CS 52

LECTURE NOTES ON A COURSE IN SYSTEMS PROGRAMMING

BY

ALAN C. SHAW

ERRATA SHEET INCLUDED

TECHNICAL REPORT NO. 52
DECEMBER 9, 1966

These notes are based on the lectures of Professor Niklaus Wirth which were given during the winter and spring of 1965/66 as CS 236a and part of CS 236b, Computer Science Department, Stanford University.

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

January 15, 1967

ERRATA in

ALAN C. SHAW, LECTURE NOTES ON A COURSE IN SYSTEMS PROGRAMMING

CS TR No. 52, Dec. 9, 1966

---

p. 17, line 5, read "S_{i+1}" for "S_{i+k}" .

p. 34, line 4, read "operand[0:m];" for "operands[0:m];" .

p. 35, line 3, read "real or" for "real of" .

p. 39, line -7, read "KDF-9" for "KDK-9" .

p. 50, line -8, read "careful" for "cardful" .

p. 5% add an arrow from "Data Channel" to "Memory" in the diagram.

p. 74, last box on page, read "TSX ↑, 4" for "TSX 4, ↑" in all cases.

p. 75, diagram, read "SQRT" for "SØRT" .

p. 86, line 10, append "s2 := s2 + a[j] × b[j];" ,
line -10, read "C[1:ℓ,1:n];" for "C[1:ℓ,1:m];" .

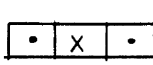
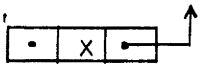
p. 91, last paragraph, replace by

"Dijkstra[11] has developed a solution to the more general problem where there are n processes, instead of only 2, operating in parallel. See Knuth[13] for a discussion and extension of Dijkstra's solution.".

p. 100, left, insert between lines -6 and -7, "TRA EQ" .
left, line -2, read "1" for "7" .

p. 105, Example, read ",.. A ×" for ",A× ".
LDQ CLA

p. 117, second diagram, read

"  " for "  " .

p. 119, line -7, read "u,v (possibly empty)" for "u,w (possibly emtpy)".

p. 120, line 1, append to-first sentence
"where the elements of $\mathcal{P}$ are of the form:

$$U \rightarrow \ldots \neq \ldots U \in V,\ x \in V^*) \ .\ ".$$

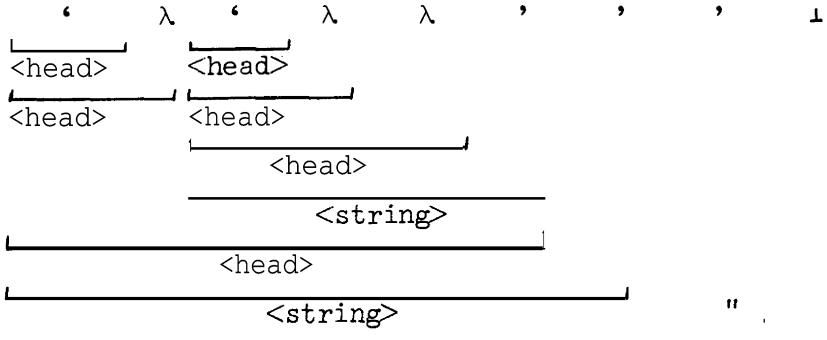
CS 52 ERRATA, Alan C. Shaw

p. 131, lines 1 and 2, interchange "iterative" and "recursive".

p. 136, replace-program by

" $S_0 := P_0$; i := 0; k := 1;
while $P_k \neq$ '⊥' do
begin i := j := i+1; $S_i := P_k$; k := k+1;
while $S_i \gtrdot P_k$ do
begin while $S_{j-1} \doteq S_j$ do j := j-1;
S_j := Leftpart(S_j . . . S_i); i := j
end
end

p. 137, replace (b) by



"

line -2, read "of ⊥<string>⊥ ." for "of ⊥<string> ."

p. 140, line -3, insert "The word "simple" is henceforth omitted.".

p. 147, lines 5 through 9, replace by
"directly reducible substrings (a) $S_i \ldots S_k$ and (b) $S_j \ldots S_\ell$. It follows from the definition of precedence relations that $S_{J-1} \lessdot S_j$ and $S_k \gtrdot S_{k+1}$. Now if $i < j$, then also $k < j$, since $i < j \leq k$ implies $S_{j-1} \doteq S_j$. If $i=j$ and $k \leq \ell$, then $k = \ell$, since $j \leq k < \ell$ implies $S_k \doteq S_{k+1}$." .

p. 151, line -13, read "conditional" for "condition" .

p. 154, line -7, read "<digit>" for "<digit" .

p. 179, insert between lines -4 and -5
"procedure Q(n); integer n;"

Lecture Notes on a Course in

SYSTEMS PROGRAMMING

December 9, 1966

These notes are based on the lectures of Professor Niklaus Wirth which were given during the winter and spring of 1965/66 as CS 236a and part of CS 236b, Computer Science Department, Stanford University.

Alan C. Shaw

SYSTEMS PROGRAMMING

I. INTRODUCTION

I-1. Advanced Programming[1,4]

In attempting to define "Advanced Programming", E. W. Dijkstra[1] described the purpose of the subject to be "Advancing Programming"; he stressed "those efforts and considerations which try to improve 'the state of the art' of programming, maybe to such an extent that at some time in the future we may speak of 'the state of the Science of Programming."' Until recently, the design of machines almost always preceded any serious thought about programming them; this had the unfortunate result that programming languages and translators had to be severely restricted to fit into the constraints imposed by machine designers. Programming beyond these restrictions succeeded only[1] "by using the machine in all kinds of curious and tricky ways which were completely unintended and not even foreseen by the designers." Programmers "have concocted thousands and thousands of ingenious tricks but they have made this chaotic contribution without a mechanism to appreciate or evaluate these tricks, or to sort them out."

Dijkstra's remarks were made in 1962. Since then, the situation has not changed significantly. New features, terminology, and "tricks" are continually being introduced with very few attempts to order or evaluate them in terms of a general framework or set of principles.

This is the challenge and function of Advanced Programming:

- to put order into the present chaos
- to develop useful principles of programming
- to apply these principles to programming languages, translators, and applications.

I-2. Purpose and Prerequisites of the Course

The intent of the course is to treat the design and implementation of Programming Systems in terms of some general principles that have been extracted from this field. Emphasis is on general methods rather than specific "tricks". It is assumed that the reader is familiar with the fundamentals of computer programming including:

(1) coding in machine, assembly, and higher-level languages, and

(2) the use of a supervisory or monitor system.[2]

Because of its important role in the evolution of language and compiler design and its usefulness as a vehicle for expressing algorithms, ALGOL 60[3] should be thoroughly understood. Most of the examples and algorithms discussed in this course are presented as ALGOL programs.

Systems Programs, such as assemblers, interpreters, compilers, and monitors can all be regarded as translators; from this point of view, Systems Programming is the science of designing and constructing translators. It is thus worthwhile at this point to examine the idea of a translator before looking into the specific details of various types of translators.

1-3. Translators

A translator can be viewed as a device which transforms an input string A into an output string B; schematically:

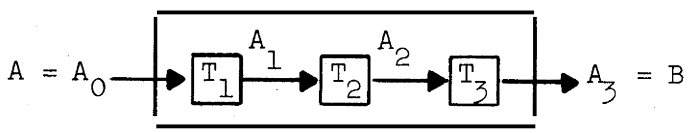
$$A \rightarrow \underset{a}{T} \rightarrow B$$

$$B := T(A)$$

Examples

A	B	T is called
(1) Binary Code (A_2)	Results (A_3)	Computer (or Interpreter) (T_3)
(2) Symbolic Code (A_1)	Binary Code (A_2)	Assembler (T_2)
(3) Phrase Structure Language (A_0)	Symbolic Code (A_1)	Compiler (T_1)

Multi-pass systems are those which require passes through several translators to produce the final output string. For example, the familiar translations-from compiler language to assembly language to binary code to computed results - can be represented:

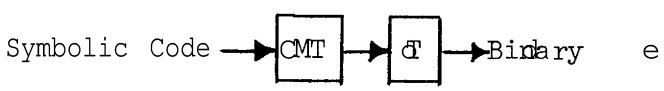


where the notation corresponds to the last example.

$$A_3 = B = T_3(T_2(T_1(A))) = T(A)$$

$$\text{where } T = T_3T_2T_1 .$$

Translators are often multi-pass systems internally but appear as single pass to the user. An assembler with "macro" facilities can be such an "invisible" multi-pass system.



Here MT is a macro translator which expands all macro calls in the input and T performs the basic assembly. A macro definition such as

MACRO X(Y, Z)(....Y....Z) ,
macro body

where X is the macro name and Y, Z are parameters, signals MT to replace macro calls in the input, such as X(A, B), by the "body" of the macro, substituting A for Y and B for Z in this example.

I-4. References

1. Dijkstra, E. W. Some Meditations on Advanced Programming. Information Processing 62, Popplewell, C. M. (Ed.) 535-538, North-Holland, Amsterdam, 1963.

2. Leeds, H. D. and Weinberg, G. M. Computer Programming Fundamentals. McGraw-Hill, New York, 1961.

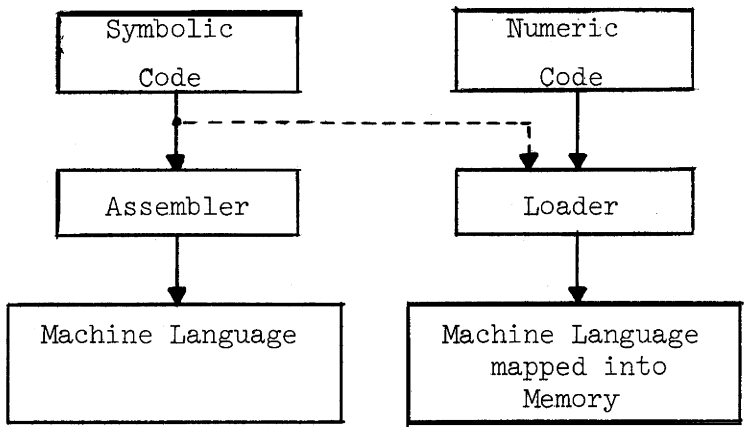
3. Naur, P. (Ed.) Revised Report on the Algorithmic Language ALGOL 60. Comm. ACM 6 (Jan. 1963), 1-17.

4. Barton, R. A Critical Review of the State of the Programming Art. Proc. Spring Joint Computer Conference 1963. 169-177.

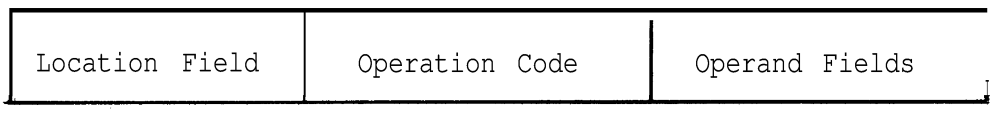
II. ASSEMBLERS

11-1. Basic Concepts

An assembler is usually understood to be a translator which produces machine language code as output from an input language which is similar in structure to the output; the natural symbolic units of the assembly language or input correspond to the natural units of the computer for which the assembly is intended. From another point of view, an assembler can be considered a sophisticated loader. A loader accepts numeric code containing machine language instructions, location addresses, relocation designators, and header information, translates this into directly executable code, and inserts the code into computer memory; this interpretation of an assembler is sketched below:



The form of an assembly language instruction, assembler record, or natural symbolic unit is:

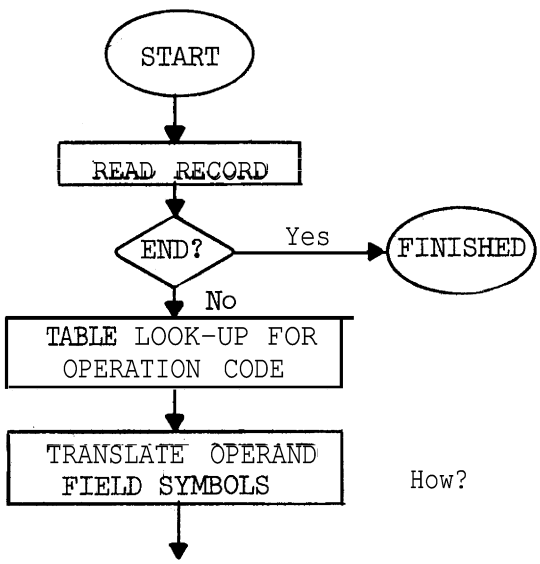


for example,

LOOP	CLA	RATE, 1

. This record corresponds to one machine language instruction. The operation codes are symbols defined by the assembler and correspond to machine operation codes; operand fields contain programmer-chosen symbols which are translated into machine memory addresses; non-blank location fields define the values of symbols. The basic task of an assembler is to establish the correspondences between programmer-chosen symbols and machine addres-

A record-by-record total translation fails in general because it is not possible to translate operand field symbols until the entire program has been scanned. This is illustrated in the following partial flow chart:



In order for the operand field symbols to have any value, they must appear in a location field; sequential total translation cannot be done

because location field definitions of symbols often follow their first appearance in the operand field. In the skeleton program:

```
(1)          BATE BSS 10
(2)          LOOP CLA RATE, 1
```

the assembler can assign the symbol RATE to the next open address at point (1); then, on reaching point (2), the assigned address for BATE can be correctly inserted. However, if the program is

```
(1)'         LOOP CLA BATE, 1
(2)'         BATE BSS 10
```

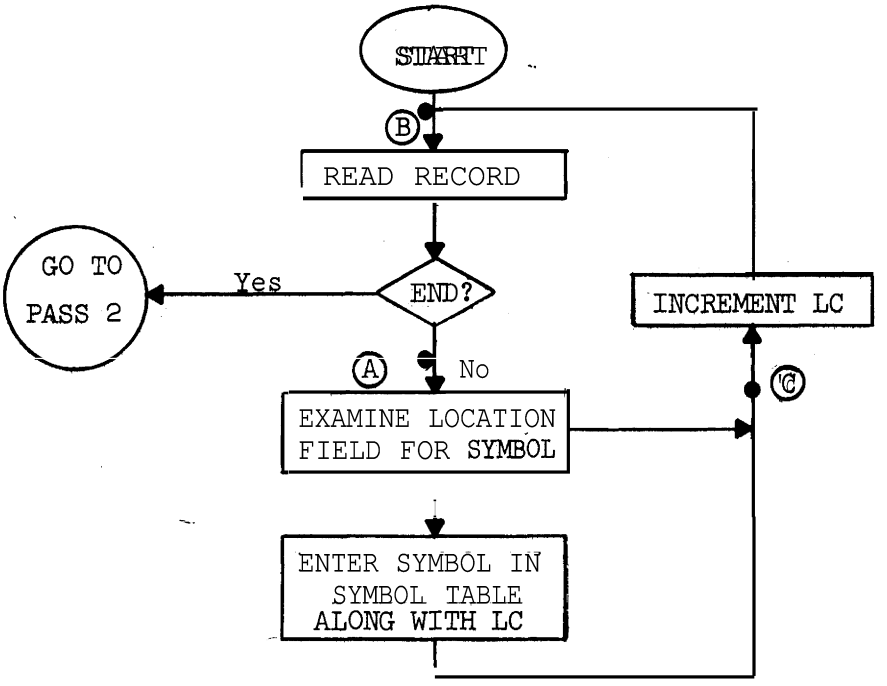
BATE has no value at point (1)' and complete translation of (1)' is impossible.

II-2. Multi-Pass Systems

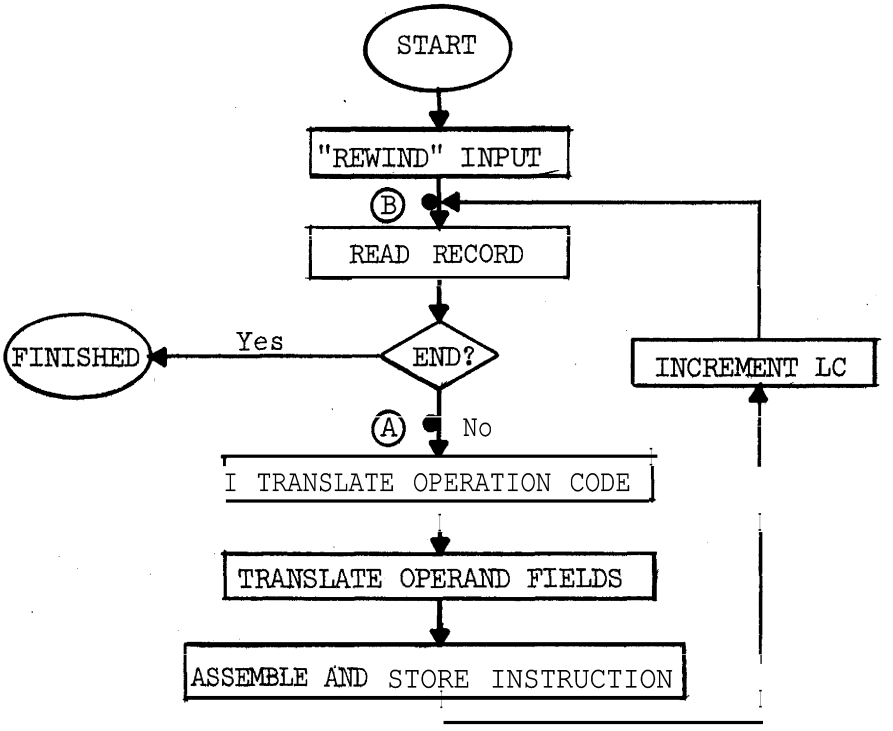
The simplest and most common solution to the above problem is to use a 2-pass system. The first pass assigns values (addresses) to all symbols. A location counter (LC) steps through the assembler records so that at each record, LC contains the address where the corresponding machine instruction will be located in computer memory (ignoring relocation); when a symbol is encountered in the location field, it is assigned the current value of LC. Symbols and their values are stored in a symbol table. Pass 2 performs a record-by-record total translation, referring to the symbol table for the values of location field symbols. A general flow chart of this method of assembly follows:

SIMPLE TWO PASS ASSEMBLER

PASS 1



PASS 2



These charts become more complex when the additional facilities provided by practical assemblers are inserted. These are the "pseudo-codes" or assembly instructions; they do not translate into executable code but are instructions to the assembler, for example, for the allocation of data and instruction space, and the assignment of values to symbols. using examples from the MAP Assembler for the IBM 7090/7094 computers,[1] the most important pseudo-operations are:

1. Location Counter Pseudo-Operations

These allow the programmer to control the operation of location counters, e.g.

	ORG	315

resets the location counter to 315 causing the assembler to start or continue the assembly from computer storage location 315.

2. Storage Allocation Pseudo-Operations

The instructions in this class reserve blocks of storage and increment the location counter to reflect this, e.g.,

MATRIX	BSS	25

assigns the current value of LC to the symbol MATRIX and increments LC by 25, effectively allocating a 25-word block of storage identified by the symbol MATRIX.

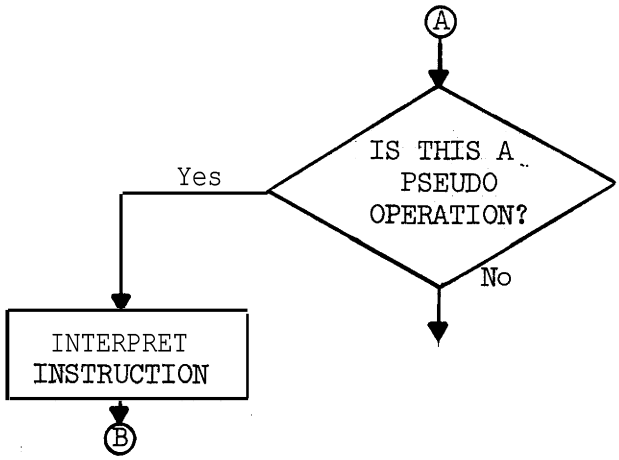
3. Data Generating Pseudo-Operations

These are used to define constants of various types, e.g.,

YEAR	DEC	1966

instructs the assembler to insert the decimal constant 1966 at the address defined by LC and to assign this address to the symbol YEAR.

The 2-pass assembler outlined above can handle these pseudo-operations by adding some blocks to its flow charts at points A in pass 1 and 2:



INTERPRET'INSTRUCTION usually involves incrementing LC and assigning a value to a symbol.

Most assemblers allow the use of actual operands or literals in the operand fields; for example, | | ADD | =1 | indicates that the operand field is to contain the address of the constant 1 after translation. The easiest way to translate literals within our 2-pass assembler is to invent symbols for them during pass 1 and add definitions of these symbols to the program; then, literals do not have to be considered in the second pass, e.g., | | ADD | =1 | is translated during pass 1 to | | ADD | ONE | ... | ONE | DEC | 1 | . At point C in the flow *chart, the block

TRANSLATE LITERALS

may be added.

Modern assemblers usually have a host of other features but most of these can be easily handled within the simple 2-pass system described here.

It is necessary at each pass of a multi-pass assembler to reread the source program. Small programs may be stored in main memory for the duration of the assembly. Systems allowing large programs usually write the source program on second-level storage such as magnetic tape or discs; the program must then be read from this storage at each pass. Partial or complete overlapping of processing and input-output operations can be accomplished by careful program organization; e.g., the following sequence enables process and input-output overlapping:

	Read	Process	Write
Record No.	i+1	i	i-1

Defining and translating symbols during assembly requires the building and searching of symbol tables. Since assemblers spend much of their time performing these functions, it is important to investigate efficient methods for table organization and searching.

II-3. Organizing and Searching Symbol Tables

Tables of all types have the general form:

Argument	Value
--------	----------
-----	--------
--------	--------

where the left-side is a list of arguments and the right side is a list of values associated with the arguments. Here, the arguments are symbols and the values are addresses.

II-3.1 Unordered Tables

The easiest way to organize a table is to add elements as they appear without any attempt at ordering. A table search requires a symbol by symbol comparison with each element in the table until a match is found; for a table of n elements, $\frac{n}{2}$ comparisons would have to be made on the average, before a match between the input and table arguments is found. This method has merit only for extremely small tables which are searched infrequently.

II-3.2 Ordered Tables

An ordered table is one in which (1) an ordering relation $>$ (or $<$) exists between any pair of arguments of the table, and (2) if S_i represents the ith element of the table, then for all i and j, $S_i > s_j$ if and only if $i > j$ (or $S_i < s_j$ if and only if $i < j$). The table is then ordered in ascending (or descending) sequence.

The most efficient general method for searching ordered tables is the binary search; starting with the complete table, the table subset under consideration is successively divided by 2 until a match is found. An ALGOL binary search procedure for a table ordered in ascending sequence follows:

```
procedure  Binary Search (S, n, arg, k);
value n, arg; integer array S[1]; integer  n, arg, k;
comment S is array of table arguments, n is length of table,
        arg is search argument,  S[k] = arg on return;
begin integer i, j;
     i := 1; j :=n;
   for k := (i+j) ÷ 2 while  S[k] ≠ arg do
   if S[k] > arg then j := k-1
        else i := k+1
end Binary Search
```

It is assumed that arg is in the table in the above program. A binary search requires $\log_2 n$ comparisons at most to search an ordered table of n elements. In order to find a match in a table of length $128(2^7)$, a binary search would require 7 comparisons at most while an element by element scan would require 64 comparisons on the average.

Instead of using one large table, it is sometimes more convenient to set up several smaller tables; for example, one could set up 26 tables for an assembler symbol table, each table corresponding to symbols starting with the same letter of the alphabet. The search then becomes a multi-level search; at the top level, the particular table is found and at the next level, the table is searched. In the above example of 26 tables, an even distribution of first letters of symbols over the letters of the alphabet would be necessary for efficient use of table storage. The advantage of multi-level schemes is that the relatively small tables may be searched very quickly; however, organization and

searching is more complex and use of storage is not always as efficient as the simpler 1-level system. These alternate methods have to be evaluated in terms of specific systems and goals in order to select the best method for a particular application.

II-3.3 Sorting

If an ordered table is desired, some type of sorting method must be employed to order the elements. There are many ways to sort a table or a file; sorting may be done internally in main storage or, when large files are to be sorted, with the aid of auxiliary storage devices such as tapes, discs, or drums. Only a few of the most important will be discussed here. Reference 2 contains a detailed presentation and evaluation of many sorting methods.

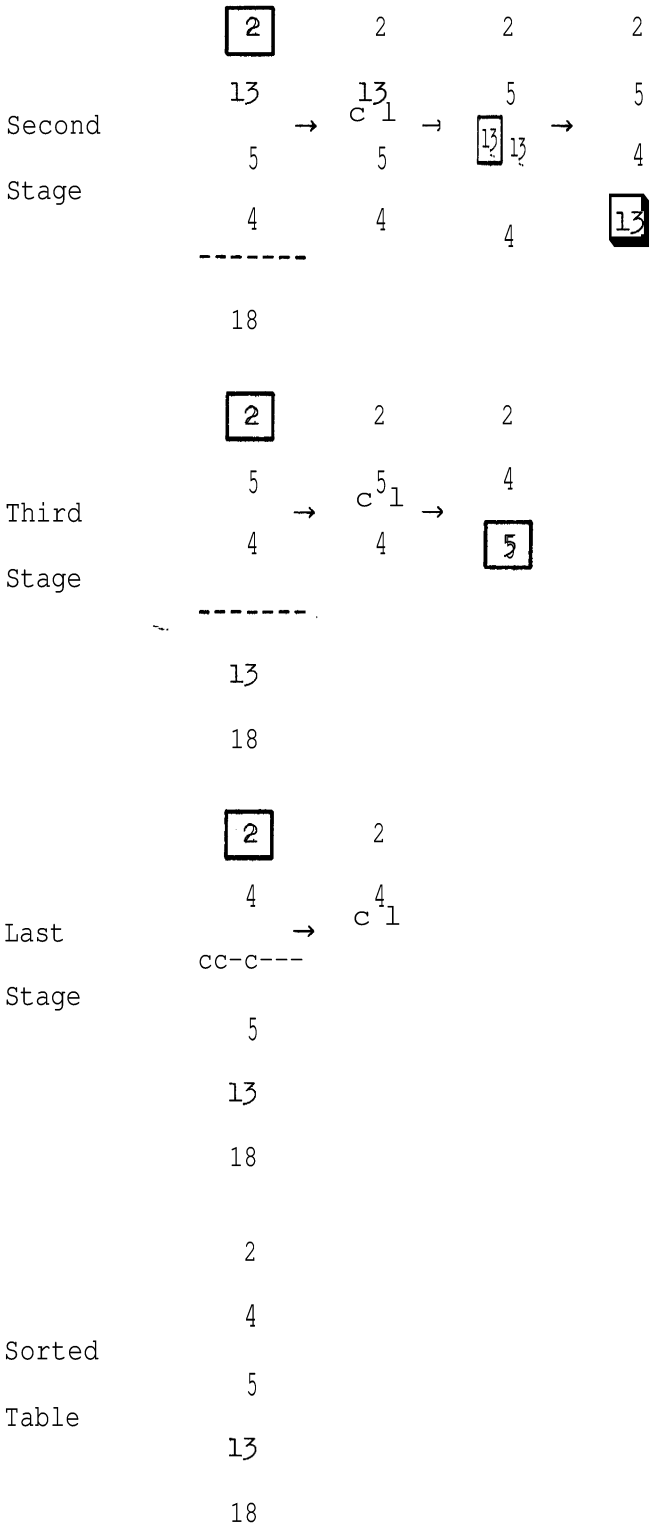
II-3.3.1 Bubble Sort

The basic idea is to successively pass through the unsorted part of the table, "bubbling" to the top the maximum (or minimum) unsorted *element; this is done by repeated comparisons and interchanges as illustrated in the following example:

To sort the table: 13 2 18 5 4

	[13]		2		2		2	
First	2	→	[13]	→	13	→	13	→ 13
Stage	18		18		[18]		5	5
	5		5		5		[18] 18	4
	4		4		4		4	18

c1



An ALGOL procedure for a simple Bubble Sort is:

```
procedure  Bubble Sort (S, n);
     value n;  integer array  S[1]; integer n;
     comment Bubble Sort sorts array  S[1:n] in ascending sequence;
     begin  integer i, j, k;  boolean tag;
            procedure  interchange  (X, Y);
                 value X, Y;  integer X Y;
                 begin  integer T;
                        T := X; X := Y; Y := T;
                        tag := true
                 end    interchange;
           tag := true;
           for j:= 1 step 1 while tag do
                        begin tag := false; k:= n-j;
                              for i := 1 step  1 until  k do
                                   if S[i] > S[i+1] then
                                        interchange  (S[i], S[i+1])
                        end
     end Bubble Sort
```

For fewer memory references, this may be modified to eliminate the interchanges; instead, the largest element of the unsorted table is found and interchanged with the top element at each stage.

II-3.3.2 Ranking by Insertion

Starting with an empty ordered table and a given unordered table, at each stage, the next element of the unordered table is inserted in

the correct position in the ordered table; this process is terminated when the original unordered table is empty. Thus, if $S_1\ S_2\ \ldots\ S_i\ S_{i+1}\ \ldots\ S_k$ represents the ordered table (ascending sequence) at the kth stage and the next element U of the unordered array is such that $S_i < U < S_{i+1}$, then U is inserted between S_i and S_{i+1}. $S_{i+1}\ \ldots\ S_k$ then have to be moved to make room for U. This block movement can be very inefficient unless the machine has a block transfer command. On the other hand, a binary search can be used to rank U and in the case of assembler symbol table construction, the table can be ordered continuously as it is built up. These features make the method useful for large symbol tables.

II-3.3.3 Other Common Methods

There are many other sorting methods in common use as well as variations of the above two methods. Other methods include the radix sort, various merge sorts, odd-even transposition, and selection sort.[2]

Sorting of a symbol table in a 2-pass assembler would occur at the end of pass 1 or beginning of pass 2.

II-3.4 Scrambling Methods

Scrambling or "hash addressing" is a fast method for converting symbols to addresses. Addresses are obtained by performing some simple arithmetic or logical operation on the symbol. For example, one method is to square the numeric representation of the symbol and select the central bits or numbers of the result as its table address; if a particular symbol, say XI, is represented numerically as 3275 and we wish

a 3-digit address, the computation would proceed as follows:

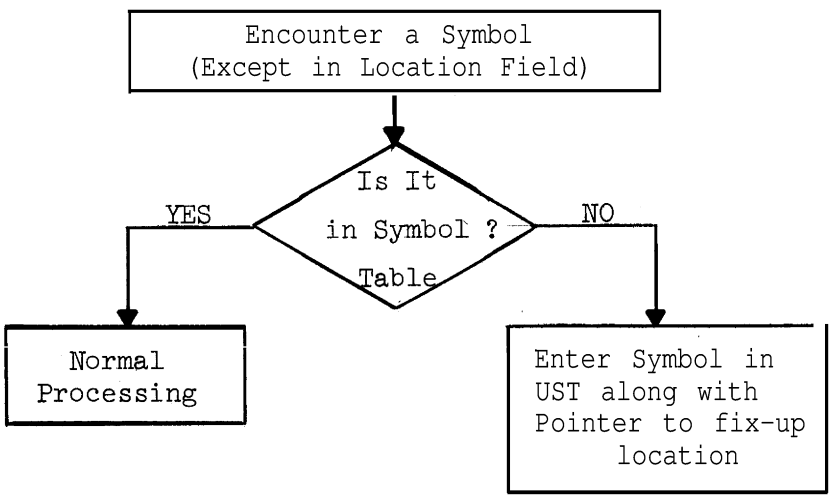
$$327\ 5^2 = 10725625$$

address of XI = 725

Care must be exercised to either prevent or account for non-unique mappings of identifiers and to use table storage efficiently; this work often negates the advantage of the fast address calculation.

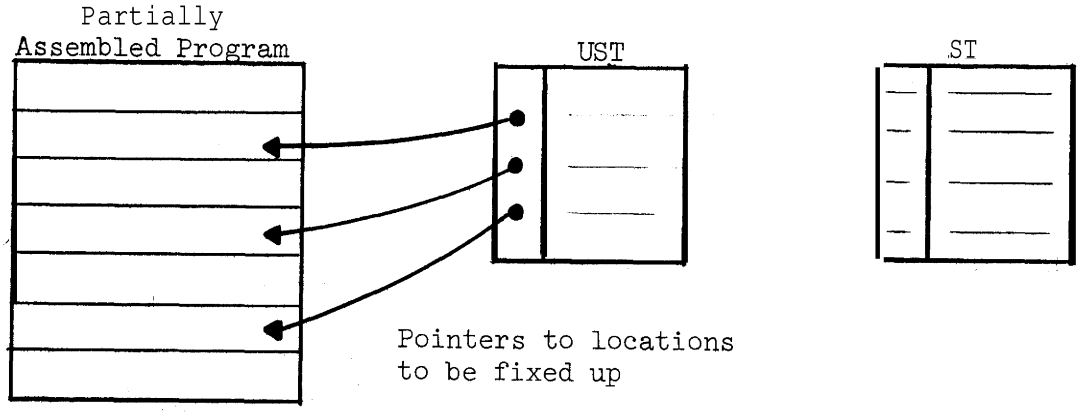
II-4. One-Pass Assembly

One-pass assembly can be accomplished despite the problem raised at the end of section II-1. The "forward reference" problem is solved by maintaining a list of undefined symbols with pointers to the locations where they are to be "fixed-up" upon definition. A flow of this scheme is

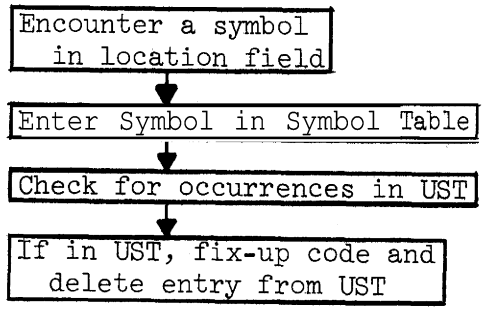


UST: Undefined Symbol Table

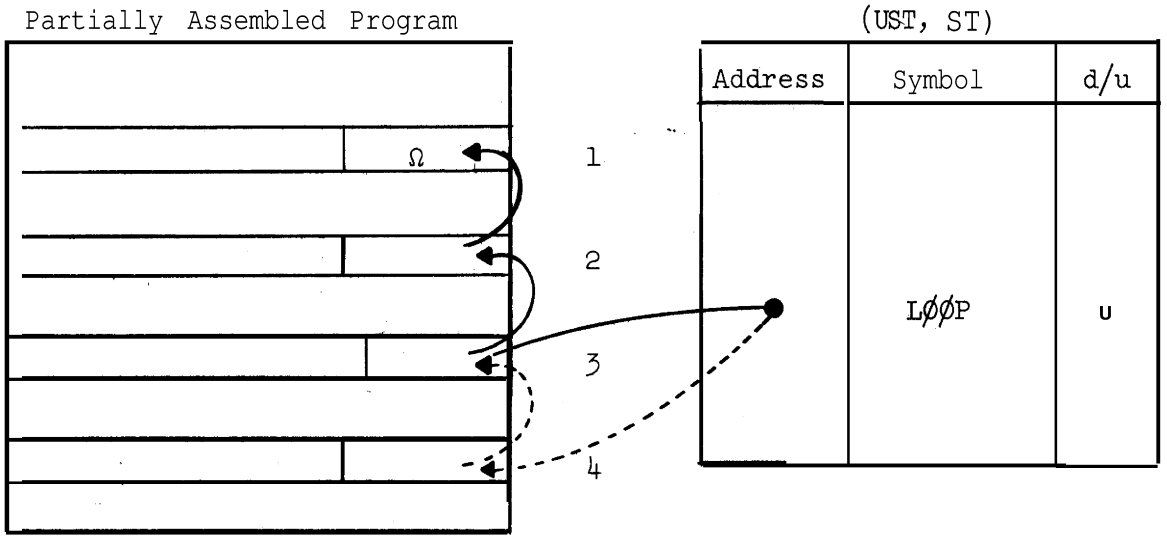
During assembly, a symbol table (ST) and UST are constructed:



On finding a symbol in the location field, the assembler flow is:



When the same undefined symbol is encountered more than once, a chaining method provides a convenient means for recording their appearances and for later fix-ups. Multiple appearances/of undefined symbols can then be recorded as below:



Address: Symbol Location

or

Fix-up Location

d/u : defined/undefined flag

The address part of the entry for the undefined symbol LØØP points to the last location seen by the assembler where LØØP appeared; pointers to 2 and 1 produce a chain through the earlier fix-up locations for loop. Ω (undefined) indicates the end of the chain. If LØØP again occurs at point 4 and is still undefined, the pointers change as indicated by the dotted lines and the pointer from the address part of LØØP to 3 is deleted. When LØØP is defined, its addresses are inserted in the places occupied by the chain pointers.

One-pass assembly has the advantage that the source program is read only once; this advantage is gained at the expense of more complex routines for handling symbols. The assembled program and various tables

must be stored in main memory during assembly or the above advantage over multi-pass systems no longer holds. Assemblers with block structure can be constructed conveniently by the one-pass method.

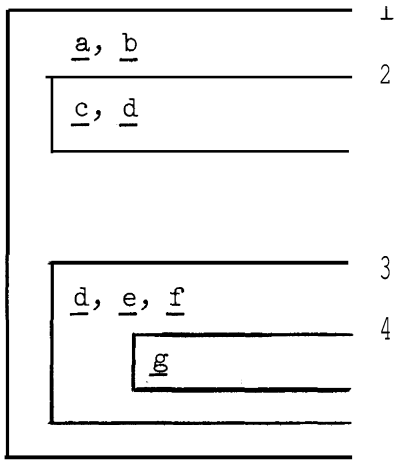
II-5. Block Structure In Assemblers

While few assembly languages have the block structure of ALGOL, it is still useful to study the implementation of block structure by assemblers for several reasons:

1. Many assemblers have limited forms of block structure, usually allowing symbols to have local and global significance. e.g., MAP programs may be structured through the use of the QUAL pseudo-operation
2. The basic methods employed by compilers to handle block structure can also be used for assemblers and thus can be illustrated in a less complex setting.
3. Many compilers translate source language into "intermediate" languages which retain the original block structure and are similar to assembly languages.

In general, a block is a delineated section of source language code having explicit or implicit declarations for some of the symbols used in the code; symbols may be declared explicitly by formal declarations (e.g., ALGOL identifiers) or implicitly by their use (e.g., ALGOL labels). Symbols defined within a block may only be referenced in that block.

Example:



This representa a program with 4 blocks, each having symbols defined within it. a and b may be referenced throughout the program; c and d are only "known" in block 2, d, e, and f in block 3 and 4, and g is known only in block 4. Note that the d in block 2 is different from the d in block 3; each has its own scope of validity.

The effect of a defined area of validity for symbols in assemblers is to allow sharing of symbol table storage among "parallel" blocks; in the above example, blocks 2 and 3 are in parallel. If opening and closing of blocks are indicated by left and right parentheses, the depth or level of a block in a program can be found by numbering the matched parentheses pairs; using our example again, we have

block No.		1		2			3		4			
		(a,	b	(c,	d)		(d,	e,	f	(g	)	))
block level		1		2		2	2		3	3	2	1

In a one-pass assembler, symbol table space may then be released on exiting from a block. On entering a block, a block marker is set; when leaving the block, the marker is reset to that of the last enclosing block. This scheme can be implemented by using the first symbol table entry for each block as a pointer to the previous block 'head" or entry.

Let ST[i] be contents of the ith symbol table entry and j be a pointer to the first symbol table entry of the current block. Then symbol table housekeeping can be done as follows:

```
                    i := 0; j := 0;

block entry:        i := i+1;
                    ST[i] := j;
                    j := i;
                      .
block exit:         i := j-1;
                    j := ST[j];
                      .
```

The evolution of the symbol table of our previous example is:

ST / i	ST_1	ST_2	ST^2	ST_3	ST_4	ST^4	ST^3	ST^1
1		0	0	0	0	0	0	0
2		a	a	a	a	a	a	a
3		b	b	b	b	b	b	b
4			1	(1)	1	1	1	(1)
5			c	(c)	d	d	d	(d)
6			d	(d)	e	e	e	(e)
7					f	f	f	(f)
8						4	(4)	(4)
9						g	(g)	(g)
10								
j=	0	1	4	1	4	8	1	1

ST_k is the symbol table at blockentry for block k; ST^k is the symbol table at blockexit for block k . Elements in parenthesis are in the table (because they haven't been destroyed) but inaccessible.

This method has to be modified to handle forward references. For the program with structure:

```
begin
    begin
        L          Use of L
        .
    end;
    L:             Declaration of L
end
```

the global identifier L is used in an inner block before it is declared in the enclosing block, On reaching block exit, all undefined symbols may be carried out into the enclosing block and filled in the symbol table; undefined symbols may then be treated correctly using the chaining and fix-up method described for one-pass assemblers. Care must also be taken in generating the correct reference in the following case:

```
begin
    L:                      First Declaration of L
    .
    begin
        go to L;            Use of L
        L:                  Second Declaration of L
    end
end
```

Here, the use of L refers to the L in the inner block (second declaration); possible forward references within a block have to be considered before treating symbols as global to that block.

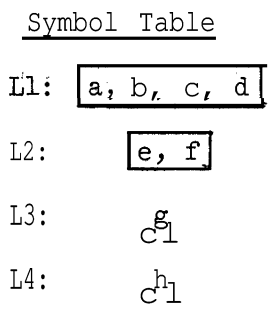
5e conventional two-pass assembler can be modified for languages with block structure properties by grouping the symbol table on a per block basis and maintaining a dictionary pointing to the symbol table blocks.

Example:

```
1: begin real a, b, c, d;
    2: begin real e, f;
       end;
       3: begin real g;
          4: begin real h;
             end
          end
    end
```

Dictionary

Block Number	Index to Symbol Table	Number of Entries in Block	Ancestor Block
2	L2	2	1
4	L4	1	3
3	L3	1	1
1	L1	4	0



Dictionary entries are made on exiting from a block. The symbol table can be one large table grouped on a block basis. To translate symbols during pass 2, the dictionary is searched with the block number as the search argument; from the dictionary entry, the pointer to the correct place in the symbol table is obtained. If a symbol is global, the ancestor entry of the dictionary which points to the enclosing block, can be similarly used.

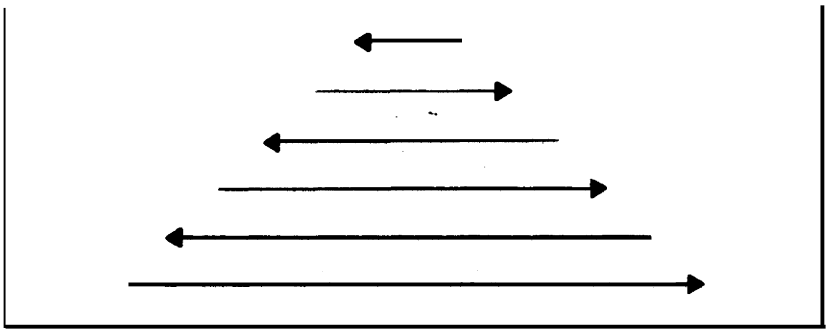
II-6. References

1. IBM 7090/7094 IBSYS Operating System, Version 13, Macro Assembly Program (MAP) Language. Form C28-6392-0. International Business Machines Corporation, 1963.

2. Papers presented at an ACM Sort Symposium. Comm. ACM-, 6, 5 (May 1963).

II-7. Problems

1. One useful variant of the bubble sort is to alternately pass through the table in both directions, bubbling the largest element in one direction and the smallest in the other.



Code this variant as an ALGOL procedure.

2.

Computer Science 236a
Winter 1966

N. Wirth
Due Date: Feb. 10

Term Problem I

Design an assembler according to the following description.

Input: One instruction per record (card), consisting of location fields (cols. 1-10), operation field (cols 12-14) and operand field (cols. 16-72).

output: Listing of assembled instruction in hexadecimal form along with instruction counter and given symbolic instructions.

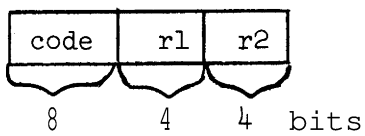
Symbols: Symbols are either names, literals or constants not containing blank characters. A name is a sequence of 1 to 10 letters or digits starting with a letter. A constant is a decimal integer, possibly preceded by a sign. A literal is a constant preceded by an equal sign (=). It denotes the address of any storage cell into which the constant is assembled.

Fields: The location field is blank or contains a name (left-adjusted in the field) in which case it is the definition of that name. The operation field must contain an instruction code (cf. Table 1), or an assembler instruction (left-adjusted in the field). The operand field is divided into two or three subfields depending on the form of the instruction. The subfields are separated by commas. A missing subfield is interpreted as 0.

Target code: An array of individually addressable 8 bit characters (bytes), listed in hexadecimal form, each character as a pair of hexadecimal digits.

Instruction Formats: Instructions are grouped into two categories to be translated into the following forms:

RR: Instruction occupies 2 bytes. Form of operand field is "r1, r2" where r1 and r2 are integers.



RX: Instruction occupies 4 bytes. Form of operand field is "r1, a2, r2" where r1, r2 are integers, and a2 is a symbol. --.

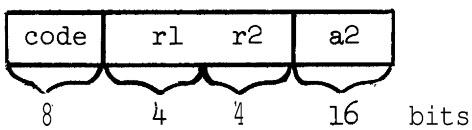


Table 1: Instruction codes

RR Form		RX Form	
Symbolic	Hexadecimal	Symbolic	Hexadecimal
AR	1A	A	5A
BCR	07	BAL	45
CR	19	BC	47
DR	1D	C	59
LR	18	D	5D
LCR	13	IC	43
MR	1C	L	58
SR	1B	LA	41
HLT	00	M	5C
		R	4A
		SL	48
		SR	49
		ST	50
		STC	42
		W	4B

Assembler Instructions:

1. Define name and increment location counter. Symbolic code: DS .

The name in the location field is defined and subsequently the location counter is incremented by the integer in the operand field. (The loc. counter addresses bytes.)

2. Set the location counter. Symbolic code: ORG . The location counter is set to the value of the constant in the operand field.

3. Terminate assembly and print the produced output in condensed hexadecimal form. Symbolic code: END .

Example of an assembly listing:

```
0000    41000000    START    LA      0,0
0004    41100000             LA      1,0
0008    41200190             LA      2,400
oooc    5A01001C    LOOP     A       0,ARRAY,1
0010    1A12                 AR      1,2
0012    591001AC             C       1,=400
0016    4720000C             BC      2, LOOP
001A    0000                 HLT
001C                ARRAY    DS      400
                             END
01AC    00000190
```

Notes:

1. Program the assembler in Extended ALGOL on the B5500 computer and test it. The program should contain comments to explain the main points and to facilitate the understanding of its principles. It is stressed that the program be presented in a neat and well structured form.

2. A few days before the due date, a sample program will be available to test the assembler. It is advised that the student test his program before that date with his own test cases.

3. At the due date, submit the program together with the output resulting from the distributed test case.

III. INTERPRETERS

III-1, Definition and Examples

Corresponding to each statement of a language, there exists an interpretation rule or action representing its meaning. An interpreter is a language translator whose primary task during translation is to perform the actions dictated by the meaning of the statements of the language, In more concrete terms, interpreters read and obey the statements of languages. By contrast, assemblers translate assembly language into another language which is later interpreted or obeyed.

Interpreters are commonly used in the following applications:

1. Simulation of real computers

A given computer can simulate the operation of another computer -- either a proposed computer or one already in existence. For example, the Burroughs B5500 can be simulated on the IBM 7090 and vice versa.

2. Simulation of hypothetical computers

Hypothetical machines are studied and used by simulating them on existing machines. Examples of such machines are the list processing machine (or language) IPL V and the "polish string" machines used by the early ALGOL compiler systems.

3. Interpretive Compilers

Instead of translating higher-level languages into machine language programs and then executing these programs, some systems execute the source language directly via an interpreter. LISP 1.5 on the IBM 7090 is such a system.

4. Simulation languages

Languages, such as SIMSCRIPT, SOL, and GPSS, which are designed to describe parallel processes, are often implemented on conventional sequential machines by interpreters.

5. Monitor systems

Control of batch processing, real-time, and time-sharing monitor systems is accomplished by user-written control instructions which are interpreted by the system.

Instead of using interpreters for the above, one could translate into equivalent machine language programs - as in assembler systems - and then execute these programs. Both approaches are employed. Interpreters are usually much easier to write, debug, and modify but can be extremely slow and wasteful of storage. For these reasons, interpreters are written 1. for research or exploratory purposes, 2. when the language is not used on a "production" basis, 3. for very complex systems, or 4. for a combination of the above.

This chapter examines interpreters of sequential computer code, as opposed to higher-level language interpreters or systems allowing parallel processing, The operation of typical von Neumann and stack machines are described via interpreter programs.

III-2. Basic Interpreter of Sequential Code

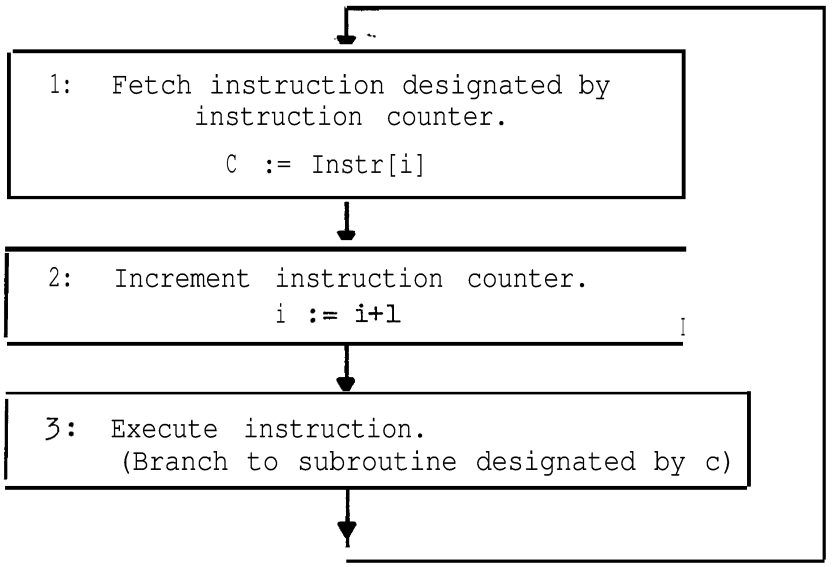
Let Instr = a vector containing the instruction sequence, such that

Instr[i] contains the ith instruction in the sequence,

i = instruction counter, and

C = current instruction.

The main loop of an interpreter of the program represented by Instr is:



Step 1 may be divided into several substeps by breaking Instr[i] into its component parts:

Instr[i][0] = operation code

Instr[i][1], Instr[i][2],..., Instr[i][n]

= operation parameters .

C is also divided into corresponding parts:

c[0], c[1], . , c[n] .

n=0 corresponds to a no-address computer;

n=1 corresponds to a 1-address computer;

n=2 corresponds to a 2-address computer;

etc.

Then, step 1 becomes:

```
c[0] := Instr[i][0];
c[1] := Instr[i][1];...; c[n] := Instr[i][n] ,
```

and step 3 may be expressed:

```
Execute(c[0](c[1], c[2],..., c[n])) .
```

111-3. Interpreter for a von Neumann Machine

These machines may be classified into (a) single address, single register computers and (b) multi-address and/or multi-register computers. In the former, operations on operands are performed in a single register, usually called the accumulator; for operations requiring two operands, the address of one is implicitly understood to be the accumulator while that of the other is contained in the instruction, e.g., IBM 7090, DEC PDP-1. In the latter, operations may be performed in one of several addressable registers and instructions may contain several addresses, e.g., IBM 360. An interpreter program for a simple single address, single register machine is presented below:

PROGRAM		REMARKS
integer array	operator, address[0:ℓ],	instr[i] = (operator[i], address[i])
	operands[0:m];	data memory
integer	op,	operation code
	adr,	operation address
	reg,	single register
	count;	instruction counter
	count := 0;	
1:	op := operator[count];	
	adr := address[count];	Fetch
2:	count := count + 1;	Increment instr. counter
3:	if op = 1 then	
	reg := operand[adr] else	Load
	if op = 2 then	
	operand[adr] := reg else	Store
	if op = 3 then	
	reg := reg + operand[adr] else	Add
	if op = 4 then	
	reg := adr else	Load immediate
	if op = 5 then	
	count := adr else	Transfer
	if op = 6 then begin if reg = 0	
	then u n t := adr end else	Conditional Transfer
	go to 1;	

While this program or a similar one may be adequate for some applications, there are several inaccuracies and omissions which must be corrected in order to precisely describe the operation of any real of hypothetical machine of this class:

1. The word length of the machine has been ignored.
2. Logical and arithmetic operations cannot be handled at the bit level since all variables are of type integer.
3. Data and program should reside in the same memory.

An interpreter for a binary computer can be written in ALGOL taking the above factors into account. The key change is to define all variables as type Boolean.

```
comment The computer has (n+1) words of memory M and word length of
        (ℓ+1) bits. Operation code, op, is (11 + 1) bits; operation
        address adr is (ℓ2 + 1) bits; (ℓ1 + 1) + (ℓ2 + 1) = ℓ + 1 .
        reg is a (ℓ+1) bit register and count is a (ℓ3 + 1) bit instruc-
        tion counter.  2↑(ℓ3 + 1) = n + 1;
Boolean array M[0:n, 0:ℓ], op[0:ℓ1], adr[0:ℓ2], reg[0:ℓ], count[0:ℓ3];
integer procedure number(x, k);
  Boolean array x[0]; integer k;
comment number treats the array x as a positive binary number of (k+1)
        bits and converts this to an integer;
  begin integer i, n;
        n := 0;
        for i := 0 step 1 until k do
        n := n × 2 + (if x[i] then 1 else 0);
        number := n
  end number;
```

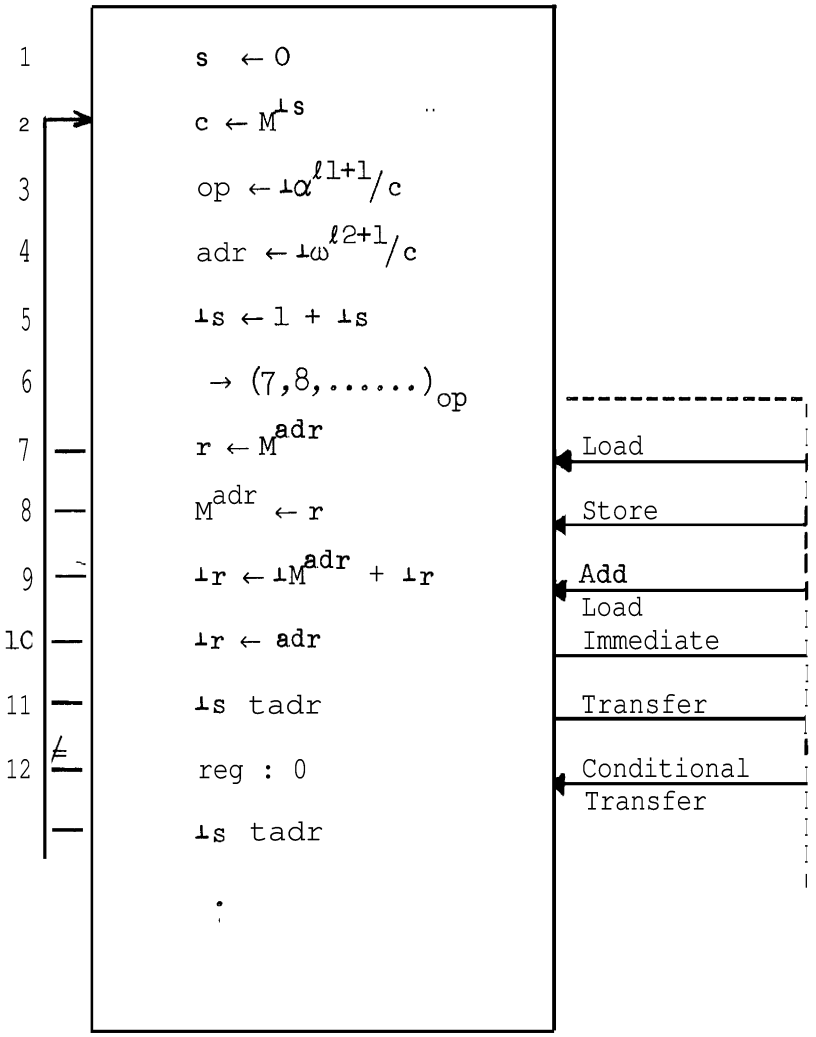
```
comment  initialize number(count, ℓ3) to 0;
for i := 0 step 1 until ℓ3 do count[i] := false;
comment begin interpreter cycle;
1:   n := number(count, ℓ3);
     for i := 0 step 1 until ℓ1 do op[i] := M[n, i];
     for i := 0 step 1 until ℓ2 do adr[i] := M[n, ℓ1 + 1 + i];
2:  n := n+1;
     binary(n, count, ℓ3);
comment the procedure, binary, converts the integer n to a ℓ3 + 1 bit
        binary number, count;
3:   if number(op, ℓ1) = 1 then
        begin    := number(adr, ℓ2);
             comment load;
             for i := 0 step 1 until ℓ do reg[i] := M[n, i]
        end else


        etc.
```

If the reader has followed this program, he is aware of the awkwardness of ALGOL for describing the operation of an interpreter at the bit level. Clearly, another language or notation is desirable. A powerful notation for this type of description is the Iverson language.[1] The following "Iverson" description of a single address, single register binary machine illustrates the elegance and power of the notation. (The reader should consult Reference 1 for more details on the notation and its application.)

Variable	Meaning;
M	computer memory
	$\nu(M) = \ell+1$ word length
	$\mu(M) = n+1$ no. of words in memory
r	register
	$\nu(r) = \ell+1$ register length
S	instruction counter
	$\nu(s) = 11 + 1,\ 2\uparrow(\ell 1 + 1) = n+1$
C	instruction register
	$\nu(c) = \ell+1$ instruction length

M, r, s, and c contain binary components. See next page for Iverson program.

Language interpretation can thus occur at different levels of detail. If the interpreter is testing the design of a new computer, then complete details of word length, radix, registers, handling of address and arithmetic overflows, etc. have to be included; on the other hand, interpretation at the level easily handled by ALGOL programs may be sufficient if the purpose of the system is to evaluate the usefulness or power of a particular language.



III-4. Polish String or Stack Organized Machines

Polish notation and stacks will be discussed further in Chapter VI. In this section, some of the basic ideas are introduced to illustrate the operation of stack machines.

The reverse or postfix polish form of a statement or expression of a language is obtained by reordering the elements of the expression so that operators appear after their operands rather than before or between them as is normally the case.

Examples

	Conventional Form	Reverse Polish
1.	a + b + c	a b + c +
2.	x := b X c↑d - e	xbcd↑ X e - :=
3.	a/(b↑c - e) + (e - f) xg	a bc↑e - / ef - g X +

(Note the elimination of parentheses)

This form of an expression is very convenient for compilation or interpretation and has led to the development of computer organizations that can handle reverse polish expressions easily. These are the stack-organized computers, such as the Burroughs B5500 or the English Electric KDK-9. They contain a stack or "cellar" which is a first-in, last-out store used for temporary storage of operands. Many of the instructions in such a machine have no address fields but implicitly refer to the top element or elements of the stack. "Pushing-down" or "popping-up" of the 'stack is performed automatically during instruction execution.

The following partial interpreter is for a machine with a stack mechanism:

PROGRAM

```
integer array  instr[0:ℓ],                  instruction sequence
               M[0:m],                      data memory
               S[0:n];                      stack or cellar
integer        op,                          operation code
               count,                       instruction counter
               s;                           stack pointer
   S := 0; count := 0;                      Initialization
1: op := instr[count];                      instruction fetch
2: count := count + 1;                      increment instr. counter
3: if op = 1 then
      begin    := s + 1;
          S[s] := M[instr[count]];          Load
          count := count + 1;
      end else
   if op   2 then
      begin M[instr[count]] := S[s];        Store
            S := s-1; count := count + 1
      end else
   if op = 3 then
      begin S[s-1] := S[s-1] + S[s]         Add
            S := s-1
      end else
        :
      etc .
```

Each word in instr[ ] is either an operation code or an operand; for Loads and Stores, the required address is in the word following that containing the operation code.

Using the expression in 2. from the reverse polish examples, the instruction sequence in the program format is:

```
i:          0  1  2  3  4  5  6  7  8  9  10  11
instr[i]:  l b, l c, l d  6  5  l e  4  2
```

where the operation codes have the meaning:

code:	1	2	3	4	5	6
meaning:	Load	Store	Add	Subtract	Multiply	Exponentiation

The stack contents are displayed below after each instruction is executed in this example:

S[] after	instr[1]	instr[3]	instr[5]	instr[6]	instr[7]	instr[9]	instr[10]	instr[11]
	b	b	b	b	b↑c↑d	b↑c↑d	b↑c↑d-e	
		c	c	c↑d		e		
			d					
s =	1	2	3	2	1	2	1	0

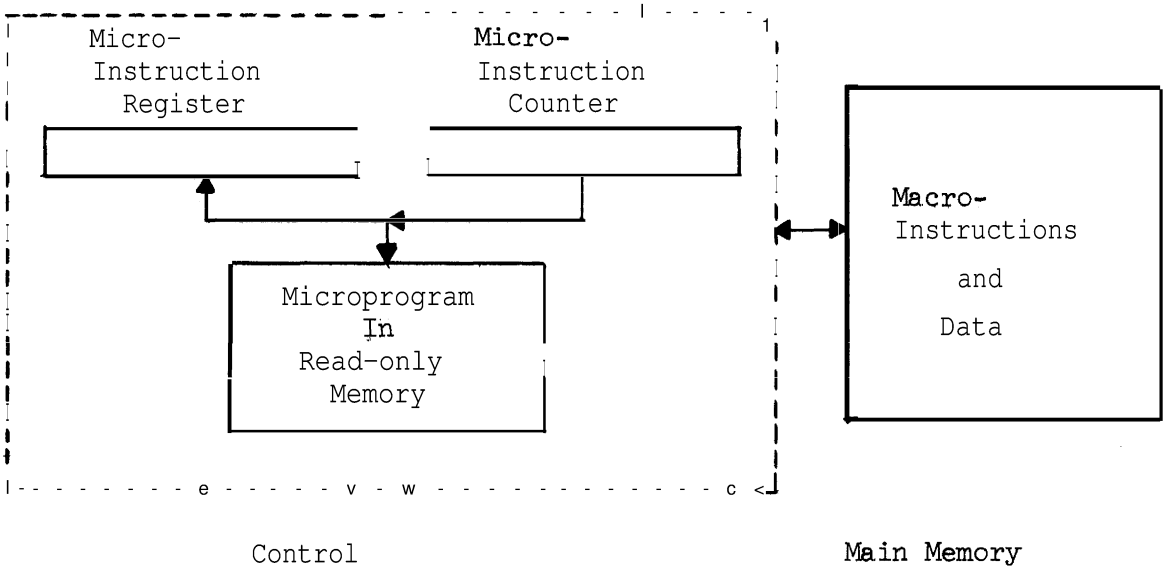
Because all operations are performed on elements of the stack, the stack access time must be small. Fast registers are therefore used for the top elements of a stack; since these are expensive, their number must be severely limited. This limitation causes major systems programming problems related to stack administration, stack overflow, and code optimization.

III-5. Interpretive Computers

The execution of machine language instructions by conventional computers occurs via an interpretive process. Instructions are translated

into mechanical or electrical operations, such as opening or closing 'data paths, setting and testing internal registers, switching memory cores, etc.

Recently, the interface between hardware and software interpretation has become less distinct. In many modern computers, machine language instructions are executed interpretively by microprograms which reside in a read-only memory in the control unit.[2] These microprograms translate machine code into microinstructions which are the basic executable instructions of the permanent hardware. Changes in machine language can be made by reprogramming the control unit to perform the desired translation. A schematic of the organization of such a computer is:



Operations at this level follow the same basic interpretive cycle as the other examples of this chapter.

III-6. References

1. Iverson, K. E. A Programming Language. Wiley, New York, 1962.

2. Fagg, J., Brown, J. L., Hipp, J. A., Doody, D. T., Fairclough, J. W., Green, J. IBM System/360 Engineering. AFIPS Conference Proceedings, Fall 1964, Spartan Books, Inc.

111-7. Problems

Computer science 236a
Winter Quarter, 1966

N. Wirth
February 2, 1966

Term Problem II

Construct an interpreter which represents a computer with the following specifications:

The computer consists of

1. A memory consisting of 4096 consecutively addressed bytes, each byte consisting of 8 bits;
2. 16 registers, each with 32 bits;
3. A condition register, able to represent 4 distinct states;

4. An instruction register, (32 bits);

5. An instruction counter (12 bits).

Instructions have the formats as indicated in Term Problem I, and cause the following actions to be taken:
(To identify-an instruction, the mnemonic codes of Term Problem I are used, the instruction parameters are denoted by rl, r2, a2 .)

Group 1:

These instructions have an RR and an RX version. They designate two operands, the first of which is the register designated by rl . The second operand is the register designated by the r2 parameter in the RR case, or the consecutive four bytes of memory, the first of which is designated by the sum of a2 and the value of register r2 .

Instruction	Code	Meaning
Add	A, AR	01 := 01 + 02
Subtract	S, SR	01 := 01 - 02
Multiply	M, MR	01 := 01 x 02
Divide	D, DR	01 := 01 / 02
Load	L, LR	01 := 02
Compare		Condition register := $\begin{Bmatrix}0\\1\\2\end{Bmatrix}$, if 01 $\begin{Bmatrix}=\\<\\>\end{Bmatrix}$ 02

Moreover, if the result of any arithmetic operation is $> 2^{31}$ in absolute value (overflow), or if a divisor is = 0, then the next-instruction in sequence is taken from location 4 of memory. In the case of overflow, the condition register is set to 3 .

Group 2:

The parameters of the instruction are interpreted as in Group 1.

Instruction	Code	Meaning
Branch	BC	Branch to 02, if the state bit* corresponding to the condition register value is 1 .
Branch	BCR	Branch to the address contained in register r2, if the state bit* corresponding to the condition registervalue is 1 .
Branch and Link	BAL	Branch to 02 . Assign the address of the next instruction after the BAL to register rl .
Load Compliment	LCR	01 := -02

*The field rl -contains 4 bits, called state bits, numbered 0,1,2,3 .

(Continued)

Instruction	Code	Meaning
Insert Character	IC	The right-most 8 bits of register r1 are made equal to the single byte 02 .
Store Character	STC	The single byte 02 is made equal to the right-most 8 bits of register r1 .
Load Address	LA	Register r1 is assigned the address which designates 02 .
Store	ST	02 := 01
Shift left	SHL	Shift to the left the bits in register r1 by as many positions as indicated by a2 plus the value of register r2 . Vacated bit positions are assigned 0's .
Shift right	SHR	Analogous to SHL.
Read	R	Read a card, assign the 80 characters read to the 80 bytes the first of which is (a2, r2) . In each byte, the first two bits are set to 0, the remaining 6 bits are assigned the corresponding BCD character.
Write	W	Analogous to Read; the register r1 indicates the number of characters to be printed on the output line.

If in any instruction, an effective address $\geq$ 4096 is created, the next instruction in sequence will be taken from location 8 in the memory.

Programming Notes:

The interpreter is to be programmed in Extended ALGOL for the B5500 computer. After debugging, it should be merged with the assembly program of Term Problem 1 in the following way:

```
BEGIN COMMENT OUTER BLOCK;
   BEGIN COMMENT ASSEMBLER;
         . . . . . .
   END;
   BEGIN COMMENT INTERPRETER;
         . . . . . .
   END
END.
```

The "outer block" contains declarations of quantities shared by the two programs, such as the array of assembled program instructions. The interpreter is then supposed to execute the code which was assembled by the assembler.

You may assume that the first 4 bytes of the memory will never be used.

Before the due date, a problem will be given to be programmed in the assembly language as described in the Term Problem. At the due date, submit

1. a listing of the combined assembler/interpreter program,
2. the solution of the programming problem in the form of
    a. an assembly listing, and
    b. the output from the interpreter executing this program.

Supply (but do not overburden) your program with comments at appropriate places.

C.S. 236a
Winter, 1966

Test Programs for Term Problem II

The following are Test Problems to be programmed in the Assembly Code of Term Problem I. They are to be assembled and interpreted by your Assembler and Interpreter Programs.

1. Read a card, sort the first 30 characters according to their BCD key, and print the resulting string of 30 characters. Repeat this process for as many cards as provided.

2. (Optional) Read from cards the sequence of integers

$$n,\ a_1,\ a_2,\ \ldots\ a_n,\ b_1, b_2, \ldots,\ b_n$$

compute and print .

$$\sum_{i=1}^{n} a_i, \quad \sum_{i=1}^{n} b_i, \quad \sum_{i=1}^{n} a_i b_i \quad .$$

Perform reading and printing with the use of subroutines, which read and print one number respectively. A number should be acceptable if it consists of a. sequence of digits, possibly preceded by a. sign, and if it is separated from other numbers by at least one blank space.

IV. INPUT-OUTPUT PROGRAMMING

IV-1. The Input-Output Problem

The components of a large computer system can be ordered in a hierarchy according to their speed of operation:

Central Processors

- Control Circuitry
- Registers
- Arithmetic Units

Main Storage

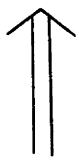
e.g., Thin Film
Cores
Drums

Secondary or Auxiliary Storage

e.g., Cores
Drums
Disks
Tapes

Pure Input-Output Devices

e.g., Card Readers, Punches
Printers
Display Devices
Typewriters
Paper Tape Readers and Punches



The rate at which information can be handled varies fantastically throughout this hierarchy - from one or fewer characters per second at the lowest level to billions of characters per second in the central processor. This is a factor of approximately 10^9 !

Each of the above components can be viewed as input-output (I-O) devices in some contexts; for example, information on a secondary storage device, such as a drum, can often be sent to or received from a central processor, main storage, other secondary storage devices, or any of the pure input-output devices. One of the most important objectives of I-O hardware and program design is to utilize all components of the computer system at their maximum rate; no component should ever be idle because it is waiting for another one to complete its operation.

Communication scheduling between the central processor and main storage is performed mainly by hardware; to counteract the relatively long access time to storage, instruction look ahead and interleaved storage are used on some large computers. The systems programming problem is to schedule and organize I-O among the elements of main storage, secondary storage, and the pure I-O devices. To do this, various techniques and devices, such as I-O buffering, interrupts, channels, and I-O processors, may be employed.

This chapter briefly examines some of the methods used to schedule and organize I-O. One multiple buffering scheme is presented in detail.

IV-2. Immediate I-O

Many of the earlier computers and some of the smaller modern computers have immediate I-O instructions; by "immediate", we mean that the complete I-O operation is handled directly by the central processor immediately upon receiving the I-O instruction. This includes initiating the I-O, specifying the I-O areas, maintaining a count of the number of characters transmitted, and testing for errors.

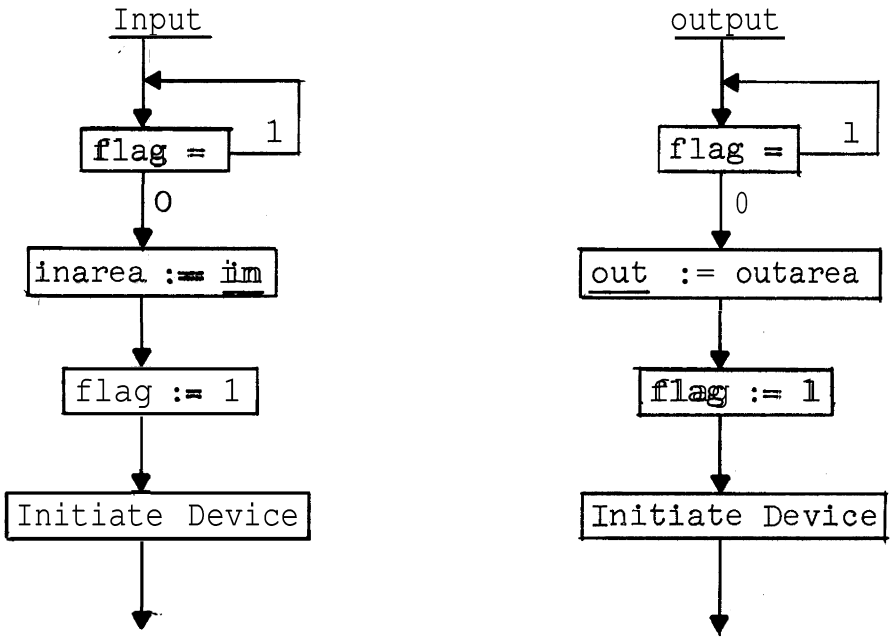
IV-2.1 No "Busy" Flag

The most primitive implementation of immediate I-O instructions has no provision for testing, by program, the status of the I-O units. If a unit is busy when an I-O command is given for it, program execution cannot continue until the unit is free and the command is accepted. Careful spacing of I-O operations can minimize this waiting time. Often, output instructions to a console typewriter are of this type.

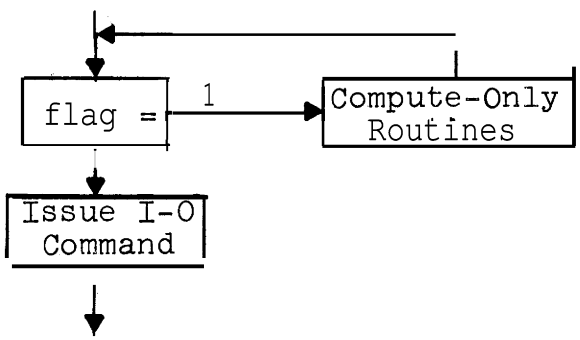
Many computers have hardware buffering for pure I-O devices with fixed record lengths, such as card readers and printers, An input (output) instruction empties (fills) the buffer into (from) storage and activates the device to automatically refill (empty) the buffer while the program proceeds, The device is always one I-O operation ahead of (behind) the program. The advantage here is that, with cardful spacing, the I-O instructions are completed at electronic speeds while many pure I-O devices actually operate at electro-mechanical speeds.

IV-2.2 "Busy" Flag

A program-addressable flag bit is automatically set when an I-O unit becomes busy and is reset when the unit becomes free, For a simple computer with I-O buffering and out, an I-O instruction produces the following hardware actions:



where inarea and outarea are storage areas for input and output. Since the flag or "busy" bit is addressable, the programmer may use it to branch to routines involving no I-O while waiting for the unit to become free:



This requires much programming of an administrative nature for testing of the flag and computing when the I-O unit is busy.

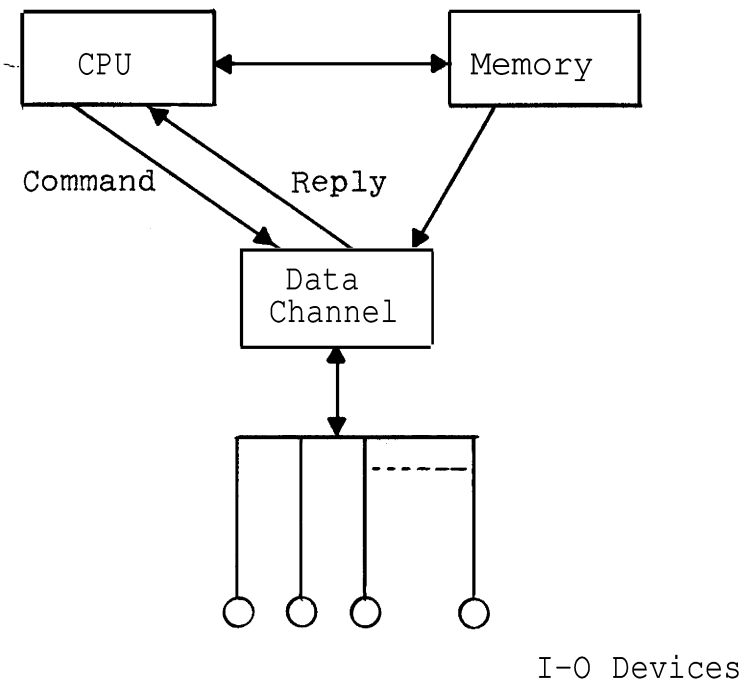
IV-3. Indirect I-O

Most computers presently available have some form of indirect I-O. The central processor only initiates the operation; the operation is

performed by an independent unit, such as a channel or I-O processor. Once an I-O operation has been initiated correctly, the computer can continue processing concurrent with the execution of the I-O command.

IV-3.1 Channels

A data-channel is a control device which acts as an interface between the processor and memory on the one hand and one or more I-O devices on the other:



An I-O command from the processor consists of a request for an I-O operation and control information (or an address containing control information). The control information usually includes the device address, I-O area address, and number of units of data to be transmitted. The channel performs the work of initiating the I-O device, counting the data units transmitted, and testing for errors. With concurrent computing

and I-O, there is competition for memory cycles; the channel "steals" its cycles when needed.

Two methods of communication between a data channel and central processor are possible:

(1) The CPU may interrogate the status of the channel, e.g., Is the channel busy? or

(2) The channel can interrupt the CPU on termination of the I-O operation or on an error condition.

IV-3.2 CPU Interrogates Channel

,A simple example of an input-output routine written in FAP for the IBM 7090 with one channel is presented. In this program no use is made of the channel as a separate independent unit since the CPU is held up until the input or output is finished. A typical call of LINE is:

```
        TSX     LINE,4
        PZE     COUNT
        PZE     BUF
         .
         .
COUNT   DEC     20
BUF     BSS     20
```

To allow overlap of I-O with computing, a simple software buffering scheme may be used; LINE moves the information from the I-O area to a buffer and the channel works on the buffer area. The IBM FORTRAN system on the 7090 handles I-O in this way. The call of the modified LINE given below is the same as in the previous example:

Simple Example of Input-Output Routine

Where CPU Interrogates Channel

```
*        FAP
         COUNT    100
         LBL      CRLN
         ENTRY    CARD
         ENTRY    LINE
 I       TAPENO A 2
 0       TAPENO A 3
*
 CARD    CLA      1,4
         STA      IOI
         RTDI
         RCHI     IOI
         TCOI     *
         TEFI*    2,4
         TRCI*    304
         TRA      4,4
*
 LINE    CLA*     1,4
         ALS      18
         STD      IOO
         CLA      2,4
         STA      IOO
         WTDO
         HCHO     IOO
         TCOO     *
         ETTO
         TRA      ETT
         TRA      3,4
*
 ETT     RUNO
         HTR      LINE
*
 IOI     IORT     **,,14
 IOO     IORT     **,,**
         END
```

Buffered Input-Output Routine Where CPU Interrogates Channel

```
*       FAP
        COUNT   200
* C A RD  R E A D E R  A N D L I N E  P R I N T E R
        LBL     CRDLIN
        ENTRY   CARD
        ENTRY   LINE
*       CALLING SEQUENCE . o
*            CARD (BUFFER, EOF EXIT, REDUN ERR EXIT)
I       TAPENO A 2
 CARD   CLA     1,4
        SXA     X1,1
        SXA     X2,2
        PAC     092
 C1     TXH     C2,0,0
        TRCI    *+1
        TEFI    *+1
        RTDI
        RCHI    IOD
        CLS     C1
        STO     C1
 C2     AXT     5,1
        TCOI    *
        TEFI    EOF
        TRCI    ERR
        AXT     0,1
        CLA     INBUF,1
        STO     0,2
        TXI     *+1,2,-1
        TXI     *+1,1,-1
        TXH     *-4,1,-14
        RTDI
        RCHI    IOD
 X1     AXT     **,1
 x2     AXT     **,2
        THA     4,4
*
 'EOF   BSRI
        TRA*    2,4             END OF FILE EXIT
 ERR    BSRI
        RTDI
        RCHI    IOD
        TIX     C2+1,1,1
        TRA*    3,4             REDUNDANCY ERROR EXIT
 100    IORT    INBUF,0,14
 INBUF  BSS     14
```

```
*       CALLING SEQUENCE IS,.  LINE(WORDCOUNT, BUFFER)
0       TAPENO A  3
 LNEST BOOL     77
 LNCNT BOOL     141
 LINE C L A *   1,4            WORD COUNT
        SXA     S1,1
        SXA     S2,2
        SXA     S4,4
        PAX     0,1
        TXL     **+2, 1822
        AXT     22,1
        SXD     IOX,1
        CLA     2,4            OUTPUT AREA
        PAC     0,2
        AXT     Or 4
        TCOO    *
        ETTO
        TRA     ETT
 L2     CLA     Or2
        STO     OBUF,4
        TXI     **+1,4,-1
        TXI     **+1,2,-1
        TIX     *-4,1,1
        WTDO
        RCHO    10x
        CAL     LNCNT
        ANA     =077777
        ADD     =1
        STA     LNCNT
        SUB     LNEST
        TPL     QUIT
 S1     AXT     **,1
 S2     AXT     **,2
 s 4    AXT     **,4
        TRA     3,4            EXIT "LINE"
*
 QUIT CAL       *              TOO MANY LINES PRINTED
        STP     LNEST
        TSX     $EXIT,4
*
 ETT    WEFO
        RUNO
        HTR     L2
 10X    IORT    OBUF,0,**
 OBUF BSS       22
        END
```

This scheme begins to take advantage of the ability of the channel to function independently of the CPU; for example, in LINE, the CPU may perform any non-I-O operation as the buffer is emptied by the channel to the I-O device. However, an inherent limitation exists when the CPU is required to interrogate the status of the channel. If bursts of I-O occur at infrequent intervals during a program, the CPU would often be idle while these bursts were taken care of. Interrupts allow the I-O to be scheduled more uniformly over the processing time.

IV-3.3 Channel Interrupts CPU

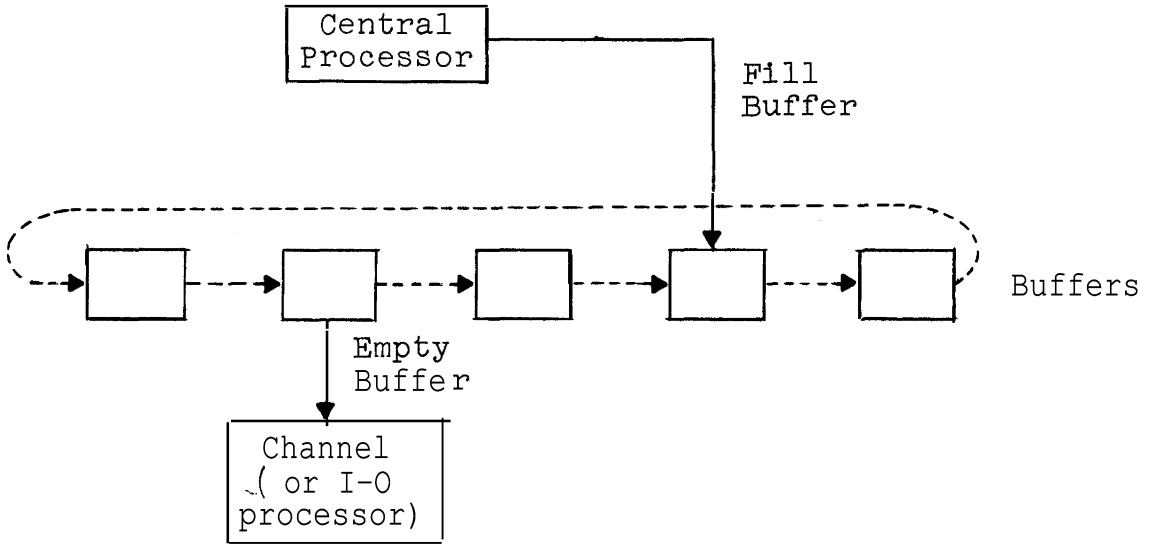
Interrupts are automatic hardware transfers and "saves" that occur when unusual or infrequent conditions result during program execution. For example, if an overflow occurred during the execution of "a : = b+c", most machines would automatically reset the instruction counter to a fixed location in the machine where an error routine resides, Without this facility, at each add, the careful programmer would have to write the equivalent of:

```
a := b + c;   if overflow then goto error;
```

where overflow is a Boolean variable set by the add operation when an overflow occurs and error is the error routine entry. In the same way, interrupts occur on termination of I-O and I-O errors.

By using interrupts in conjunction with several buffers, channels can operate almost completely independently of the CPU. The degree of parallelism obtained depends on the number of buffers, the number of channels, and the amount of I-O called for. With a reasonable number of buffers, the processor should rarely be in a "wait" loop waiting for

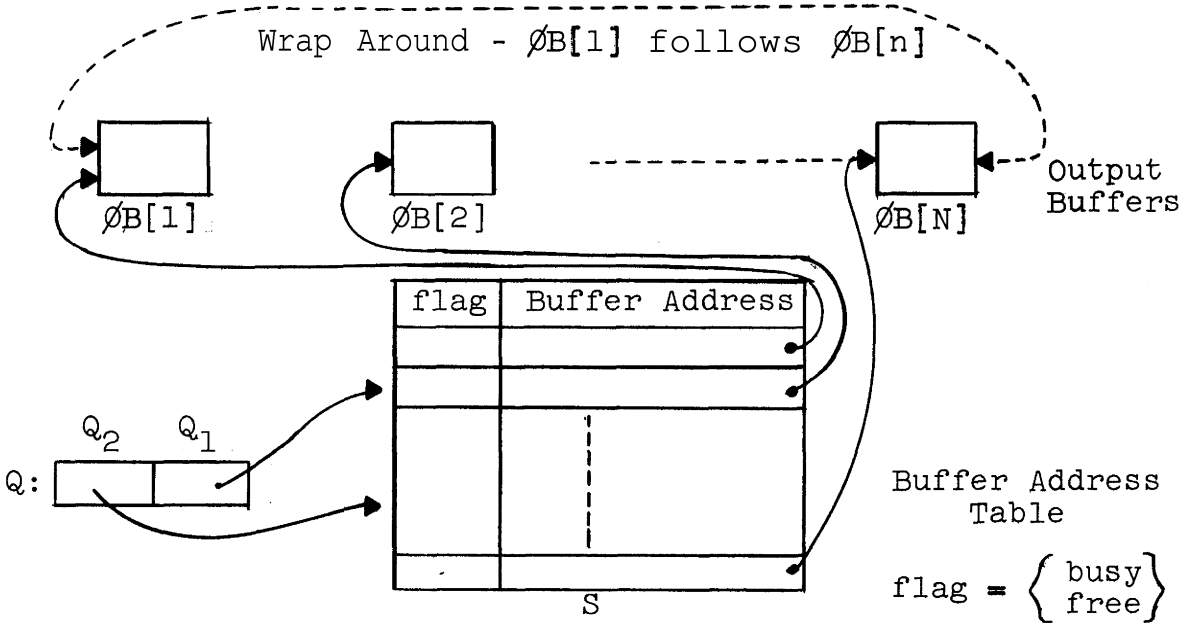
a channel to be free. Buffer handling by interrupts by an output routine can be organized as follows:



A detailed description of such a multiple buffering output routine using interrupts is given on the following pages. The FAP program also includes a similarly constructed input routine.

Multiple Buffer Output System Using Interrupts

1. Program, Buffer, and Pointer Organization



Q_1 : next free buffer, or

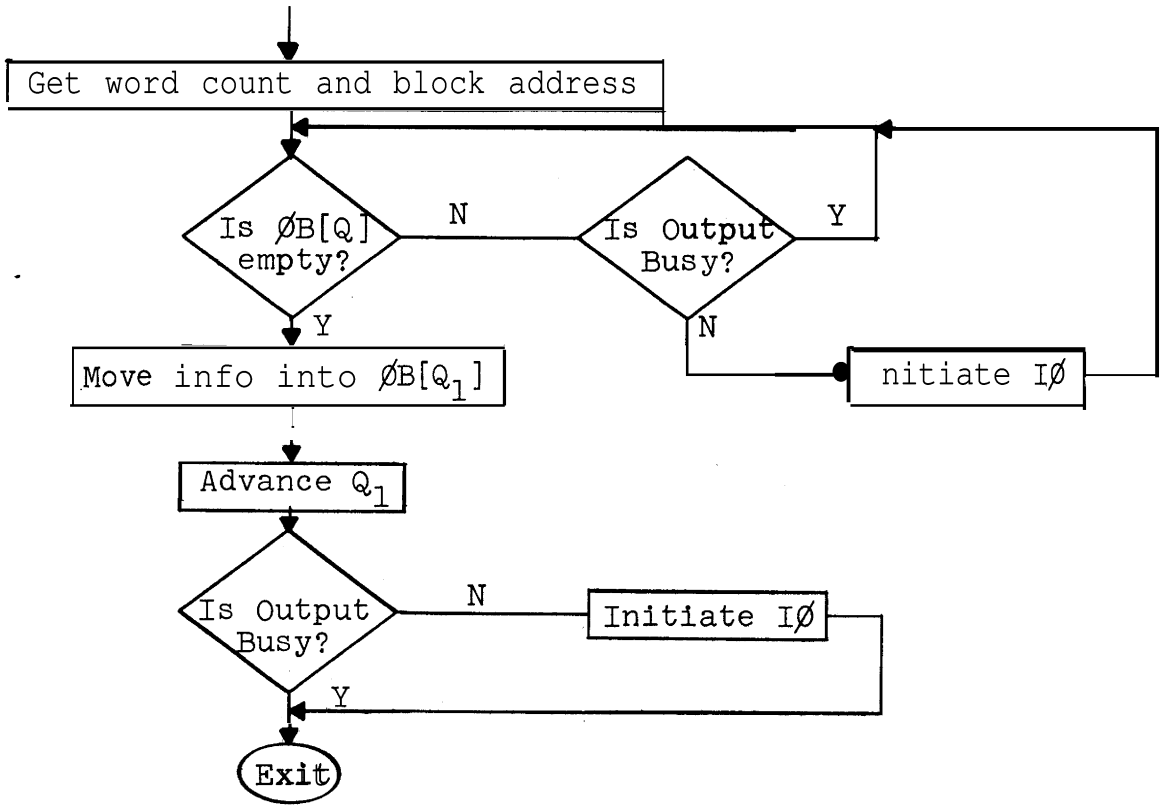
buffer that will be available first

Q_2 : buffer being emptied by channel

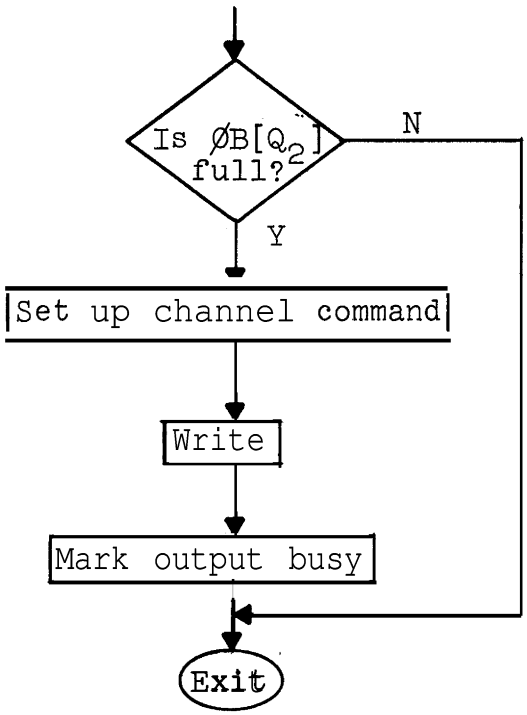
The CPU fills buffer Q_1 and the channel empties buffer Q_2. The program is organized so that Q_1 chases Q_2. An interrupt occurs after a buffer Q_2 has been emptied by the channel; the interrupt program adjusts the Q_2 pointer and initiates another output if the- new $OB[Q_2]$ is full. The routine LINE is called from the main program whenever output is required. LINE fills $OB[Q_1]$ and increments Q_1.

2. Flow Charts

LINE : Activated by Main Program

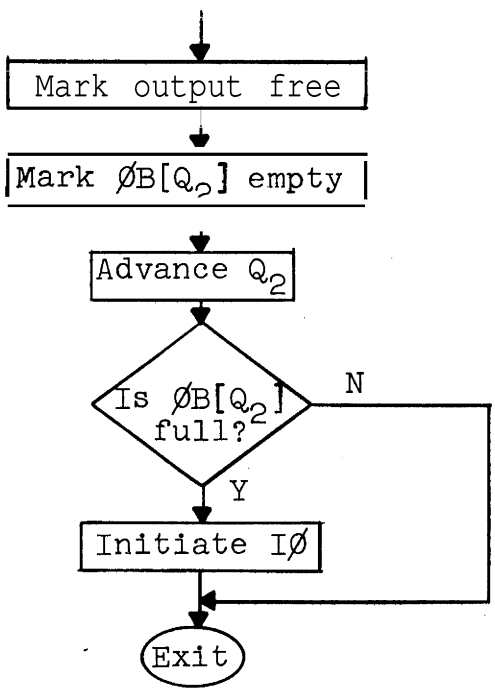


IØ: Initiate IØ (subroutine)



T2: Output Termination

Activated by a channel interrupt when channel terminates an output instruction.



3. **FAP** Program

```
*        I N P U T
I        TAPEND   A2
CARD S   X A      CEX,2
         SXA      CEX+1,4
         CLA      1,4
         STA      C5
C2       LXA      P,4
         CLA      T,4          IS BUFFER FULL
         TMI      C4           YES
         ZET      END
         TRA      QUIT
         NZT      BUSYI
         TSX      I1,4
         ENR      MSK
         TRA      C2
*
C4       PAC      0,2          TRANSFER INPUT DATA
         AXT      0,4
         CLA      0,2
C5       STO      **,4
         TXI      *+1,2,-1
         TXI      *+1,4,-1
         TXH      *-4,4,-14
*
         LXA      P,4          MOVE POINTER P1
         ZAC
         STP      T,4
         TIX      *+2,4,1
         AXT      N,4
         SXA      P,4
*
         NZT      BUSY1
         TSX      I1,4
*
CEX      AXT      **,2
         AXT      **,4
         TRA      04,4
*
I1       SXA      C6,4         INITIATE INPUT
         LXD      P,4
         CLA      T,4
         TMI      C6
         STA      ICOM
         RTOI
         RCHI     ICOM
         CLA      T1
         STO      11
         STO      BUSYI
C6       AXT      **,4
         TRA      1,4
         EJECT
T1       TRA      *+1          INPUT INTERRUPT
         SXA      T12,4
         STQ      MQ
         LGR      2
```

```
        STO     AC
        STZ     BUSYI
        LXD     10.4
        TXH     ENDF,4,2
        TXH     RED,4,1
T13     CLA*    ICOM
        SUB     FINIS
        TZE     ENDF
        LXD     P,4
        CLA     T,4
        SSM
        STO     T,4             MARK SUFFER FULL
        TIX     *+2,4,1
        AXT     N,4
        SXD     P,4
        CLA     T,4
        TMI     *+2             IS NEXT BUFFER EMPTY
        TSX  --.  II,4          YES
T11     CLA     AC
        LGL     2
        LDQ     MQ
T12     AXT     **,4
        RCT
        TRA*    10
*
ENDF    STL     END             END OF FILE
        STZ     BUSYI
        TRA     T11
RED     AXT     3,4             REDUNDANCY CHECK ERROR
        BSRI
        RTDI
        RCHI    ICOM
        TCOI    *
        TRCI    *+2
        TRA     T13
        TIX     RED+1,4,1
        STL     ERR
        TRA     ENDF
*
QUIT LXA        CEX+1,4
        ZET     ERR
        TRA*    2,4
        TRA*    3,4
```

```
*       O U T P U T
 0      TAPENO  B3
 LIKE SXA       LEX,1
        SXA     LEX+1,2
        SXA     LEX+2,4
        CLA*    1,4
        ALS     18
        STO     WC
        CLA     2,4
        STA     L5
 L2     LXA     Q,4
        CLA     S,4
        TPL     L4
        NZT     BUSY0
        TSX     IO,4
        ENB     MSK
        TRA     L2
*
 L4     PAC     0,2             TRANSFER OUTPUT DATA
        CLA     WC
        STD     S,4
        POX     0,1
        AXT     0,4
 L5     CLA     **,4
        STO     0,2
        TXI     *+1,2,-1
        TXI     *+1,4,-1
        TIX     *-4,1,1
*
        LXA     Q,4             MOVE POINTER Q1
        CLS     S,4
        STO     S,4
        TIX     *+2,4,1
        AXT     M,4
        SXA     Q,4
*
        NZT     BUSY0
        TSX     IO,4
*
 LEX    AXT     **,1
        AXT     **,2
        AXT     **,4
        TRA     3,4
        EJECT
 IO     SXA     L6,4            INITIATE OUTPUT
        LXQ     Q,4
        CLA     S,4
        TPL     L6
        STA     OCOM
        STD     OCOM
        WTDO
        RCHO    OCOM
        CLA     T2
        STO     13
    STO         BUSY0
```

```
L6      AXT     **,4
        TRA     1,4
*
T2      TRA     0 +1            OUTPUT INTERRUPT
        SXA     T22,4
        STQ     MQ
        LGR     2
        STO     AC
        STZ     BUSYO
        ETTO
        TRA     ETP
        LXD     Q,4
        ZAC
        STP     S,4             MARK BUFFER EMPTY
        TIX     *+2,4,1
        AXT     M,4
        SXD     Q,4
        CLA     S,4
        TPL     *+2             IS NEXT BUFFER FULL
T21     TSX     IO,4            YES
        CLA     AC
        LGL     2
        LDQ     MQ
T22     AXT     **,4
        RCT
        TRA*    12
*
ETP     RUNO
        HTR     T21
```

```
N       EQU     4               NUMBER OF INPUT BUFFERS
M       EQU     4               NUMBER OF OUTPUT BUFFERS
WC      PZE                     LENGTH OF OUTPUT RECORD
ERR     PZE                     FLAG SET BY REOUNOANCY ERROR
P       PZE     N,,N            INPUT TABLE POINTERS
Q       PZE     M,,M            OUTPUT TABLE POINTERS
ENC     PZE                     FLAG SET BY ERROR CONDITION
BUSY I  PZE                     FLAG ON IF INPUT CHANNEL BUSY
BUSY0   PZE                     FLAG ON IF OUTPUT CHANNEL BUSY
ICOM    IORT    **,,14
OCOM I O R T    **,,**
MSK     PZE     3,,1
FINIS   BCI     1,FINIS
MQ      PZE
AC      PZE
*
        PZE     IB1
        PZE     IB2
        PZE     103
        PZE     IB4
T       SYN     *               INPUT BUFFER TABLE
*
        PZE     OB1,,0
        PZE     OB2,,0
        PZE     OB3,,0
        PZE     OB4,,0
S       SYN     *               TABLE OF OUTPUT BUFFERS
*
IB1     BSS     14
IB2     BSS     14
IB3     BSS     14
IB4     BSS     14
OB1     BSS     22
OB2     BSS     22
OB3     BSS     22
OB4     BSS     22
        END
```

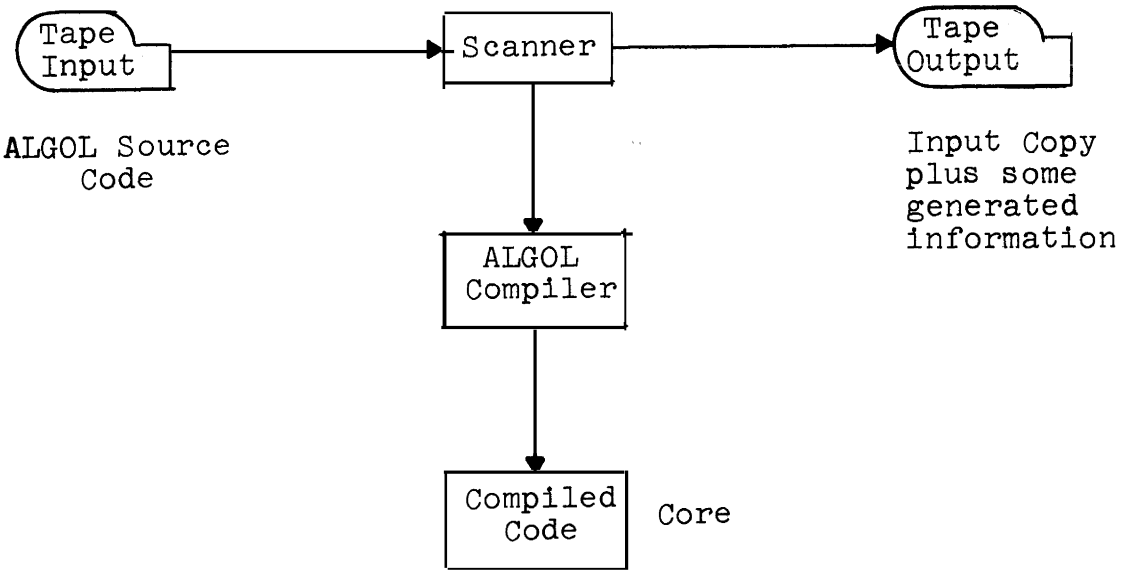
For computers with several channels, there is the possibility of interrupts from different sources occurring simultaneously. To handle this, there must be hardware or software provision for determining priorities of interrupts and storing pending interrupts in a priority queue. It is the task of monitor or supervisory programs to administer these interrupts correctly.

IV-4. I-O Processors

The next level of sophistication after channels is the use of separate I-O processors to process I-O. In this context, a channel is a crude I-O processor. With I-O processors that approach the power of a computer, I-O data can be edited, checked, and manipulated before it reaches the central processor; that is, all the I-O housekeeping tasks can be delegated to the I-O processor. An I-O processor can be a specifically designed unit for a particular machine, as in the DEC PDP-6 system, or it may be another computer attached to the main machine, as in the IBM 7090-7040 direct-coupled system.

IV-5. Experimental Comparison of Several Methods of I-O Organization

Several methods of organizing I-O for the scanner portion of an experimental ALGOL compiler on the IBM 7090 were examined by N. Wirth. The basic problem is illustrated below:



For the experiment, 630 card records were put on tape as the input and then compiled into core producing an output tape ($5,550_8$ machine language instructions were compiled). Four I-O schemes were investigated:

A: No Buffers

1 Channel

CPU interrogates channel

B: 1 Buffer

1 Channel

CPU interrogates channel

C: 1 Buffer

2 Channels (one for input, other for output)

CPU interrogates Channel

D: 4 Buffers

2 Channels (one for input, other for output)

Channel interrupts CPU

(FAP program of section IV-3.2.)

Test Results

Method

	A	B	C	D
Compile Time (Seconds)	16.11	12.12	7.05	6.31
*Comparison	1	1.33	2.29	2.55
Length of I-O Program	26_8	51_8	51_8	244_8
Total Buffer Length	0	44_8	44_8	220_8

* The comparison gives the ratio of the Compile Time of A to that of B, C, and D .

For this particular application, the greatest gain is for method C where separate channels are used for input and output. The multiple buffering scheme is marginal here. However, the I-O occurs uniformly over the processing time in this experiment. It is predicted that D would show a greater gain for applications where the I-O occur in bursts.

IV-6. I-O and Systems Programming

Today the applications programmer seldom worries about the detailed scheduling and programming of I-O. In fact, it is very difficult, if not impossible in some instances, for the user to gain access to the machine I-O commands. Monitor programs carry out the details of the 'I-O tasks requested, even at the assembly language level for some systems. Therefore, most I-O programming for computer systems is carried out "centrally" by the systems programmer.

V. SUPERVISORY PROGRAMS (MONITORS)

V-1. Monitor Tasks

Historically, monitor or supervisory programs were developed to ensure the continuous operation of computer systems with little or no human intervention. As systems became more complex, monitors assumed the responsibility of scheduling and allocating computer resources, such as storage, channels, I-O units, and processors. To accomplish these tasks, it is necessary that ultimate control within and among user jobs resides in the monitor.

Monitor systems perform the following general functions:

1. Job-to-Job Control

This consists of the automatic termination and initiation of jobs. Jobs may be terminated "naturally" or on error conditions; termination tasks include sign-off accounting, closing of files, and compilation of job statistics. Job initiation includes sign-on accounting and interpreting user monitor control commands for opening files and program loading.

2. Accounting

Records of use of the computer system components during a job are kept and the user is charged accordingly.

3. Program Loading and Merging

Prior to or during execution, user programs and subroutines must be loaded into storage and linkages established among them. The monitor allocates storage to the programs, loads them into storage performing the necessary address relocations, and sets up linkages among the programs so-they may communicate with one another.

4. Accessing and Maintenance of Library Programs

Most monitors maintain a library of systems and applications programs that may be "called" by a user; these include compilers, assemblers, I-O routines, and common mathematical functions. Loading and merging, and inserting and deleting library programs are handled by the monitor.

5. I-O Processing

In order to maintain job-to-job control and to obtain optimum use of I-O facilities, most modern systems delegate all I-O to the supervisor. These systems often have hardware supervisory and problem modes of operation. Hardware I-O instructions are "supervisory" type, that is, only the monitor is permitted to use them. To perform an I-O operation, the user issues an I-O request to the monitor which does the actual execution.

6. Error Checking and Recovery

Run-time errors, such as overflow, use of illegal or "privileged" instructions (e.g., I-O instructions), exceeding run time limit, memory-protect violations, etc., result in interrupts or calls on the supervisor; the supervisor determines the cause of the error, decides whether to terminate execution or not, and produces diagnostic information for the user.

7. Interrupt Handling

Monitors are responsible for the analysis and disposition of all interrupts that may occur during systems operation; this may include maintenance of pending interrupt queues and priority scheduling of interrupt handling.

8. Scheduling and Allocation of Resources

When computer resources are insufficient to satisfy the total demand on them or when it is desired to maintain a high degree of parallel operation of the system components, resource allocation and scheduling routines are necessary. These become part of the monitor program.

This chapter outlines the three basic types of monitors and discusses some general methods of allocation and relocation which are central to the above tasks. A separate section describes some approaches to solving an important control problem for parallel processes.

v-2. Types of Monitors

V-2.1 Batch Processing Monitors

This is the simplest and oldest type of monitor. In this type of systems, jobs arrive sequentially in "batches" usually from one input source. Normally, one job at a time is processed; where multiprogramming is possible, several jobs may be in storage simultaneously and the monitor controls the switching among jobs. Typical conventional monitors are the IBM 7090/7094 IBSYS System1 and the B5500 Operating System[2] (multiprogramming).

V-2.2 Real Time Monitors

Interrupts from external devices command the attention of the system and must be processed within a given time interval. Interrupt times are unpredictable but several may occur during the processing of another interrupt. Airlines reservation systems[3] and computer control of physics experiments[4] are applications of this type.

The major task of a real time monitor is the handling of interrupts. In addition, most systems batch process "background" programs while there are no interrupts pending; on an interrupt, the real time monitor transfers control from the "background" program to the particular interrupt processing routine.

V-2.3 Time Sharing Monitors

A time-shared digital computer system[5] is "a system from which many people (or machines) may demand access and expect to receive responses after short enough delays to satisfy them." Batch processing and real time operations may be included as part of the capabilities of a general time-sharing system.

The most common method of implementing a time-sharing system is through multiprogramming where[5] "several programs are maintained in an active state (with others probably waiting in a queue), and at various times each is given control of some part of the computer, until one or another of them is finished, or until a new task is brought in to replace an older one, according to some scheduling algorithm. Fast response by the computer to many users (e.g., 150 to 200 or more) requires that each task be given a "time slice", and if the task cannot be completed during its "time slice", that it must be interrupted to allow another task its turn."

A time-sharing monitor has the following demands and requirements:[5]
"(1) At any moment in time one may expect to find a great many partially completed programs, each waiting for a turn at the central processor, an input-output processor or some other part of the computer.

(2) Very effective use must be made of high speed storage, since many programs must have access to it, but usually only a fraction of these programs can reside there at any one time.

(3) The overhead incurred in keeping track of the programs which are partially completed or not yet begun and the overhead incurred in switching control among them (while protecting each from the others), must be reduced to a minimum; otherwise, it will quickly become intolerable."

Methods for allocating high speed storage and satisfying requirement (2) for any type of monitor are discussed in the next section. The papers on the MULTICS system' contain a good discussion and bibliography on time-sharing.

v-3. Storage Allocation Methods

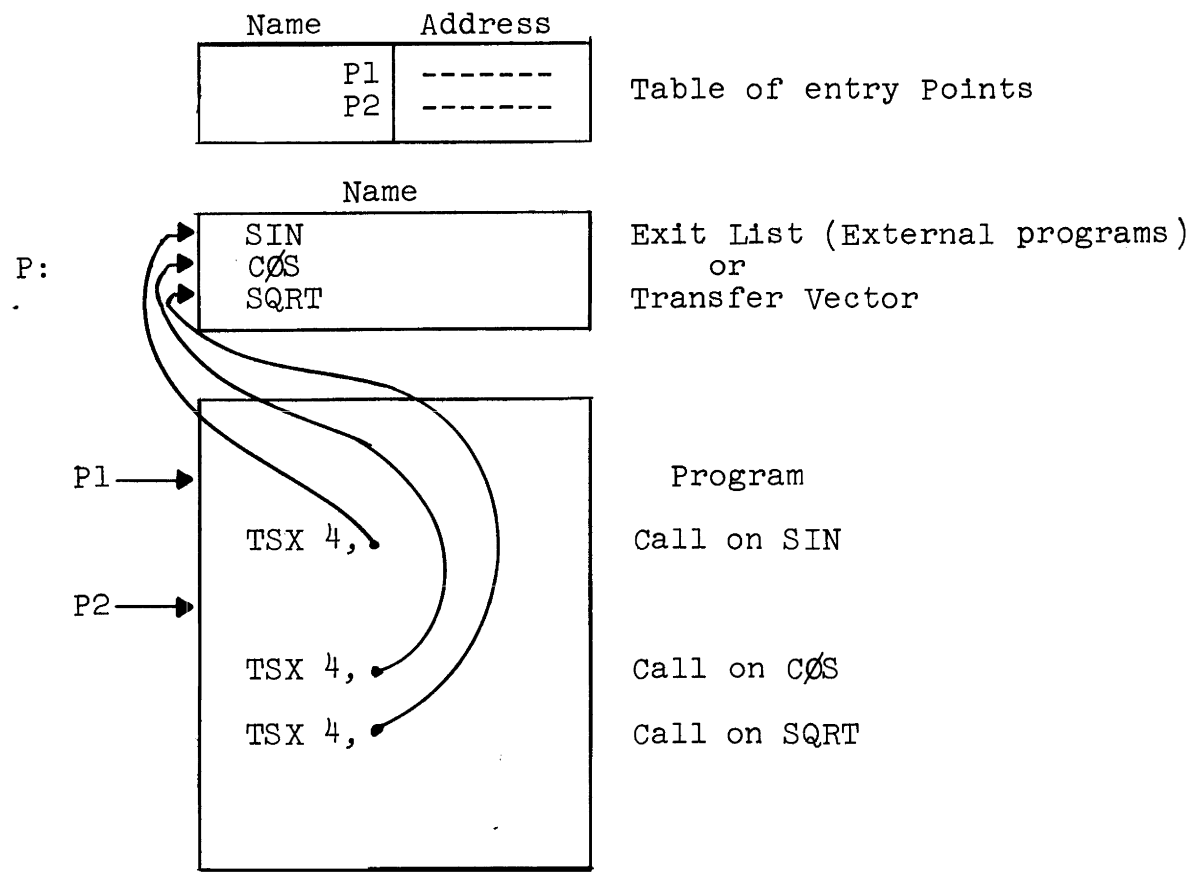
Storage may be allocated to a program at the time it is translated, before execution, or during execution. In the first case, a translator, such as an assembler, generates absolute addresses for data and instructions and the entire program including subroutines must be translated at the same time; merging of independently translated programs can only be done with great difficulty since address conflicts easily occur.

Because storage is allocated after translation in the latter two cases, the translation must result in a program with relocatable addresses; e.g., instruction addresses, data addresses, and operand addresses may all be translated relative to a given base address, commonly 0 . Loading of programs, parts of programs, or subroutines into storage is done before or during execution by adding relocation constants to the addresses. Relocation performed before execution is called static relocation; relocation performed during execution is called dynamic relocation.

V-3 .1 Static Relocation

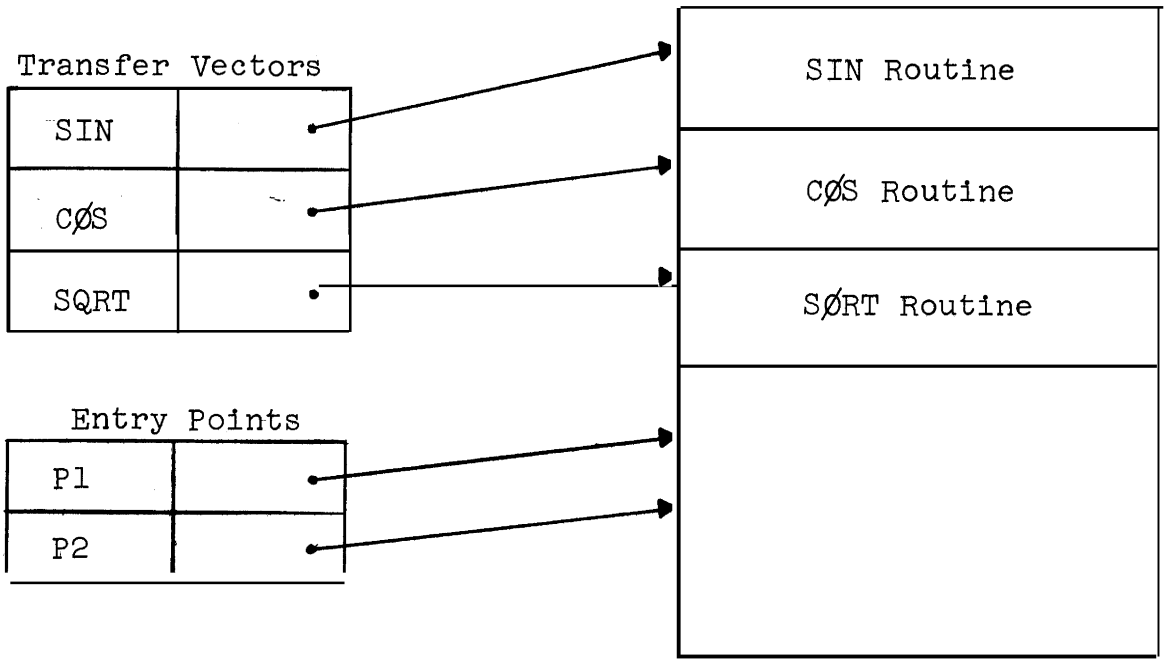
Static relocation is performed by a relocation loader as the program is loaded into storage. A number of programs comprising a job may be translated independently; the relocation loader allocates storage to the programs, relocates addresses to reflect this allocation, establishes linkages between programs, and places them in storage ready to be executed. During translation, flags can be set for each instruction to indicate which addresses in that instruction are relocatable and which are absolute (e.g., immediate type addresses); calls on "external" programs and program entry points are tabulated so these correct addresses may be inserted at load time.

The IBM 7090 FAP system[7] relocates statically as illustrated below:



This is the input to the relocation loader. The loader reads P and merges the library programs SIN, COS, and SQRT into storage performing the required address relocations; linkages are made using the transfer vectors:

Program P and Library Programs after Loading



The use of entry and exit point tables and transfer vectors is the most common method for performing the loading task,

Loading with a relocation loader is a complex and time consuming job. If a fast assembler or compiler is available, it is sometimes more efficient to translate and load all programs required by a job each time the job is run. This is the approach taken in the B5500 operating system.

Conceptually, a computer with base addressing, such as the IBM 360, can perform relocations very easily. For example, an IBM 360 address is

formed from the contents of a specified base register and a displacement (ignoring indexing):

address = (b, d), where b - base register
d - displacement

effective address := R[b] + d where R[ ] - register

Translation could occur with respect to the displacement; in loading, a relocation constant would be inserted in R[b]. This scheme requires that certain registers be-unavailable for use by the programmer and that the displacement cover a large address range.

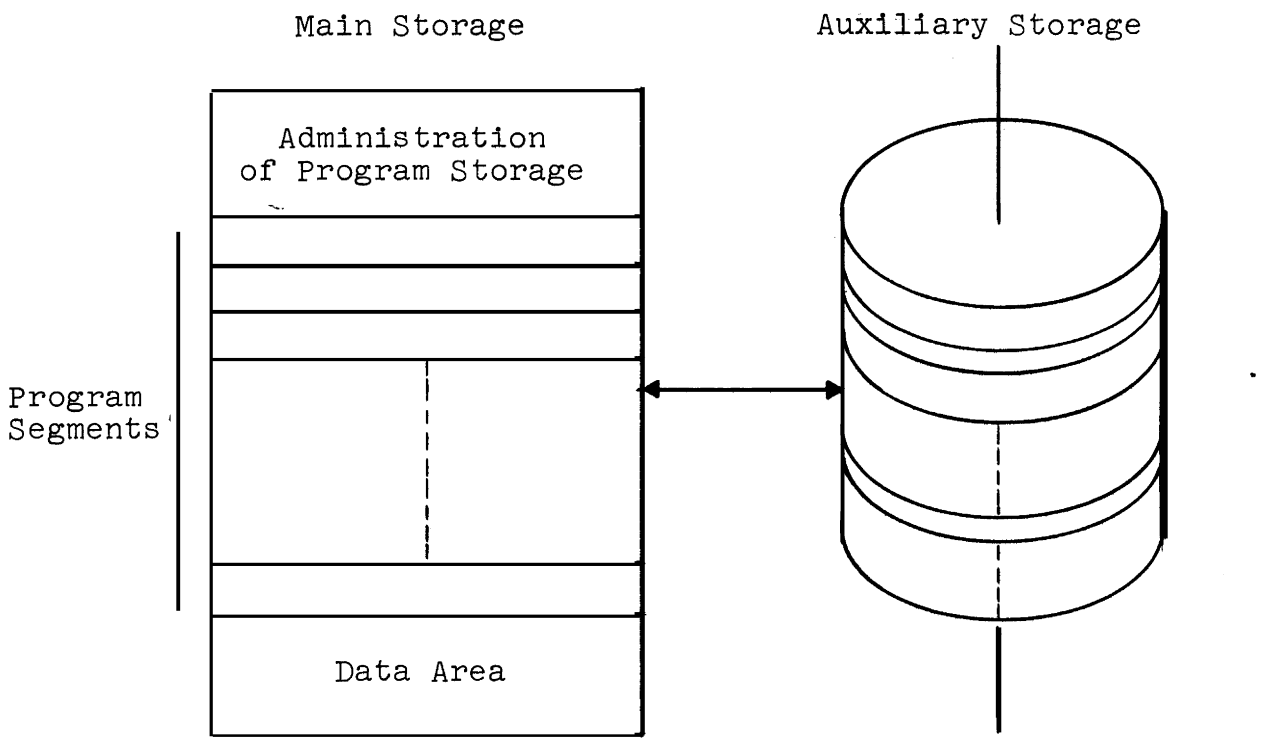
V-3.2 Dynamic Relocation

When a program is too large for main storage and auxiliary storage is available, some method for dividing the program into manageable se@;-ments and administering the swapping of these segments between main and auxiliary storage is necessary. One static technique that has been used is the following:

The monitor or translator, at translation time, (or the programmer when -he codes the problem) divides the program into segments which will fit into main storage and inserts "segment calls" to bring in new segments; all segments are relocated statically before execution of the program begins. This requires that the system (or programmer) know how much storage will be available for program and data at execution time; when several programs reside in core simultaneously as in a multiprogramming or time-sharing environment, or when data can be dynamically declared, this

knowledge is not available in general. A more satisfactory method is to divide the program into fixed or variable size segments, each of which can be dynamically relocated during execution.

A description and evaluation of this technique as used for the GIER ALGOL system is given by P. Naur in Reference 8. A typical picture of the allocation during execution is:



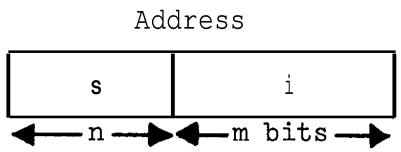
The data area is dynamically allocated by the program during execution using a stack mechanism. Programs are divided into small segments so that there is room for several segments in main storage at any time; segment to segment transfers are controlled by the Program Storage Administration - if the required segment is in core, (a table is kept of all segments in core) the transfer is made; if not, then the segment is brought into main storage from auxiliary storage. Segments which are unused for the longest

times are the candidates for replacement. Naur's conclusions were that the simple segment administration method used yielded satisfactory results in terms of run time efficiency and that a significant performance increase could be achieved by adding a hardware instruction to perform segment to segment transition (and thus reduce the segment table searching time).

This ability to insert segments anywhere in main storage during execution requires that all addresses be dynamically relocatable; addresses take the form of a pair (s, i) where s is a segment number and i represents the address within s . (This form of address was originated in the Ferranti ATLAS computer.) During execution of a segment, the pair (s, i) is translated to the correct absolute address, usually by hardware (however, the GEIR ALGOL system does this by software). Some of the hardware methods for implementing dynamic relocation are described next. Reference 9 gives a good general discussion of these methods.

V-3.2.1 Ferranti ATLAS Method

The upper part of the 20 bit machine language address is interpreted as the page number (page is synonymous with segment here) and the low order part as the address within the page or line number:



n = 11
m = 9

The addressing structure thus allows a program of up to 2^n pages, each page consisting of 2^m words. -However, in general, main storage consists

only of 2^{m+k} words, where $k < n$. Associated with each of the 2^k pages that may be in main storage is a hardware page register. Generation of the actual address from the relocatable address (s, i) proceeds as follows:

1. Search all page registers for s.
2. if value(Register[j]) = s then

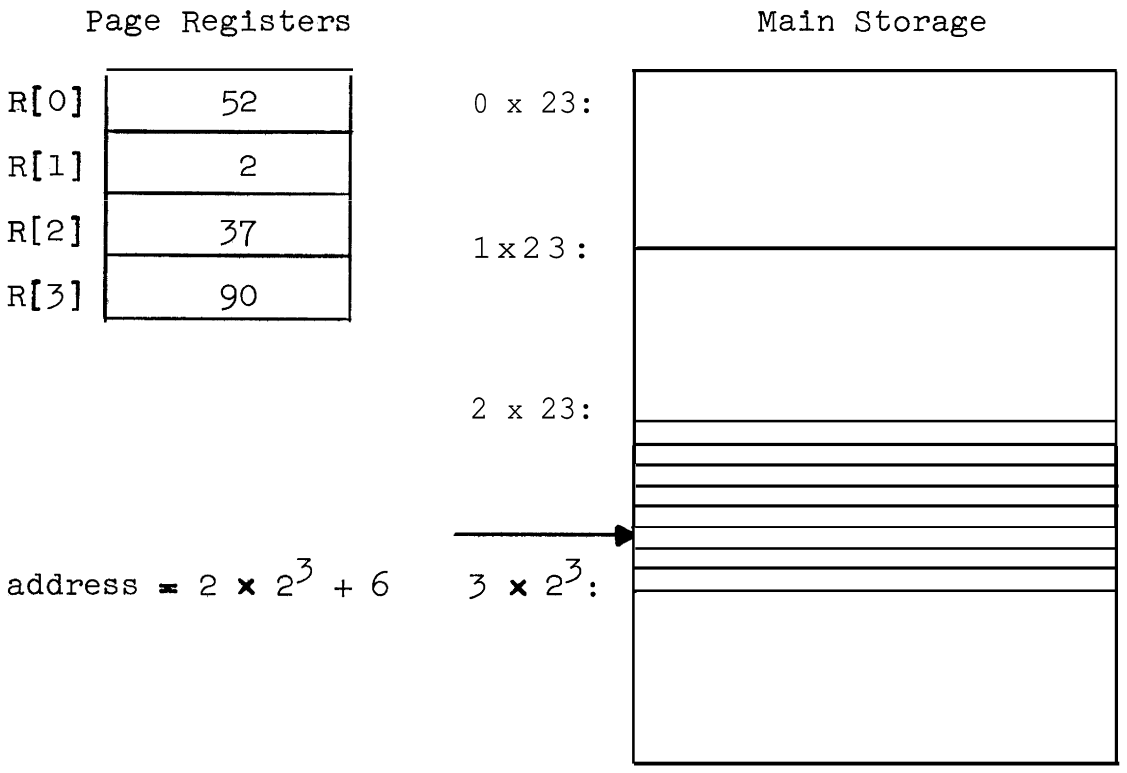
   address := $j \times 2^m + i$

   (i.e., page is in core)
3. Otherwise, fetch page from drum;

Steps 1 and 2 are performed by the hardware; a hardware interrupt to the supervisor occurs if the page is not in core.

Example

$k = 2$, $m = 3$
$(s,i) = (37,6)$



Administration hardware keeps track of page usage; when a new page is required from the drum and core is full, the page with the least usage is replaced. The relocation method applies both to data and program. The programmer sees a "virtual" memory of 2^{m+n} words and does not have any control over the segmenting and dynamic relocation processes.

V-3.2.2 Burroughs B5500:

B5500 ALGOL is compiled so that segments consist of ALGOL blocks, data, and control information. A program reference table (PRT) contains block and array "descriptors" which point to the core area containing the segment. Addresses of the form (s, i) are translated by:

Physical address := M[b+s] + i

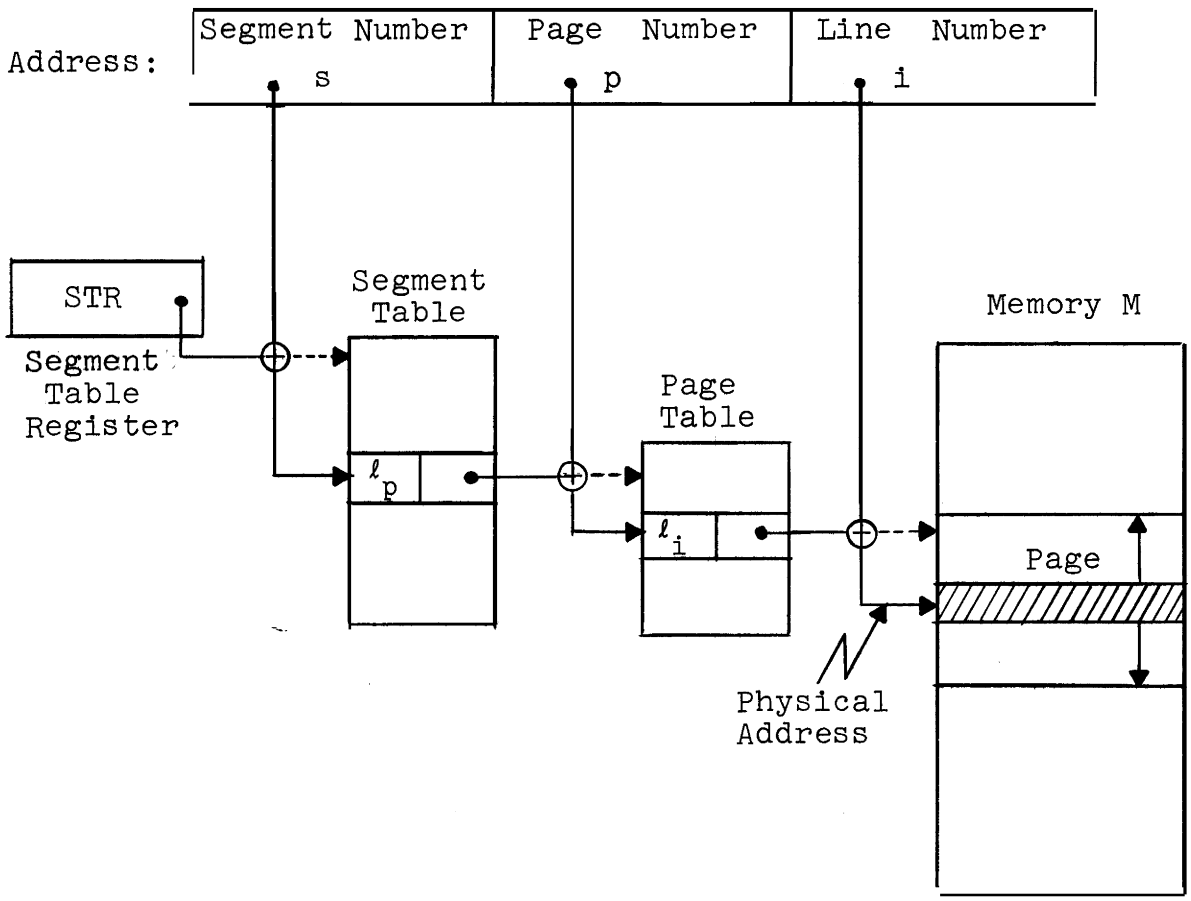
M: memory

b: base of PRT

Segments are not of fixed length but contain a size limit entry that enables an automatic check, e.g., if subscripts exceed their declared bounds. The advantage of making blocks equivalent to segments is that segments (or blocks) can then only be entered from the top and left either from the bottom or by a go to statement (ALGOL requirements).

V-3.2.3 Arden, et al. Scheme[5]

The scheme developed by Arden, et al., (and implemented on the General Electric 645 and IBM 360/67 computers) considers a machine address to be a triplet (s, p, i) rather than a pair (s, i) . Physical address generation can be illustrated by the following diagrams:



Physical Address := M[M[M[STR+S] + p] + i]

ℓ_p and ℓ_i indicate page table lengths and page lengths so that automatic error checks occur if $p > \ell_p$ or $i > \ell_i$.

In this scheme, which is proposed for time-sharing systems, each user has his own segment table and the STR register contains the segment table base for the user currently in control; the page and segment table entries also have an availability bit to indicate whether the page or segment is in memory or not. The triplet is used since it is anticipated that pages and page tables will be shared by many users (see section on Invariant Programs).

To save storage references through page and segment tables, several associative registers containing (s, p, physical page base) can be used. Address generation then consist of a parallel hardware search through the associative registers; if a match is found, the line number is added to the physical page base stored in the register; otherwise, the segment and page tables must be searched, as before. The associative registers are controlled by the monitor so that the most frequently used page addresses are stored there. It appears that this method will be in common use in the future.

V-3.3 Memory Protection

When user programs run under the control of a monitor, it is imperative that there be hardware and/or software to also control and restrict the blocks of memory that are available and unavailable to a particular user. A block, page, or segment of memory may have one of four types of access allowed:

1. Read and Write

   This is the "classical" type of access; the block may be read from or written into - both loads and stores are allowed. Program data blocks are usually read and write,

2. Read only

   A block may be read but not written into - loads but no stores. When several programs share the same procedure, the shared procedure is read only.

3. Write only

   Only stores are allowed to the block.

4. Neither read nor write

Both read and write access are prohibited. This protects independent programs and data from access by other programs.

The IBM 360 provides read write, read only, and neither read nor write access. A 4-bit "key" identifies each memory block; each program is also given its own "key". For read-write access, program keys must match memory block keys: an additional fetch protect bit is used for read-only protection, on the 360/67. Hardware interrupts occur on protection violations.

Segment and page table entries have length indicators indicating the segment or page size; these are checked during physical address computation to further check for memory protect violations.

V-3.4 Invariant Programs

In the early days of computer programming, there was much emphasis on computer instruction codes that modified themselves during the computation. For example, to compute a sum, the following self-modifying instruction sequence in MAP could be used:

```
        Initialize
LOOP    CLA    *+3
        ADD     =1
        STO    *+1
        ADD      A
        STO    SUM
       -end    test-
        TRA    LOOP

 SUM    BSS 1
   A    BSS 1
        BSS 100
```

Later, the use of index registers to store and compute addresses made instruction self-modification unnecessary. Looping and subroutine transfers, the two principal areas where programs might have to change themselves, can be accomplished easily with index registers:

1. Looping:

   The loop: "for i := 1 step 1 until M do S"
($M \geq 1$) can be written in MAP as:

```
        CLA   M
        ALS   18
        STD   B
        AXT   1, 1
L
        ---
         S
        ---
A       TXI   *+1, 1,1
B       TXL   L, 1,**
```

2. Subroutine Transfer and Return

```
        TSX   SUB,4                    SUB  SXA   L,4

                                       L    AXT   **,4
                                            TRA   1,4
```

The current trend is to eliminate self-modifying programs. In multiprogramming and time-sharing systems, invariant procedures, that is, procedures that do not modify themselves, are shared by many programs (page and segment tables of several programs point to the same area for these procedures). The invariant procedures may be library programs of several different types - evaluation of mathematical functions, sorting routines, editing and formating routines, etc. It is these invariant procedures that must be read-only protected.

v-4. Loosely Connected Parallel Processes[10, 11, 12, 13]

To achieve faster speeds and allow computer-to-computer communication, computer systems designers connect several independent processors to common memory banks and control circuitry, and run these in parallel. This includes central processors, I-O processors, data channels, and special purpose processors, such as a floating point arithmetic processor. With this type of arrangement, more than one program and parts of a single program can be executed in parallel and communicate with each other.

In general, we have many processes operating in parallel and communicating with one another by means of common variables. In such a situation it is necessary to ensure that no conflicts arise in accessing these variables. Two examples of these loosely connected processes should clarify these ideas:

1. I-O processing

An I-O area in storage (or buffer area) may be filled or emptied by the central processor or by I-O processors. The system must be programmed so that the common variables, the I-O area, are not accessed by more than one processor at a time. One special method for this case is the multiple buffer system described in chapter IV.

2. General file Processing

When several central processors have access to a common file, such as a payroll, accounting, or inventory file, access must be restricted to one processor at a time in order to maintain accurate files; if not, it is possible for the same item to be updated simultaneously by more than one processor and only one of the updates would then be recorded instead of all of them.

V-4.1 Programming Conventions for Parallel Processing

Following Wirth,[10] the parallel execution of two or more ALGOL statements will be indicated by replacing the -semicolon separating the statements by the symbol and . For example, to compute $\sum_{i=1}^{N} a_i b_i$ in two parallel parts, the program (minus declarations) is:

```
s1 := s2 := 0;
for i := 1 step 1 until N ÷ 2 do
            s1 := s1 + a[i] X b[i]

     and

for j := N ÷ 2 + 1 step 1 until N do
s := s1 + s2
```

A matrix multiplication program[10] computing A := B X C, where all elements of A can be computed simultaneously is:

```
integer array A[1:m, 1:n], B[1:m, 1:ℓ], C[1:ℓ, 1:m];
procedure product(i, j);
value i, j; integer i, j;
begin
     integer k; real s;
     s := 0;
     for k := 1 step 1 until ℓ do
          s := s + B[i, k] X C[k, j];
     A[i, j] := s
end product;
```

```
procedure  column(i, j);
value i, j;  integer i, j;
product(i, j) and
     if j > 1 then  column(i, j - 1);
procedure  row(i);
value  i; integer i;
column(i, n)  and
     if  i > 1 then row(i - 1);
row(m)
```

V-4.2 The Control Problem for Loosely Connected Processes

The problem and its environment can now be stated more precisely. We are given several sequential processors which can communicate with each other through a common data store. The programs executed by the processors each contain a "critical section" (CS) in which access to the common data is made; these programs are considered to be cyclic. The problem is to program the processors so that, at any moment, only one of the processors is in its critical section; once a processor, say A, enters its critical section, no other processor may do the same until A has left its CS .

The following assumptions are made about the processors:

1. Writing into and reading from the common data store are each undividable operations; simultaneous reference to the same location by more than one processor will result in sequential references in an unknown order.

2. Critical sections may not have priorities associated with them.

3. The relative speeds of the processors are unknown.

There are two possible types of blocking which the solution to the problem must prevent:

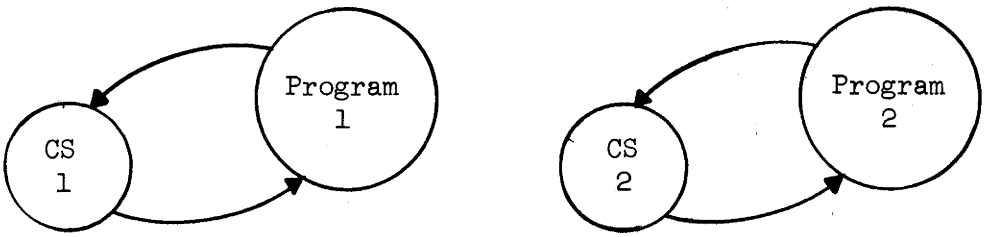
1. A program operating well outside its CS cannot then be blocking another program from entering its CS .

2. Several programs (or processors) about to enter their CS's cannot, by an "after you" - "after-you"" type of intercommunication, postpone indefinitely the decision on which one actually enters.

We will now try to develop solutions to the problem and illustrate some of the pitfalls that exist.

V-4.3 Solving the Problem

The problem will be restricted to 2 processors, each with its own CS:



Processor 1 Processor 2

1. Example 1

```
begin integer turn; turn := 2;
      P1: begin L1:  if turn = 2 then go to L1;
                 CS1:  turn := 2;
                 program 1; go to L1
          end       and
      P2: begin L2:  if turn = 1 then go to L2;
                 cs2;  turn := 1;
                 program 2; go to L2
          end
end
```

Unfortunately, neither P1 nor P2 may enter its CS twice in succession; the program insists that they enter alternately.

2. Example 2

An attempt is made to avoid the mutual blocking in example 1 by defining two common variables, C1 and C2 .

```
Begin l e a n  C1, C2; C1 := C2 := true;
     P1: begin  L1:  if  ¬ C2 then go to L1;
                     C1  := false; CS1;
                     C1  := true; program 1;
                     go to  L1
         end      and
     P2: begin  L2:  if ¬ C1  then go to  L2;
                c2  := false; CS2;
                c2  := true; program 2;
                go to  L2
         end
end
```

When C1 or C2 is false (true), the corresponding process is inside (outside) its critical section. The mutual blocking of example 1 is now not possible but both processes may enter their CS's together; the latter can occur since both programs may arrive at L1 and L2 together with C1 = c2 = true.

3. Example 3

The mutual execution of example 2 is avoided by setting C1 and C2 false at L1 and L2 respectively:

```
begin Boolean C1, C2; C1 := C2 := true;
      P1: begin A1: C1 := false;
                 L1: if ¬ C2 then go to L1;
                CS1; C1 := true;
                 program 1;  go to A1
          end       and
      P2: etc. . . .
end
```

The last difficulty has been resolved but mutual blocking is now possible again. C1 may be set false at A1 at the same time that C2 is set false at A2; in this case, both P1 and P2 will loop indefinitely at L1 and L2 . The obvious way to rectify this is to set C1 and C2 true after testing whether they are false at L1 and L2 .

4. Example 4

```
begin Boolean C1, C2; C1 := C2 := true;
      P1: begin L1: C1 := false;
                 if 1 C2 then begin C1 := true;
                                    go to L1
                              end;
                CS1; C1 := true;
                program 1; go to L1
          end      and
      P2: etc.----
end
```

Unfortunately, this solution may still lead to the same type of blocking as in the last example; if both processes are exactly in step at L1 and L2 and their speeds are exactly the same for each succeeding instruction, the same loop as before will develop around L1 and L2 .

The above attempts illustrate some of the subtleties underlying this problem. The following solution was first proposed by Th. J. Dekker:

```
begin integer  turn;  Boolean "C1, C2;
      C1 := c2  := true; turn := 1;
     P1: begin  A1: C1  := false;
                L1:  if ¬ C2 then
                     begin if  turn = 1 then go to L1;
                           C1 := true;
                       B1:  if turn =  2 then go to B1;
                       go to  A1
                     end
                CS1; turn := 2;
                C1 := true; program 1;
                go to  A1
          end       and
     P2: etc. ---
end
```

C1 and C2 ensure that mutual execution does not occur; "turn" ensures that mutual blocking does not occur.

Dijkstra[11] has developed a solution to the more general problem where there are n processes, instead of only 2, operating in parallel. If it was further stipulated that no individual process be indefinitely blocked, both the above solution and Dijkstra's solution would fail; for example, if in Dekker's program, the speed of processor 2 is much greater than that of processor 1, it is possible for processor 1 to loop indefinitely at L1 while processor 2 executes its cycle continuously. This problem is considered in Reference 13.

V-4.4 The Use of Semaphores

While Dekker's and Dijkstra's programs solve the given problem, there are, nevertheless, two unappealing features of them:

1. The solution is mystifying and-unclear in the sense that a simple conceptual requirement, mutual exclusion, leads to cumbersome additions to programs.

2. During the time when one process is in its critical section, the other processes are continually accessing and testing common variables; to do this, the waiting processors must "steal" memory cycles from the active one. The result is a general slowing down of the active process by other processes that are not doing any useful work.

An improved solution can be obtained by adding two new primitive or basic operations (Dijkstra[12]). These primitives, designated V and P, operate on integer non-negative variables, called "semaphores"; it is the semaphores that perform the communications among processes. The V and P operations are defined as follows:

1. V(S) (S a semaphore variable). S is increased by 1 . This is not equivalent to S := S+1 . e.g., If S = 5 and 2 processes call V(S) simultaneously, both V-operations will be performed (in some order) with the result that S = 7; however, if the ALGOL S := S+1 is executed by each process, it is possible for each process to fetch S when it is 5, increment it by 1, leaving S = 6 - i.e., S has only been incremented once instead of twice. V(S). does the fetch, increment, and store as one operation.

2. P(S) (s a semaphore variable). P(S) decrements S by one, if possible. If s = 0, then it is not possible to decrement S and remain in the domain of non-negative integers; in this case, the P-operation waits until it is possible.

Let us apply these primitives to the mutual exclusion problem with n processes:

```
begin integer mutex; mutex := 1;
     P1: begin . . n end   d
     P2: . . . . . .
     Pi: begin Li: P(mutex); CSi; V(mutex);
                   program i; go to Li
         end   and
        .
        .
    Pn: . . . . . , .
end
```

mutex = 0 when one of the processes is in its critical section; otherwise, mutex = 1 . Mutual execution of CS's cannot happen since mutex can't be decremented below zero by the P-operation. It should be noted how much simpler and clearer the solution is when the V and P-operations are employed. Some more general applications of semaphores will be illustrated next.

V-4.4.1 2-Processes Communicating via an Unbounded Buffer

A "producer" process produces information for the buffer and a "consumer" process consumes information from the buffer; this is analogous to the situation where-a CPU fills an output buffer and a data

channel consumes or empties the buffer contents. The following two semaphores are used:

n = number of queued portions of output of the producer and input to consumer,

b = { 0 indicates adding to or taking from buffer is occuring
    { 1 indicates buffer access routines are not active.

The critical sections are the buffer access routines, "Add To Buffer" and "Take From Buffer".

```
begin integer n, b; n := 0; b := 1;
     producer: begin L_p: produce next portion of data;
                          P(b); Add To Buffer; V(b);
                          V(n); go to L_p
               end        and
     consumer: begin L_c: P(n);
                          P(b); Take From Buffer; V(b);
                          Process Portion; go to L_c
               end
end
```

The two most common methods of organizing a buffer are the cyclic method (Chapter IV) and the chaining method, where each portion of the buffer is an element in a linked list or chain. In the latter case, adding or taking from the buffer simultaneously can disturb the linkages; the semaphore b ensures the mutual exclusion of the critical sections, Add To Buffer and Take From Buffer.

In general, it is always possible to replace a general semaphore (taking all non-negative integer values) by one or more binary semaphores (taking 0 or 1). Below, the last example is programmed using binary semaphores only; the simple integer variable n and the binary semaphore d are used instead of the general semaphore n :

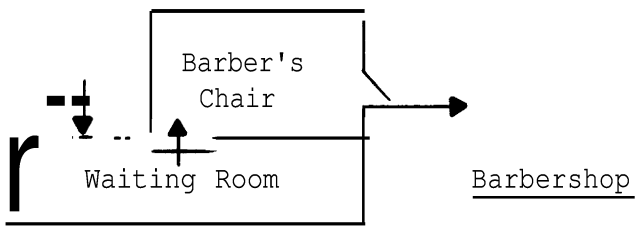
```
begin  integer  b, n, d; b := 1; n := d := 0;
     producer: begin: L_p : produce next portion;
                      P(b); Add To Buffer; n := n+1;
                      if  n = 1  then  V(d);
                      V(b); go to L_p
               end          and
     consumer: begin integer oldn;
                      L_c: P(d);
                      L_x: P(b); Take From Buffer; n := n-1;
                           oldn := n; V(b); Process portion;
                           if oldn ≠ 0 then go to L_x  else go to  L_c
               end
end
```

Another solution, called "The Sleeping Barber", presents the actions of the producer and consumer more clearly:

```
begin integer b, n, d; b := 1; n := d := 0;
      producer: begin L_p : produce next portion;
                      P(b); Add To Buffer; n := n+1;
                      if  n = 0  then  V(d);
                      V(b); go to L_p
                end       and
```

```
consumer: begin L_c: P(b); n := n-1;
                  if  n= -1 then
                  begin V(b); P(d); P(b) end;
                  Take From Buffer;
                  V(b); Process portion;
                  go to L_c
          end
end
```

When n= -1 outside of CS execution, the buffer is empty and the consumer, having noted this, is waiting. The "sleeping barber" story goes as follows:



Customers enter the waiting room and the Barber's room through a sliding door that only admits entrance to one of the rooms at a time (mutual exclusion of customer producer and consumer); the entrances are designed so that only 1 customer may come into or leave the waiting room at a time. When the barber finishes a haircut, he inspects the waiting room by opening the door (P(b) at L_c); if the room is not empty, the next customer is invited in ($n \neq -1$); if the room is empty ($n = -1$), the barber goes to sleep (waiting at P(d)) . When a customer enters and finds a sleeping barber, he awakens him.

V-4.4.2 Processes Communicating via a Bounded Buffer

The general semaphore is applied to the last problem in a. more realistic setting - a bounded buffer. N is the buffer size, in portions and is a global variable in the program. Two general semaphores are used:

n = number of empty portions in buffer

m = number of queued portions

b is a binary semaphore ensuring mutual exclusion of critical sections.

```
begin integer m, n, b; m := 0; n := N; b := 1;
     producer: begin  L_p : produce next portion; P(n);
                      P(b); Add To Buffer; V(b);
                      V(m); go to L_p
               end       and
     consumer: begin  L_c: P(m);
                      P(b); Take From Buffer; V(b);
                      V(n); process portion;
                      go to L_c
               end
end
```

V-5. References

1. Noble, A. S., Jr. Design of An Integrated Programming and Operating System. Part I: System Considerations and the Monitor. IBM Systems Journal 2, (June 1963),153-161.

2. Master Control Program Characteristics, B5500 Information Processing System. Bulletin 5000-21003-D, Burroughs Corp. May 1962.

3. Desmonde, W. H., Real-Time Data Processing Systems: Introductory Concepts Prentice-Hall, Inc., N. J., 1964.

4. Clark, R., Miller, W. F., Computer-Based Data Analysis Systems. Methods of Computational Physics, 5 (1966). Academic Press. pp. 47-98.

5. Arden, B. W., Galler, B. A., O'Brian, T. C., and Westervelt, F. H., Program and Addressing Structure in a Time-Sharing Environment. J. ACM 13 (January 1966), 1-16.

6. A New Remote Access Man-Machine System. AFIPS Conference Proceedings Fall 1965 Part 1 Spartan Books. pp. 185-247.

7. IBM 7090/7094 Programming Systems, FORTRAN II Assembly Program (FAP). Form C28-6235-4. IBM Corporation, 1963.

8. Nauer, P., The Performance of a System for Automatic Segmentation of Programs Within an ALGOL Compiler (GEIR ALGOL). Comm. ACM, 8, 11 (Nov. 1965) 671-676.

9. McGee, W. C., On Dynamic Relocation. IBM Systems Journal, 4, 3 (1965) 184-199.

10. Wirth, N., A Note on "Program Structures for Parallel Processing." Comm. ACM 9, 5 (May, 1966), 320-321, (letter to the editor).

11. Dijkstra, E. W., Solution of a Problem in Concurrent Programming Control. Comm. ACM 8, (September, 1965), 569.

12. Dijkstra, E. W., Cooperating Sequential Processes (Preliminary Version). Mathematics Department, Technological University, Eindhoven, The Netherlands, September, 1965.

13. Knuth, D. W., Comm. ACM 9, 5 (May, 1966), 321-322, (letter to the editor).

14. Dennis, J. B., Segmentation and the Design of Multiprogrammed Computer Systems. J. ACM 12, 4 (Oct. 1965) 589-602.

v-6. Problem

"Prove" that Dekker's solution, to the mutual exclusion problem is correct.

VI. COMPILERS - AN INTRODUCTION

The next 3 chapters are devoted to the description of the main techniques and formal methods that are useful for designing mechanical languages and their compilers.

VI-1. Tasks of a Compiler

A translator whose input is a language with some "structure' will be called a compiler; most interpretations of the word "compiler" are included in this definition. Specific examples will be restricted to compilers of algebraic languages - ALGOL and FORTRAN being the two most common ones.

To understand the meaning of 'structure" in the above definition, solutions to the same problem are coded in MAP, FORTRAN, and ALGOL:

Problem

Given: $a_i, b_i \quad i = 1, 100$

compute: $c_i = \begin{cases} a_i & \text{if } a_i > b_i \\ b_i & \text{if } a_i \leq b_i \end{cases} \quad i = 1, 100$

MAP Solution

```
      AXT   1,1
LOOP  CLA   A,1
      CAS   B,1
      TRA   UNEQ
      TRA   EQ
UNEQ  STO   C,1
BUMP  TXI   *+1,1,1
      TXL   LOOP,1,101
      HTR
EQ    CLA   B,7
      TRA   UNEQ
```

FORTRAN Solution

```
      DO  100 I = 1, 100
          IF (A(1) - B(1)) 10,10,20
10        C(I) = B(1)
          GO TO 100
20        C(I) = A(1)
100   CONTINUE
```

ALGOL Solution

```
begin real array  A,B,C[1:100]; integer i;
     for i  := 1 step 1 until 100 do
          C[i] := if' A[i] > B[i] then A[i] else B[i]
end
```

The most significant feature that distinguishes these three solutions (and the languages) from each other is the degree of structure in the programs. The logical flow of the MAP solution is indicated through the extensive use of labels and transfer instructions. The statements are simple, almost independent of each other, and it is easy to decompose them into component parts. In contrast, the ALGOL solution is highly structured; the structure itself exhibits the logical flow. Each ALGOL statement must be analyzed into component statements and parts; for example, in the above solution there is a Boolean expression which is part of an arithmetic expression which is part of an assignment statement which is part of a block which constitutes the program. The FORTRAN solution lies somewhere between these two extremes.

The basic tasks of a compiler are:

1. Recognition of the Basic Parts of the Source or Input Language.

The source program must be exhaustively scanned to recognize and construct its primitive components; these may include identifiers, numbers, delimiters, and other basic units.

2. Analysis of the Structure of the Language.

The scope and constituent parts of the input statements are determined. This is a recursive process since statements may consist of sets of other statements each of which again must be analyzed for scope and constituents. Output reflecting this structure is produced.

3. Processing of Symbolic Names.

The declaration and use of symbols must be linked; this is very similar to the symbol processing performed in an assembler.

4. Transformation of Arithmetic Expressions Into a Sequence of Simple Operations.

Arithmetic expressions are analyzed to transform them into sequences of elementary arithmetic operations. Structure in arithmetic expressions was a feature of most of the early algebraic languages and many techniques were developed to analyze them.

5. Storage Allocation.

When the output language is a machine language, real or "virtual" storage must be allocated for programs and data.

Expressions compilation methods are briefly surveyed in the remainder of this chapter. The environment is relatively simple, yet it provides insights and clues to compilation methods in general.

VI-2. Heuristic Techniques for Expression Compilation'

VI-2.1 Rutishauser (1952)

The expression is repeatedly scanned, each time extracting the innermost subexpression; elementary arithmetic operations are generated for the selected subexpression and it is replaced by a single operand in the original. The first scan, from left-to-right, assigned level numbers to each element of the expression - operands and "(" increment level numbers while operators and ")" decrement them. The innermost subexpressions are defined by the highest level number; the numbers are updated as subexpressions are replaced.

Example

Level numbers appear under the expression elements.

Scan No.	Expression After Scan	Generated Operations
1	$(A_1 \div (A_2 + A_3)) - (A_1 \times A_2 \times A_3)$ 012 1 23 2 3 21 0 12 1 2 1 2 10	
	$(A_1 \div R_1) - (A_1 \times A_2 \times A_3)$ 012 1 2 1 0 12 1 2 1 2 10	$R_1 := A_2 + A_3$
	$R_2 - (A_1 \times A_2 \times A_3)$ 01 0 12 1 2 1 2 10	$R_2 := A_1 \div R_1$
4	$R_2 - R_3$ 01 010	$R_3 := A_1 \times A_2 \times A_3$
	R 010	$R := R_2 - R_3$

VI-2.2 FORTRAN Compiler (1954 +)

The emphasis in the first FORTRAN compiler was placed on producing efficient code for the 701 computer. Expression compilation was a 5-pass task with the following functions:

PASS 1 : Replace all constants and subscripted variables by simple variables. e.g., A +B↑3/Y(6) becomes A + B↑C/D

PASS 2 : Insert all parenthesis in expression so that operator precedences are explicit. e.g., A + B↑C/D becomes

(((A))) + (((B))↑(C))/((D)))

PASS 3 : Break expression into subexpressions or "segments." e.g., the expression (((A+B) - C)/((Dx(E+F)/G) - H+J)) (extra parentheses are omitted for simplicity) breaks into 6 segments:

1. (A + B)
2. ((A + B) - C)
3. (E + F)
4. (D x (E + F)/G)
5. ((D x (E + F)/G - H + J)
6. (((A + B) - C)/((D x (E + F)/G) - H + J))

PASS 4 : Triplets of the form (segment no., operator, operand) are compiled from each segment. The segments of pass 3 are translated into the triplets:

(1, +, A)	(1, +, B)	
(2, +, 1)	(2, -, C)	
(3, +, E)	(3, +, F)	
(4, x, D)	(4, x, 3)	(4, /, G)
(5, +, 4)	(5, -, H)	(5, +, J)
(6, x, 2)	(6, /, 5)	

PASS 5 : Repeated scans of the triplets are made-deleting redundant parenthesis, removing triplets corresponding to common subex-

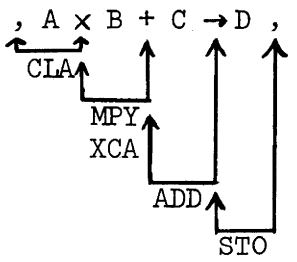
pressions, re-ordering triplets to minimize fetch and stores, and finally, generating assembly code.

VI-2.3 NELIAC (a dialect of ALGOL 58)[2]

A tabular technique was used in which pairs of operators, the current operator (COP) and the next operator (NOP), are used to generate code in a single scan from left to right.

Example

COP \ NOP	,	+	→	×
,		CLA	CLA	LDQ
+	ADD	ADD	ADD	STO T LDQ
→	STO			
x		MPY XCA	(ADD T) MPY XCA	



generates

```
LDQ  A
MPY  B
XCA
ADD  C
STO  D
```

The method is very fast but expressions are severely restricted so that only 1 temporary storage cell T is needed--no parenthetical nesting of expressions is allowed and only 2 levels of operator hierarchy exist. The pair (COP, NOP) actually acts as a 2-dimensional switch to branch to an appropriate subroutine.

VI-2.4 Samelson and Bauer (1959)[3]

Two symbols at a time were compared as in the NELIAC method but Samelson and Bauer introduced the push-down store (stack or cellar) for saving operators and temporary results:' Symbol pairs were used to access an element of a two-dimensional "transition matrix" which selected the appropriate action.

Example: $(a \times b + c \times d)/(a - d)$ is translated into:

```
R1 := a; R2 := b; R1 := R1 × R2;
R2 := c; R3 := d; R2 := R2 × R3;
R1 := R1 + R2; R2 := a; R3 := d;
R2 := R2 - R3; R1 := R1/R2;
```

where Ri are the stack elements.

VI-2.5 Dijkstra (1960)[4]

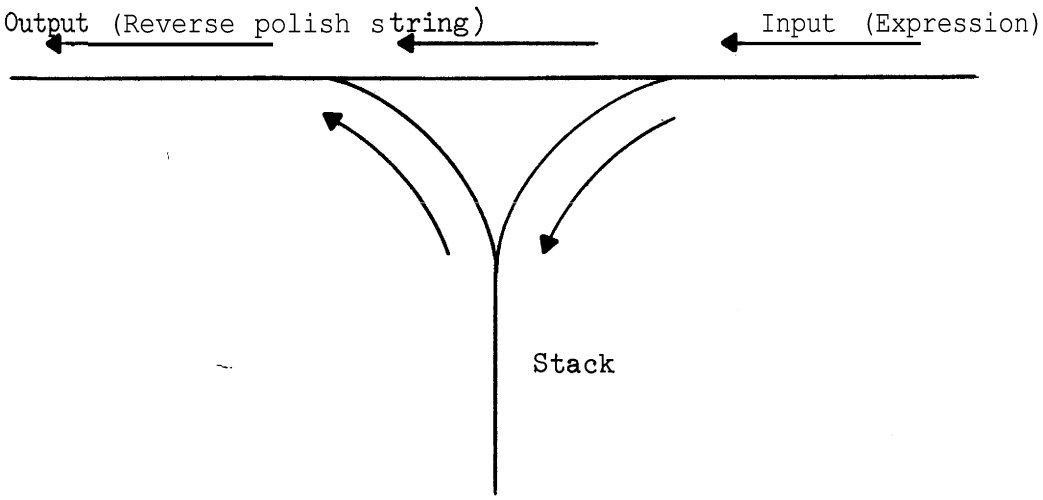
Dijkstra used an extension of the stack techniques of Samelson and Bauer in his implementation of the first ALGOL 60 compiler. He demonstrated that the cellar principle is also appropriate for other constructions of ALGOL beyond expressions. Dijkstra's method and modifications of it form the basis for many algebraic compilers; the next section presents a general description of it.

VI-3. Compilation of Expressions Using a Stack[1]

An arithmetic expression can be easily converted to a reverse or postfix Polish string with the aid of a stack. This string can be viewed

as the sequence of elementary arithmetic operations represented by the original expression.

The process is analogous to a "T-shaped" railway shunting system with the shunting or re-ordering performed in the vertical bar of the "T":



Operands take the direct route to the output while operators pass through the stack. Priorities are defined for the operators to reflect their precedences; for example:

priority("↑") > priority("x") > priority("+") .

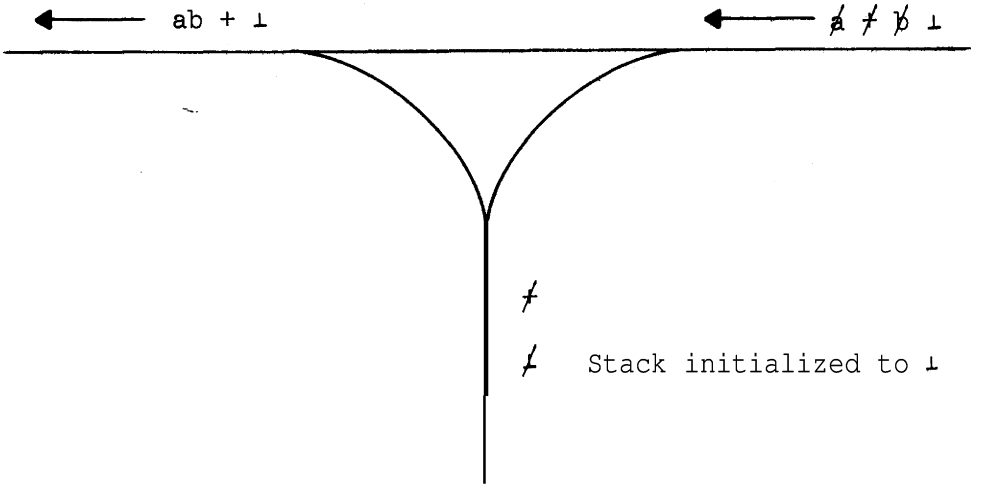
Assuming the input string is an arithmetic expression consisting of operators and operands, conversion to reverse Polish goes as follows:

1. if nextsymbol(input) = operand then pass it through to the output
   else
2a. if priority(operator at top of stack) ≥ priority(incoming operator)
   then pass stack operator to output else
2b. move incoming operator to top of stack.

Example 1

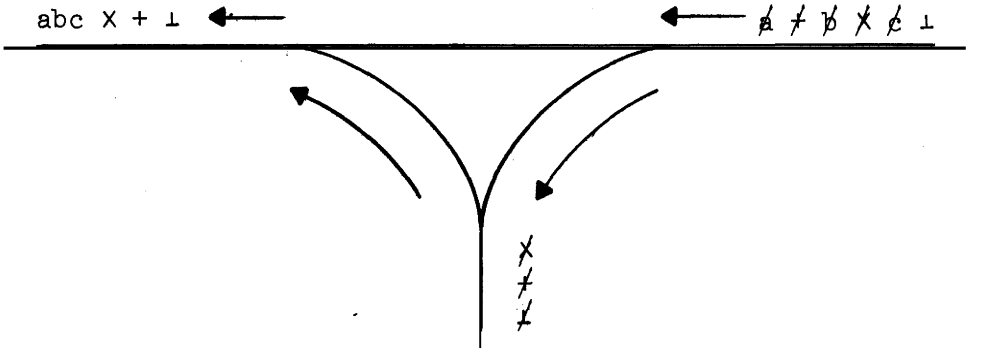
Priority Table

Operator	Priority
+	1
X	2
↑	3
⊥	-∞ (expression termination operator)



The termination symbol ⊥ at the end of the expression is not put into the stack (a special case); its use is to cause total unstacking at the end of the expression.

Example 2



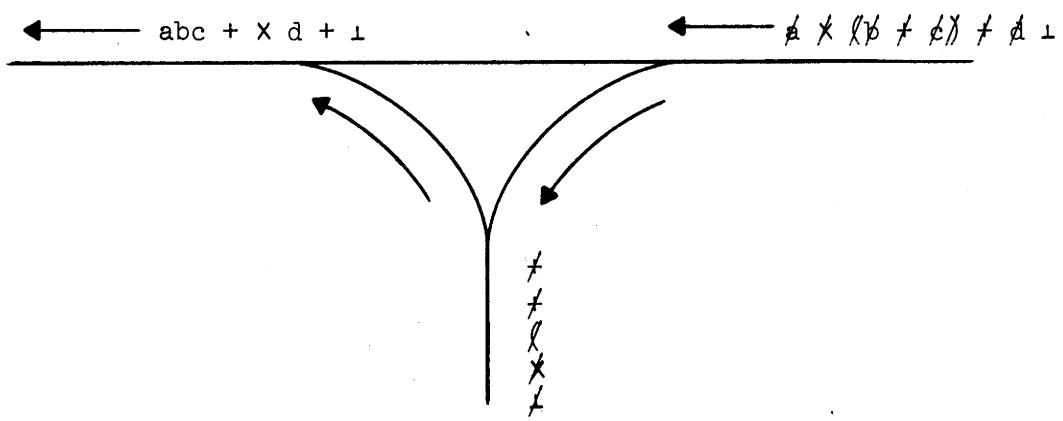
Parenthesis may be handled by modifying the algorithm. Two kinds of priorities are defined for operators - a stack priority which holds when the operator is in the stack and a compare priority which holds with the operator is the incoming symbol. The priorities are determined so that a "(" is automatically stacked and remains there until its corresponding ")" arrives; the ")" then causes unstacking to its" (" .

Step 2a. must be changed to:

2a'. if stackpriority(operator at top of stack) ≥ comparepriority (incoming operator) then pass stack operator to output e

Example 3

Operator	Stack Priority	Compare Priority
(	0	4
+	1	1
×	2	2
↑	3	3
)	—	1
⊥	-∞	-∞



")" is never stacked; after unstacking down to "(", both "(" and ")" are deleted. Disecting the operation of the method in this example, we have:

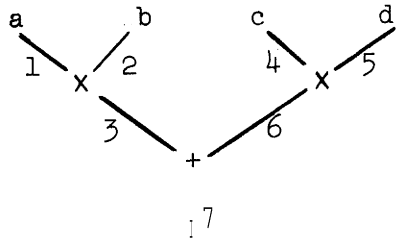
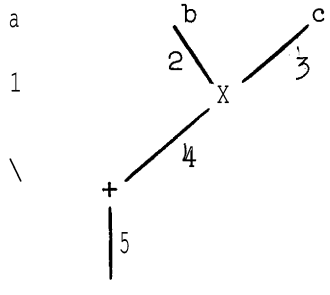
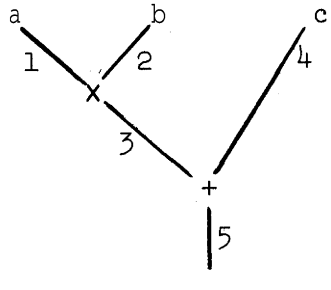
Incoming Symbol S	After Processing S	
	Stack	output
a	⊥	a
X	⊥X	a
(	⊥X(	a
b	⊥X(	ab
+	⊥X(+	ab
c	⊥X(+	abc
)	⊥X	abc+
+	⊥+	abc+X
d	⊥+	abc+Xd
⊥		abc+Xd+⊥

Relational operators ($<$, $\leq$, $\geq$, ...), Boolean operators ($\wedge$, $\vee$, $\neg$), and the remaining arithmetic operators can be included by adding their priorities to the table. Subscripted variables can be handled by treating the subscript brackets, "[" and "]", and the commas separating the subscripts in a similar manner as parentheses. Finally, conditional expressions, simple statements, conditional statements, and compound statements can all be transformed into a meaningful sequence of reverse Polish operations by establishing priorities for the delimiters and using the shunting algorithm.

The transformed expression - the reverse Polish representation of the input string - can directly correspond to a sequence of instructions for a stack computer (see the stack interpreter in Chapter III).

VI-4: Phrase Structure Methods

These methods use the formal definition of the language directly. Expression compilation - and compilation in general - is based on a mechanical parse of the input program which exhibits its structure. These parses may be conveniently represented as trees:

Expression	Tree Representation
a X b + c X d	
a + b X c	
a X b + c	

The numbering of the tree elements is performed by a left-to-right and top-to-bottom systematic count. If the elements are ordered according to number, the result is their reverse Polish representation. This is

not an accident. Precedences are implicit in the formal definition of the language and the parse automatically produces the reverse Polish.

Present production compilers are based on the heuristic and stack methods. The more formal phrase structure schemes are of recent origin and have been applied to several successful experimental systems. They appear to offer great promise for changing compiler writing from an art to a science, The next chapter develops the main ideas of Phrase Structure Programming Languages and their translators.

VI-5. References

1. Randall;' B., and Russell, L. J., ALGOL 60 Implementation. Academic Press, London and New York, 1964.

2. Halstead, M. H., Machine-Independent Computer Programming. Spartan Books, Washington, D.C., 1962.

3. Samelson, K., and Bauer, F. L., Sequential Formula Translation. Comm. ACM, Vol. 3, pp. 76-83 (Feb. 1960).

4. Dijkstra, E. W., Making a Translator for ALGOL 60. Annual Review In Automatic Programming, Vol. 3, pp. 347-356 (1963).

VI-6. Problems

1. Produce the reverse Polish representation of the following arithmetic expressions:

(1) a + b x c↑(d÷e)/f

(2) (((axb+c)xd+e)xf+g)↑2

(3) a+ 3 x (b-c+d) - i x(j/e↑2x(b+3xi)+c)

2. Expand the priority tables to include the Boolean operators ($\equiv$, $\supset$, 1, $\vee$, and A), the relational operators ($>$, $\geq$, $<$, $\leq$, $=$, and $\neq$), and all the arithmetic operators ($+$, $-$, $/$, $\div$, X, $\uparrow$) .

Note: special cases must be made for the unary operators. Use the shunting algorithm to translate:

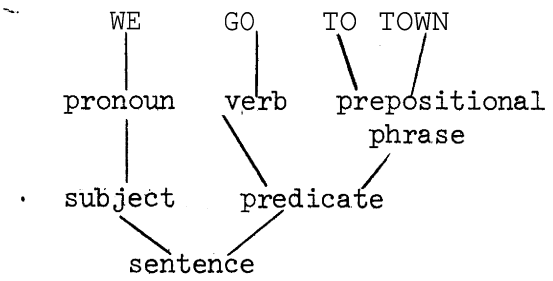
$b < - c - (d+e) \equiv e \times f + g \uparrow h > i \wedge \neg j$ into reverse Polish.

VII. PHRASE STRUCTURE PROGRAMMING LANGUAGES

VII-1. Introduction

Intuitively, a language is a set of sentences or word sequences; each sentence is formed by concatenating some words in the language vocabulary according to given composition rules. The composition rules are called the syntax of the language and define its structure. An analysis of a sentence that produces its structure or syntactical components is a parse of the sentence. A language is ambiguous if there exist sentences to which more than one structure can be assigned.

Example 1



A possible set of rules or syntax which underlies this parse is:

pronoun → WE

pronoun → YOU

noun → CHILDREN

verb → GO

verb → DRIVE

prepositional-phrase → TO TOWN

subject → pronoun

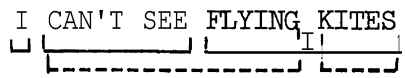
subject → noun

predicate → verb

predicate → verb prepositional-phrase

sentence → subject predicate

Example 2



The sentence is ambiguous since it can have either of the two indicated structures.

Usually, a set of rules and a string are given and the question "Is the string a sentence of the language?" must be answered; if the string can be parsed, the answer is "Yes". It is rarely required to do the opposite - i.e., generate a sentence from a given set of rules. (In computing, programmers generate strings of code; compilers analyze them.) A syntactic analysis can be used to help determine the meaning or semantics of sentences; for example, given the meaning of the subject and predicate in Example 1, the meaning of the entire sentence can be determined. Meaning is obtained by associating a semantic or interpretation rule with each syntactical rule. Semantic rules can also indicate when "meaningless" sentences have been successfully parsed.

These notions will now be formalized, extended, and applied to programming languages and compilers.

VII-2. Representation of Syntax

The most common method for expressing the syntactical rules of a language is by a straightforward list of productions, each of the form:

$$x \rightarrow y$$

where x and y are strings over the vocabulary of the language. The vocabulary consists of non-terminal symbols, such as ⟨term⟩ or (factor), and basic or terminal symbols, such as begin, else, or + .

Example:

(if clause) → if (Boolean expression) then

(term) → ⟨term⟩ X (factor)

ADC → XC

The representation used in the ALGOL report, the Backus Normal Form (BNF), is an abbreviation of the above which allows several productions to be given on one line and uses ::= instead of → .

e.g., ⟨term⟩ ::= (factor) \ (term) X (factor)

Both will be used where convenient.

A graphical specification of syntax can be very useful, especially when writing a compiler. B5500 ALGOL syntax is expressed in a chart' using the following graphic symbols:

Symbol	Meaning
	symbol definition
	reference to symbol
	terminal

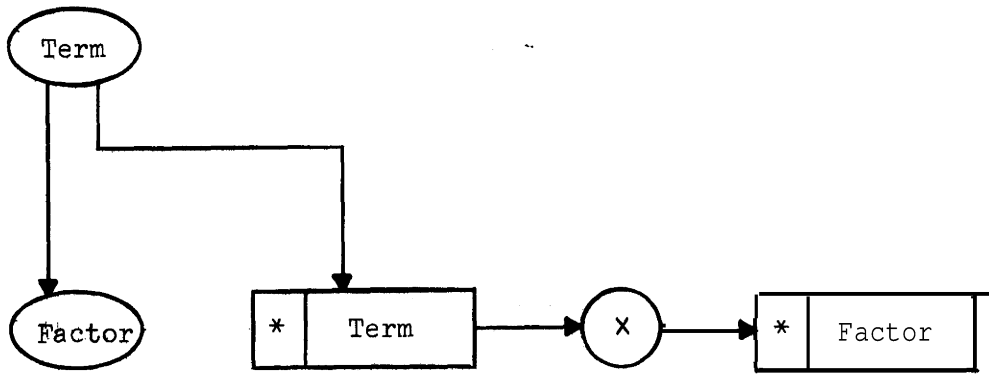
T: terminal symbol

NT: non-terminal symbol

Example

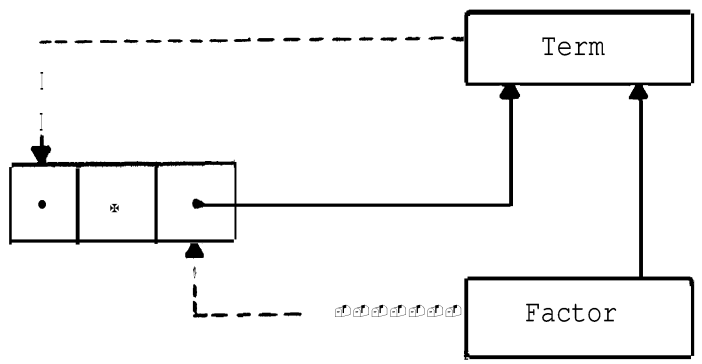
(term) ::= ⟨factor⟩ | (term) X (factor)

is expressed:

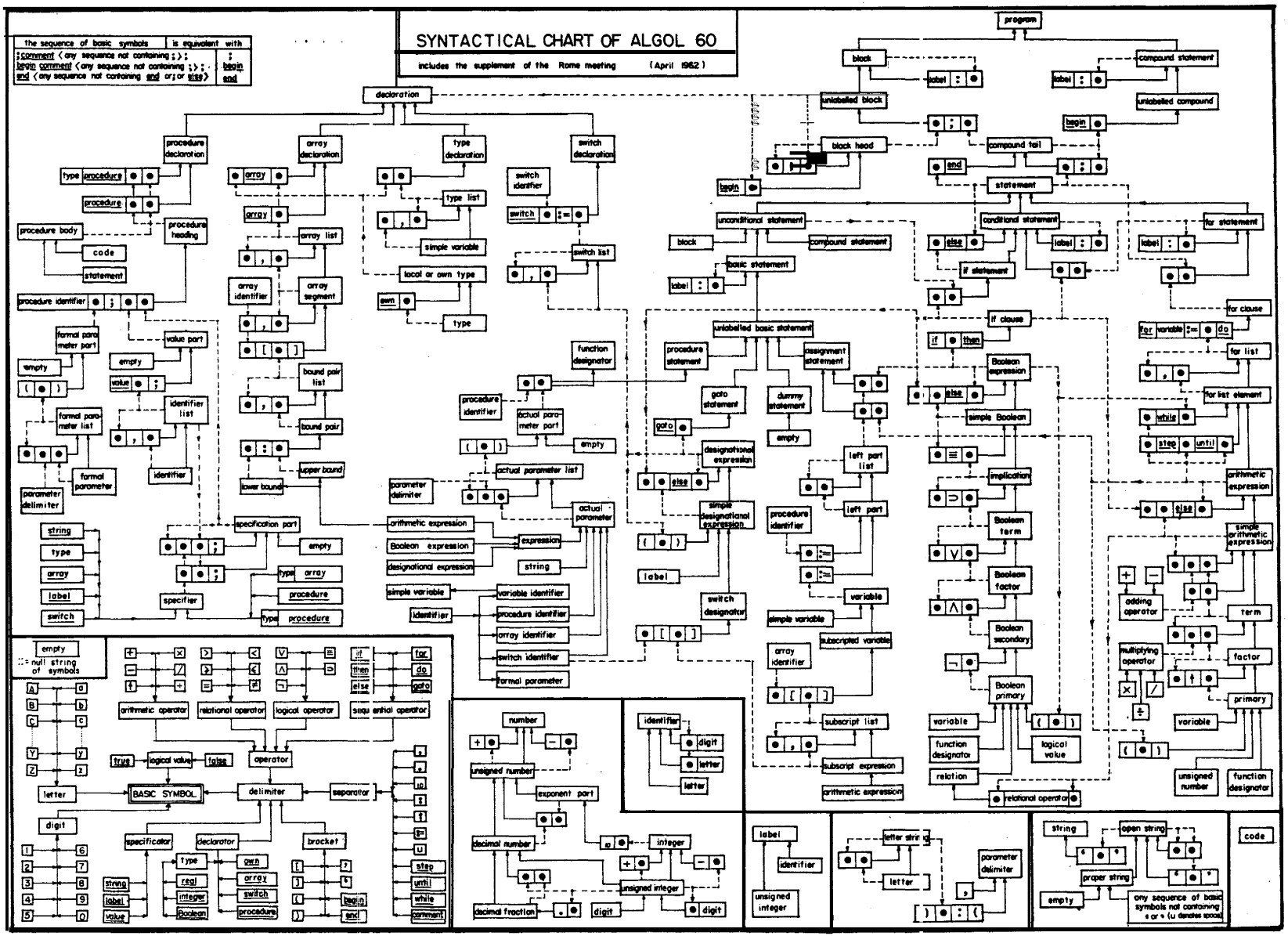


* gives a "coordinate" reference to the point of definition of the symbol in the box.

Another graphical method replaces the coordinate references by dotted lines pointing to the occurrences of the symbol:[2]



Here the directions of the arrows have been reversed to indicate reductions rather than productions. A complete specification for ALGOL using this method is given on the next page.


SYNTACTICAL CHART OF ALGOL 60
includes the supplement of the Rome meeting (April 1962)
the sequence of basic symbols | is equivalent with
;comment ⟨any sequence not containing ;⟩; | ;
begin comment ⟨any sequence not containing ;⟩; | begin
end ⟨any sequence not containing end or ; or else⟩ | end
program
block
compound statement
unlabelled block
unlabelled compound
label
begin
block head
compound tail
end
statement
declaration
procedure declaration
array declaration
type declaration
switch declaration
type procedure
procedure
procedure heading
procedure body
code
statement
procedure identifier
value part
empty
value
formal parameter part
formal parameter list
formal parameter
parameter delimiter
identifier list
identifier
array
array list
array segment
array identifier
bound pair list
bound pair
lower bound
upper bound
type list
simple variable
local or own type
own
type
switch identifier
switch list
unconditional statement
basic statement
unlabelled basic statement
conditional statement
if statement
if clause
if
then
else
for statement
for clause
for
variable
:=
do
for list
for list element
while
step
until
Boolean expression
simple Boolean
implication
Boolean term
Boolean factor
Boolean secondary
Boolean primary
variable
function designator
logical value
relation
relational operator
arithmetic expression
simple arithmetic expression
adding operator
term
multiplying operator
factor
primary
unsigned number
function designator
procedure statement
assignment statement
goto statement
dummy statement
goto
designational expression
simple designational expression
switch designator
label
left part list
left part
procedure identifier
variable
simple variable
subscripted variable
array identifier
subscript list
subscript expression
actual parameter
actual parameter list
actual parameter part
function designator
procedure identifier
empty
parameter delimiter
arithmetic expression
Boolean expression
designational expression
expression
string
simple variable
variable identifier
procedure identifier
array identifier
switch identifier
formal parameter
identifier
specification part
specifier
string
type
array
label
switch
procedure
type procedure
empty
::= null string of symbols
letter
digit
BASIC SYMBOL
arithmetic operator
relational operator
logical operator
sequential operator
operator
delimiter
separator
logical value
true
false
specificator
declarator
bracket
type
real
integer
Boolean
own
array
switch
procedure
string
label
value
step
until
while
comment
begin
end
number
unsigned number
exponent part
decimal number
decimal fraction
integer
unsigned integer
identifier
letter
digit
label
identifier
unsigned integer
letter string
parameter delimiter
letter
string
open string
proper string
empty
any sequence of basic symbols not containing ' or ; (␣ denotes space)
code

VII-3. Notation and Definitions[3]

Capital letters and letter sequences enclosed in "⟨" and "⟩" denote symbols; e.g., ⟨term⟩, A, U, ⟨sentence⟩, T . Small letters denote strings of symbols. The empty string is designated Λ . Script letters are used for sets; e.g., $\mathcal{L}$, $\mathcal{V}$, $\mathcal{P}$.

The set $\mathcal{V}$ of symbols is called the vocabulary. $\mathcal{V}^*$ is the set of strings generated over $\mathcal{V}$; formally:

$$\mathcal{V}^* = \{s \mid s = A \text{ or } (s = s'S \text{ with } s' \epsilon \mathcal{V}^*, S \epsilon \mathcal{V})\}$$

Example: $\mathcal{V} = \{A, B\}$

$\mathcal{V}^* = (A, A, B, AB, AA, AAB, \ldots.\}$

$\mathcal{P}$ is a set of syntactic rules of the form:

$$X \rightarrow y;\ x, y \epsilon \mathcal{V}^* .$$

A string x directly generates y if and only if there exist strings u, w (possibly emtpy) such that x = uvw, y = uzw and $v \rightarrow z \epsilon \mathcal{P}$. This is denoted $x \overset{\cdot}{\rightarrow} y$. e.g., Using Example 1 of section 1,

$$\langle \text{verb} \rangle \langle \text{prepositional phrase} \rangle \overset{\cdot}{\rightarrow} \langle \text{verb} \rangle \text{ TO TOWN}$$

x generates y ($x \overset{\cdot}{\rightarrow}^* y$) if there exists a sequence of strings $x = x_0, x_1, x_2, \ldots, x_n = y$ such that $x_{i-1} \overset{\cdot}{\rightarrow} x_i$, i = 1,..., n . e.g.,

$\langle \text{verb} \rangle \langle \text{prepositional phrase} \rangle \overset{*}{\rightarrow}$ GO TO TOWN

$\langle \text{sentence} \rangle \overset{*}{\rightarrow}$ WE GO TO TOWN

A phrase structure system is a pair (V, Φ). A phrase structure language (V, V_T, Φ, S) is defined:

$$\mathcal{L}(V, V_T, \Phi, S) = \{s \mid s \in V_T^*, V_T \subset V, S \in V - V_T, \text{ and } S \overset{*}{\rightarrow} s\}$$

V_T, the set of basic or terminal symbols, is the subset of V such that no element of V_T occurs as the left part of any production,

Example 1

$V = \{A, B, C, S\}$

$V_T = \{A, B, C\}$

$\Phi = \{S \rightarrow ABC\}$

$\therefore \mathcal{L} = \{ABC\}$

Example 2

$V = \{S, A, B, C, D, E\}$

$\Phi = \{S \rightarrow AB, B \rightarrow CD, C \rightarrow E\}$

$\therefore V_T = \{A, D, E\}$

The generation of $\mathcal{L}$ from S is

$S \rightarrow AB \rightarrow ACD \rightarrow AED$

$\therefore \mathcal{L} = \{AED\}$

Example 3

$V = \{S, A, B, C, D, E\}$

$V_T = \{A, D, E\}$

$\mathcal{P} = \{S \rightarrow AB,\ B \rightarrow CD,\ B \rightarrow DC,\ C \rightarrow E\}$

$\mathcal{L} = \{AED, ADE\}$

Example 4

$Y \in \{S, A, B, C\}$

$V_T = \{B, C\}$

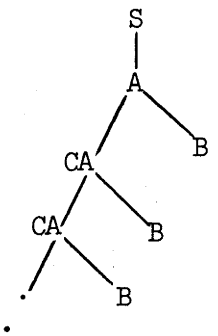
$\mathcal{P} = \{S \text{ "}A,\ A \rightarrow B,\ A\text{"} CA\}$

$\mathcal{L} = \{B, CB, CCB, CCCB, \ldots\}$ or

$\mathcal{L} = \{C^n B \mid n=0, 1, \ldots\}$

A is defined recursively here, that is, in terms of itself.

The language derivation or generation can be represented as a tree:



Example 5 Replacing the rule A → CA by A → CAC in Example 4,

$\mathcal{L}$ becomes:

$$\mathcal{L} = [B, CBC, CCBCC, \ldots\}$$
$$= \{C^n BC^n \mid n=0, 1, \ldots)$$

VII-4. Chomsky's Classification of Languages[4]

Chomsky has classified languages according to the type of productions used to generate them:

Class 0: No restrictions.

Class 1: All productions are of the form:

$$uAv \rightarrow uav$$

(u, v may be Λ).

This is sometimes called context-dependent since $A \rightarrow a$ only in the context of u, v .

Class 2: Productions are restricted to the form:

$$A \rightarrow a$$

Class 2 languages are also called context-free.

Class 3: Productions are severely restricted to either of the forms:

$$A \rightarrow B \text{ or } A \rightarrow BC$$

with $A, C \epsilon V - V_T$

$$B \epsilon V_T$$

This class of languages is also called finite-state.

There are class i languages which are not in class i+1 (for i = 0, 1, 2), so that the class to which a language belongs is some indication of its power. Most programming languages can be (almost) formulated as members of Class 2.

VII-5. The Parsing Problem[3]

A direct reduction of b into a, designated $b \xrightarrow{\cdot} a$, is an application of the production $X \rightarrow y$, where $b = uyv$ and $a = uXv$ for some u, $v \epsilon V^*$. A reduction of b into a, $b \xrightarrow{*} a$, is a sequence of direct reductions $x_i \xrightarrow{\cdot} x_{i+1}$ for i = 0,..., n-1, such that $x_0 = b$, $x_n = a$; this is also called a parse.

Example 1

$A \rightarrow BC$

$B \rightarrow DE$

$C \rightarrow FG$

Parsing or reducing the string DEFG gives:

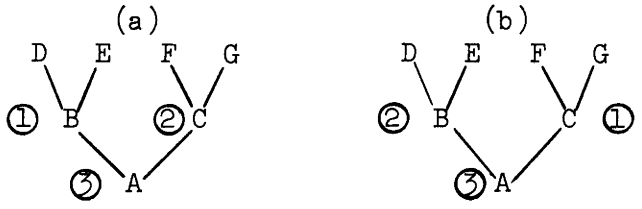
(a) $DEFG \overset{\cdot}{\rightarrow} BFG \overset{\cdot}{\rightarrow} BC \overset{\cdot}{\rightarrow} A$

or

(b) $DEFG \overset{\cdot}{\rightarrow} DEC \overset{\cdot}{\rightarrow} BC \overset{\cdot}{\rightarrow} A$

$\therefore DEFG \overset{*}{\rightarrow} A$

These reductions may be expressed as trees:

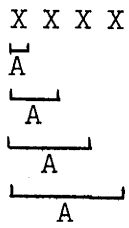


Circled numbers indicate the order of the reduction; the resulting trees are identical. The above difference in parsing, due to the order of application of the reductions, is trivial and can be eliminated by introducing a canonical ordering to parses. The canonical parse is the one that proceeds from left to right in a sentence and reduces a left-most part of a sentence as far as possible before proceeding further to the right. Thus, if $x = x_1x_2$ and $x_1 \overset{\cdot}{\rightarrow} s_1$, $x_2 \overset{\cdot}{\rightarrow} s_2$, then the reduction $x_1 \overset{\cdot}{\rightarrow} s_1$ is performed first. In this example (a) is the canonical parse.

Example 2

$A \rightarrow X$

$A \rightarrow AX$



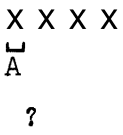
Parse

The sequence of X's is defined using a left-recursive definition.

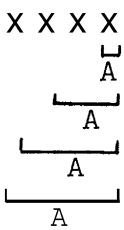
Example 3

$A \rightarrow X$

$A \rightarrow XA$



We have run into a dead end by starting the parse from the left.



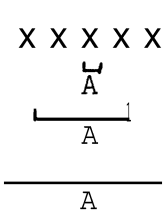
A successful parse is obtained by starting from the right.

Here, the sequence of X's is defined by a right-recursive definition.

Example 4

$A \rightarrow X$

$A\,'XAX$



The parse must start at the middle of the string at each stage.

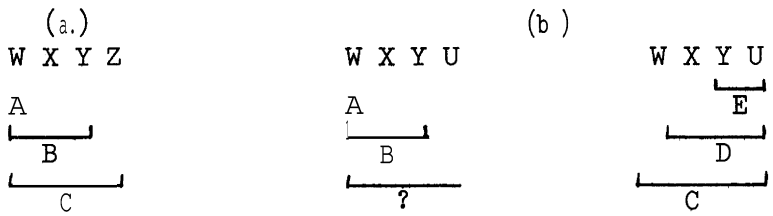
Example 5

A → BY|CZ
B → X|BX
C → X|XC

This example illustrates how the input string determines the direction and position of the reductions.

Example 6

A → WX
B → AY
C → BZ|WD
D → XE
E → YU



In (b), the first try leads to a dead end. The second, and successful, parse starts with the next reducible substring from the left, namely YU .

The parsing problem is to analyze sentences efficiently; the ideal system would have a "recognizer" that recognizes productions and determines the correct reduction to be made at any stage.

VII-6. Irons' Classification of Languages According to Parsing Difficulty[5]

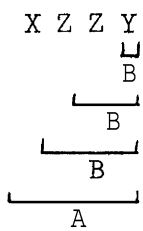
Irons suggests that languages be classified "according to the complexity of interaction between parses or disjoint subs-kings of a parsed string."[5] Several examples will illustrate the basic idea of his scheme.

Example 1 $A \rightarrow X|AX$

In the string X X X X X, each X is immediately reduced to A without any need to examine its surrounding symbols.

Example 2 $A \rightarrow XB$

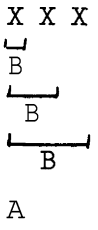
$B \rightarrow Y|ZB$

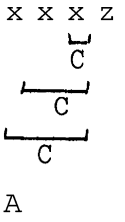


Preceding symbols must be stored until Y is reached but each reduction can be made, e.g., $ZB \rightarrow B$, without examining any symbols not in the reduction itself.

Example 3

A → BY|CZ
B → X|BX
C → X|XC

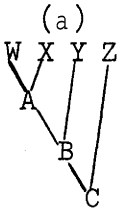


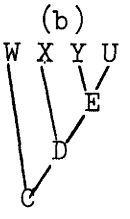


To reduce X it is necessary to look ahead to the end of the string to see whether the reduction should be to a B or to a C .

Example 4 --

A " wx
B → AY
C → BZ|WD
D → XE
E → YU





The substring WX cannot be reduced to an A until we have looked 2 symbols to the right of it. This language is then classified as OSL,2SR . (SL - symbols left; SR - symbols right.)

Generally, Irons classifies a language as $n\{^{S}_{B}\}$, $m\{^{S}_{B}\}R$, where

s = symbol in input string

B = "bracketed" string, meaning a string that has already been reduced

L = left

R = right

n, m are numbers.

This defines "the extent to which symbols surrounding a string determine its parse."[5] Example 1 and 2 are both OSL,OSR (or "unconnected") languages. Example 3 is OSL but it is impossible to fix m since one must always look to the end of the string, whatever its length may be.

Example 5

A → WX
B → AY
C → BZ|UD
D → XE
E → YZ

Here, YZ cannot be reduced in isolation. One must first look two symbols to the left - if a UX is found, YZ can be reduced to E; otherwise it cannot. This language is then 2SL,OSR .

By classifying a language in terms of its parsing difficulty, we gain a clearer understanding of what is needed for its automatic analysis. Some general parsing methods are discussed in the next section.

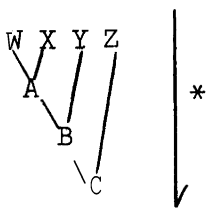
VII-7. Parsing Methods

VII-7.1 A "Top Down" Method[6]

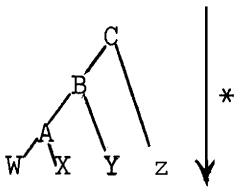
A "bottom up" parse of the string s of the language $\mathcal{L}(V, V_T, \mathcal{P}, S)$ starts with s and looks for a sequence of reductions so that $s \overset{*}{\rightarrow} S$; the parses in the examples of the last few sections have been implicitly

of this type. A "top down" parse starts with S and looks for a sequence of productions such that $S \overset{*}{\rightarrow} s$. The same parsing trees are produced but they appear with the root at the top in the latter case and at bottom in the former. The tree of Example 5(a) of the last section is:

bottom up



top down



Given the syntax $\left\{\begin{array}{l} E ::= T|T+E, \\ T ::= F|F\times T, \\ F ::= \lambda|(E) \end{array}\right\}$ the following ALGOL procedures, in conjunction with some symbol pointer and storage administration which have been intentionally omitted, will perform a "top down" analysis:

```
Boolean procedure E;
   E := if T then  (if issymbol ('+') then E else true) else
   false    (issymbol (arg) is a Boolean procedure which com-
   pares the next symbol in the input string with its argument, arg.)
Boolean procedure T;
   T := if  F then (if issymbol ('×') then T else true) else
   false
Boolean procedure F;
   F := if issymbol  ('λ') then true else
        if  issymbol ('(') then  (if E then
        (if issymbol (')') then true else false)
            else false) else false
```

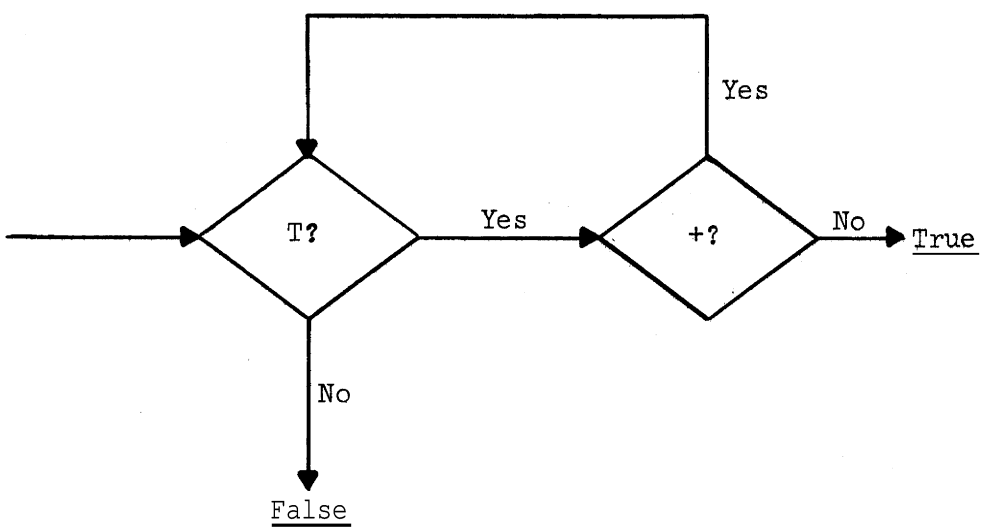
If the last production in the syntax were changed to E ::= T|E+T, a straightforward application of the general method will yield the new procedure for E :

```
Boolean procedure E;
   E := if T then true else if E then . . .
```

For the string, λ + A, the procedure E will call T which calls F which tests for 'λ' and gives the result true; E then is true, but only the first element of the string is in the analysis; i.e., the analysis stops before completion!

λ + λ
␣
F
␣
T
␣
E

If the input string is not a member of the language, T is false and we can easily get into an infinite loop on E . (The problem is that $E \epsilon \mathcal{L}(E)$ - see next section on precedence grammars). The usual solution to the problem is to replace the recursive definition of E by an iterative definition:



A possible extension of BNF that replaces iterative definitions by recursive ones is

$$E ::= T\{+T\} ,$$

where the quantity in the braces can be repeated any number of times, including 0 .

This method has been implemented on several compilers, for example, the B5000 EXTENDED ALGOL compiler. It has the advantage of being conceptually simple. However, it has some severe disadvantages:

(1.) Many false paths can be tried before the correct one is found; a failure on any path requires backtracking to the last successful recognition.

(2.) It is difficult to insert semantic rules, such as code generators, into the system.

(3.) There is no systematic way to determine the success or failure of the method, except by exhaustion.

In general, we can classify the "top down" method as being a heuristic solution to the parsing problem.

VII-7.2 Eickel, Paul, Bauer, and Samelson[7]

This method deals with productions whose right sides are of length 1 or 2; i.e., U ::= R and U ::= ST are the only forms allowed. No generality is lost with this restriction since the production $U ::= S_1S_2 \ldots S_n$ can be replaced by the equivalent set $(U ::= S_1U_1, U_1 ::= S_2U_2, \ldots, U_{n-1} ::= S_n\}$. A stack is used to store symbols and

reduced. substrings; at any point, only the top two elements in the stack need be examined. A table of possible symbol triples is built from the syntax; each element of the table has the form $(S_1S_2S_3)$ n N, with the interpretation:

> If S_1S_2 are the top two elements of the stack and S_3 is the incoming symbol of the input string, then we are in case n and action N is performed.

case		action
n = 1	$U ::= S_1S_2\epsilon$ ℘	Pop stack and replace S_1S_2 by U .
n = 2	$U ::= S_2\epsilon$ ℘	Replace S_2 by U in stack.
n = 3	No production exists.	Push down stack, insert S_3 in stack, and read next input symbol.

This is a systematic mechanical method for parsing strings; the authors claim that the method can handle any unambiguous class 2 language. Semantic rules could be easily included in the parsing algorithm at the points where the triples and action are determined. The method should be able to easily "recover" from syntax errors (an important consideration for programming languages). The main disadvantages are the large storage requirements for the tables and the relatively long time it takes to scan the table of triples for matches.

VII-7.3 Precedence Methods

Floyd[8] has developed a method of syntactic analysis for class 2 languages, which is based on the use of "precedence" relations between pairs of terminal symbols. Productions are restricted so they cannot be of the form:

$$U \rightarrow xU_1U_2y, \quad \text{where } U_1,\ U_2 \epsilon (V - V_T)\ ;$$

the resulting language is called an operator language. The beauty of Floyd's method is that it admits a very simple and efficient parsing algorithm which produces the unique parse.

Wirth and Weber[3] have generalized Floyd's results and shown how efficient compilers for practical non-trivial programming languages may be implemented using precedence methods in conjunction with semantic rules. Wirth and Weber's precedence grammars and their application to compiler writing is discussed in the remainder of the chapter.

VII-8. Precedence Phrase Structure Systems

VII-8.1 Precedence Relations and the Parsing Algorithm

For all $S_iS_j \epsilon V$, it is either possible or impossible for the string s_is_j to appear in a successful parse. When they do appear, there are only 3 ways in which they may be reduced:

1. $\ldots \underbrace{S_i\ S \ldots}$
   reducible substring

S_j is the first or left most symbol of a reducible substring. Using Floyd's notation, this is indicated by $S_i \lessdot S_j$

2. $\ldots \underbrace{S_i\ S \ldots}$

s_is_j is part of a reducible substring.

$$S_i \doteq S_j$$

3. $\underbrace{\ldots S_i}\, S_j \ldots$

S_i is the last or rightmost part of a reducible substring

$$S_i \gtrdot S_j$$

$\lessdot$, $\doteq$ 4 are 3 precedence relations that may exist between ordered pairs of symbols.

Example 1

Input String $S_1\ S_2\ S_3\ S_4\ S_5\ S_6$

Given Relations $\lessdot\quad \lessdot\quad \underbrace{\doteq}\quad \gtrdot\quad \gtrdot$

Since $S_2 \lessdot S_3$, $S_3 \doteq S_4$, $S_4 \gtrdot S_5$, there must exist a symbol $U_1 \epsilon V$ such that

$$U_1 \rightarrow S_3 S_4 \epsilon \mathcal{P} \ .$$

Reduced String $S_1\ S_2\ U_1\ S_5\ S_6$

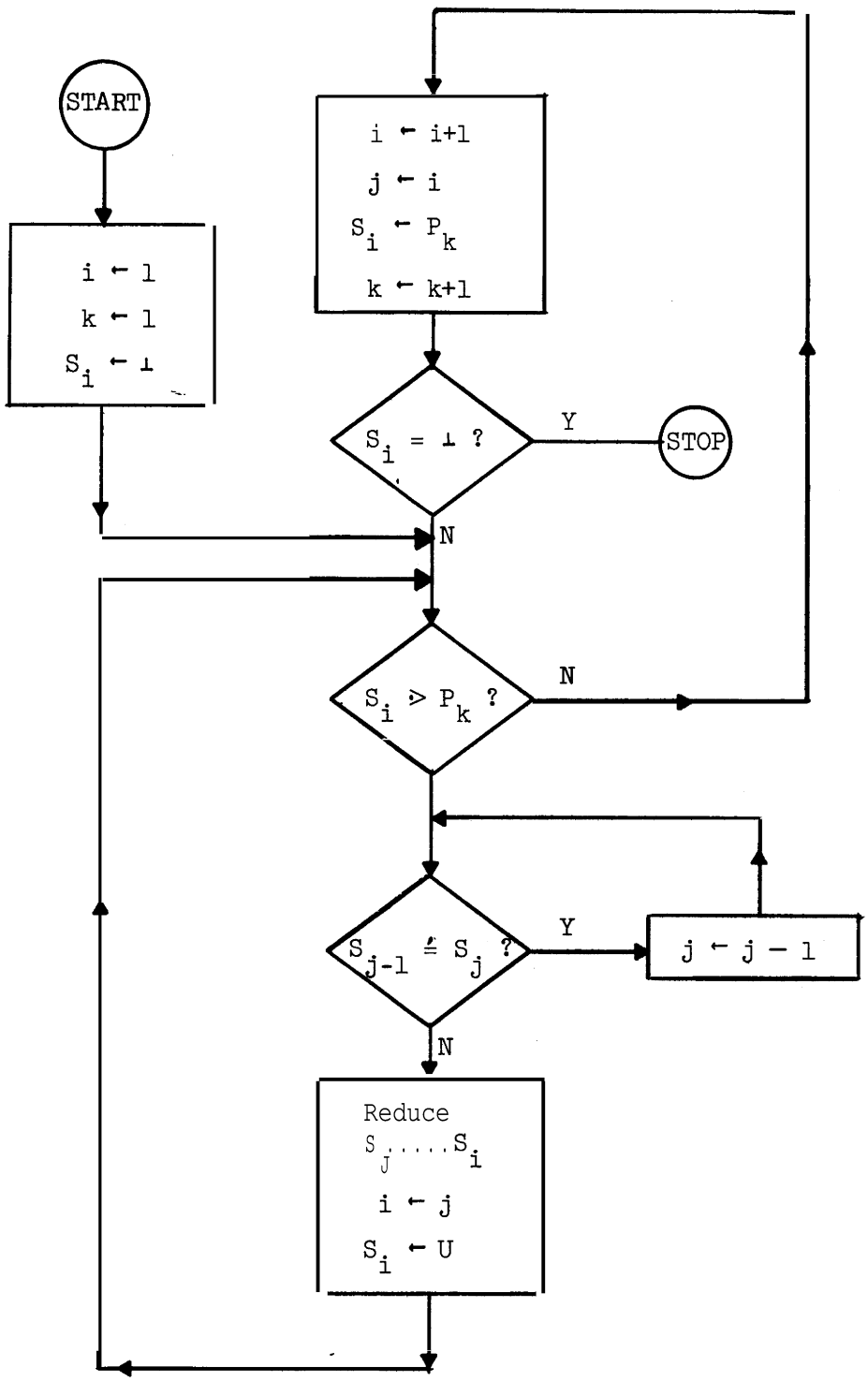
Given Relations $\lessdot\quad \doteq\quad \doteq\quad \gtrdot$

$$\therefore \exists\ U_2 \epsilon V \text{ such that } U_2 \rightarrow S_2 U_1 S_5 \epsilon \mathcal{P} \ .$$

The reduced string then is $S_1 U_2 S_6$.

Let p be any input string, where $p = P_1 P_2 \ldots P_n$; enclose p by the terminating symbol $\perp$ so that $P_0 = P_{n+1} = \perp$; for any symbol

$S \in V$, $\perp \lessdot S$ and $S \gtrdot \perp$. Given one precedence relation between any two symbols that may occur together, p may be parsed using the following algorithm:



S is a stack which contains the partially reduced string at any stage. P is copied into S until the relation $\gtrdot$ is encountered, Then, we retreat backward through S until the beginning of the reducible substring is found. We are then guaranteed (if the string is in the language) that there is a production whose rightside is $S_j, \ldots, S_i$. "Reduce $S_j, \ldots, S_i$" replaces the substring by the left side of that production.

An ALGOL-like program for the algorithm is:

```
i := 0; k := 0;
while P_k ≠ '⊥' do
begin
    while S_i ⋗ P_k do
    begin
        while S_(j-1) = S_j do j := j-1;
        S_j := Leftpart(S_j .... S_i);
        i := j
    end
    i := j := i+1; S_i := P_k;
    k := k+1
end
```

Note that the algorithm involves no backtracking.

Example 2

(entire string) ::= ⊥⟨string⟩⊥

(string) ::= (head)'

(head) ::= ' | (head) λ | ⟨head⟩⟨string⟩

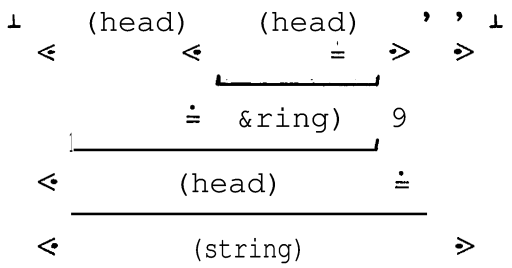
The precedence relations may be described in a precedence matrix M:

	⟨string⟩	⟨head⟩	λ	‘	’
⟨string⟩	⋗	⋗	⋗	⋗	⋗
⟨head⟩	≐	⋖	≐	⋖	≐
λ	⋗	⋗	⋗	⋗	⋗
‘	⋗	⋗	⋗	⋗	⋗
’	⋗	⋗	⋗	⋗	⋗

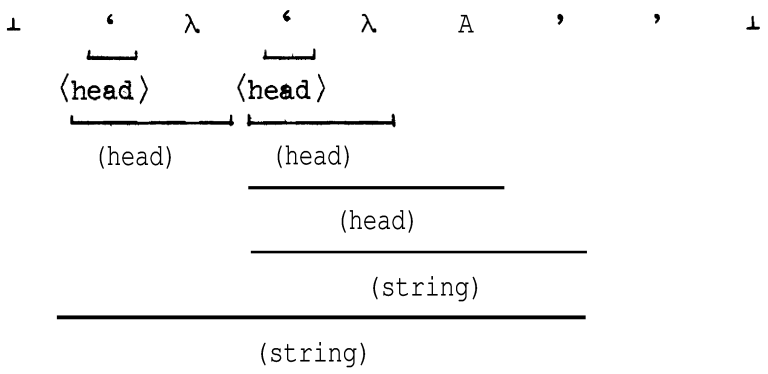
The elements M_{ij} represent the relation between the symbols S_i and S_j ; e.g., (head) ≐ λ

λ ⋗ (head)

(a) Parse using algorithm:



(b) Parse using algorithm:



The parse terminates while the stack contains ⊥⟨string⟩’⊥ instead of ⊥⟨string⟩⊥. This indicates that the string is not a member of the language.

VII-8.2 Finding the Precedence Relations

The precedence relations definitions are first formalized:

1. $S_i \doteq s_j$ if and only if there is a rule $U \rightarrow xS_iS_jy$.
2. $S_i \lessdot s_j$ if and only if there is a rule $U \rightarrow xS_iU_\ell y$ and $U_\ell \overset{*}{\rightarrow} S_jz$.
3. $S_i \gtrdot s_j$ if and only if there is a rule $U \rightarrow xU_kS_jy$ and $U_k \overset{*}{\rightarrow} zs_i$ or $U \rightarrow xU_kU_\ell y$ and $U_k \overset{*}{\rightarrow} zS_i$ $U_\ell \overset{*}{\rightarrow} S_jw$.

The strings w, x, y, z may be empty in the above definitions.

Example 3

$A \rightarrow BC$
$B \rightarrow WX$
$c \rightarrow YZ$

From definition 1. : $B \doteq C$, $W \doteq X$, $Y \doteq Z$

From definition 2. : $B \lessdot Y$

From definition 3. : $X \gtrdot C$, $X \gtrdot Y$

The leftmost symbols of a non-terminal symbol U are defined

$$\mathfrak{L}(U) = \{S \mid \exists z(U \overset{*}{\rightarrow} Sz)\} .$$

The rightmost symbols of a non-terminal symbol U are

$$\mathfrak{R}(U) = \{S \mid \exists z(U \overset{*}{\rightarrow} zS)\} .$$

The precedence relations can now be alternately defined:

$$1.\ S_i \doteq S_j \leftrightarrow \exists p(p: U \rightarrow xS_iS_jy)$$
$$p \in \mathcal{P}$$

$$2.\ S_i \lessdot S_j \leftrightarrow \exists p(p: U \rightarrow xS_iU_\ell y)$$
$$\wedge\ S_j \in \mathcal{L}(U_\ell)$$

$$3.\ S_i \gtrdot S_j \leftrightarrow \exists p(p: U \rightarrow xU_kS_jy) \wedge S_i \in \mathcal{R}(U_k)$$
$$\vee\ \exists p(p: u \rightarrow xU_kU_\ell y) \wedge S_i \in \mathcal{R}(U_k)$$
$$\wedge\ S_j \in \mathcal{L}(U_\ell)\ .$$

The use of these definitions directly leads to an efficient mechanical algorithm for finding the relations. The sets $\mathcal{L}$ and $\mathcal{R}$ may be found by using their recursive definition:

$$\mathcal{L}(U) = \{S \mid \exists z(U \rightarrow Sz) \vee$$
$$\exists z, U'(U \rightarrow U'z \wedge S \in \mathcal{L}(U'))\}$$
$$\mathcal{R}(U) = \{S \mid \exists z(U \rightarrow zS) \vee$$
$$\exists z, U'(U \rightarrow zU' \wedge S \in \mathcal{R}(U'))\}$$

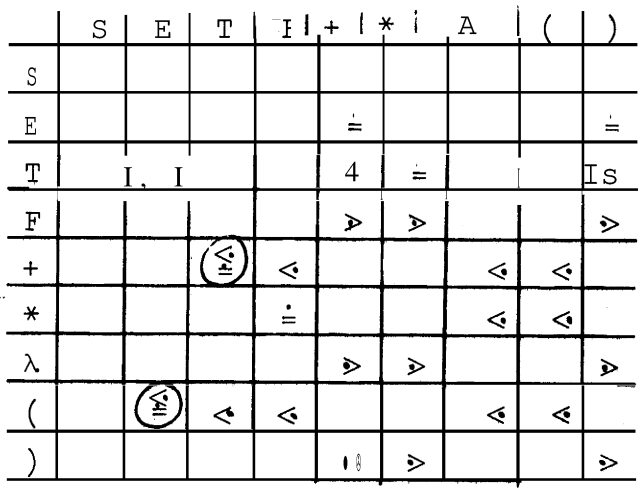
These are easier to work with than the original definitions; however, some complex administration is needed to ensure that the program does not fall into an infinite recursion, for example, in the case where $A \rightarrow B$, $B \rightarrow A$ are in $\mathcal{P}$.

Example 4

$$S \rightarrow E$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow \lambda \mid (E)$$

U	$\mathfrak{L}(U)$	$\mathfrak{R}(U)$
S	E, T, F, λ, (	E, T, F, λ,)
E	E, T, F, λ, (	T, F, λ,)
T	T, F, λ, (	F, λ,)
F	λ, (	λ,)

Precedence Matrix



Note that there are 2 relations for the ordered pair (+, T) and for the pair ((, E) . i.e.,

$$+ \doteq T \quad \text{and} \quad + \lessdot T$$

$$(\doteq E \quad \text{and} \quad (\lessdot E \ .$$

A syntax is a simple precedence grammar (or simple precedence syntax) if and only if at most one of the relations $\doteq$, $\lessdot$, and $\gtrdot$ holds between any ordered pair of symbols. Thus, example 4 is not a precedence grammar; it can be made into a precedence grammar by modifying the syntax as follows:

$$S \rightarrow E,\ E \rightarrow E',\ E' \rightarrow E' + T \mid T,\ T \rightarrow T',$$

$$T' \rightarrow T' * F \mid F,\ F \rightarrow \lambda \mid (E)$$

If none of the relations holds between a given ordered symbol pair, then the appearance of this ordered pair during a parse indicates a syntax error, i.e., the input string is not a member of the language.

For a practical language, the number n of symbols in the vocabulary is very large (ALGOL has n ~ 220, ~ 110 symbols in V_T and ~ 110 symbols in $V - V_T$). A precedence matrix then has n^2 elements. To compact the precedence information, Floyd' introduced "precedence functions".

VII-8.3 Use of Precedence Functions

We try to find two functions, f and g, such that for any ordered symbol pair (S_i, S_j):

$$f(S_i) = g(S_j) \leftrightarrow S_i \doteq S_j$$
$$f(S_i) < g(S_j) \leftrightarrow S_i \lessdot S_j$$
$$f(S_i) > g(S_j) \leftrightarrow S_i \gtrdot S_j$$

At least 2 functions are required since 2 symbols S_i and S_j may be related $S_i R_1 S_j$ and $S_j R_2 S_i$, where $R_1, R_2 \in \{\Lambda, \doteq, \lessdot, \gtrdot\}$ and $R_1 \neq R_2$ (see Example 1). If f and g exist, then only 2n elements are necessary to store the precedence relations and the relations can be found much faster.

Example 5

$$E \rightarrow E'$$
$$E' \rightarrow T|E' + T|E' - T$$
$$T \rightarrow T'$$
$$T' \rightarrow F|T' \times F|T' / F$$
$$F \rightarrow F'$$
$$F' \rightarrow P|F' * P$$
$$P \rightarrow \lambda|(E)$$

V_T = CA, (,), *, x, /, +, -3

$V - V_T$ = { E, E', T, T', F, F', P}

U	c(u)	R(U)
E	E' T T' F F' P A (	E' T T' F F' P A)
E'	E' T T' F F' P A (	T T' F F' P A)
T	T' F F' P A (	T' F F' P A)
T'	T' F F' P A (	F F' P A)
F	F' P λ (	F' P λ)
F'	F' P λ (	P λ)
P	λ (	λ)

Precedence Matrix

f \ g	8 (A	8 P*	7	6 F′	6 F	5 /	4 X	4 T′T	4	3 -	2 +	2 E′	1	1)E
8)				⋗			3	4			3 ⋗		⋗	
8 λ				⋗			⋗	⋗			⋗ ⋗		⋗	
7 P				⋗			3	3			⋗ ⋗		⋗	
7 *	⋖	⋖	≐											
6 F′			≐				⋗	⋗			⋗ ⋗		⋗	
6 F							⋗	⋗			⋗ ⋗		⋗	
5 /		⋖	⋖	⋖	≐									
5 ×		⋖	⋖	⋖	≐									
4 T′							≐	≐			⋗ ⋗		⋗	
4 T											⋗ ⋗		⋗	
3 -	⋖	⋖	⋖	⋖	⋖				⋖	≐				
3 +	⋖	⋖	⋖	⋖	⋖				⋖	≐				
2 E′											≐ ≐		⋗	
1 E												≐		
1 (	⋖	⋖	⋖	⋖	⋖				⋖	⋖		⋖		≐

The functions exist when we can permute the rows and columns of the precedence matrix so that it is divided into 3 areas, only one relation holding per area. This has been done in the above example. If the matrix division is of the form  , f and g function values can be assigned starting from the bottom left corner of the array. An algorithm for determining f and g, if they exist, is published in the Algorithm section of the Communications of the ACM.[9]

Unfortunately, f and g do not always exist; however, it is often possible to make minor changes to the syntax that allow f and g to be found.

Example 6

$$T \rightarrow PB\}$$
$$P \rightarrow \{$$
$$B \rightarrow xx|xT|Tx|TT$$

This grammar generates list structures,

e.g., {{xx}x}

U	$\mathfrak{L}(U)$	$\mathfrak{R}(U)$
T	P{	}
P	{	{
B	xT{P	xT}

Precedence Matrix

	{	P	x	T	}	B
{	>	>	>	>	*	>
}	>	>	>	>	>	
x	<	<	≐	≐	>	
T	<	<	≐	≐	>	
P	<	<	<	<		≐
B				≐		

The entry designated by * is empty; therefore, the f and g functions may be found. However, if we add T → {} to the production in order to allow the empty list, the relation {≐} holds and * becomes ≐ . f and g then do not exist even though the syntax is still a precedence syntax. If we change the first rule to T → PB}|P}, the empty string is allowed but f and g can be found.

A comparison of these results with Dijkstra's priority methods discussed in the last chapter leads to the following important connection:

> The f and g functions for precedence grammars are exactly equivalent to the stack and compare priorities used for the transformation of expressions into reverse Polish form.

The contribution here is a formalization and extension of the early priority ideas so that the following compilation problems can be handled by general algorithms:

1. determining whether a given syntax is a precedence syntax
2. finding the precedence relations
3. computing the f and g functions, if they exist
4. parsing strings in a precedence language in an efficient manner.

An open problem is how to transform a syntax so that it is a precedence grammar. As shown after example 4, it is presently necessary to add some "artificial" productions to a grammar to make it a precedence grammar.

VII-8.4 Ambiguities

An unambiguous syntax is a phrase structure syntax (the ordered quadruple $\mathcal{G} = (V, V_T, \mathcal{P}, S)$) with the property that for every string $x \in \mathcal{L}(\mathcal{G})$ there exists only one canonical parse.

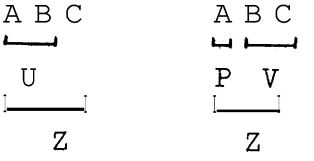
Example 7

Z → UC|PV

U → A13

V → BC

P ' A



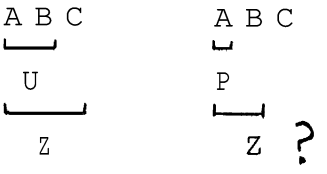
ABC has 2 canonical parses and is therefore ambiguous. A local ambiguity occurs where a substring may have more than one canonical parse:

Example 8

Z → UC|PB

U → AB

P ' A



i.e., local ambiguities lead to backtracking.

Theorem: The parsing algorithm described in this section yields the canonical form of the parse for any sentence of a precedence phrase structure language if there exist no two syntactic rules with the same right part. Furthermore, this canonical parse is unique.

Proof:

This theorem is proven, if it can be shown that in any sentence its directly reducible parts are disjoint. Then the algorithm, proceeding strictly from left to right, produces the canonical parse, which is

unique, because no reducible substring can apply to more than one syntactical rule.

The proof that all directly reducible substrings are disjoint is achieved indirectly: Suppose that the string $S_1 \ldots S_n$ contain two directly reducible substrings $S_i \ldots S_k$ (a.) and $S_j \ldots S_\ell$ (b.), where $1 \leq i \leq j \leq k \leq \ell \leq n$. Then because of a. it follows from the definition of the precedence relations that $S_{j-1} \doteq S_j$ and $S_k \gtrdot S_{k+1}$, and because of b. $S_{j-1} \lessdot S_j$ and $S_k \doteq S_{k+1}$. Therefore this sentence cannot belong to a precedence grammar.

Since in particular the left most reducible substring is unique, the syntactic rule--to be applied is unique. Because the new sentence again belongs to the precedence language, the next reduction is unique again. It can be shown by induction, that therefore the entire parse must be unique.

By associating semantic rules with the syntactical rules of a precedence phrase structure language, the meaning is also unambiguous.

VII-9. Association of Semantics with Syntax

VII-9.1 Mechanism for Expressing Semantics

An environment $\mathcal{E}$ is a set of variables whose values define the meaning of a sentence. $\mathcal{T}$ is a set of interpretation rules each of which define an action (or a sequence of actions) involving the variables in $\mathcal{E}$. A phrase structure programming language $\mathcal{L}_p(V, V_T, \mathcal{P}, S, \mathcal{T}, \mathcal{E})$ is a phrase structure language $\mathcal{L}(V, V_T, \mathcal{P}, S)$ where $\mathcal{T}$ is a set of interpretation rules in one-to-one correspondence with the elements of $\mathcal{P}$ and $\mathcal{E}$ is an environment for the elements of $\mathcal{T}$. The meaning m of sentence

$x \epsilon \mathcal{L}_P$ is the effect of the execution of the sequence of interpretation rules $t_1, t_2, \ldots, t_n$ on the environment $\mathcal{E}$, where $P_1, P_2, \ldots, P_n$ is a parse of x into $\mathbf{S}$ and t_i corresponds to p_i for all i .

The fact that the precedence grammar parsing algorithm never backtracks allows us to attach semantic rules to each syntactical unit or reduction. It will therefore be assumed that we are dealing with precedence grammars. Corresponding to the symbol stack S used in the algorithm, we maintain a value stack V . At the same time the syntactical reduction $U \rightarrow S_j \ldots S_i$ is made, a similar semantic "reduction" or rule is obeyed for the elements $V_j \ldots V_i$ in the value stack.

Example 1

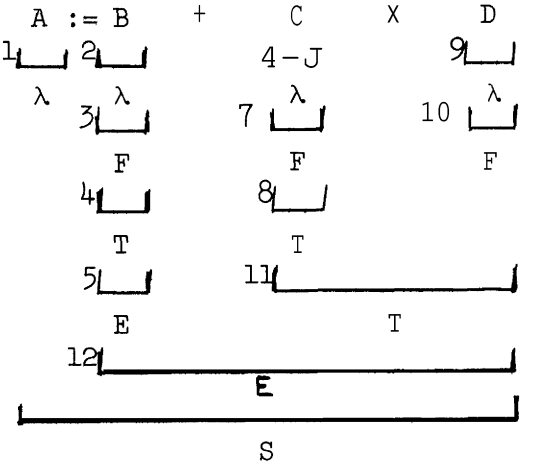
Syntactic Rules	Semantic Rules
$S \rightarrow \lambda := E$	$V_{V_j} \leftarrow V_i$
$E \rightarrow T \mid E + T$	$\Lambda \mid V_j \leftarrow V_j + V_1$
$T \rightarrow F \mid T \times F$	$\Lambda \mid V_j \leftarrow V_j \times V_i$
$F \rightarrow \lambda \mid (E)$	$V_j \leftarrow V_{V_j} \mid V_j \leftarrow V_{j+1}$

V_i represents the value associated with the stack symbol S_i . The semantic rule $V_j \leftarrow V_j + V_1$ corresponding to $E \rightarrow E + T$ can also be written "value(E) $\leftarrow$ value(E) + value(T)". The first way makes explicit reference to the parsing algorithm block "reduce $S_j \ldots S_i$". In the rule $V_j \leftarrow V_{V_j}, V_J$ originally holds the address of the particular variable used. λ is a representative for all possible variable identifiers.

Example 1 gives semantic rules for an interpreter. The next example shows how semantic rules for a compiler for a stack machine may be associated with the same syntax as above.

Example 2

Syntax	Semantics
X → λ := E	store λ
E → T	Λ
E → E + T	add
T → F	Λ
T ' T X F	multiply
F → λ	load λ
F → (E)	Λ



The numbers indicate the order of the reductions.

Obeying the semantic rules, the statement compiles into:

	Reduction Step
load B	3
load C	7
load D	10
multiply	11
add	12
store A	13

In these examples, it has been assumed that the specific variable names and values are available. (V_{v_j} of the interpreter, λ in the compiler.) We now show how this may be accomplished,

VII-9.2 Handling of Declarations

A common way of putting declaration lists (DL) and statement lists (SL) into the syntax is illustrated by the following simple example:

Example 3

P → begin DL; SL end
DL → D|DL, D
SL → S|SL, s

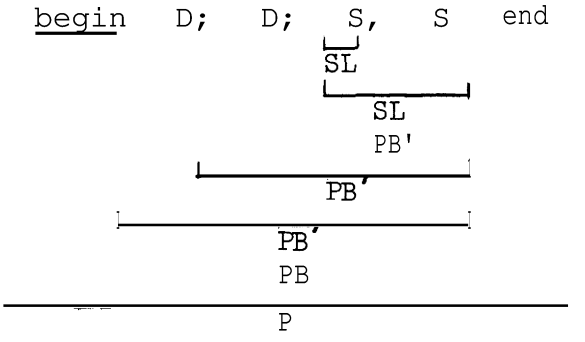
begin D, D, D; S end

The difficulty here is that when the parse reaches the statement S, the stack contains "begin DL;". What is needed is to retain the declarations D in the stack so the semantic rules for S may refer to them for addresses or values of specific variables.

Example 4

P → begin end
PB' → D; PB'|SL
SL → S|SL, s
PB → PB'

(PB' must be included to make the syntax a precedence grammar.)



Syntax and semantics for declarations D and variables V are:

Syntax	Semantics
D → real I	$V_j \leftarrow \{\Omega, I\}$
V → I	Search Stack for I

Ω (undefined) is the initial value of I .

After reducing to D, the value stack contains a value (Ω) and a name; when the statements S are reduced, they may refer to the values and names in the stack. In a compiler, the declarations would produce "reserve storage" instructions.

VII-9.3 Conditional Statements and Expressions

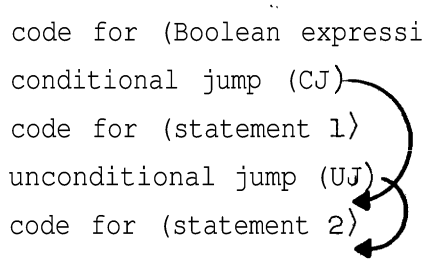
For a first try, the syntax for a condition statement is defined in an obvious way:

(conditional statement) ::= if (Boolean expression) then
(statement 1) else
(statement 2)

The reduction to (conditional statement) occurs when the symbol stack of the parse contains:

if
(Boolean expression)
then
(statement 1)
else
(statement 2) ← top of stack

If code is being generated by the semantic rules, it is then too late to compile "jumps" around the statements. The semantic rules should compile:



The ALGOL definition of conditional statement is:

⟨conditional statement⟩ ::= ⟨if clause⟩⟨statement 1⟩
else ⟨statement 2⟩

⟨if clause⟩ ::= if ⟨Boolean expression⟩ then

Here, we may attach the semantic rule for ⟨if clause⟩:

Generate CJ Ω
Set V(⟨if clause⟩) = Pointer to Generated code CJ Ω .

This will take care of the first part of the conditional statement. Unfortunately, the else is not reduced in time for a UJ; ⟨statement 2⟩ has been reduced and its semantics obeyed before the entire ⟨conditional statement⟩ with the else is recognized.

To allow the syntax to correspond with the desired semantics and vice versa, the conditional statement is further divided:

(conditional statement) ::= (if clause)⟨true part⟩
(statement 2)
(if clause) ::= if (Boolean expression) then
⟨true part⟩ ::= (statement 1⟩ else

The desired meaning can then be attached; for example,

V(⟨if clause⟩) = Pointer to Generated code CJ Ω
V(⟨true part⟩) = Pointer to Generated code UJ Ω
V(⟨conditional statement⟩) = Insert Jump addresses
in CJ and UJ commands

Conditional expressions can be treated in a similar manner.

VII-9.4 GO TO and Labelled Statements

It is difficult to give a clean set of interpretation rules for the GO TO statement, GO TO (label) for an interpreter since the (label) might not have a value at that point. However, a compiler can use:

Semantics (GO TO (label)) = Search Symbol Table for (label)
and emit UJ instruction

"Chaining" (see Chapter II-4 on One-Pass Assembly) or indirect addressing can be used to solve the forward reference problem.

The ALGOL definition of (basic statement) is:

(basic statement) ::= (label) : (basic statement)

A problem similar to that in conditional statements exists here; (basic statement) must be recognized and compiled before the label definition "(label) :" is detected. The syntax is therefore changed to:

(basic statement) ::= (label definition)⟨basic statement⟩
(label definition) ::= (label) :

The location counter can then be assigned to the (label) before the (basic statement) following it is compiled:

Semantics ((label definition)) = Enter (label) together with the location counter into the Symbol Table.

VII-9.5 Constants

Conversion of catenated symbols representing constants to their numerical values can be handled by rules of the following type:

Syntax	Semantics
(integer) ::= ⟨digit⟩	Λ
⟨integer⟩⟨digit	$V_j \leftarrow 10 \times V_j + V_i$
(digit) ::= 0	$V_j \leftarrow 0$
1	$V_j \leftarrow 1$
⋮	⋮
9	$V_j \leftarrow 9$

The important point to note in the preceding examples is that it is both desirable and feasible to explicitly exhibit the natural relationship that exists between the structure and the meaning of a programming language. An unambiguous syntax then guarantees that every sentence (program) in the language has one and only one well-defined meaning. Precedence grammars offer a powerful framework in which to design, experiment with, and implement programming languages. The reader should consult reference 3 for an example of a language more general than ALGOL that has been implemented using these methods.

VII-10. References

1. Taylor, W., Turner, L., Waychoff, R., A syntactical Chart of ALGOL 60. Comm. ACM 14, 9 (Sept. 1961) 393.

2. Anderson, C., An Introduction to ALGOL 60.

3. Wirth, N., Weber H., EULER - A Generalization of ALGOL, and its Formal Definition: Part I, Part II. Comm. ACM, Vol. 9, pp. 13-25, 89-99, (Jan/Feb. 1966).

4. Chomsky, N,, Schutzenberger, M. P., The Algebraic Theory of Context- - Free Languates. Computer Programming and Formal Systems, North-Holland, Amsterdam, 1963.

5. Irons, E. T., Structural Connections in Formal Languages. Comm. ACM, Vol. 7, pp. 67-71 (Feb. 1964).

6. Leavenworth, B. M., FORTRAN IV as a Syntax Language. Comm. ACM, Vol. 7, pp. 72-80 (Feb. 1964).

7. Eichel, J., Paul, M., Bauer, F. L., Samelson, K., A Syntax-controlled Generator of Formal Language Processors. Comm. ACM, Vol. 6, pp. 451-455 (Aug. 1963).

8. Floyd, R. W., Syntactic Analysis and Operator Precedence. J. ACM, vol. 10, pp. 316-333 (July, 1963).

9. Wirth, N., Find Precedence Functions. Algorithm 265. Comma ACM, 8, 10 (Oct. 1965) 604-605.

Additional References

1. Brooker, R. A., Morris, D., A General Translation Program for Phrase Structure Languages. J. ACM, Vol. 9, pp. 1-10 (Jan. 1962).

2. Knuth, D. E., On the Translation of Languages from Left to Right. Information and Control, (1965).

3. Irons, E. T., The Structure and Use of the Syntax-Directed Compiler. Annual Review of Automatic Programming. Vol. 3, pp. 207-227 (1963).

4. Floyd, R. W., The Syntax of Programming Languages - A Survey. IEEE Trans on EC, Vol. EC13, pp. 246-353 (August, 1964).

VII-11. Problems

CS 236a
Problem Set III

Feb. 24, 1966
N. Wirth

1. Given is the following set $\mathcal{P}_1$ of productions:

$$S \rightarrow A$$
$$A \rightarrow B \mid BCB$$
$$B \rightarrow D \mid E$$

Which are the sets of terminal and nonterminal symbols?

Which is the language $\mathcal{L}_1(V_1, V_1^T, \mathcal{P}_1, S)$?

2. Add to the set $\mathcal{P}_1$ the production

$$B \rightarrow FBG$$

obtaining $\mathcal{P}_2$. Which are the symbol sets V_2^N and V_2^T, and which is the language $\mathcal{L}_2(V_2, V_2^T, \mathcal{P}_2^T, S)$?

Use the notation X^n for the n-fold concatenation of the symbol X, and indicate which values n may assume.

3. Instead of $B \rightarrow FBG$, add the production

$$B \rightarrow FAG$$

to $\mathcal{P}_1$, thus obtaining $\mathcal{P}_3$. (and V_3).

Is the string

F F E G G C F D G

a sentence of $\mathcal{L}_3(V_3, V_3^T, \mathcal{P}_3, S)$?

Is it also a sentence of $\mathcal{L}_2$?

Does $\mathcal{L}_3$ differ from $\mathcal{L}_2$?

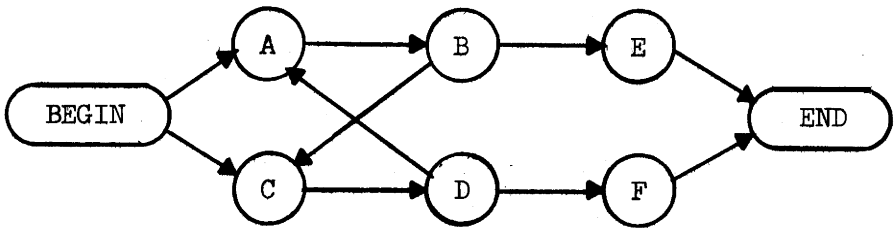
If so, construct a string which belongs to one language (indicate which) but not to the other; if not, show that they are equal.

4. Find a grammar which defines a language $\mathcal{L}$ such that a string consisting of any even number of B's with any number of A's between two consecutive B's, is a sentence of the language.

5. Find a grammar defining a language whose sentences have the form

$$X^n Y^n Z^n \qquad (n = 1,2,\ldots)$$

6.



Starting out at "BEGIN" you choose a path according to the arrows in the above notework, each time appending the encountered letter to a letter string, until you reach the "END" point.

Define the set of all strings you can construct in this way, and call them a language. Which is this language? Use the same notation as in problem 2.

7. Construct a set of productions which generate the language of problem 6. You should not need to introduce more than 6 or 7 nonterminal symbols.

8, Consider an arithmetic expression to be defined by the following syntax:

⟨expression⟩ → ⟨term⟩|⟨expression⟩ + ⟨term⟩|⟨expression⟩ - ⟨term⟩
⟨term⟩ → ⟨factor⟩|⟨term⟩ × ⟨factor⟩|⟨term⟩ / ⟨factor⟩
⟨factor⟩ → ⟨primary⟩|⟨factor⟩ * ⟨primary⟩
⟨primary⟩ → ⟨letter⟩|(⟨expression⟩)
⟨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

Which are the values of the priority functions used in the "railway shunting yard" algorithm for producing a polish postfix notation of such an expression.

Symbol	Stack priority	Compare priority
(		
+		
-		
x		
/		
*		
)		

9. Write in B5500 ALGOL and test on the computer a program performing the following tasks:

a. Read from a card an expression as defined in problem 8. (The correctness of the input need not be checked).

b. Print this expression.

c. Use the "railway shunting yard" algorithm to produce polish postfix notation of the read expression, and print it on one line.

d. From the result of step c., compile a sequence of "machine instructions" representing the read expression, and print it (one instruction per line). The underlying machine is supposed to be a multi-register computer, where all its 9 registers are alike. The form of those printed machine instructions shall be

(result) ← ⟨operand⟩⟨operator⟩⟨operand⟩

where

(operator) ::= +|-|x|/|*
(result) ::= (register)
(operand) ::= ⟨letter⟩|⟨register⟩
(register) ::= 1|2|3|4|5|6|7|8|9

The sequence shall be such that the result is left in register 1. Does your compiled code represent the minimum number of instructions necessary to evaluate the expression, and is your code such that the minimum number of registers is used necessary to evaluate the expressions on this machine? Use the following expression as test cases:

A×B+C

A+B×C-D

A×B+C-D

A×(B+C)/D*E

A+B-C×D/F*F

((A×B+C)×D+E)×F+G

A+ (B+ (C+(D+ (E+F))))

10. Is the syntax of problem 2a (simple) precedence syntax? The one of problem 3? For both syntaxes, construct the sets of leftmost and rightmost symbols and the matrix of precedence relations.

11. Replace in the syntax of problem 2 the symbol G by the symbol F . Is the resulting syntax (and language) unambiguous? Explain. Make the same replacement in the syntax of problem 3, and given the answer to the same question.

Problem Set III: Solution to Problem 9

```
BEGIN COMMENT CS 236, N.WIRTH. EXAMPLE OF AN EXPRESSION COMPILER;
    INTEGER I,J,K,U,V,X,R;
    INTEGER ARRAY A,B[0:31], S[0:9], F,G[0:63];
    FORMAT FO (32A1);
    LABEL L1,L2;
    F["+"] ← 3; F["-"] ← 3; F["x"] ← 5 ; F["/"] ← 5; F["*"] ← 7;
    G["+"] ← 2; G["-"] ← 2; G["x"] ← 4 ; G["/"] ← 41 G["*"] ← 6;
    F["("] ← 1; G["("] ← 81 G[")"] ← 1 ; G[" "] ← 1;
L1: READ (FO, FOR I ← 0 STEP 1 UNTIL 31 DO A[I]) [L2];
    WRITE (FO, FOR I ← 0 STEP 1 UNTIL 31 DO A[I]);
COMMENT PART 1:B← POLISH POSTFIX (A) ;
    I ← J ← K ← X ← S[0] ← 0;
    WHILE S[K] ≠ " " DO
    BEGIN X ← A[I]; I ← I+1;
        IF G[X] = 0 THEN
        BEGIN B[J] ← X; J ← J+1
        END ELSE
        BEGIN WHILE F[S[K]]>G[X] DO
              BEGIN B[J]← S[K]; J ← J+1; K ← K-1
              END ;
              IF F[S[K]] =G[X] THEN K ← K-1 ELSE
              BEGIN K ← K+1; S[K]← X
              END
        END
    END ;
    B[J] ← " ";
    WRITE (FO, FOR I ← 0 STEP 1 UNTIL J DO B[I]);
COMMENT PART2: GENERATE MACHINE INSTRUCTION SEQUENCE)
    J ← K ← R ← 0;
    WHILE B[J] ≠ " " DO
    BEGIN X← B[J]; J ← J+1;
        IF G[X] = 0 THEN
        BEGIN K ← K+1; S[K]← X
        END ELSE
        BEGIN U t S[K]; IF U < 10 THEN R ← R-1; K← K-1;
              V← S[K]; IF V < 10 THEN R ← R-1; S[K]← R ← R+1;
              WHITE (<X3,A1," ←",3(X1,A1)>, R,V,X,U)
        END
    END ;
    WRITE ([DBL]); GO TO L1;
L2:
END .
```

```
A×B+C
AB×C+
    1 ← A × B
    1 ← 1 + C

A+B×C-D
ABC×+D-
    1 ← B × C
    1 ← A + 1
    1 ← 1 - D

A×B-C×D
AB×CD×-
    1 ← A × B
    2 ← C × D
    1 ← 1 - 2

A×(B+C)/D*E
ABC+×DE*/
    1 ← B + C
    1 ← A × 1
    2 ← D * E
    1 ← 1 / 2

A+B-C×D/E*F
AB+CD×EF*/-
    1 ← A + B
    2 ← C × D
    3 ← E * F
    2 ← 2 / 3
    1 ← 1 - 2

((A×B+C)×D+E)×F+G
AB×C+D×E+F×G+
    1 ← A × B
    1 ← 1 + C
    1 ← 1 × D
    1 ← 1 + E
    1 ← 1 × F
    1 ← 1 + G

A+(B+(C+(D+(E+F))))
ABCDEF+++++
    1 ← E + F
    1 ← D + 1
    1 ← C + 1
    1 ← B + 1
    1 ← A + 1
```

Problem Set A
N. Wirth

C.S. 236b
April, 1965

1. Devise grammars G(V,T,P,S) which generate strings according to the following specifications:
   a. $a^{(n)}b^{(n)}$, where $x^{(n)}$ signifies a string of n x's for arbitrary n.
   b. strings consisting of a's and b's, such that there is always an even number of b's in the string, and there are either 0 or more than 1 a's between any two b's.
   c. $a^{(n)}b^{(n)}c^{(n)}$

2. Devise a set of ALGOL procedures which analyse strings generated by the following grammar:

```
S ::= A
A ::= AaB
A ::= B
B ::= BbC
B ::= C
C ::= cAc
C ::= d
```

   Assume the presence of a Boolean procedure "issymbol('x')", which tests the next symbol of the input string. Choose the names of the procedures in correspondence with the nonterminal symbols of the vocabulary.

3. Consider the grammars given below. Determine whether they are precedence grammars. If so, indicate the precedence relations between symbols and find the precedence functions f and g; if not, indicate the symbol pairs which do not have a unique procedence relationship. Also, list the sets of leftmost and rightmost symbols L and R. .

   a.
```
S::=E
E ::=F
E ::= FcF
F ::=x
F ::= GEz
F ::= Gz
G ::= GE,
G ::= a
```

   b.
```
S ::= A
A ::=B
A ::= x,A
B ::= B,y
B ::=Y
```

Problem Set A, Solutions

c. s. 236b
May, 1965
N. Wirth

1a. $S \rightarrow A$
$A \rightarrow aAb \mid \Lambda$

1b. $S \rightarrow 3 \; U \mid aU \mid Ua \mid aUa$
$U \rightarrow V \mid w$
$V \rightarrow W \mid bWb$
$W \rightarrow A \mid \Lambda$
$A \rightarrow Aa \mid aa$

1c. $S \rightarrow A$
$A \rightarrow abBAc \mid C$
$bBa \rightarrow abB$
$bBC \rightarrow Cb$
$aC \rightarrow a$

2.
```
Boolean procedure S; S := A;
Boolean procedure A;
   A := if B then
        (if is symbol ('a') then A else true) else false;
Boolean procedure B;
   B := if C then
        (if is symbol ('b') then B else true) else false;
Boolean procedure C;
   C := if is symbol ('c') then
        (if A then is symbol ('c') else false) else is symbol ('d');
```

3a.

	$\mathcal{L}$	$\mathcal{R}$
S	EF Gx a	E F x z
E	F Gx a	F x z
F	G x a	x z
G	G a	, a

	S	E	F	G	c	x	z	a,
S								
E						=		=
F					=	>		>
G		=	<	<		<	=	<
c			=	<		<		<
x					>		>	>
z					>		>	>
a		>	>	>		>	>	>
,		>	>	>		>	>	>

	S	E	F	G	c	x	z	a,
f	1	1	2	1	2	3	3	4	4
g	1	1	2	3	2	3	1	3	1

G3a is a precedence grammar.

3b.

	$\mathcal{L}$	$\mathcal{R}$
S	ABxy	ABy
A	Bxy	ABy
B	By	Y

G3b is not a precedence grammar, since

, $\lessdot$ Y and , $\doteq$ y

VIII. ALGOL COMPILATION

VIII-1. The Problems of Analysis and Synthesis

The tasks of a compiler can be divided into two distinct phases--the analysis of the source program and the synthesis of an equivalent object language program. It was argued in the last chapter that these phases may occur in parallel by obeying semantic rules as the input is reduced to its syntactic components. Some production compilers for ALGOL generate object code as the source program structure is analyzed' but most perform several analysis passes first; the analysis passes check for errors and transform the input into a more convenient form. For example, Naur's GEIR ALGOL compiler[2] consists of 9 passes--the first 6 analyze the input and the last 3 synthesize object code:

GEIR ALGOL COMPILER

Pass	Task
1.	Check and convert input to ALGOL symbols.
2.	Associate each identifier with an integer.
3.	Analyze and Check delimiter structure.
4.	Collect declarations and specifications. Rearrange procedure calls.
5.	Allocate storage for variables.
6.	Check operand types. Convert strings to reverse Polish notation.
7.	Generate machine instructions for expressions.
8.	Final program addressing and drum allocation.
9.	Rearrange drum segments.

Much of the complexity (and challenge) in ALGOL compilers lies in the allocation of storage for handling block structure, especially recursive procedures and dynamic arrays. The latter two features make it impossible to compile pure machine language addresses for variables. Instead, what is needed is the generation of calls to run time administration routines (RTA) that allocate storage and compute addresses. Assuming a stack computer, this chapter discusses one particular RTA scheme that correctly reflects the dynamic behavior in an ALGOL program; most of the material may be found in Reference 1. The reader should review the discussion of assembler block structure in Chapter II.

VIII-2. Run Time Storage Administration

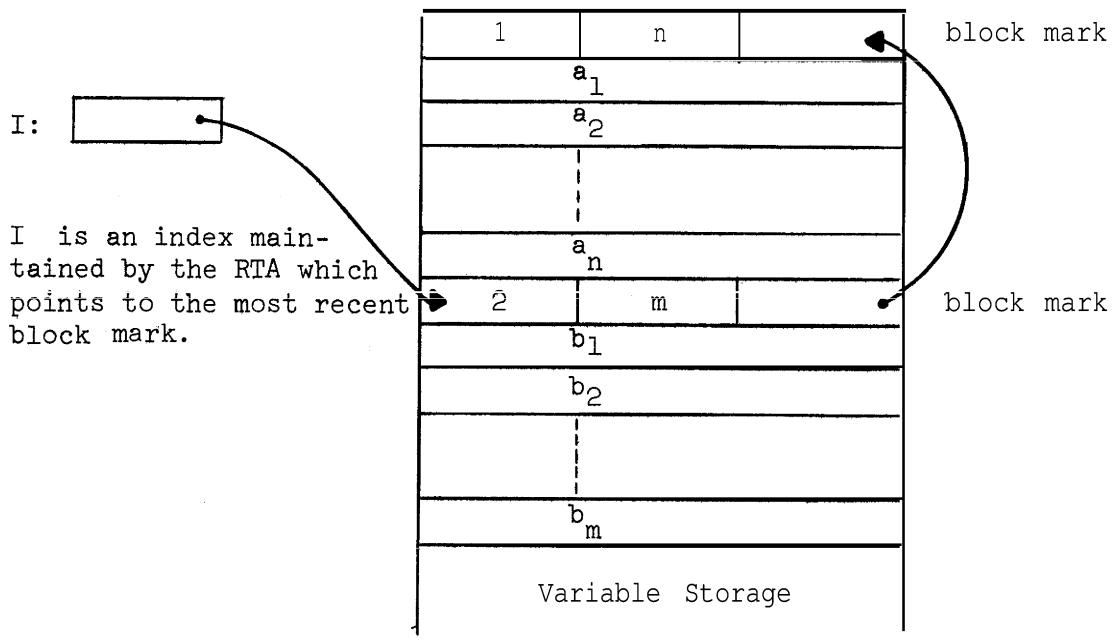
Execution of an ALGOL (block) requires that storage be allocated for its declared variables and temporary results. Temporaries present no problem since they are automatically created and destroyed in the run-time stack. During compilation, reduction to a (block head) causes the generation of the code BLOCKENTRY(bn, n), where n is the amount of space to be assigned for variables (obtained from the variable declarations), bn is the block number, and BLOCKENTRY is a RTA subroutine that performs the storage allocation. An address for a variable is produced as a pair (bn, on), where on is the order number or address relative to the base of the variable storage area for block bn . The block number bn indicates the level or depth of nesting of the block. A call to the RTA subroutine BLOCKEXIT is produced at the end of the block to release the storage.

Example 1

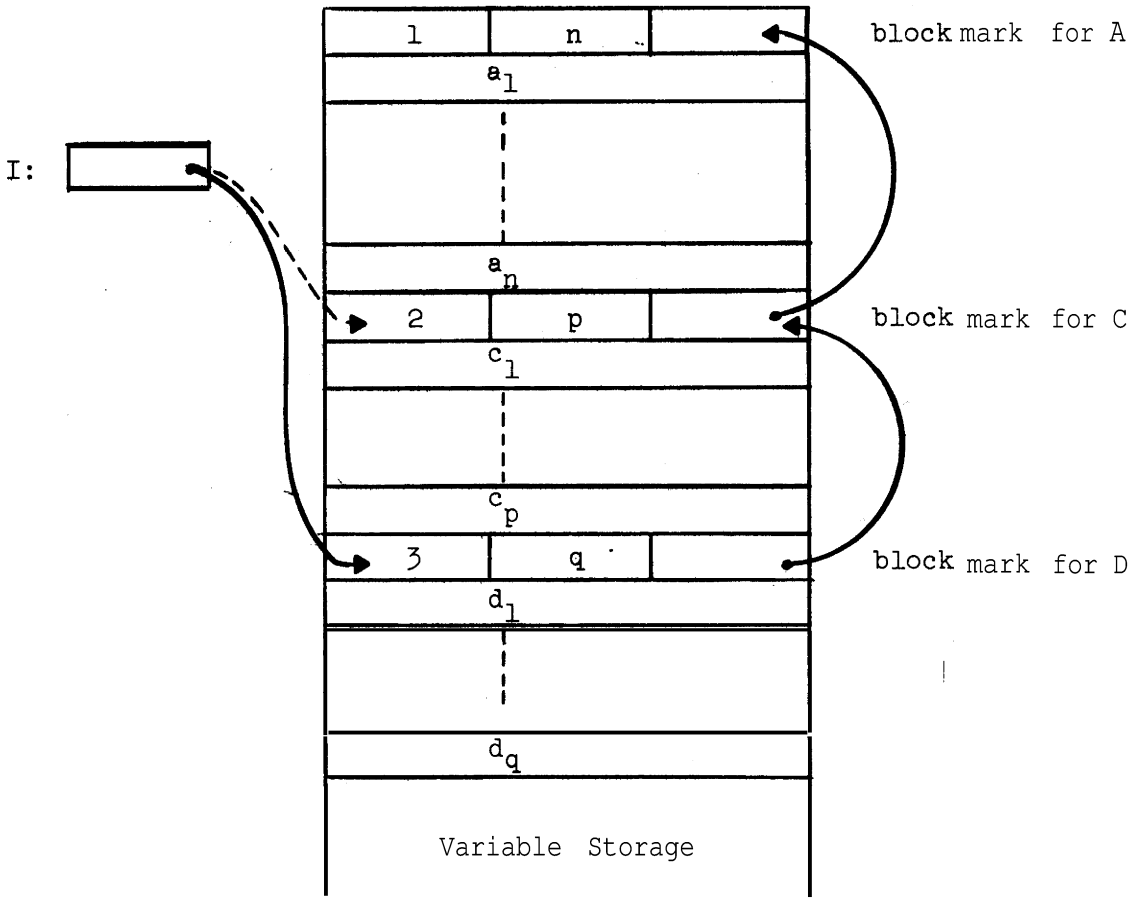
Program Skeleton	Generated Code
A: begin real $a_1, a_2, \ldots, a_n$;	BLOCKENTRY(1, n)
B: begin real $b_1, \ldots, b_m$;	BLOCKENTRY(2, m)
end B;	BLOCKEXIT
C: begin real $c_1, c_2, \ldots, c_p$;	BLOCKENTRY(2, p)
D: begin $d_1, \ldots, d_q$;	BLOCKENTRY(3, q)
end D	BLOCKEXIT
end C	BLOCKEXIT
end A --	BLOCKEXIT

d_k has the address (3, k).

(a) During execution of the block labeled B BLOCKENTRY(1, n) and BLOCKENTRY(2, m) have been invoked and the storage is allocated as follows:

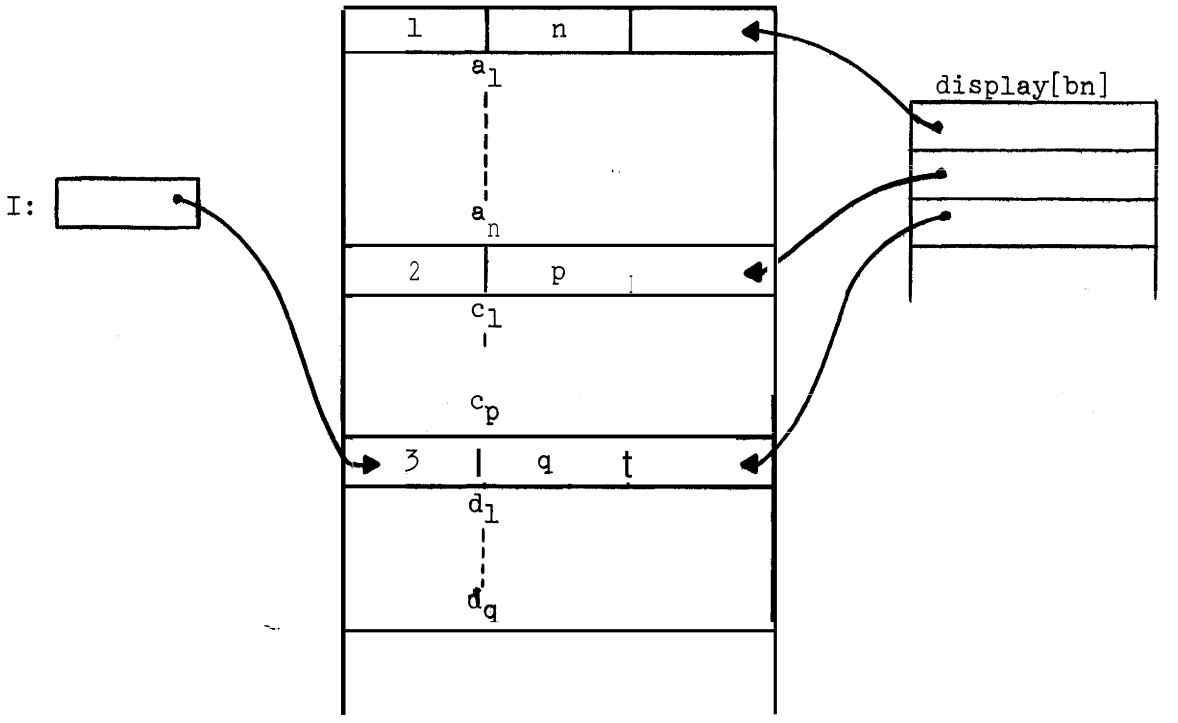


(b) After entry into block D, the variable storage has been re-allocated:



Note the space for B has been released. The dotted arrow indicates the change in I after BLOCKEXIT for D . To locate a symbol, $a_k = (1, k)$, while in block D, we chain back through the block marks until the bn in the block mark agrees with the bn in the address, i.e., at the block mark for A; a_k is then at the kth location following the block mark.

Variables may be located more efficiently by adding a block mark pointer array or display to the scheme described above. Then the address (bn, on) is translated immediately to display[bn] + on . Example 1 (b) can be redrawn to indicate the display:



VIII-3. Treatment of Procedures

The RTA scheme of the last section must be expanded in order to handle procedure calls correctly. Ignoring parameters for the moment, storage for procedure variables can be allocated in the same way as that for blocks; however, when a procedure is called, two types of information are needed:

(1) what variables are accessible to the procedure, and

(2) the return address of the procedure.

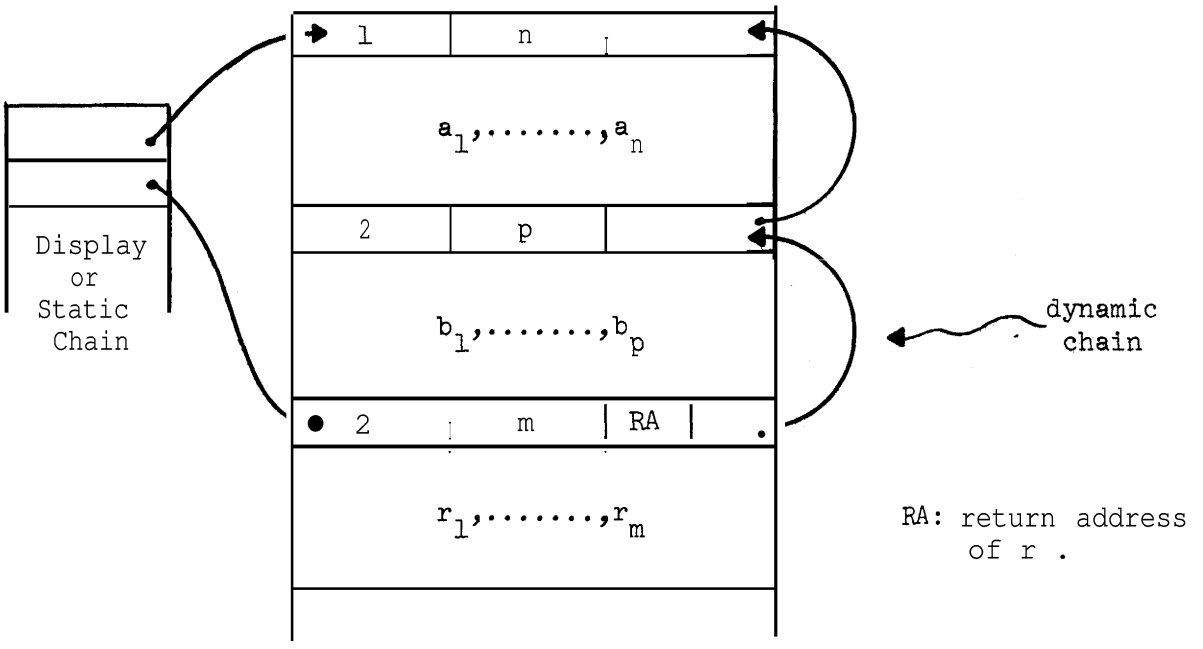
The first is given by the static block structure of the program, regardless of where the procedure was called from. The second is given by the dynamic behavior of the program at execution time; both are complicated by the possibility of recursion. The solution is to maintain two sets of pointer chains, a static chain or display of the last section, and a dynamic chain. The static chain determines which variables have currently

valid declarations; the dynamic chain points to the blocks that have been activated and not yet completed. Calls to the RTA routines BLOCKENTRY and BLOCKEXIT are produced for both procedure declarations and blocks; procedure calls are compiled into transfers to the procedure BLOCKENTRY . At run time, the RTA's allocate storage, update static and dynamic pointers, and keep track of procedure return addresses (RA) .

Example 1

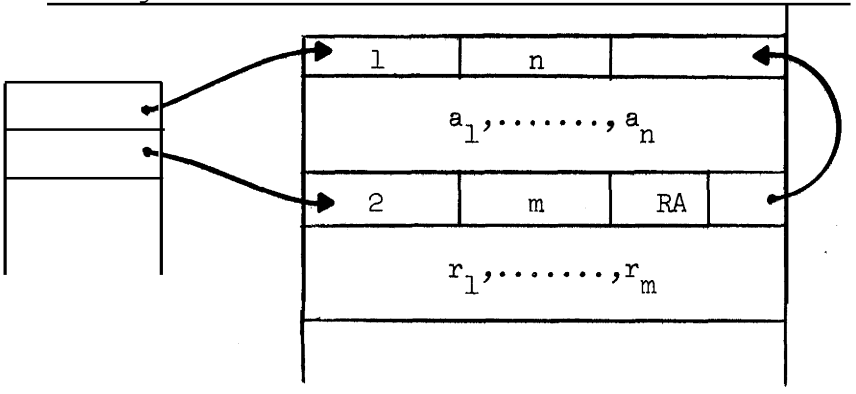
```
A: begin real a1,..., an;
   procedure r;
   begin real r1,..., rm;
      R: r1 := 0
   end r;
B: begfn real b1,..., bp;
   B1: r
   end B;
A1: r
end A
```

Storage Allocation at R when r is called at B1



The variables of block B are inaccessible since B and r are at the same level.

Storage Allocation at R when r is called at A1

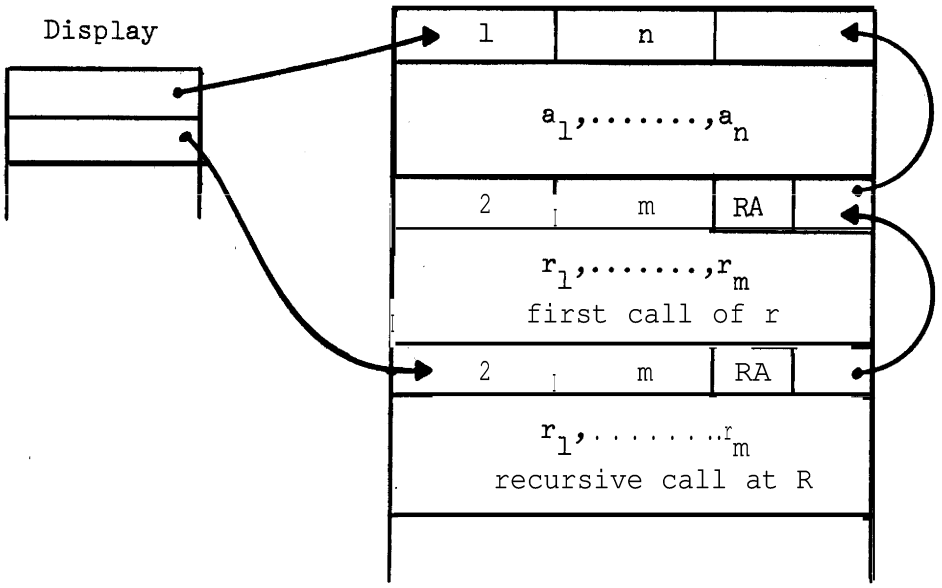


Storage allocation for recursive calls are correctly administered by the same method.

Example 2

```
A: begin real a1,..., an;
   procedure r;
   begin real r1,..., rm;
     .
     R: r
   end r;
Al: r;
end A
```

r is called at Al; after r is called recursively for the first time at R, storage and pointers appear as follows:



The dynamic chain gives the correct linkage upon return from r; the static chain makes inaccessible the original set of variables for r .

An additional mechanism is needed for procedure parameters:

Example 3

```
begin procedure P(a); real a;
                  a := a+1;
      B: begin real b;
               P(b)
         end
end
```

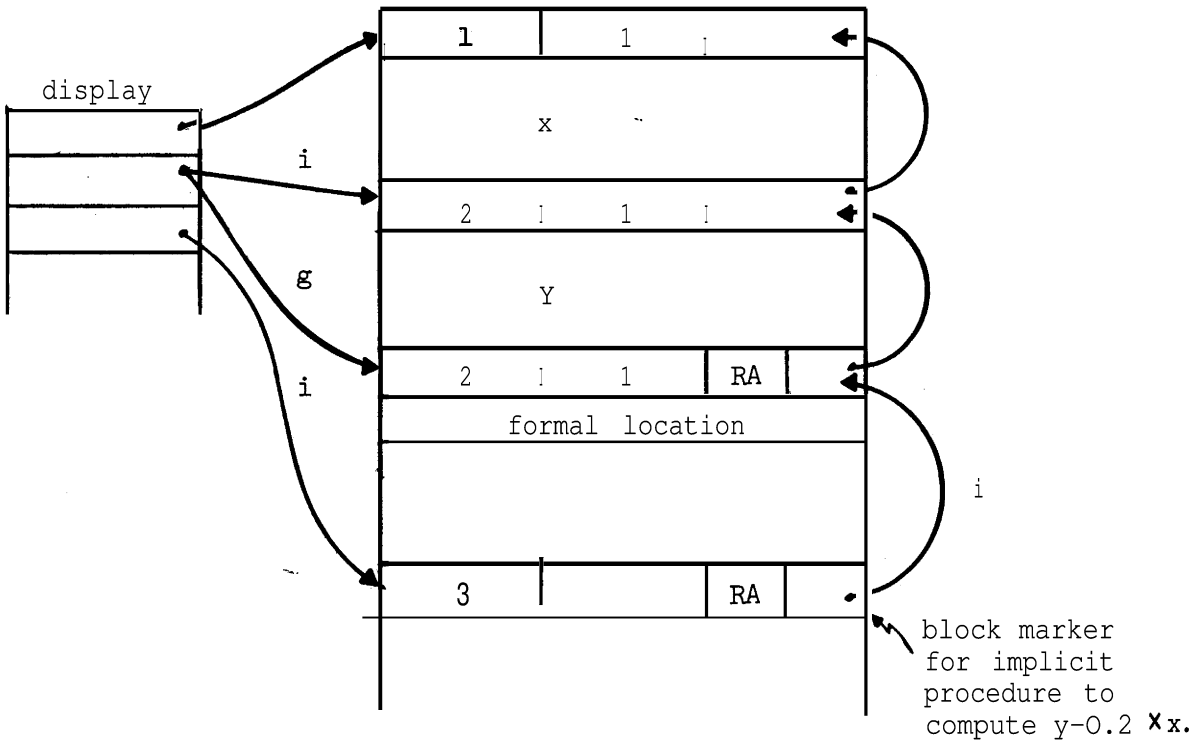
Normally, the variable b is inaccessible to procedure P since B and P are on the same block level. However, in this case, b is called by name and is therefore known in P .

On entering a procedure, storage is reserved for parameters as well as the variables declared in the procedure body. The parameter locations (called formal locations) contain transfers to implicit subroutines which compute the value or address of the actual parameters; at each use of a formal parameter inside a procedure, a transfer to the formal location is compiled (or the formal location could be indirectly addressed). Declarations which are valid at the place, C, of the procedure call are made accessible by regarding the implicit subroutine as a block inserted at C .

Example 4

```
begin real x;
      procedure g(t); real t;
          begin x := x+t;  x := x/(1-t)
          end g;
      begin real y;
           y := 0.5; x := 1.0;
           g(y-0.2 x x)
      end.
end
```

During the execution of g(y-0.2 x x), storage allocation is as follows:



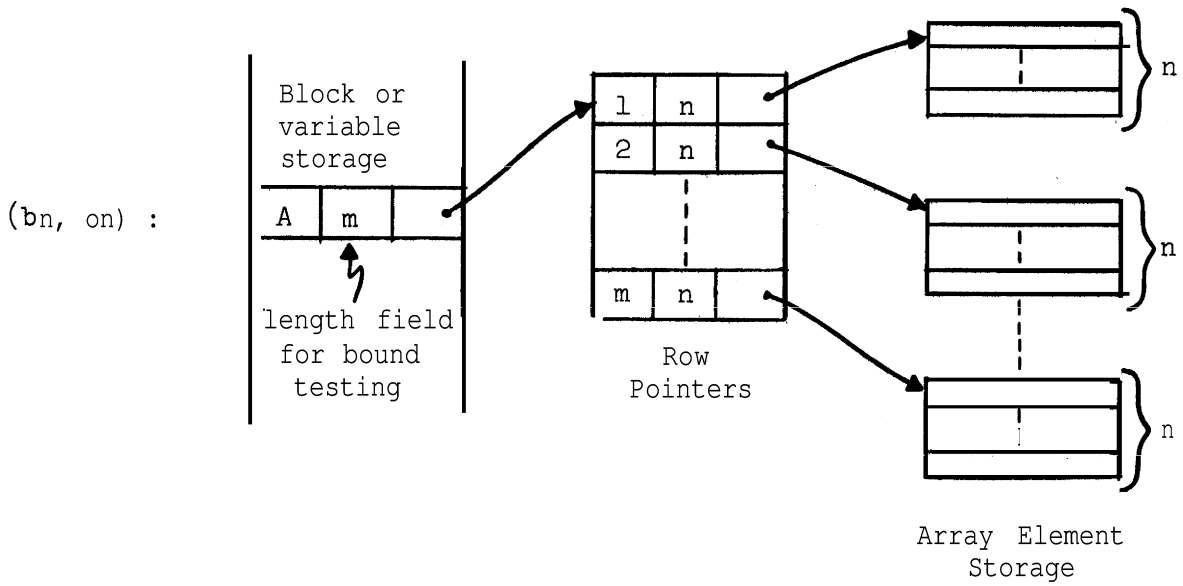
When the program is in the implicit subroutine, the pointers labeled i are in force; inside g, the i pointers are deleted and display [2] contains the g pointer.

It is apparent that a great deal of housekeeping must be done at run time to cope with the full generality of ALGOL. The administrative work can be reduced by eliminating the implicit subroutine for some types of name parameters, such as constants and simple variables, and inserting their value or actual address in the formal location. Value parameters can also be compiled in a simple manner so that they appear in the formal location. To avoid copying arrays, it is generally more efficient to pass them as name parameters. Arrays have been neglected in the preceding discussion; they present some special problems in storage allocation and addressing, and are examined next.

VIII-4. Arrays

For each array declared in a block, a storage location with address (bn, on) is reserved in the variable storage area. The RTA allocates array storage in a separate area and inserts both a pointer to that area and some mapping data in (bn, on) . Reference to a subscripted variable generates a call to a mapping function that computes the physical address of the element at run time.

One method for organizing array storage is to allocate a separate area for each row or column. The B5000 ALGOL compiler stores each row of an array as a linear string. For example, the declaration real array A[1:m, 1:n] results in the following run time organization:



The contents of element A[i, j] is then (((A) + i) + j) .

A second method is to completely linearize the array and store it in one contiguous area. A[1:m, 1:n] is stored A_{11}, A_{12},..., A_{1n}, A_{21}, ..., A_{2n},..., A_{m1}, A_{m2},..., A_{mn} (by row). The element A[i, j] has

the address $(i-1) \times n + j - 1$ relative to the base of A's storage area. The general case is treated as follows:

$$\underline{\text{array}}\ A[\ell_1:u_1, \ell_2:u_2, \ldots, \ell_k:u_k]$$

Let $A_i = u_i - \ell_i + 1$

The address of $A[n_1, n_2, \ldots, n_k]$ is then:

$$(\ldots((n_1 - \ell_1)\Delta_2 + (n_2 - \ell_2))\Delta_3 + \ldots + (n_{k-1} - \ell_{k-1}))\Delta_k + n_k - \ell_k$$

$$= ((\ldots(n_1\Delta_2 + \ldots + n_{k-2})\Delta_{k-1} + n_{k-1})\Delta_k + n_k$$

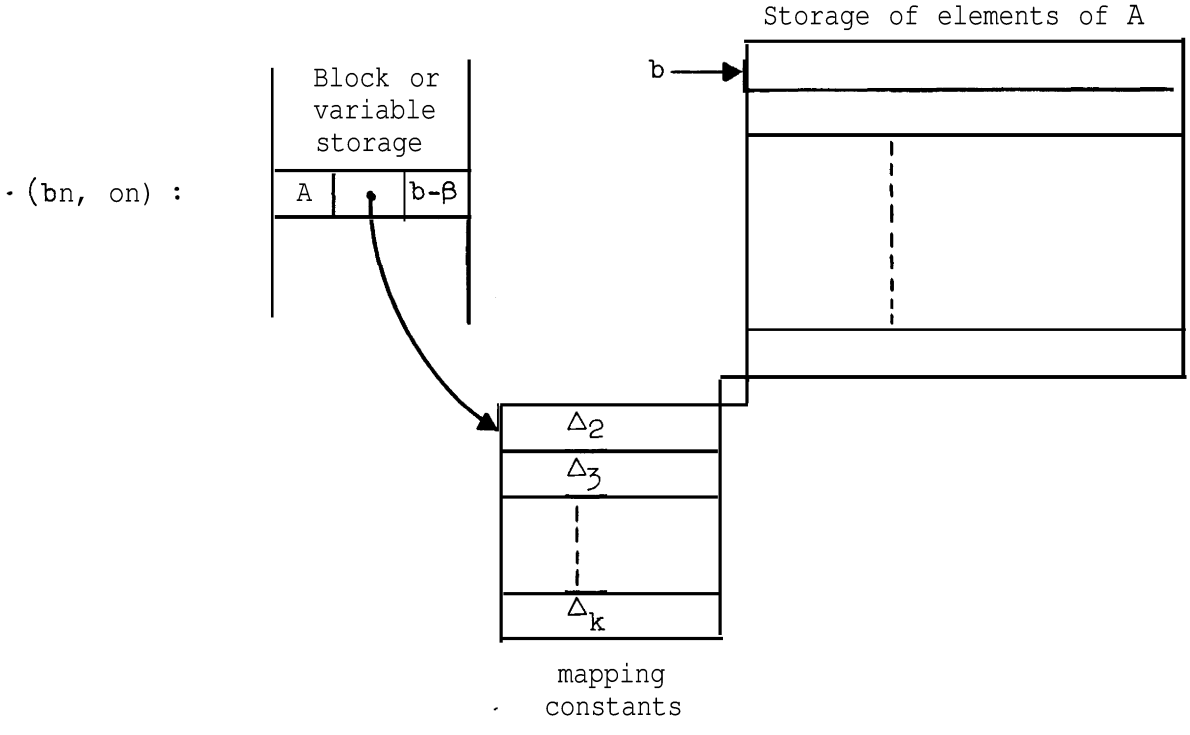
$$- ((\ldots(\ell_1\Delta_2 + \ldots + \ell_{k-2})\Delta_{k-1} + \ell_{k-1})\Delta_k + \ell_k$$

$$= \alpha - \beta .$$

β can be computed by the RTA when storage is allocated for A;

α is computed by the mapping function when the element is accessed.

Storage allocation after run time processing of the array declaration is:



b is the physical address of the first element of A, i.e.,

$A[\ell_1, \ell_2, \ldots, \ell_k]$.

$b - \beta$ is the address of the (fictitious) element $A[0, 0, \ldots, 0]$.

The physical address of an element is then $\alpha + b - \beta$.

The storage allocation problem for ALGOL compilers has been solved by a run time interpretive scheme (the RTA's). There is a similarity between this solution and that of the dynamic relocation problem described in Chapter V. The paging and segmentation methods can be conveniently employed to handle the dynamic allocation problems in an ALGOL compiler --in fact, the B5000 ALGOL compiler does this.

VIII-5. References

1. Randall, B. and Russell, L. J. ALGOL 60 Implementation. Academic Press, London and New York, 1964.

2. Naur, P. The Design of the GEIR ALGOL Compiler. BIT 3 (1963) 124-140, 145-166.

VIII-6. Problems

CS 235 B
Problem Set I

March 31, 1966
Prof. N. Wirth

The purpose of this problem set is to draw your attention to certain facilities and problems of ALGOL 60 which must be clearly understood before one discusses the implementation of ALGOL 60 on any computer.

For each of the problems list the output produced by the "write

statements". Not all of the programs are correct ALGOL 60, and even fewer can be handled by the B5500 system. Along with the numeric results, state in a brief comment what, if anything, is incorrect or at least controversial. Indicate where the B5500 system deviates from ALGOL 60.

You may do so, but you are not expected to use the computer for this problem set. If you had to resort to the computer, indicate for which problems. Give the answers on a separate paper.

```
1: begin integer i, j, m, n;
       i := 3; j := -2; m := 8 ÷ 2↑i; n := 8 ÷ 2↑j;
       write (m); write (n)
   end
```

```
2: begin integer procedure f(x); value x; integer x; f := x + 0.25;
       write(f(1.3))
   end
```

```
3: begin integer i; array A, B, C[1:1];
       integer procedure j; begin j := 1; i := i+1 end;
       i := 0; B[1] := 1; C[1] := 3;
       for A[j] := B[j] step B[j] until C[j] __;
       write(i)
   end
```

```
4: begin integer i, k, f;
       integer procedure n; begin n:= 5; k := k+1 end;
       procedure P(n); value n; integer n;
       for i := 1 step 1 until n do f:= fxi;
       k := 0; f := 1; P(n); write(f,k);
       k := 0; f := 1; Q(n); write (f,k)
   end
```

```
5: begin integer i, s; integer array A[0:n];
        i := n;
        while  i ≥ 0 A A[i] ≠ s  do  i := i-1;
        comment  anything wrong with this?;
   end
```

```
6: begin real x;
        procedure g(t); value t;  real t;
        begin t := x+t; x := t/(1-t)  end;
        x := 0.5;
        begin real x; x:= 0.5;
             g(x-o.2); write(x)
        end;
        write(x)
   end
```

```
7: begin integer array A[1:5], B[1:5, 1:5];
        integer  i,j;
        integer procedure  S(k,t); integer k, t;
        begin integer s; s := 0;
             for k := 1 step  1  until  5 do s := s+t;
             S := s
        end;
        comment  initialize  A and B to:
```

$$A = \begin{pmatrix} 3 \\ -2 \\ 0 \\ 4 \\ -1 \end{pmatrix} \quad B = \begin{pmatrix} 8 & 2 & 3 & 0 & 0 \\ 5 & -3 & 2 & 1 & 0 \\ 0 & -1 & 7 & 1 & 0 \\ 0 & 0 & 4 & 6 & 3 \\ 0 & 0 & 8 & -5 & 9 \end{pmatrix} ;$$

```
        write(S(i, A[i]));
        write(S(j, A[j] x B[j, j]));
        write(S(i, S(j, B[i, j])))
   end
```

```
8: begin
       procedure p(r,b); value b; Boolean b; procedure r;
       begin integer i;
            procedure q; i := i+1;
            i := 0;
            if b then p(q, ¬b) else r;
            write(i)
       end;
       p(p,true)
   end
```

```
9: begin procedure C(x); value x; integer x;
            begin own integer k;
                 if i=0 then k := x else begin k := kx2; write (k) end;
                 if x > 1 then C(x-1)
            end;
       integer ;
       for i := 0, 1 do C(5)
   end
```

```
10: begin integer i;
       real procedure P(k, x1, x2, x3, x4, x5); value k; integer k;
       begin real procedure ;
            begin k := k-1; Q := P(k, Q, x1, x2, x3, x4)
            end Q;
            P := if k < 0 then x4 + x5 else Q
       end P;
       for i := 1 step 1 until 4 do write(Pi, 1, 2, 3, 4, 5))
   end
```

11: TERM PROBLEM

You are to write an interpretive program for a simple programming language to be described presently. The language is designed for simple computations of the desk-calculator type with immediate response. The computer to be used is the Burroughs B5500, using a programmed character-input routine simulating characterwise input from a typewriter. The language is by no means a complete and very useful tool, but exhibits the basic features upon which extensions could easily be built.

Description of the Language

The elements are numbers, variables and operators. The numbers are decimal integers denoted in standard form. The variables are named by the letters A through Z. Thus there exist exactly 26 variables which do not have to be declared. There exist the following operators; listed in order of increasing priority:

$	logical OR
&	logical AND
< ≤ = ≠ ≥ >	relational (resulting in 1 or 0)
" @	min, max
+ -	add, subtract
× /	multiply, integer divide
*	exponentiate

These operators can be followed by a period (.) and are then to be understood as unary operator-s with the following meanings:

$.a = a

&. a = a

<. a = 0 < a (same for other relational ops.)

". ☋ ☐ ☋

@. a = a

+. a = a

-. a = 0 - a

×. a = a

/.a = 1 / a

*. a = 2 * a

The standard precedence rules can be over-ruled by use of parenthetical grouping in the conventional way. Using numbers, variables, operators and parentheses expressions can be formed, whose resulting value can then be assigned to any variable through an assignment statement of the form

$$v \leftarrow Exp;$$

The multiple assignment shall be admitted and is of the form

$$v1 \leftarrow v2 \leftarrow \ldots \leftarrow vn \leftarrow Exp;$$

The interpreter shall upon each execution of an assignment print the name of the variable and the value to be assigned. (This constitutes the output of the program.) Every assignment statement shall be terminated by a semicolon (;).

So far we have described Subset A of the language.

The language is able to handle vectors (linear arrays), represented as follows:

$$[E, F, \ldots . H]$$

where E, F through H are expressions. A vector can be assigned to a variable, but only if this variable has been previously declared as vector. A vector declaration takes the following form:

$$v : Exp;$$

and means that the variable v shall consist of as many elements as the value of the expression Exp indicates. Upon assignment, the vector to be assigned must be compatible, i.e., of equal length, with the variable v . A multiple vector declaration is written as

$$v1 : v2 : \ldots . : Exp;$$

All existing operators are now extended to apply to vectors (c.f. Iverson's notation),

according to the following definitions:

Let a, b be scalars, $\underline{x}$, y vectors, and 0 a binary operator, then

$$a\ 0\ \underline{x} = [a\ 0\ \underline{x}_1,\ a\ 0\ \underline{x}_2,\ \ldots,\ a\ 0\ \underline{x}_n]$$

$$x\ 0\ b = [\underline{x}_1\ 0\ b,\ \underline{x}_2\ 0\ b,\ \ldots,\ \underline{x}_n\ 0\ b]$$

$$\underline{x}\ 0\ \underline{y} = [\underline{x}_1\ 0\ \underline{y}_1, \underline{x}_2, \ldots\ \underline{x}_n\ 0\ \underline{y}_n]$$

Let $\underline{x}$ be a vector and 0 an operator, then

$$0.\ x = [o.\ \underline{x}_1\ 0\ \underline{x}_2\ 0\ \ldots\ 0\ \underline{x}_n] \qquad \text{(reduction of x)}$$

where 0. is the unary operator corresponding to the binary 0 .

Expressions involving vectors may, of course, also use parenthetical groupings.

Examples of statements:

$X \leftarrow A + B \times C$;

$Y \leftarrow A + [\ 1.\ 4.\ g.\ 16\ ]$

$z \leftarrow +.\ (\ X \times Z\ )$ (scalar product)

$X \leftarrow *.\ (\ [\ 1,\ A + B\ ] + [\ *.\ Z,\ 55\ ]\ ) + 1$

Hints

The implementation of this interpreter requires a combination of what usually is called a translator and an interpreter of sequential code. Instead of having the translator produce a list of code and then have the interpreter process it after termination of the translation process, the interpreter immediately processes an instruction when it is issued by the translator. The implementation of this method is greatly facilitated by the absence of conditional and go to statements.

Vectors must be created dynamically. The interpreter shall be written in such a fashion that after the execution of an assignment statement all storage used for temporary vectors is released again. Upon termination your program should print out a message indicating how much total vector storage space has been used up (through permanent vector declarations). This space shall initially include 1000 cells.

An example of a character-input routine is listed below and makes use of the following declarations:

```
INTEGER CC, WC;
ARRAY CARD[0:14];
LABEL EXIT;

STREAM PROCEDURE CLEAR (D);
BEGIN DI ← D; DS - 8 LIT " "; SI ← D; DS - 14 WDS END;

STREAM PROCEDURE TRCH (S, M, D, N); VALUE M, N;
BEGIN DI ← D; DI ← DI + N; SI ← S; SI ← SI + M; DS ← CHR END;

PROCEDURE INSYMBOL (S); INTEGER S;
BEGIN INTEGER T; LABEL L;
L:   IF CC = 7 THEN
     BEGIN IF WC = 8 THEN
           BEGIN READ (CARDFIL, 10, CARD [*]) [EXIT];
                 WRITE (PRINFIL, 15, CARD[*]; WC ← 0
           END
           ELSE WC ← WC + 1;
           CC ← 0
     END
     ELSE CC ← CC + 1;
     TRCH (CARD[WC], CC T, 7);
     IF T = " " THEN GO TO L ELSE S ← T
END
```

At the due date, submit

A: A statement whether you implemented Subset A or the entire language;

B. A syntactic description of the language you implemented;

C. A block diagram indicating the main principles of the system, (this diagram should not exceed one page);

D. A table of the basic characters of the language and their priorities (if you used such);

E. A B5500-ALGOL listing of the system followed by

F. The output produced by your system and a test program to be issued one week before the due 'date. (One will be issued for Subset A; one for the entire language.)

Solution To TERM PROBLEM

```
BEGIN COMMENT  CLEVER TYPEWRITER.      N.WIRTH     MARCH 1965;
   INTEGER R,X, NUMBER;
   INTEGER CC, WC;        COMMENT INPUT POINTERS;
   INTEGER I;             COMMENT TRANSLATOR STACK POINTER;
   INTEGER J;             COMMENT ARRAY STORE POINTER;
   INTEGER K;             COMMENT WORKSTACK POINTER;
   INTEGER ARRAY CARD [0:14];      COMMENT INPUT BUFFER;
   INTEGER ARRAY T[0:31];          COMMENT TRANSLATOR STACK;
   ARRAY S,V[0:127]; COMMENT THE WORKING STACK;
   ARRAY A[0:1022]; COMMENT SECONDARY ARRAY STORAGE;
   INTEGER ARRAY F,G[0:63];        COMMENT PRIORITY FUNCTIONS;
   LABEL L1,L2,L3, NEXT, EXIT;
LABEL DIG, NIL, MAX, ARY, GTR, GEQ, ADD, VAR, DOT, LBK, LAN, LPA,
   LSS, ASS, MUL, LOR, MIN, SUB, RPA, SCL, LEQ, DVD, CMA, NEQ, EQL,
   RBK, PWR, UMX, UGR, UGQ, UAD, UAN, ULS, UML, UOR, UMN, USB, ULQ,
   UDD, UNQ, UEL, UPR;
SWITCH EVALUATE ←
   DIG, DIG, DIG, DIG, DIG, DIG, DIG, DIG,
   DIG,DIG, NIL, MAX, NIL, ARY, GTR, GEQ,
   ADD, VAR, VAR, VAR, VAR, VAR, VAR, VAR,
   VAR, VAR, DOT, LBK, LAN, LPA, LSS, ASS,
   MUL, VAH, VAR, VAR, VAR, VAR, VAR, VAR,
   VAR, VAR, LOR, PWR, SUB, RPA, SCL, LEQ,
   NIL, DVD, VAR, VAR, VAR, VAR, VAR, VAR,
   VAR, VAR, CMA, RBK, NEQ, EQL, NIL, MIN,
   NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL,
   NIL, NIL, NIL, UMX, NIL, NIL, UGR,UGQ,
   UAD, NIL, NIL, NIL,NIL,NIL,NIL,NIL,
   UAD, NIL, NIL, NIL, UAN, NIL, ULS, NIL,
   UML, NIL, NIL, NIL, NIL,NIL,NIL,NIL,
   NIL, NIL, UOR, UPR, USB, NIL, NIL, ULQ,
   NIL, UDD, NIL, NIL, NIL, NIL, NIL, NIL,
   NIL, NIL, NIL, NIL, UNQ, UEL, NIL, UMN;
   DEFINE TYPE = [1:2] #, LOW =[10:10]#, UP = [20:10] #;
   DEFINE FLAG = [3:1] #, ADR = [30:10] #;
   DEFINE VALTYPE = 1 #, ADRTYPE = 2 #, ARYTYPE = 3 #;
   DEFINE THRU = STEP 1 UNTIL #;
   DEFINE Z = @56 #;

STREAM PROCEDURE CLEAR(D);
 BEGIN      DI ← D; DS ← 8 LIT" "; SI ← D; DS ← 14 WDS  END ;
STREAM PROCEDURE TRCH (S,M,D,N); VALUE M,N;
 BEGIN DI ← D; DI ← DI+N; SI ← S; SI ← SI+M; DS ← CHR END ;
PROCEDURE INSYMBOL(S); INTEGER S;
   BEGIN INTEGER T; LABEL L;
   L: IF CC = 7 THEN
      BEGIN IF WC=8 THEN
         BEGIN READ (CARDFIL,10, CARD[*])[EXIT];
            WRITE (PRINFIL,15,CARD[*]); WC ← 0
         END
         ELSE WC ←WC+1;
         cc ← 0
      END
      ELSE CC←CC+1;
      TRCH (CARD[WC], CC, T, 7);
```

```
      IF  T = " "  THEN  GO  TO  L  ELSE  S ←T
   END ;
PROCEDURE ERROR(N); VALUE N; INTEGER N;
   BEGIN LABEL L; COMMENT PRINT MESSAGE AND RESUME PROCESSING;
   SWITCH FORMAT TEXT ←
   ("PARENTHESES  00  NOT MATCH"),
   ("INCOMPATIBLE ARRAYS"),
   ("INCOMPATIBLE ASSIGNMENT"),
   ("ASSIGNMENT TO UNKNOWN QUANTITY"),
   ("ILLEGAL LIST ELEMENT"1 ,
   ("ILLEGAL OPERATOR"),
   ("ARRAYSPACE EXHAUSTED"),
   ("DIVISION BY ZERO");
      WRITE(TEXT[N]);
      FOR K ← K STEP -1 UNTIL 64 00
      IF S[K].TYPE = ARYTYPE A N D S[K].FLAG = 1 THEN J ← S[K].LOW;
   L: IF R ≠ ";" THEN BEGIN INSYMBOL(R); GO TO L END;
      I ← 0; GO  TO  Lli
   END;
PROCEOURE FETCH;
      BEGIN  V[K] ← V[S[K].ADR]; S[K] ← S[S[K].ADR] END ;
PROCEDURE UNARY(FCT,NULL);
   REAL PROCEDURE FCT; REAL NULL;
   B E G I N INTEGER L,U,X; REAL E; E ← NULL;
      I F S[K].TYPE = ADRTYPE THEN FETCH;
      IF S[K].TYPE = ARYTYPE THEN
      B E G I N  L ← S[K].LOW; U ← S[K].UP;
         IF  S[K].FLAG = 1 THEN  J  ← L;
         FOR  X ← L THRU U DO E ← FCT (E, A[X]);
         V[K] ← E; S[K].TYPE ← VALTYPE;
      END ELSE
      V[K] ←  FCT (NULL, V[K]);
   END UNARY ;
PROCEDURE BINARY(FCT);
   REAL PROCEDURE FCT;
   BEGIN
      I F S[K].TYPE = ADRTYPE THEN FETCH;
      K ← K-1;
      I F S[K].TYPE = ADRTYPE T H E N FETCH;
      IF S[K].TYPE = AHYTYPE THEN
      BEGIN IF S[K+1].TYPE = AHYTYPE THEN
         BEGIN INTEGER L1,L2,U1,U2,X,Y;
            L1 ← S[K].LOW; U1 ← S[K].UP; L2 ← S[K+1].LOW;
            U2 ← S[K+1].UP; Y ← L2;
            I F U1-L1 ≠ U2-L2 T H E N ERROR(1);
            IF S[K+1].FLAG = 1 T H E N J ← L2;
            IF S[K].FLAG = 1 T H E N J ← L1;
            S[K].LOW ← J; S[K].UP ← J+U1-L1; S[K].FLAG ← 1;
            FOR X • L1 THRU UI D O
            BEGIN AC J1 ← FCT(A[X], A[Y]); Y ← Y+1; J ← J+1 END ;
         END ELSE
         BEGIN INTEGER L,U,X; REAL Y;
            L ← S[K].LOW; U ← S[K].UP; Y ← V[K+1];
            IF S[K].FLAG = 1 THEN J • L;
            S[K].LOW ← J; S[K].UP ← J+U-L; S[K].FLAG ← 1;
```

```
            FOR X ← L THRU U DO
            BEGIN A[J] ← FCT (A[X], Y); J ←J+1 END
         END
      END ELSE
      BEGIN IF S[K+1].TYPE = ARYTYPE THEN
         BEGIN INTEGER L,U,X; REAL Y:
            L ← S[K+1].LOW; U ← S[K+1].UP; Y ← V[K];
            IF S[K+1].FLAG = 1 THEN J ←L; S[K].FLAG ← 1;
            S[K].TYPE ← ARYTYPE; S[K].LOW ← J; S[K].UP ← J+U-L;
            FOR X ← L THRU U 00
            BEGIN A[J] ← FCT (Y, A[X]); J ← J+1 END ;
         END ELSE
         V[K] ← FCT(V[K],V[K+1])
      END;
   END BINARY;
REAL PROCEDURE SUM(X,Y); VALUE X,Y; REAL X,Y; SUM ← X + Y;
REAL PROCEDURE DIFF(X,Y); VALUE X,Y; REAL X,Y; DIFF ← X - Y;
REAL PROCEDURE PROD(X,Y); VALUE X,Y; REAL X,Y; PROD ← X x Y;
REAL PROCEDURE QUOT(X,Y); VALUE X,Y; REAL X,Y;
   IF Y = 0 THEN ERROR(7) ELSE QUOT←X DIV Y;
REAL PROCEDURE EXPO(X,Y); VALUE X,Y; REAL X,Y; EXPD ← X*Y;
REAL PROCEDURE LESS(X,Y); VALUE X,Y; REAL X,Y; LESS ←REAL(X<Y);
REAL PROCEDURE LEQL(X,Y); VALUE X,Y; REAL X,Y; LEQL ←REAL(X≤Y);
REAL PROCEDURE EQUL(X,Y); VALUE X,Y; REAL X,Y; EQUL ←REAL(X=Y);
REAL PROCEDURE NEQL(X,Y); VALUE X,Y; REAL X,Y; NEQL ←REAL(X≠Y);
REAL PROCEDURE GEQL(X,Y); VALUE X,Y; REAL X,Y; GEQL ←REAL(X≥Y);
REAL PROCEDURE GRTR(X,Y); VALUE X,Y; REAL X,Y; GRTR ←REAL(X>Y);
REAL PROCEDURE INFI(X,Y); VALUE X,Y; REAL X,Y;
   INFI ←IF X < Y THEN X ELSE Y;
REAL PROCEDURE SUPR(X,Y); VALUE X,Y; REAL X,Y;
   SUPR ← IF X> Y THEN X ELSE Y;
REAL PROCEDURE UNON(X,Y); VALUE X,Y; REAL X,Y;
   UNON ← REALCBOOLEANCX) OR BOOLEAN(Y));
REAL PROCEDURE INSC(X,Y); VALUE X,Y; REAL X,Y;
   INSC ← REALCBOOLEANCX) AND BOOLEAN(Y));

COMMENT INITIALIZE POINTERS AND TABLES;
      WC ← 8; CC ← 7; CLEAR (CARD[0] );
      I ← J ← NUMBER ← 0; T[0] ← "#";
      FOR K ← 0 THRU 63 DO S[K].TYPE ← VALTYPE; K ← 63;

COMMENT PRIORITY FUNCTIONS OF BASIC SYMBOLS.
      SYMBOL                  F     G     # (OCTAL)
        (                     1     19    35
        [                     20    19    33
        %                     1     19    73
        $  (OR)               6     5     52
        &  (AND)              A     9     34
        < ≤ = ≠ ≥ >           10    11    36  57  75  74  17  16
        " @  (MIN MAX)        12          77  13
        +  -                  14    13    20  54
        x  /                  16    15    40  61
        *                     18    17    53
        LETTER                20    19
        DIGIT                 23    19
```

```
        )              20              1       55
        1              20              3       76
        ←                                      37
        SEMICOLON   2   1  19      0           56
        ,              20              3       72
        .
        # (FILEMARK) 20  -1  19  0              12
        :  (ARRAY)             2      19       15          ;
    FILL F[*] WITH
       20,20,20,20,20,20,20,20,20,20,-1,12, 0, 2,10,10,
       14,20,20,20,20,20,20,20,20,20,20,20, 8, 1,10, 2,
       16,20,20,20,20,20,20,20,20,20, 6,18,14,20, 1,10,
        0,16,20,20,20,20,20,20,20,20,20, 1,10,10,20,12;
    FILL G[*] WITH
       19,19,19,19,19,19,19,19,19,19, 0,11, 0,19, 9, 9  ,
       13,19,19,19,19,19,19,19,19,19,19,19, 7,19, 9,19,
       15,19,19,19,19,19,19,19,19,19, 5,17,13, 1, 0, 9,
        0,15,19,19,19,19,19,19,19,19, 3,19, 9, 90 1,11;

COMMENT READ AND REORDER BASIC SYMBOLS. BRANCH TO INTERPR,RULES;
L1:   INSYMBOL(R);
L2:   IF F[T[I],[42:6]] ≤ G[R] THEN
         BEGIN I ← I+1; T[I]←R; GO TO L1 END ;
L3:   IF F[T[I-1].[42:6]] = G[T[I].[42:6]] THEN
         BEGIN I ←I-1;;GO TO L3 END ;
      G O T O EVALUATE[T[I]+1];
NEXT: I ← I-1; GO TO L2;

ADD:  BINARY(SUM); GO TO NEXT;
sue :  BINARY(DIFF); GO TO NEXT;
MUL:  BINARY(PROD); GO TO NEXT;
DVD:  BINARY(QUOT); GO TO NEXT;
PWR:  BINARY(EXPO); G O T o NEXT;
MIN:  BINARY(INFI); GO TO NEXT;
MAX:  RINARY(SUPR); GO TO NEXT;
LSS:  BINARY(LESS); GO TO NEXT;
LEG:  BINARY(LEQL); GO TO NEXT;
EQL:  BINARY(EQUL); G O TO NEXT;
NEG:  BINARY(NEQL); GO TO NEXT;
GEQ:  BINARY(GEQL); GO TO NEXT;
GTR:  RINAHY(GRTR); GO TO NEXT;
LAN:  BINARY(INSC); GO TO NEXT;
LORa  BINARY(UNON); G O TO NEXT;
UAC:  UNAHY(SUM,0); GO TO NEXT;
USB:  UNARY(DIFF,0); GO TO NEXT;
UML:  UNARY(PROD, 1); GO TO NEXT;
uoc:  UNARY(QUOT,1); GO TO NEXT;
UPR:  UNARY (EXPO, 2); GO TO NEXT;
UMN:  UNARY (INFI, Z); GO TO NEXT;
UMX:  UNARY (SUPR,-Z); GO TO NEXT:
ULS:  UNARY (LESS, 0); GO TO NEXT;
ULG:  UNARY (LEQL, 0); GO TO NEXT;
UEL:  UNARY(EQUL, 0); GO TO NEXT;
UNC:  UNARY (NEQL, 0); GO TO NEXT;
UGQ:  UNAHY(GEQL,0); GO TO NEXT;
```

```
UGR:  UNARY (GRTR, 0); GO TO NEXT;
UAN:  UNARY (INSC, 1); GO TO NEXT;
UOR:  UNARY (UNON, 0); GO TO NEXT;
VAR:  K ← K+1; S[K].TYPE ← ADRTYPE; S[K].ADR ← T[I]; GO TO NEXT;
DIG:  NUMBER ← NUMBER × 10 + T[I];
      IF R ≥ 10 THEN
      BEGIN K ← K+1; S[K].TYPE ← VALTYPE; V[K] ← NUMBER; NUMBER ←0 END;
      GO TO NEXT;
DOT:  T[I-1] ← T[I-1]+64; GO TO NEXT;
ARY:  X ← S[K-1].ADR;
      IF S[K].TYPE = ADRTYPE THEN FETCH;
      S[X].TYPE ← ARYTYPE; S[X].LOW ← J; S[X].UP ← J+V[K]-1;
      S[X].FLAG ← 0;  J ← J + V[K]; IF J > 1022 THEN ERROR(6);
      K ← K-1; S[K] ← S[K+1]; V[K] ← V[K+1]; GO TO NEXT;
LBK:  K ← K+1; S[K].TYPE ← ARYTYPE; S[K].FLAG ← 1;
      S[K].LOW ← J; S[K].UP ← J-1; T[I] ← "%"; I ← I+1; GO TO NEXT;
RBK:
CMA:  IF S[K].TYPE = ADRTYPE THEN FETCH;
      IF S[K].TYPE ≠ VALTYPE OR S[K-1].TYPE ≠ ARYTYPE THEN ERROR(4);
      K ← K-1; S[K].UP ← J; A[J] ← V[K+1]; J ← J+1;
      IF J > 1022 THEN ERROR(6); GO TO NEXT;
ASS:  IF S[K-1].TYPE ≠ ADRTYPE THEN ERROR(3);
      IF S[K].TYPE = ADRTYPE THEN FETCH;
      X ← S[K-1].ADR;
      IF S[X].TYPE ≠ S[K].TYPE THEN ERROR(2);
      IF S[X].TYPE = ARYTYPE THEN
      BEGIN INTEGER L1,L2,U1,U2;
         WRITE ([NO], <"*  ",A1," ←">, X);
         L1 ← S[X].LOW; U1 ← S[X].UP; L2 ← S[K].LOW; U2 ← S[K].UP;
         IF U1-L1 ≠ U2-L2 THEN ERROR(1);
         FOR X ← L1 THRU U1 DO BEGIN A[X] ← A[L2]; L2 ← L2+1 END ;
         WRITE (<X6,10I10>, FOR X ← L1 THRU U1 DO A[X]);
      END ELSE
      BEGIN V[X] ← V[K]; WRITE (<"*  ",A1," ←",I10>, X, V[X]);
      END ;
      K ← K-1; S[K] ← S[K+1]; V[K] ← V[K+1]; GO TO NEXT;
RPA:  ERROR(0);
SCL:  IF S[K].TYPE = ARYTYPE AND S[K]. FLAG = 1 THEN J ← S[K].LOW;
      K ← K-1; GO TO NEXT;
LPA:  GO TO NEXT;
NIL:  ERROR (5);
EXIT:
   WRITE (</"ARRAYSPACE USED:",I4," CELLS">, J);
END .
```

cs 236 B

April 5,1966

Problem 2

N. Wirth

Design a simple programming language for complex arithmetic and implement it on the B5500 computer. The implementation shall consist of a translator based on a precedence syntax analyzer, and an interpreter of the compiled code.

I. The language.

The language should include facilities to express arithmetic operations on complex numbers and variables, such as addition, subtraction, multiplication, division, taking absolute value, sign inversion, comparison and possibly others. On the statement level there should exist the assignment operation, an output operation, and facilities for conditional and repeated execution of statements. Variables are designated by identifiers in the usual sense. A program shall be preceded by some form of declaration of those identifiers.

II* The translator.

The translator should consist of three main parts:

1. A routine reading basic symbols from the input source. It is recommended that this routine reads entire identifiers (and possibly numbers) which are considered in the syntax as a basic symbol. The source program should be listed by the printer.

2. A set of interpretation rules, corresponding to the syntactic rules of the language.

3. The algorithm for syntactic analysis.

The compiled code should be printed in a readable form upon completion of the compilation.

III* The Interpreter.

The interpreter executes the program compiled by the translator. The computer represented by-this interpreter should consist of an instruc-

tion register, an instruction counter, a set of arithmetic registers, and a memory, divided into a program-, and a data-part. The interpreter shall not include a stack mechanism.

In order to determine the precedence relations and functions, a syntax processing program is available on the B5500 computer. This program accepts a sequence of syntactic productions, one per card, punched in the following format: Each card consists of 6 fields, each 12 characters long, each representing a symbol (blank spaces count!). The first field represents the left part symbol of the production; if it is left entirely blank, then the left part symbol from the previous production is copied into it.

The syntax processor is called in the following way:

a. In the "system" field of the type-II card write "DISKIO".

b. The type-11 card is followed by a card containing

? EXECUTE SYNTAX/PROC1

where ? is a 2-8 punch in column 1. This is followed by a "Green card", followed by the data.

Use a time estimate of 2 minutes.

The total available machine time for this problem is 30 minutes. On April 19, submit the output produced by the syntax processor from the syntax underlying your language. May 6 is the final due date, when you should submit:

1. your tested compiler and interpreter program,

2. a clear and systematic description of your language, and of the organization and the instruction code of your interpreter, and

3. an output produced from a sample program. This sample program should demonstrate the main features of your language, and the correctness of your translator.

cs 236 B

N. Wirth

May 1966

A SOLUTION TO PROBLEM 2

Introduction

A description is given of a simple programming language to express computational processes involving complex numbers. The structure of the language is defined by a syntax (described in BNF). To each syntactic construction corresponds a certain operation which is systematically described by the processor. This processor has been chosen to consist of two parts:

1. a translator (compiler), and
2. an interpreter, closely reflecting the design and capabilities of a present-day computer.

The Language

The basic constituents of the language are complex numbers and variables. They can be used as operands in expressions, containing the dyadic operators of addition, subtraction, multiplication and division, and the monadic operators of sign inversion, exponentiation (e^x), selection of the real or imaginary part of a complex number (real x, im x), taking the absolute value (modulus), and of identity.

Expressions are constituents of assignment statements, which specify that the value of the expression be assigned to a variable. Statements can be executed conditionally, depending on whether a relation between two complex numbers holds or not. In the same fashion, a statement may be' executed repeatedly as long as (while) a relation is satisfied. Sequences of statements may be bracketed and thus be subjected to conditions as a unit. Relations on complex numbers are understood as the ordering relations taken on their absolute values.

Variables are denoted by freely chosen names, so called identifiers, i.e. sequences of letters and digits the first element being a letter. All identifiers must, however, be declared in the heading of the program. Since, due to the limited character set of the equipment available,

certain operators and delimiters are represented by sequences of letters, the following such sequences may not be chosen as identifiers:
NEW, BEGIN, END, IF, THEN, ELSE, WHILE, DO, OUT, EXP, ABS, REAL, IM

Numbers are denoted as follows (they are treated as basic constituents of the language and are therefore not described in the general syntax):

Syntax of numbers:

(number) ::= (real part)I(imaginary part)|(real part)

(real part) ::= (real number)

(imaginary part) ::= (real number)|-(real number)

(real number) ::= (digit sequence)|
(digit sequence) . (digit sequence)

(digit sequence) ::= (digit)|(digit sequence)(digit)

Examples of numbers:

1 12.5 91.5I23.8 0I-0.75 0.8311

The Processing System

The processing system is given as a B5500 Extended Algol program. It utilizes the techniques of precedence syntax analysis as discussed in class and as described in Wirth and Weber [1].

The syntax of the language is analyzed by a program which determines the precedence relations (printed below in the form of a matrix) and the precedence functions (F and G) of the symbols of the language. These functions, along with tables representing the productions of the syntax (KEY and PRTB), occur in the program of the compiler. The organization of the two latter tables is as follows:

KEY[i] represents for the i'th symbol the index in the production table PRTB, where those productions are listed whose right part string begins with the i'th symbol. For each production, the right part is listed without its leftmost symbol, followed by the identification number of the listed production and the left part symbol of the production. The end of the list of productions referenced via KEY[i] is marked with a 0 entry in PRTB.

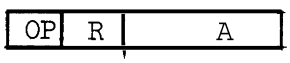
This representation of the productions was chosen to speed up the table lookup process. Clearly, even more efficient methods could be devised.

A program listing the compiled code in mnemonic form is activated before execution of the code.

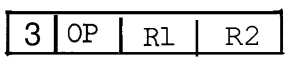
The ficticious computer, represented by the interpreter, consists of the following elements:

1. A program storage area (PROGRAM), into which the code is compiled.
2. A data storage area (DATAR, DATAC), into which constants (numbers) are compiled.
3. A set of 16 "registers" (REGR, REGC), upon which aritmnetic operations can be performed.
4. An instruction register (IR), holding the currently executed instruction.
5. An instruction counter (IC), holding the address of the next instruction in sequence.
6. A condition register (TOGGLE), holding the result of a comparison.

The instruction formats are the following:

a. 

OP ≠ 3

b. 

In case (a), the OP field designates the operations of a fetch, a store, or a branch, involving the register specified by the R field and the storage cell addressed by the A field (in the case of a branch, the R field determines whether the branch is taken unconditionally or depending on the value of the condition register). In case (b), the OP field specifies the operation to take place on the registers specified by the R1 and R2 fields.

Two Examples

Two examples of short programs are given below. The first is intended to illustrate the main features of the language. The second example was executed with a modified output operator, providing a primitive graphic representation of the complex plane.

Reference:

1. "EULER: A Generalization of ALGOL and its formal definition," Comm. ACM 9/1,2 (Jan. Feb. 1966)

PRODUCTIONS

1	<PROGRAM>	←	$	<HEADING>	<COMP STAT> $
2	<HEADING>	←	<DECLAR>	;	
3	<DECLAR>	←	NEW	<ID>	
4		←	<DECLAR>	,	<ID>
5	<COMP STAT>	←	<COMP ST H>	END	
6	<COMP ST H>	←	BEGIN		
7		←	<COMP ST H>	<STAT>	;
8	<STAT>	←	<STAT*>		
9	<STAT*>	←	<SIM STAT>		
10		←	<COND STAT>		
11		←	<ITER STAT>		
12	<COND STAT>	←	<IF CLAUSE>	<STAT+>	
13		←	<IF CLAUSE>	<TRUE PART>	<STAT*>
14	<IF CLAUSE>	←	IF	<RELATION>	THEN
15	<TRUE PART>	←	<SIM STAT>	ELSE	
16	<ITER STAT>	←	<WHILE CL>	<STAT*>	
17	<WHILE CL>	←	<WHILE HD>	<RELATION>	DO
18	<WHILE HD>	←	WHILE		
19	<RELATION>	←	<EXPR>	<	<EXPR>
20		←	<EXPR>	≤	<EXPR>
21		←	<EXPR>	=	<EXPR>
22		←	<EXPR>	≠	<EXPR>
23		←	<EXPR>	≥	<EXPR>
24		←	<EXPR>	>	<EXPR>
25	<SIM STAT>	←	<ASS STAT>		
26		←	<COMP STAT>		
27		←	<OUT STAT>		
28	<OUT STAT>	←	OUT	<EXPR>	
29	<ASS STAT>	←	<VARIABLE>	←	<EXPR>
30		←	<VARIABLE>	←	<ASS STAT>
31	<EXPR>	←	<EXPR*>		
32	<EXPR*>	←	<TERM>		
33		←	<EXPR*>	+	<TERM>
34		←	<EXPR*>	-	<TERM>
35		←	+	<TERM>	
36		←	-	<TERM>	
37	<TERM>	←	<TERM*>		
38	<TERM*>	←	<FACTOR>		
39		←	<TERM*>	×	<FACTOR>
40		←	<TERM*>	/	<FACTOR>
41	<FACTOR>	←	<PRIMARY>		
42		←	EXP	<FACTOR>	
43		←	ABS	<FACTOR>	
44		←	REAL	<FACTOR>	
45		←	IM	<FACTOR>	
46	<PRIMARY>	←	<NUMBER>		
47		←	<VARIABLE>		
48		←	(	<EXPR>	)
49	<VARIABLE>	←	<ID>		

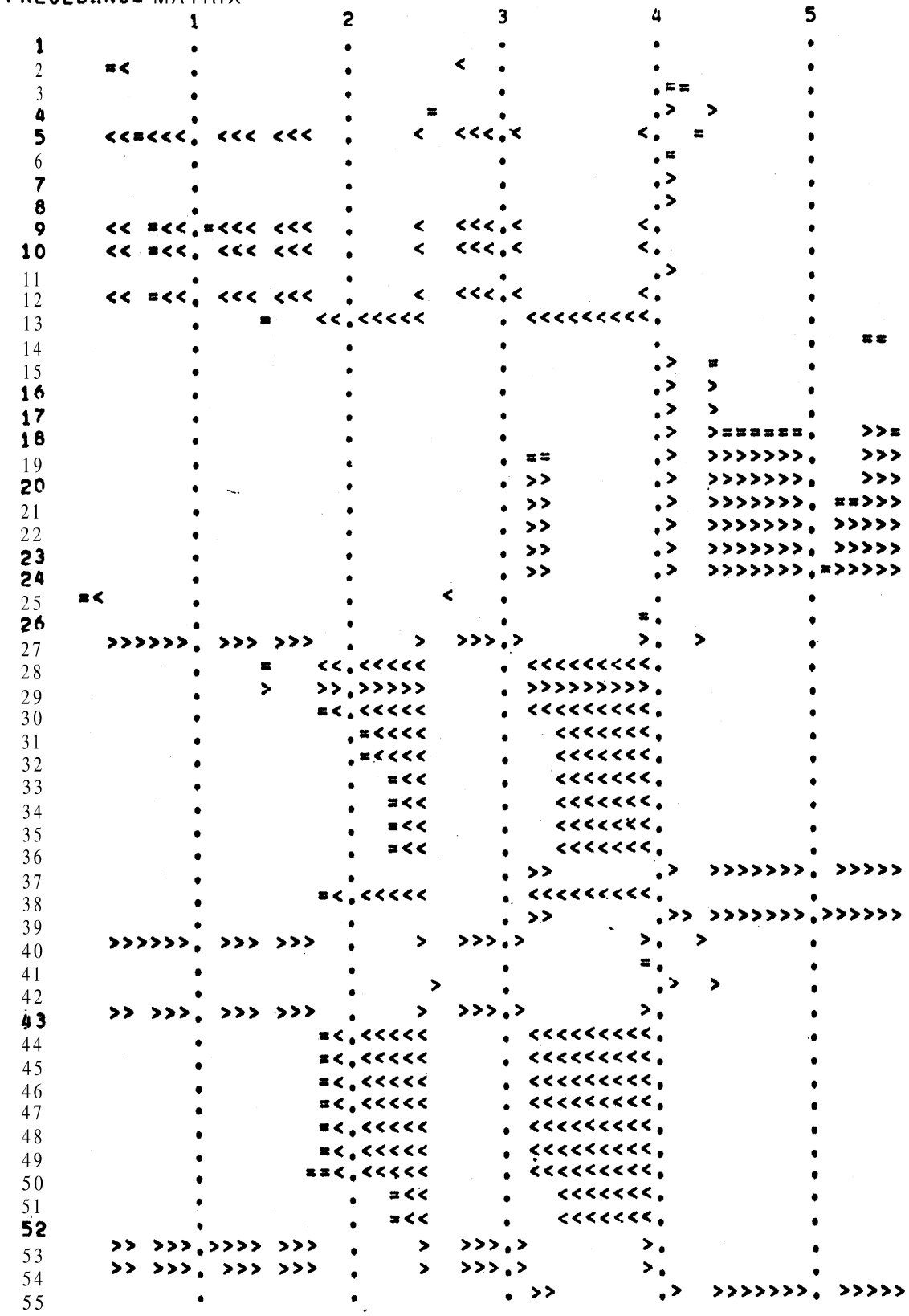
NONBASIC SYMBOLS

1	<PROGRAM>	2	<HEADING>	3	<DECLAR>	4	<COMP STAT>	5	<COMP ST H>
6	<STAT>	7	<STAT*>	8	<COND STAT>	9	<IF CLAUSE>	10	<TRUE PART>
11	<ITER STAT>	12	<WHILE CL>	13	<WHILE HD>	14	<RELATION>	15	<SIM STAT,
16	<OUT STAT>	17	<ASS STAT>	18	<EXPR>	19	<EXPR*>	20	<TERM>
21	<TERM*>	22	<FACTOR>	23	<PRIMARY>	24	<VARIABLE>		

BASIC SYMBOLS

25	$	26	NEW	27	BEGIN	28	IF	29	WHILE
30	OUT	31	+	32	-	33	EXP	34	ABS
35	REAL	36	IM	37	<NUMBER>	38	(	39	<ID>
40	,	41	.	42	END	43	ELSE	44	<
45	≤	46	=	47	≠	48	≥	49	>
50	←	51	×	52	/	53	THEN	54	DO
55	)								

PRECEDENCE MATRIX



PRECEDENCE FUNCTIONS

1	<PROGRAM>	1	1
2	<HEADING>	3	1
3	<DECLAR>	1	2
4	<COMP STAT>	3	3
5	<COMP ST H>	1	4
6	<STAT>	1	1
7	<STAT*>	2	2
8	<COND STAT>	7	3
9	<IF CLAUSE>	2	3
10	<TRUE PART>	2	2
11	<ITER STAT>	3	3
12	<WHILE CL>	2	3
13	<WHILE HD>	1	3
14	<RELATION>	1	1
15	<SIM STAT>	2	3
16	<OUT STAT>	3	3
17	<ASS STAT>	3	3
18	<EXPR>	3	3
19	<EXPR*>	4	4
20	<TERM>	5	4
21	<TERM*>	5	5
22	<FACTOR>	6	5
23	<PRIMARY>	6	6
24	<VARIABLE>	6	6
25	$	1	3
26	NEW	4	2
27	BEGIN	7	4
28	IF	1	3
29	WHILE	7	3
30	OUT	3	3
31	+	4	4
32	-	4	4
33	EXP	5	6
34	ABS	5	6
35	REAL	5	6
36	IM	5	6
37	<NUMBER>	6	6
38	(	3	6
39	<ID>	7	6
40	)	7	1
41	,	6	1
42	END	4	1
43	ELSE	7	2
44	<	3	3
45	≤	3	3
46	=	3	3
47	≠	3	3
48	≥	3	3
49	>	3	3
50	←	3	6
51	×	5	5
52	/	5	5
53	THEN	7	1
54	DO	7	1
55	)	6	3

```
BEGIN COMMENT COMPILES CS?360.  SPRING 1966. N.WIRTH;
   INTEGER LENGTH;        COMMENT LENGTH Of THE PROGRAM COMPILED:
   INTEGER ARRAY PROGRAM[0:255];        COMMENT PROGRAM STORE:)
   REAL ARRAY DATAR, DATAC [0:255];     COMMENT DATA STOR: REAL/IM PART;
   DEFINE ONE = [32:4]#, TWO = [36:4]#, T H R E E = [40:4]#, FOUR =[44:4]#;
   DEFINE ADR = [40:8]#;
   LABEL ALLTHRU;


BEGIN COMMENT THIS BLOCK IS THE TRANSLATOR;
   INTEGER I,J,K,L;     COMMENT I N D I C E S USED BY SYNTAX-ANALYSER;
   INTEGER LOC;         COMMENT INDEX OF DATA-STORE;
   INTEGER PLOC;        COMMENT INDEX Of PROGRAM-STORE;
   INTEGER NX;          COMMENT INDEX O   f NAME LIST;
   INTEGER CHAR;        COMMENT LAST CHARACTER READ RY "NEXTCHAR";
   BOOLEAN LORD;        COMMENT "CHAR" IS A LETTER OR A DIGIT;
   INTEGER WC,CC;       COMMENT WORD- AND CHAR-COUNTER ON INPUT-BUFFER;
   INTEGER SYMBOL,SYMBOLVALUE;  COMMENT LAST SYMBOL READ BY "INSYMBOL";
   INTEGER R;           COMMENT REGISTER NO. LAST USED BY CODE;
   ARRAY BUFFER[0:14]; COMMENT INPUT BUFFER;
   INTEGER ARRAY WORDDELIMITER, DELIMITERNUMBER[0:12];
    NTEGER ARRAY OPERATOR, OPCODE[0:15];
   INTEGER ARRAY F,G[0:55];     COMMENT PRIORITY FUNCTIONS OF SYMBOLS)
   INTEGER ARRAY KEY[0:55];     COMMENT KEY INDEX TO PRODUCTION TABLE;
   INTEGER ARRAY PRTB[0:205];   COMMENT PRODUCTION TABLE;
   INTEGER ARRAY S,V [0:49];    COMMENT SYMBOL- AND VALUE-STACKS)
   INTEGER ARRAY NAME, LOCATION[0:99];
   DEFINE ENDFILE = 25 #;

STREAM PROCEDURE CLEAR ( D);
   BEGIN      DI ← D; 15(DS ← 8 LIT "")
   END ;

BOOLEAN STREAM PROCEDURE ALFA(S,N,D); VALUE N;
   BEGIN  TALLY  ← 1; SI ← S; SI  ← SI+N; 01  ← D; DI ← DI+7;
      IF SC = ALPHA THEN ALFA ← TALLY; n   s ← CHR
   END ;

PROCEDURE ERROR (N); VALUE N; INTEGER N;
   COMMENT MARK POSITION OF INPUT POINTER AND PRINT E R R O R MESSAGE.
      NO ATTEMPT TO CONTINUE COMPILATION IS MADE;
   BEGIN INTEGER K,M;
      SWITCH FORMAT MESSAGE ←
         ("SYNTACTIC ERROR IN PROGRAM"),
         ("ILLEGAL CHARACTER TN PROGRAM"),
         ("UNDECLARED IDENTIFIER"),
         ("TOO MANY REGISTERS REQUIRED"),
         ("PROGRAM IS TOO LONG"),
         ("TOO YANY VARIABLES OR CONSTANTS");
      M ← WC×8 +    CC:
      WRITE(<80A1>, F O R K ← 1 STEP 1 UNTIL M DO " ", "←");
      WRITE(MESSAGE[N]);
      G O TO ALLTHRU
   END ERROR;
```

```
PROCEDURE NEXTCHAR;
   COMMENT ASSIGNS THE NEXT CHARACTER IN THE SOURCE STRING TO "CHAR",
      A S S I G N S "TRUE" TO "LORD", IF THE CHARACTER I S A LETTER A R DIGIT;
   BEGIN IF CC = 7 THEN
      BEGIN IF WC = 8 THEN
         BEGIN READ (CARDFIL, 10, BUFFER[*]); W   C ← 0;
               WRITE (PRINFIL, 15, BUFFER[*])
         END ELSE
         WC ← WC+1;
         CC ← 0
      END ELSE
      c c ← CC+1;
      LORD ← ALFA(BUFFER[WC], C C , C H A R )
   END ;

PROCEDURE READNUMBER;
   COMMENT READS A COMPLEX NUMBER AND ALLOCATES IT IN THE DATA STORE.
      "SYMBOL VALUE" I S ASSIGNED ITS I N D E X I N THE DATA STORE;
   BEGIN OWN REAL M,N; OWN I N T E G E R I; BOOLEAN SIGN;
   PROCEDURE READINTEGER;
      WHILE CHAR < 10 DO
      BEGIN N ← N×10 + CHAR; I ← I-1; NEXTCHAR
      END ;

      M ← N ← 0; I ← 0;
      READINTEGER; M ← N;
      IF CHAR = "." T H E N
      BEGIN N ← 0; I ← 0; NEXTCHAR; READINTEGER; M ← 10*I×N+M
      END ;
      DATAR[LOC] ← M;
      IF C H A R = "I" THEN
      BEGIN M ← N ← 0; I ← 0; NEXTCHAR;
         SIGN ← C H A R = "-"; IF SIGN THEN NEXTCHAR;
         READINTEGER; M ← N;
         I F CHAR = "." THEN
         BEGIN N ← 0; I ← 0; NEXTCHAR; READINTEGER; M ← 10*I×N+M
         END ;
         DATAC[LOC] ← IF S I G N THEN -M ELSE M;
      END ELSE
      DATAC[LOC] ← 0;
      SYMBOLVALUE ← LOC; LOC ← LOC+1
   END READNUMBER ;

PROCEDURE INSYMBOL;
   COMMENT A S S I G N S T H E NUMERIC CODE OF THE NEXT SYMBOL IN THE SOURCE
      S T R I N G TO "SYMBOL". IDENTIFIERS AND NUMBERS ARE CONSIDERED AS
      SYMBOLS, AND ARE NOT FURTHER DECOMPOSED BY THE SYNTAX;
   BEGIN INTEGER I,T; LABEL EXIT;
      WHILE CHAR = " " DO NEXTCHAR;
      IF C H A R < 10 THEN
      B E G I N READNUMBER; SYMBOL ← 37
      END ELSE
      IF LORD THEN
      B E G I N T ← CHAR; NEXTCHAR;
         WHILE      LORD DO
```

```
        BEGIN T ← C H A R & T[12:18:30]; NEXTCHAR
        END;
        FOR I ← 0 STEP 1 UNTIL 12 DO
        IF WORDDELIMITER[I] = T THEN
        BEGIN SYMBOL ← DELIMITERNUMBER[I]; G O TO EXIT
        END;
        SYMBOLVALUE ← T; SYMBOL ← 39; C O M M E N T IDENTIFIER;
     END ELSE
     BEGIN FOR I ← 0 STEP 1 UNTIL 15 DO
        IF OPERATOR[I] = C H A R THEN
        BEGIN SYMBOL ← OPCODE[I]; NEXTCHAR; GO TO EXIT
        END ;
        ERROR(1)
     END ;
  EXIT:
  E N D INSYMBOL;

PROCEDURE EDITX (CLASS,REG,ADDR);
  VALUE CLASS,REG,ADDR; INTEGER CLASS,REG,ADDR;
  BEGIN INTEGER COM; COM ← 0;
     COM.ONE ← CLASS; COM.TWO ← REG; COM.ADR ← ADDR;
     PROGRAM[PLOC] ← COM; PLOC ← PLOC+1; IF PLOC > 255 THEN ERROR(4)
  END ;

PROCEDURE E D I T 3 (OP,R1,R2);
  VALUE OP,R1,R2; INTEGER OP,R1,R2;
  BEGIN INTEGER COM; COM ← 0;
     COM.ONE ← 3 ; COM.TWO ← OP; COM.THREE ← R1; COM.FOUR ← R2;
     PROGRAM [PLOC] ← COM; PLOC ← PLOC+1; I F PLOC > 255 THEN ERROR(4)
  END ;

PROCEDURE FIXUP (WHERE,WHAT);
  VALUE WHERE,WHAT; INTEGER WHERE,WHAT;
  PROGRAM[WHERE].ADR ← WHAT;

PROCEDURE INTERPRET(N); VALUE N; INTEGER N;
  COMMENT EXECUTES T H E N"TH INTERPRETATION RULE;
  CASE N OF BEGIN
  ;
  ;
  BEGIN NAME[NX] ← V[I]; NX ← NX+1; LOC ← LOC+1 END ;
  BEGIN NAME[NX] ← V[I]; NX ← NX+1; L O C ← LOC+1 END ;
  ;
  ;
  ;
  ;
  ;
  ;
  ;
  FIXUP (V[J],PLOC);
  BEGIN FIXUP (V[J],V[J+1]+1); FIXUP (V[J+1],PLOC) E N D;
  BEGIN V[J] ← PLOC; EDITX (2,0,0) E N D;
  BEGIN V[J] ← PLOC; EDITX (2,1,0) E N D;
  BEGIN EDITX(2,1,V[J].[16:16]); FIXUP (V[J].[32:16],PLOC) END ;
  BEGIN V[J].[32:16] ← PLOC; EDITX (2,0,0) END ;
```

```
V[J].[16:16] ← PLOC;
BEGIN EDIT3 (5,R-1,R); R ← R-2 END;
BEGIN EDIT3 (6,R-1,R); R ← R-2 END;
BEGIN EDIT3 (7,R-1,R); R ← R-2 END;
BEGIN EDIT3 (8,R-1,R); R ← R-2 END;
BEGIN EDIT3 (9,R-1,R); R ← R-2 END;
BEGIN EDIT3 (10,R-1,R); R ← R-2 END;
;
;
;
BEGIN EDIT3 (15,R,0); R ← H-1 END ;
BEGIN EDITX (1,R,V[J]); R ← Q-1 END;
EDITX (1,R+1,V[J]);

;
BEGIN EDIT3 (0,R-1,R); R ← R-1 END;
BEGIN EDIT3 (1,R-1,R); R ← R-1 END;
;
EDIT3 (12,R,0);
;
;
BEGIN EDIT3 (2,R-1,R); R ← R-1 END;
BEGIN EDIT3 (3,R-1,R); R ← R-1 END;
I
EDIT3 (4,R,0);
EDIT3 (11,R,0);
EDIT3 (13,R,0);
EDIT3 (14,R,0);
BEGIN R ← R+1; IF R > 15 THEN ERROR(3);
   EDITX (0,R,V[J]);
END;
BEGIN R ← R+1; IF R > 15 THEN ERROR(3);
   EDITX (0,R,V[J]);
END;
;
BEGIN INTEGER K,ID; K ← NX; ID ← V[J];
   WHILE NAME[K] ≠ ID DO
   BEGIN IF K = 0 THEN ERROR(2) ELSE K ← K-1
   END;
   V[J] ← K
END
END INTERPRET;

COMMENT INITIALIZE THE TABLES AND READ THE FIST SYMBOL)
FILL WORDDELIMITER[*] WITH
   "IF","THEN","ELSE","WHILE","DO","ABS","OUT","REAL","IM","EXP",
   "BEGIN","END","NEW";
FILL DELIMITERNUMBER[*] WITH
   28,53,43,29,54,34,30,35,36,33,27,42,26;
FILL OPERATOR[*] WITH
   "←", "+", "-", "×", "/", ";", "<", "≤", "=", "≠", "≥", ">",
   "(", ")", ",", "$";
FILL OPCODE[*] WITH
   50,31,32,51,52,40,44,45,46,47,48,49,38,55,41,25;
FILL F[*] WITH
```

```
    0 . . 1, 3, 1, 3, 1, 1, 2, 2, 2, 2, 2,
    2, 1, 1, 2, 3, 3, 3, 4, 5, 5, 6, 6,
    6, 1, 6, 7, 1, 7, 3, 4, 4, 5, 5, 5,
    5, 6, 3, 7, 7, 6, 4, 7, 3, 3, 3, 3,
    3, 3, 5, 5, 7, 7, 6;
        3,
FILL G[*]1,WITH 1, 2, 3, 4, 1, 2, 3, 3, 2, 3,
    3, 3, 1, 3, 3, 3, 3, 4, 4, 5, 5, 6,
    6, 3, 2, 4, 3, 3, 3, 4, 4, 6, 6, 6,
    6, 6, 6, 6, 1, 1, 1, 2, 3, 3, 3, 3,
    3, 3, 6, 5, 5, 1, 1, 3;
FILL KEY[*] WITH
    0, 1, 2, 3, 11, 14, 22, 23, 26, 29, 37, 38,
   41, 45, 50, 51, 57, 60, 63, 88, 99, 102, 113, 116,
  119, 130, 136, 140, 143, 148, 151, 155, 159, 163, 167, 171,
  175, 179, 182, 187, 190, 191, 193, 193, 194, 195, 196, 197,
  198, 199, 200, 201, 202, 203, 204, 205;
FILL PRTB[*] WITH
    0, 0, Or 40, -2, 2, 41, 39, -4, 3, 0, -26,
   15, Or 43, -5, 4, 6, 40, -7, 5, 0, 0, -8,
    6, 0, -10, 7, 0, 7, -12, 8, 10, 7, -13, 8,
    0, 0, -11, 7, 0, 7, -16, 11, 0, 14, 54, -17,
   12, 0, 0, -9, 7, 43, -15, 10, 0, -27, 15, Or
  -25, 15, 0, 44, 18, -19, 14, 45, 18, -20, 14, 46,
   18, -21, 14, 47, 18, -22, 14, 48, 18, -23, 14, 49,
   18, 24, 14, 0, -31, 18, 31, 20, -33, 19, 32, 20,
  -34, 19, 0, -32, 19, 0, -37, 20, 51, 22, -39, 21,
   52, 22, -40, 21, 0, -38, 21, 0, -41, 22, 0, 50,
   18, -29, 17, 50, 17, -30, 17, -47, 33, 0, 2, 4,
   25, -1, 1, 0, 39, -3, 3, 0, -6, 5, 0, 14,
  538 -14, 9, o, -18, 13, 0, 18, -28, 16, 0, 20,
  -35, 19, Or 200 -36, 19, 0, 23, -42, 22, 0, 22,
  -43, 22, 0, 32, -44, 22, 0, 22, -45, 22, 0, -46,
   23, 0, 18, 55, -48, 23, Or -49, 24, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0;
CLEAR(BUFFER[0]);
CC t 7; WC t 8; NEXTCHAR; INSYMBOL;
J ← 1; S[1] ← ENDFILE;
NX ← LOC t PLOC ← 0; R ← -1;

COMMENT ALGORITHM FOR PRECEDENCE S Y N T A X ANALYSIS;
WHILE SYMBOL ≠ ENDFILE DO
BEGIN I←J ← J+1; S[J] ← SYMBOL; V[J] ← SYMBOLVALUE; INSYMBOL;
   WHILE F[S[J]] > G[SYMBOL] DO
   BEGIN WHILE F[S[J-1]] = G[S[J]] AND J > 1 DO J ← J-1;
      COMMENT S[J]... S[I] IS THE REDUCIBLE STRING, NOW FIND
         THE CORRESPONDING LEFTPART FROM T H E PRODUCTION TABLE;
      L ← KEY[S[J]];
      W H I L E PRTB[L] ≠ 0 DO
      B E G I N K ← J+1;
         WHILE K≤I AND S[K]=PRTB[L] DO
         BEGIN K ← K+1; L ← L+1
         END ;
         IF K>I AND PRTB[L]<0 T H E N
```

```
            BEGIN INTERPRET(-PRTB[L]);S[J]←PRTB[L+1];L←0
            END ELSE
            BEGIN WHILE PRTB[L]>000 L←L+1;L←L+2
            END
         END ;
         IF L≠0 THEN ERROR(0);
         I← J
      END
   END ;
   LENGTH←PLOC;EDITX(4,0,0);
END COMPILER;


BEGIN COMMENT LIST THE COMPILED PROGRAM USING SYMBOLIC CODES;
   INTEGER K; ARRAY MNEMONIC [0:15];
   FILL MNEMONIC[*] WITH
      "  ADD ","  SUB ","  MUL ","  DIV "0"  EXP ",
      "  LSs ","  LEQ ","  EQL ","  NEQ ","  GEQ ","  GTR ",
      "  ABS ","  NEG ","  REAL","  IMAG","  OUT ";

   WRITE(<//"COMPILED CODE:"/>);
   FOR K←0 STEP 1 UNTIL LENGTH DO
   CASE PROGRAM[K].ONE+1 OF BEGIN
      WRITE (<I8,"  LOAD",I4,",",I3>,
            K, PROGRAM[K].TWO, PROGRAM[K].ADR);
      WRITE(<I8,"  STOR",I4,",",I3>,
            K , PROGRAM[K].TWO, PROGRAM[K].ADR);
      WRITE (<I8,A6,I8>, K, IF BOOLEAN (PROGRAM[K].TWO) THEN
            " JUMP" ELSE "  IFJP", PROGRAM[K].ADR);
      WRITE(<I8,A6,I4,",",",I3>,K, MNEMONIC [PROGRAM[K].TWO],
            PROGRAM[K].THREE, PROGRAM[K].FOUR);
      WRITE(<I8,"  HALT">, K)
   END ;
END LISTER ;


BEGIN COMMENT THIS BLOCK I   S THE INTERPRETER;
   BOOLEAN TOGGLE;
   INTEGER IR,IC;        COMMENT INSTRUCTION REGISTER AND-COUNTER;
   REAL ARRAY REGR, REGC[0:15];          COMMENT REGISTERS: REAL/IMPART;
   LABEL CYCLE, FINIS;

REAL PROCEDURE ABSV( I ); VALUE I; INTEGER I;
   BEGIN REAL X,Y; X←REGR[I]; Y← REGC[I];
      ABSV ← IF Y = 0 THEN ABS(X) ELSE
            IF X = 0 THEN ABS(Y) ELSE SQRT(X*2+Y*2)
   END ABSV;

   WRITE(<//"EXECUTION"/>);
   IC ← 0;
CYCLE:
   IR ← PROGRAM[IC]; IC ← IC+1;
   CASE IR.ONE +1 OF BEGIN
      BEGIN REGR[IR.TWO]←DATAR[IR.ADR];
            REGC[IR.TWO] ← DATAC[IR.ADR];
```

```
        END ;
        BEGIN DATAR[IR.ADR] ← REGR[IR.TWO];
              DATAC[IR.ADR] ← REGC[IR.TWO];
        END ;
        BEGIN IF BOOLEAN (IR.TWO) OR TOGGLE THEN IC ← IR.ADR
        END ;
        BEGIN INTEGER M,N; M ← IR.THREE; N ← IR.FOUR;
           CASE IR.TWO +1 OF BEGIN
              BEGIN REGR[M] ← REGR[M]+REGR[N]; REGC[M] ← REGC[M]+REGC[N]
              END ;
              BEGIN REGR[M] ← REGR[M]-REGR[N]; REGC[M] ← REGC[M]-REGC[N]
              END ;
              BEGIN REAL X; X ← REGR[M]×REGR[N] - REGC[M]×REGC[N];
                 REGC[M] ← REGR[M]×REGC[N] + REGC[M]×REGR[N]; REGR[M] ← X
              END ;
              BEGIN REAL D,X; D ← REGR[N]↑2 + REGC[N]↑2;
                 X ← (REGR[M]×REGR[N] + REGC[M]×REGC[N]) / D;
                 REGC[M] ← (REGC[M]×REGR[N] - REGR[M]×REGC[N]) / D;
                 REGR[M] ← X
              END ;
              BEGIN REAL X,Y;
                 X ← EXP(REGR[M]); Y ← REGC[M];
                 REGR[M] ← COS(Y)×X; REGC[M] ← SIN(Y)×X;
              END ;
              TOGGLE ← ABSV(M) ≥ ABSV(N);
              TOGGLE ← ABSV(M) > ABSV(N);
              TOGGLE ← ABSV(M) ≠ ABSV(N);
              TOGGLE ← ABSV(M) = ABSV(N);
              TOGGLE ← ABSV(M) < ABSV(N);
              TOGGLE ← ABSV(M) ≤ ABSV(N);
              BEGIN REGR[M] ← ABSV(M); REGC[M] ← 0 END ;
              BEGIN REGR[M] ← -REGR[M]; REGC[M] ← -REGC[M] END ;
              REGC[M] ← 0;
              REGR[M] ← 0;
              WRITE (<F20.10," I",E18.10>, REGR[M], REGC[M]);
           END
        END ;
     GO TO FINIS
     END ;
     GO TO CYCLE;
FINIS:
END COMPUTER ;

ALLTHRU:
END .
```

```
NEW A,B,C,D;
BEGIN A ← 5I5; B ← -3I-8,5; C ← A×(B+A)-2,5; OUT C;
   OUT CABS B = REAL A + IM B);
   IF A > 0 THEN OUT -A ELSE OUT A;
   A ← 0; D ← 0I0,7853981634;
   WHILE A < 10 DO BEGIN OUT EXP Al A ← A+D; END ;
END $
```

COMPILED CODE:

```
 0  LOAD  0,  4
 1  STOR  or  0
 2  LOAD  0,  5
 3  NEG   0,  0
 4  STOR  or  1
 5  LOAD  0,  0
 6  LOAD  1,  1
 7  LOAD  2,  0
 8  ADO   1,  2
 9  MUL   or  1
10  LOAO  1,  6
11  SUB   08  1
12  STOR  0,  2
13  LOAD  0,  2
14  OUT   0,  0
15  LOAD  0,  1
16  ABS   0,  0
17  LOAD  1,  0
18  REAL  1,  0
19  SUB   0,  1
20  LOAD  1,  1
21  IMAG  1,  0
22  ADD   or  1
23  OUT   0,  0
24  LOAD  0,  0
25  LOAD  1,  7
26  GTR   0,  1
27  IFJP      32
28  LOAD  0,  0
29  NEG   0,  0
30  OUT   or  0
31  JUMP      34
32  LOAD  08  0
33  OUT   0,  0
34  LOAD  0,  8
35  STOR  0,  0
36  LOAD  0,  9
37  STOR  0,  3
38  LOAD  Or  0
39  LOAD  1,  10
40  LSS   0,  1
41  IFJP      SO
42  LOAD  0,  0
43  EXP   0,  0
```

```
 44   OUT     0,  0
 45   LOAD    0,  0
 46   LOAD    1,  3
 47   ADD     0,  1
 48   STOR    0,  0
 49   JUMP       38
 50   HALT
```

EXECUTION

```
-6.0000000000@+01 I  7.7500000000@+01
 4.0138781887@+00 |  8.5000000000@+00
-5.0000000000@+00 I -5.0000000000@+00
 1.0000000000@+00 I  0.0000000000@+00
 7.0710678119@-01 I  7.0710678119@-01
-1.4551915228@-11 I  1.0000000000@+00
-7.0710678120@-01 I  7.0710678118@-01
-1.0000000000@+00 I -1.4551915228@-11
-7.0710678118@-01 I -7.0710678120@-01
 1.4551915228@-11 I -1.0000000000@+00
 7.0710678120@-01 I -7.0710678118@-01
 1.0000000000@+00 |  1.4551915228@-11
 7.0710678116@-01 I  7.0710678120@-01
 0.0000000000@+00 |  1.0000000000@+00
-7.0710678125@-01 I  7.0710678116@-01
-1.0000000000@+00 |  0.0000000000@+00
```

```
NEW A; BEGIN A ← 1;
    WHILE A < 5 DO
    BEGIN A ← A × 110.2; OUT A; END ;
END $
```

COMPILED CODE:

```
 0   LOAD    0,  1
 1   STOR    0,  0
 2   LOAD    0,  0
 3   LOAD    1,  2
 4   LSS     0,  1
 5   IFJP       13
 6   LOAD    0,  0
 7   LOAD    1,  3
 8   MUL     0,  1
 9   STOR    0,  0
10   LOAD    0,  0
11   OUT     0,  0
12   JUMP        2
13   HALT
```

EXECUTION