expression>
<simple τ_1 type> RESULT	<τ_0 variable>
<simple τ_1 type> VALUE RESULT	<τ_2 variable>
<simple τ type> PROCEDURE	<τ function identifier>
	<τ expression>
PROCEDURE	<procedure identifier>
	<statement>
<simple τ type> ARRAY	<τ subarray designator>

The simple type τ_1 must be assignment compatible (cf. 5.2) with the simple type τ_0. The simple types τ_1 and τ_2 must be mutually assignment compatible.

As <actual parameter>s, expressions and <statement>s may serve as the implicit specifications of nameless and parameterless procedures.

examples

```
INCREMENT
COPY(A,B,M,N)
INNERPRODUCT(IP,N,A(I,*),B(J,K,*,*))
```

5.11 READ STATEMENT

Syntax

```
<read statement> ::= READ(<file part> <format and list part>)
<file part> ::= <file identifier> <record number or carriage control>
<record number or carriage control> ::= <empty>|
                                        [<integer number>]|
                                        [LINE <integer number>]|
                                        [NO]|
                                        [SKIP <integer number>]|
                                        [SPACE <integer number>]
<format and list part> ::= <empty>|
                           ,<<editing specifications>>|
                           ,<<editing specifications>>,<list>|
                           ,*,<list>|
                           ,/,<list>|
                           ,<integer number>,<τ subarray designator>
<list> ::= <τ variable>|
           <control identifier>|
           <list>,<τ variable>|
           <list>,<control identifier>
<editing specifications> ::= <editing segment>|
                             <editing specifications>/|
                             /<editing specifications>|
                             <editing specifications>/<editing segment>
<editing segment> ::= <editing phrase>|
                      <repeat part>(<editing specifications>)|
                      <editing segment>,<repeat part>(<editing
                                                  specifications>)
<editing phrase> ::= <repeat part> <editing phrase type> <field width part>
<repeat part> ::= <empty>|
                  <integer number>|
                  *
<editing phrase type> ::= <string> |A|C|D|E|F|G|H|I|J|K|L|O|
                          R|S|T|V|X
```

```
<field width part> ::= <empty>|
                       <field width> <decimal places>
<field width> ::= <integer number>|
                  *
<decimal places> ::= <empty>|
                     .<integer number>|
                     .*
```

Semantics

For semantics see the B6700/B7700 Algol Language Reference Manual.

5.12 REWIND STATEMENT

Syntax

```
<rewind statement> ::= REWIND (<file identifier>)
```

Semantics

For semantics see the B6700/B7700 Algol Language Reference Manual.

5.13 SEEK STATEMENT

Syntax

```
<seek statement> ::= SEEK(<file identifier>[<record number>])
<record number> ::= <integer expression>
```

Semantics

For semantics see the B6700/B7700 Algol Language Reference Manual.

5.14 SPACE STATEMENT

Syntax

```
<space statement> ::= SPACE(<file identifier>,<integer expression>)
```

Semantics

For semantics see the B6700/B7700 Algol Language Reference Manual.

5.15 WRITE STATEMENT

Syntax

```
<write statement> ::= WRITE(<file part> <format and list part>)
```

Semantics

For semantics see the B6700/B7700 Algol Language Reference Manual.

6. EXPRESSIONS

Syntax

<τ expression> ::= <simple τ expression>|
<case clause>(<τ expression list>)

<τ_0 expression> ::= <if clause> <τ_1 expression> ELSE <τ_2 expression>

<τ expression list> ::= <τ expression>

<τ_0 expression list> ::= <τ_1 expression list>,<τ_2 expression>

<τ block expression> ::= <blockbody> <τ expression> END

Semantics

In the above rules the symbol τ has to be replaced consistently as described in Section 1, and the triplets τ_0, τ_1, τ_2 have to be either all three replaced by the same one of the words

logical
bit
string
reference

or by any combination of words as indicated by the following table, which yields τ_0 given τ_1 and τ_2:

τ_1 \ τ_2	integer	real	complex
integer	integer	real	complex
real	real	real	complex
complex	complex	complex	complex

τ_0 has the quality "long" if either both τ_1 and τ_2 have that quality, or if one has that quality and the other is "integer".

Expressions are rules which specify how new values are computed from existing ones. These new values are obtained by performing the operations indicated by the operators on the values of the operands. The operands are either constants, variables or function designators, or other expressions, enclosed by parentheses if necessary. The evaluation of operands other than constants may involve smaller units of action such as the evaluation of other expressions or the execution of <statement>s. The value of an expression between parentheses is obtained by evaluating that expression.

The construction

<if clause> <τ_1 expression> ELSE <τ_2 expression>

causes the selection and evaluation of an expression on the basis of the current value of the <logical expression> contained in the <if clause>. If this value is TRUE, the expression following the <if clause> is selected; if the value is FALSE, the expression following ELSE is selected. If τ_1 and τ_2 are <simple type> STRING, the shorter expression will be padded on the right with blanks to make it the length of the longer one.

The construction

<case clause>(<τ expression list>)

causes the selection of the expression whose ordinal number in the expression list is equal to the current value of the <integer expression> contained in the <case clause>. In order that the case expression be defined, the current value of this expression must be the ordinal number of some expression in the expression list (> = 1). If τ is <simple type> STRING, the <string expression>s will be padded on the right with blanks to make all alternatives the length of the longest one.

The construction

<blockbody> <τ expression> END

can be considered as a <τ function procedure body> without parameters. This represents a considerable notational convenience, since it enables the function to be specified actually in the place where it is to be used, rather than disjointly in the head of some embracing block.

6.1 ARITHMETIC EXPRESSIONS

Syntax

<simple τ expression> ::= <τ term>|
+<τ term>|
-<τ term>

<simple τ_0 expression> ::= <simple τ_1 expression> + <τ_2 term>|
<simple τ_1 expression> - <τ_2 term>

<τ term> ::= <τ factor>

<τ_0 term> ::= <τ_1 term> * <τ_2 factor>

<τ_0 term> ::= <τ_1 term> / <τ_2 factor>

<integer term> ::= <integer term> DIV <integer factor>|
<integer term> REM <integer factor>

<τ_0 factor> ::= <τ_0 primary>|
<τ_1 factor> ** <integer primary>

<τ_0 primary> ::= ABS <τ_1 primary>

<τ_0 primary> ::= LONG <τ_1 primary>

<τ_0 primary> ::= SHORT <τ_1 primary>

<τ primary> ::= <τ variable>|
<τ function designator>|
(<τ expression>)|
<τ number>|
<τ block expression>

<integer primary> ::= <control identifier>

Semantics

In any of the above rules, every occurrence of the symbol τ must be systematically replaced by one of the following words (or word pairs):

integer
real
long real
complex
long complex

The rules governing the replacement of the symbols τ_0, τ_1 and τ_2 are given below.

An arithmetic expression is a rule for computing a number. According to its <simple type> it is called an <integer expression>, <real expression>, <long real expression>, <complex expression>, or <long complex expression>.

The operators +, -, *, and / have the conventional meanings of addition, subtraction, multiplication and division. In the relevant syntactic rules above the symbols τ_0, τ_1 and τ_2 have to be replaced by any combination of words according to the following tables which indicate τ_0 for any combination of τ_1 and τ_2.

Operators + | -

τ_1 \ τ_2	integer	real	complex
integer	integer	long real	long complex
real	long real	real	complex
complex	long complex	complex	complex

τ_0 has the quality "long" if both τ_1 and τ_2 have the quality "long", or if one has the quality "integer" and the other does not.

Operator *

τ_1 \ τ_2	integer	real	complex
integer	integer	long real	long complex
real	long real	long real	long complex
complex	long complex	long complex	long complex

τ_1 or τ_2 having the quality "long" does not affect the type of the result.

Operator /

τ_1 \ τ_2	integer	real	complex
integer	long real	long real	long complex
real	long real	real	complex
complex	long complex	complex	complex

τ_0 has the quality "long" if both τ_1 and τ_2 have the quality "long", or if one has the quality "integer" and the other does not, or if both are "integer".

The operation "-" standing as the first symbol of a <simple expression> denotes the monadic operation of sign inversion. The type of the result is the type of the operand. The operator "+" standing as the first symbol of a <simple expression> denotes the monadic operation of identity.

The operator DIV is mathematically defined (for B $\neq$ 0) as

```
            A DIV B = SGN(A*B)*D(ABS A,ABS B)
```

where the function procedures SGN and D are declared as

```
            INTEGER PROCEDURE SGN (INTEGER VALUE A);
               IF A < 0
               THEN -1
               ELSE 1;
            INTEGER PROCEDURE D (INTEGER VALUE A,B);
               IF A < B
               THEN 0
               ELSE D(A-B,B)+1;
```

The operator REM (remainder) is mathematically defined as

```
            A REM B = A-(A DIV B)*B
```

A and B both must be <integer expression>s.

The operator ** denotes exponentiation of the first operand to the power of the second operand. In the relevant syntactic rule above the symbols τ_0 and τ_1 are to be replaced by any of the following combinations of words:

τ_0	τ_1
long real	integer
long real	real
long complex	complex

τ_1 having quality "long" does not affect the type of the result.

The monadic operator ABS yields the absolute value or modulus of the operand. In the relevant syntactic rule above the symbols τ_0 and τ_1 have to be replaced by any of the following combinations of words:

τ_0	τ_1
integer	integer
real	real
real	complex

If τ_1 has the quality "long", then so does τ_0.

In the relevant syntactic rules above the symbols τ_0 and τ_1 must be replaced by any of the following combinations of words (or word pairs):

Operator LONG

τ_0	τ_1
long real	integer
long real	real
long complex	complex

Operator SHORT

τ_0	τ_1
real	long real
complex	long complex

Note: It is illegal to apply LONG to an expression which is already long; similarly for SHORT.

examples

```
    C + A(I) * B(I)
    EXP(-X/(2*SIGMA))/SQRT(2*SIGMA)
```

6.2 BIT EXPRESSIONS

Syntax

```
<simple bit expression> ::= <bit term>|
                            <simple bit expression> OR <bit term>
<bit term> ::= <bit factor>|
               <bit term> AND <bit factor>
<bit factor> ::= <bit secondary>|
                 ¬<bit secondary>
<bit secondary> ::= <bit primary>|
                    <bit secondary> SHL <integer primary>|
                    <bit secondary> SHR <integer primary>
<bit primary> ::= <bit sequence>|
                  <bit variable>|
                  <bit function designator>|
                  (<bit expression>)|
                  <bit block expression>
<bit sequence> ::= # <hex digit>|
                   <bit sequence> <hex digit>
<hex digit> ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
```

Semantics

The number of bits in a <bit sequence> is 48 or 12 <hex digit>s. The <bit sequence> is always represented by a 48 bit word with the specified <bit sequence> right justified in the word and zeros filled in on the left.

A <bit expression> is a rule for computing a <bit sequence>.

The operators AND, OR, and ¬ produce a result of type BITS, every bit being dependent on the corresponding bit(s) in the operand(s) as follows:

X	Y	¬ X	X AND Y	X OR Y
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

The operators SHL and SHR denote the shifting operation to the left and to the right respectively by the number of bit positions indicated by the absolute value of the <integer primary>. Vacated bit positions to the right or left respectively are assigned the bit value 0.

examples

G AND H OR #38

G AND ¬ (H OR G)SHR 8

6.3 FUNCTION DESIGNATORS

Syntax

```
<τ function designator> ::= <τ function identifier>|
                            <τ function identifier>(<actual parameter
                                  list>)
```

Semantics

A function designator defines a value which can be obtained by a process performed in the following steps:

Step 1. A copy is made of the body of the function procedure whose <τ function identifier> is given by the function designator and of the <actual parameter>s of the latter.

Steps 2,3,4 As specified in 5.10.

Step 5 The copy of the <T function procedure body>, modified as indicated in Steps 2-4, is executed. Execution of the expression which constitutes or is part of the modified procedure body consists of evaluation of that expression and the resulting value is the value of the function designator. The <simple type> of the function designator is the <simple type> in the corresponding function procedure declaration.

examples

```
    MAX(X**2,Y**2)
    SUM(I,100,H(I))
    SUM(I,M,SUM(J,N,A(I,J)))
    YOUNGESTUNCLE(JILL)
    SUM(I,10,X(I)*Y(I))
    HORNER(X,10,2.7)
```

6.4 LOGICAL EXPRESSIONS

Syntax

```
<simple logical expression> ::= <logical element>|
                                <relation>
<logical element> ::= <logical term>|
                      <logical element> OR <logical term>
<logical term> ::= <logical factor>|
                   <logical term> AND <logical factor>
<logical factor> ::= <logical primary>|
                     ¬ <logical primary>
```

```
<logical primary> ::= <logical value>|
                      <logical variable>|
                      <logical function designator>|
                      (<logical expression>)|
                      <logical block expression>
<logical value> ::= TRUE|
                    FALSE
<relation> ::= <simple τ0 expression> <equality operator> <simple
                                                   τ1 expression>|
          <logical element> <equality operator> <logical element>|
          <simple reference expression> IS <record class identifier>|
          <simple τ2 expression> <relational operator> <simple
                                                   τ3 expression>
<relational operator> ::= < | < = | > = | >
<equality operator> ::= =|¬ =
```

Semantics

In the above rules for <relation> the symbols τ_0 and τ_1 must either be identically replaced by any one of the following words:

bit
string
reference

or by any of the words from:

complex
long complex
real
long real
integer

and the symbols τ_2 or τ_3 must be identically replaced by

string

or must be replaced by any of

real
long real
integer.

A <logical expression> is a rule for computing a <logical value>.

The <relational operator>s represent algebraic ordering for arithmetic arguments and EBCDIC ordering for string arguments. If two strings of unequal length are compared, the shorter string is first extended to the right by blanks. The <relational operator>s yield the <logical value> TRUE if the relation is satisfied for the values of the two operands; FALSE otherwise. Two references are equal if and only if they are both NULL or both refer to the same record. The operator IS yields the <logical value> TRUE if the <reference expression> designates a record of the indicated record class; FALSE otherwise. The reference value NULL fails to designate a record of any record class.

The operators ¬ (not), AND, and OR, operating on <logical value>s, are defined by the following equivalences:

```
      ¬ X     IF X
              THEN FALSE
              ELSE TRUE
  X AND Y     IF X
              THEN Y
              ELSE FALSE
   X OR Y     IF X
              THEN TRUE
              ELSE Y
```

examples

```
P OR Q
(X < Y)AND(Y < Z)
YOUNGESTOFFSPRING(JACK)¬ =NULL
FATHER(JILL) IS PERSON
```

6.5 OPERATOR PRECEDENCE

The syntax of 6.1, 6.2 and 6.4 implies the following hierarchy of operator precedences:

```
LONG,SHORT,ABS
SHL,SHR,**
¬
*,/,DIV,REM,AND
+,-,OR
<,<=,=,¬=,>=,>,IS
```

example

A = B AND C is equivalent to A = (B AND C)

6.6 REFERENCE EXPRESSIONS

Syntax

```
<simple reference expression> ::= <null reference>|
                                  <reference variable>|
                                  <reference function designator>|
                                  <record designator>|
                                  (<reference expression>)|
                                  <reference block expression>
<record designator> ::= <record class identifier>|
                        <record class identifier>(<expression list>)
<expression list> ::= <τ expression>|
                      <expression list> , <τ expression>|
                      <empty>|
                      <expression list>,
<null reference> ::= NULL
```

Semantics

A <reference expression> is a rule for computing a reference to a record.

The value of a <record designator> is the reference to a newly created record belonging to the designated record class. If the <record designator> contains an <expression list>, then the values of the expressions are assigned to the fields of the new record. The entries in the <expression list> are taken in the same order as the fields in the <record class declaration> (cf.4.4), and the <simple type>s of the expressions must be assignment compatible with the <simple type>s of the record fields (cf. 5.2). The <empty> entry in the <expression list> allows for selective initialization of the record fields.

The reference value NULL fails to designate a record; if a <reference expression> occurring in a field designator (cf. 6.8) has this value, then the field designator is undefined.

example

```
    PERSON("CAROL",0,FALSE,JACK,JILL,NULL,
                 YOUNGESTOFFSPRING(JACK))
```

6.7 STRING EXPRESSIONS

Syntax

```
<simple string expression> ::= <string primary>
<string primary> ::= <string>|
                     <string variable>|
                     <string function designator>|
                     (<string expression>)|
                     <string block expression>
<substring designator> ::= <simple string variable>(<integer expression>
                                                   ‖ <integer number>)|
                           <simple string variable>[<integer expression>
                                                   ‖ <integer number>]
```

(Note: The ‖ stands for the vertical bar character |).

Semantics

A <string expression> is a rule for computing a <string>.

A <substring designator> denotes a sequence of characters of the <string> designated by the <string variable>. The <integer expression> preceding the ‖ selects the starting character of the sequence. The value of the expression indicates the position in the <string variable>. The value must be greater than or equal to 0 and less than the declared length of the <string variable>. The first character of the <string> has position 0. The <integer

number> following the | indicates the length of the selected sequence and is the length of the <string expression>. The sum of the <integer expression> and the <integer number> must be less than or equal to the declared length of the <string variable>.

example

```
    STRING(10)S;
      S(4|3)
      S(I+J|1)
    STRING(10)ARRAY T(1::M,2::N);
      T(4,6)(3|5)
```

6.8 VARIABLES

Syntax

```
<simple τ variable> ::= <τ variable identifier>|
                        <τ field designator>|
                        <τ array designator>
<τ variable> ::= <simple τ variable>
<string variable> ::= <substring designator>
<τ field designator> ::= <τ field identifier>(<reference expression>)
<τ array designator> ::= <τ array identifier>(<subscript list>)|
                         <τ array identifier>[<subscript list>]
<subscript list> ::= <subscript>|
                     <subscript list> , <subscript>
<subscript> ::= <integer expression>
```

Semantics

An array designator denotes the variable whose indices are the current values of the expressions in the <subscript list>. The value of each <subscript> must lie within the declared bounds for that <subscript> position.

A field designator designates a field in the record referred to by its <reference expression>. The <simple type> of the field designator is defined by the declaration of that field identifier in the record class designated by the <reference expression> of the field designator (cf. 4.4).

examples

X

A(I)

M(I+J,I-J)

FATHER(JACK)

MOTHER(FATHER(JILL))

6.9 STANDARD FUNCTIONS

The following are the standard functions available. They are considered to be declared in a block which encloses each ALGOL W program.

real procedure ARCTAN (real value X);

comment arctangent (radians) of X;

bits procedure BITSTRING (integer value N);

comment two's complement representation of N;

string (1) procedure CODE (integer value N);

comment character with numeric code given by abs (N rem 250);

real procedure COS (real value X);

comment cosine of X (radians);

integer procedure DECODE (string (1) value S);

comment numeric code for the character S;

integer procedure ENTIER (real value X);

comment the integer i such that

$i <= X < i + 1$;

real procedure EXP (real value X);

comment e**X;

complex procedure IMAG (real value X);

comment the complex number $0 + X_i$;

real procedure IMAGPART (complex value Z);

comment the imaginary component of Z;

real procedure LN (real value X);

comment logarithm of X to the base e;

real procedure LOG (real value X);

comment logarithm of X to the base 10;

long real procedure LONGARCTAN (long real value X);

comment arctangent (radians) of X;

```
long real procedure LONGCOS (long real value X);
        comment cosine of X (radians);
long real procedure LONGEXP (long real value X);
        comment e**X;
long complex procedure LONGIMAG (long real value X);
        comment the long complex number OL + X_iL;
long real procedure LONGIMAGPART (long complex value Z);
        comment the imaginary component of Z;
long real procedure LONGLN (long real value X);
        comment logarithm of X to the base e;
long real procedure LONGLOG (long real value X);
        comment logarithm of X to the base 10;
long real procedure LONGREALPART (long complex value Z);
        comment the real component of Z;
long real procedure LONGSIN (long real value X);
        comment sine of X (radians);
long real procedure LONGSQRT (long real value X);
        comment the positive square root of X;
integer procedure NUMBER (bits value X);
        comment integer with two's complement
                representation X;
logical procedure ODD (integer value N);
        comment the logical value
                N rem 2 = 1;
real procedure REALPART (complex value Z);
        comment the real component of Z;
integer procedure ROUND (real value X);
        comment the value of the integer expression
                if X < 0 then TRUNCATE (X-0.5)
                         else TRUNCATE (X+0.5);
real procedure ROUNDTOREAL (long real value X);
        comment the properly rounded value of X;
real procedure SIN (real value X);
        comment sine of X (radians);
real procedure SQRT (real value X);
        comment the positive square root of X;
integer procedure TIME (integer value N);
        comment   N    Result
```

1	Returns as an integer value the time of day, in sixtieths of a second.
2	Returns as an integer value the elapsed processor time of the program, in sixtieths of a second.
3	Returns as an integer value the elapsed I/O time of the program, in sixtieths of a second.
4	Returns as an integer value the contents of a 6-bit machine clock that increments every sixtieth of a second.
11	Same as TIME (1), except time is expressed in multiples of 2.4 microseconds.
12	Same as TIME (2), except time is expressed in multiples of 2.4 microseconds.
13	Same as TIME (3), except time is expressed in multiples of 2.4 microseconds.
14	Returns as an integer value the elapsed time since the last HALT/LOAD, in multiples of 2.4 microseconds;

```
integer procedure TRUNCATE (real value X);
      comment the integer i such that
            |i| < = |X| < |i| + 1 and i * X > = 0;
```

7. SYNTACTIC ENTITIES WITH SECTION NUMBERS

Appendix C

COMPILE-TIME OPTIONS

The user is provided with compile-time ability to control the manner in which the compiler processes the source input that it accepts. The compiler control statement is entered into the compiler by cards, containing only compiler control information, in the same manner as source language statements and can occur at any point in the compiler input files.

An option control card is recognised by the appearance of a dollar sign ($) in the first or second column of the card. If the $ is in column 2, the option control card image is placed in the updated symbolic file if such a file is generated. An option control card with no compiler information causes the card image in the secondary input file that has the same sequence number, to be ignored.

Compiler options are invoked by the appearance of their names on an option control card. Two states are associated with the majority of options: set and reset. Default states are assigned to these compiler options and the desired state of such an option can be specified on an option control card. The balance of options are parameter options with which no states are associated.

OPTION CONTROL CARDS

Syntax

```
<option control card> ::= $<option list>
<option list> ::= <empty>|
                  <option action> <option>|
                  <option list> <option>
```

```
<option action> ::= <empty>|
                    POP|
                    RESET|
                    SET
<option> ::= CHECK|CODE|<dump option>|FORMAT|<goto option>|
             LIST|LISTDELETED|LISTOMITTED|LISTP|
             MERGE|NEW|NEWSEQERR|OMIT|<outer level>|
             PAGE|SEQ|SEQERR|SINGLE|TIME|<user option>|
             VOID|VOIDT|$|<parameter>
<dump option> ::= DUMP <dump value>
<dump value> ::= <integer number>
<goto option> ::= <go part> <sequence number>
<go part> ::= GOTO|GO TO
<sequence number> ::= <integer number>
<outer level> ::= LEVEL <integer number>
<user option> ::= {word used for specific user option}
<parameter> ::= <sequence increment>|
                <sequence base>
<sequence increment> ::= + <integer number>
<sequence base> ::= <integer number>
```

Semantics

DUMP (default RESET)

The DUMP option causes the printout of internal compiler tables, depending on the value of the <dump value>.

0 - all
1 - betwpass
2 - nametable
3 - tree
4 - betwpass and nametable
5 - betwpass and tree
6 - nametable and tree

For any missing <dump value> or any illegal value then value of 0 is taken.

FORMAT (default RESET)

If the FORMAT option is SET while the LIST option is SET, the printout is spaced to the top of the next page after each procedure in the input printout. This aids readability.

For the semantics of the other options the reader should refer to Burroughs B6700/B7700 ALGOL Language Reference Manual, Appendix D.

Appendix D

ERROR MESSAGES

PASS ONE ERROR MESSAGES

All pass 1 error messages are of the form:

ERROR 1xxx NEAR COORDINATE yyyy - message.

yyyy corresponds to one of the coordinate numbers in the first column on the program listing. If there is more than one statement on a card, only the coordinate of the first statement is listed. Some messages are only warnings, in which case the fixup action taken is listed below, and the program proceeds to pass 2.

The messages are:
(Note errors 1000 to 1006 inclusive are all to do with compile-time table sizes. These are actually set at the specified maximums. It would be hoped that these messages would never occur, only experience with the compiler will indicate this.)

1000 ERROR TABLE OVERFLOW

Maximum number of error messages is 75. Something is drastically wrong with the program. To save time and paper the rest of the program is ignored.

1001 TOO MANY RECORD CLASSES

This is actually an overflow of the record class list array which has a maximum number of entries of 65535.

1002 ID TABLE OVERFLOW

Maximum of 65535.

1003 TABLE OF ID POINTERS OVERFLOW

Maximum of 65535. If most of the identifiers are short (less than 4 characters) this table may fill up before the id table.

1004 NAME TABLE OVERFLOW

Maximum of 65535.

1005 TEMPORARY NAME TABLE OVERFLOW

More than 65535 identifiers in current unclosed blocks.

1006 BLOCK LIST OVERFLOW

Maximum of 513.

1007 UNEXPECTED END OF INPUT

End of input encountered before an END matching each BEGIN. The coordinate indicated may be two or three more than the last coordinate in the listing. Check the block numbers in the second column of the program listing.

1008 WARNING:ILLEGAL CHARACTER

A strange character accidently keypunched (or overpunched). It is likely that the character will print as a blank, so it may be necessary to inspect the card.

Fixup:treated as a blank.

1009 WARNING:UNEXPECTED "."

An apparently final "." before expected, such as in a constant with an inadvertant space:. 123.

Fixup:treated as a blank.

1010 WARNING:EXPONENT LARGER THAN 5 DIGITS

Exponent in a constant is too large.

Fixup:exponent treated as 0.

1011 WARNING:EXPONENT UNDERFLOW

Exponent in a constant is too small.

Fixup:exponent treated as 0.

1012 WARNING:EXPONENT OVERFLOW

Exponent in a constant is 5 digits but too large.

Fixup:exponent treated as 0.

1013 WARNING:UNEXPECTED "'"

Fixup:treated as a blank.

1014 WARNING:INTEGER TOO LARGE

Integer constant too large.

Fixup:treated as 0.

1015 WARNING:MISSING "TO"

Missing TO after GO.

Fixup:supplied.

1016 WARNING:INVALID BITS LENGTH

(a) "#" not followed by hex digits.

(b) "#" followed by more than 12 hex digits.

Fixup:replaced with #0.

1017 WARNING:INVALID STRING DECLARATION

(a) STRING (n) where n is not a number

(b) STRING (0) or STRING (> 256)

Fixup:treated as STRING (24).

1018 WARNING:MISSING ")"

STRING (n with no closing ")".

Fixup:supplied.

1019 WARNING:MISSING "("

REFERENCE not followed by a "(".

Fixup:supplied.

1020 WARNING:UNMATCHED END (DELETED)

An END encountered with no matching BEGIN.

Check the block numbers in the second column of the program listing.

Fixup:END deleted.

1021 WARNING:INVALID STRING LENGTH

(a) A string constant of length > 256. The closing string quote has probably been omitted.

(b) An empty string constant ("").

Fixup:replaced by "?".

1022 WARNING:TOO MANY DIGITS

More than 256 digits in a digit sequence.

Fixup:treated as 0.

1023 WARNING:ID LENGTH > 64

One of the identifiers in the program is too long.

Fixup:truncated to first 64 characters.

1024 WARNING:MISSING FINAL "."

Program not terminated by ".". The coordinate indicated may be two or three more than the last one on the listing.

May occur if the program ends with an unterminated string constant or comment.
Fixup:supplied.

PASS TWO ERROR MESSAGES

All pass 2 error messages have the format:

```
ERROR 2xxx NEAR COORDINATE yyyy - message
      (FOUND NEAR "...")
```

yyyy corresponds to one of the coordinate numbers in the first column on the program listing. "..." is a pair of symbols in the program text being scanned at the time the error is detected, which may be somewhat after the actual point of error. In general, the first symbol terminates the phrase in which the error was detected, the second is the next symbol to be scanned.

If any pass 2 error messages occur (other than warnings), then compilation stops at the end of pass 2.

The messages are:

2000 ERROR TABLE OVERFLOW
  Something is drastically wrong with your program. To save time the rest of your program is ignored.
  Maximum is 75.

2001 xxxxxx CANNOT FOLLOW yyyyyy HERE
  There are no legal programs in which xxxxxx and yyyyyy can be written together. A semi-colon, a comma, or an operator may have been omitted.

2002 INCORRECT PARENTHESIZATION
  This often occurs in conjunction with 2020 or 2021. Usually, additional parentheses are required in the expression.

2003 WARNING: ";" SHOULD NOT FOLLOW EXPRESSION
  In BEGIN ... expression; END, the semi-colon is incorrect.
  Fixup:";" ignored.

2004 SYNTAX ERROR

This is a "catch-all" message that is produced when the compiler cannot find anything more meaningful to say. The current context will point to the part of the program being parsed when the error was detected, but in general the real error may be much earlier in the program. If the current context is at or near a semi-colon and no errors can be found there, look at the beginning of the statement which ends at that semi-colon. If the current context is at or near an END, look at the corresponding BEGIN. For example, if ELSE BEGIN ... END; occurs, but not after an IF, the compiler will not detect the error until it reaches END;.

2005 INCOMPATIBLE NUMBER TYPE

In most cases an integer number is required.

2006 xxxxxx IS UNDEFINED

The variable or label xxxxxx has not been declared in the current block or in one containing it.

2007 ILLEGAL ATTRIBUTE

Illegal initial attribute in a file declaration.

2008 SYNTAX ERROR IN ATTRIBUTE LIST

Syntax error in initial attribute list in a file declaration.

2009 ILLEGAL MNEMONIC

Illegal mnemonic in initial attribute list in a file declaration.

2010 INCOMPATIBLE MNEMONIC

Mnemonic not recognized for attribute used.

2011 SYNTAX ERROR IN READ/WRITE STATEMENT

2012 INCOMPATIBLE IDENTIFIER

Identifier of wrong type.

2013 INDEX OF ARRAY OR STRING MUST BE INTEGER

(a) In S(x|y), x is not an integer expression.

(b) In Array id (...x...), x is not an integer expression.

2014 WARNING:NUMBER IN EDITING SPECIFICATIONS TOO LARGE

Number > 2**39-2

2016 ILLEGAL LIST ELEMENT IN READ/WRITE STATEMENT

2017 TOO MANY DIFFERENT LITERALS IN PROGRAM

No more than 16383 different constants are allowed.

2018 MORE THAN ONE DECLARATION OF xxxxxx IN THIS BLOCK

The variable xxxxxx has been declared more than once in the same block.

2019 IDENTIFIER MUST BE RECORD CLASS ID

In a declaration REFERENCE (xyz), xyz is not the name of a record class.

2020 INCORRECT OPERAND TYPE FOR xxxxxx

xxxxxx is a unary operator.

(a) LONG is applied to something which is already LONG, or to STRING, BITS, LOGICAL, or REFERENCE.

(b) SHORT is applied to something which is neither LONG REAL nor LONG COMPLEX.

(c) ¬ (not) is applied to something which is neither LOGICAL nor BITS.

(d) Prefix + or - applied to something which is LOGICAL, STRING, BITS, or REFERENCE.

(e) ABS applied to something which is LOGICAL, STRING, BITS, or REFERENCE.

(f) In Record variable (x), x is not a REFERENCE.

(g) In FOR I:=x,...,x is not an integer expression.

(h) In various other contexts, an INTEGER or LOGICAL operand is required.

2021 INCORRECT OPERAND TYPE(S) FOR xxxxxx

xxxxxx is a binary operator. Even when the error is in the first operand, the error is detected after both operands are inspected.

(a) AND or OR applied to expressions which are not both BITS or both LOGICAL. This case often happens in an IF statement when necessary parentheses are left out:

```
IF X < Y OR Z = 3 THEN ...
```

As written, Y is to be ORed with Z before anything else is calculated. Try instead:

```
IF (X < Y) OR (Z = 3) THEN ...
```

(b) A relational operator (like >) is applied to something which is COMPLEX, LOGICAL, or REFERENCE.

(c) SHL or SHR is applied to something which is not BITS, or the shift amount is not INTEGER.

(d) In x IS Recordclass, x is not a REFERENCE.

(e) In x**y,x is LOGICAL, STRING, BITS, or REFERENCE, or y is not INTEGER.

(f) In a FOR statement, the UNTIL expression is not INTEGER.

(g) In various other contexts, an INTEGER operand is required.

2022 INCORRECT NUMBER OF FIELDS

In creating a record, too many initial values have been specified.

2023 SIMPLE VARIABLE USED INCORRECTLY

In x(, x is a simple variable and not STRING

2024 INCORRECT STRING LENGTH

In S(x|y), y is negative, zero, or greater than 256.

2025 INCOMPATIBLE STRING LENGTHS

(a) In STRING1 := STRING2, STRING2 is longer than STRING1.

(b) In STRING3 (x|y), y is larger than the declared size of STRING3.

(c) A long string has been passed to a shorter formal string parameter.

2026 ARRAYS USED INCORRECTLY

A simple variable must be used here.

2027 INCORRECT DIMENSION

(a) The number of dimensions of an actual parameter does not equal the number of dimensions declared for the corresponding formal parameter.

(b) The wrong number of subscripts have been used in an array element reference.

(c) Dimensions in virtual parameters don't agree.

2028 EXPRESSION MISSING IN PROCEDURE BODY

A function PROCEDURE must have its final value specified by an expression standing alone immediately before the END.

2029 PROPER PROCEDURE ENDS WITH AN EXPRESSION

A procedure which returns no value nonetheless ends with an expression. This sometimes happens when a final assignment statement has been mis-punched, e.g. A=B, instead of A:=B.

2030 IMPROPER COMBINATION OF TYPES

Mixing incompatible types as alternatives of a conditional or case expression.

2031 LEXICAL LEVEL EXCEEDS 31

Non-trivial blocks, i.e. blocks with declarations, are nested too deeply.

2032 MISMATCHED PARAMETER

(a) A procedure call is passing an actual parameter which is not of the same type as the formal parameter in the procedure declaration.

(b) Virtual parameters not of the same types.

2033 INCORRECT NUMBER OF ACTUAL PARAMETERS

(a) The number of actual parameters in a procedure call does not equal the number of formal parameters in the procedure declaration.

(b) The number of virtual parameters do not agree.

2034 INCOMPATIBLE REFERENCES

A reference variable refers to a wrong record class.

2035 RESULT PARAMETER MUST BE A VARIABLE

In a procedure declaration, a formal parameter is declared ... RESULT xyz, but a call to that procedure has passed an expression which is not a variable.

2036 ASSIGNMENT INCOMPATIBILITY

An attempt to assign an expression of one type to a variable of a different type, or pass an actual parameter to a formal parameter of a different type. The only automatic conversions allowed are INTEGER to REAL, INTEGER to LONGREAL, REAL to/from LONGREAL, INTEGER/REAL/LONGREAL to COMPLEX/LONGCOMPLEX, COMPLEX to/from LONGCOMPLEX.

PASS THREE ERROR MESSAGES

Pass 3 error messages are of the form:

ERROR 3xxx NEAR COORDINATE yyyy - message

yyyy corresponds to one of the coordinate numbers in the first column on the program listing.

Unless a warning error, compilation terminates immediately on a Pass 3 error.

Messages are:

(Note messages not shown are compiler error messages).

3000 ERROR TABLE OVERFLOW

3001 DISPLACEMENT TOO BIG

3002 TOO MANY STACK CELLS AT THIS LEVEL

3003 PROGRAM SEGMENT TOO LARGE

3004 PROGRAM TOO LARGE

3006 COMPILE AND GO ILLEGAL WITH THIS PROCEDURE

3007 WARNING:PROCEDURE VALID FOR BINDING ONLY

For other use recompile at level 2.

3008 PERIOD EXPECTED ENDING STRING

String in initial attributes should end with a period

3009 ILLEGAL ATTRIBUTE VALUE

3010 LOWER BOUND EXCEEDS UPPER BOUND

In an array declaration an upper bound is less than the specified lower bound.

3012 SUBSCRIPTS MUST PRECEDE ASTERISKS

In a generalised subarray A(n,*) the specified subscripts must precede the asterisks.

3013 NOT ENOUGH SUBSCRIPTS

A generalised subarray is being used with not enough specified subscripts, i.e. too many asterisks.

3014 TOO MANY SUBSCRIPTS

A generalised subarray is being used with too many specified subscripts, i.e. not enough asterisks.

3015 TOO MANY STATEMENTS

Only 1280 statements are allowed in a Case Statement.