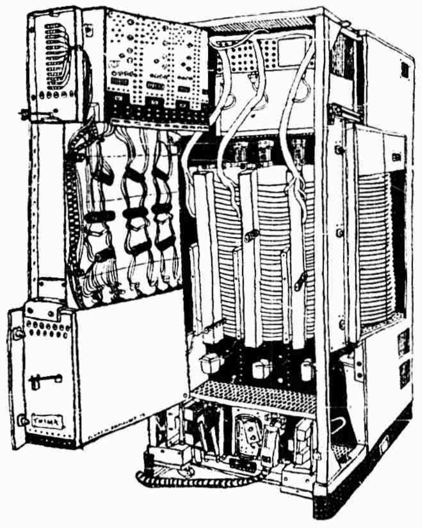
AD 240512

10

Carnegie Institute of Technology

Pittsburgh 13, Pennsylvania

Computation Center



COMPUTATION CENTER

CARNEGIE INSTITUTE OF TECHNOLOGY

SCHENLEY PARK

PITTSBURGH 13, PENNSYLVANIA

ON THE SYNTAX MACHINE AND THE CONSTRUCTION OF A UNIVERSAL COMPILER

by A. E. Glennie

Technical Report No. 2
July 10, 1960

PREPARED UNDER CONTRACT Nonr-760(18)
(NR 049-141)
for
OFFICE OF NAVAL RESEARCH

TABLE OF CONTENTS

		page
PART 1		1
1.2	A Notation for Syntax	3
1.3	Syntax Notation as Program. The Syntax Machine	5
1.4	Flow Diagrams for the Syntax Machine	8
1.5	Recursive Programs for the Syntax Machine	9
1.6	The Algorithmic Form of the Syntax Formalism	10
1.7	The Syntax Machine with Output	12
1.8	The Syntax of the Output	13
1.9	The Algebra of the Algorithmic Syntax	15
1.10	Examples	18
PART 2	Applications of the Syntax Machine	21
2.1	Example 1. Addition and Subtraction of Floating-Point Numbers	22
2.2	Example 2. Extension of Example 1 to Include Storage Operations	24
2.3	Example 3. Arithmetic expressions using +, -, *, /, and Parentheses	25
2.4	Example 4. Assignment Statements using the Expressions of Ex.3	31
2.5	Example 5. Simple Relational Expressions	33
2.6	Combinations of Relations	38
2.7	Example 7. Simple Branching Instructions	42
2.8	Example 8. Iteration Statements	43
PART 3		
3.1	Declarations	46
3.2	Declarations about Syntactic Properties	47
3.3	Substitution Declarations	49
3.4	Declarations about Macro-instructions	52

Table of contents continued on next page

TABLE OF CONTENTS -- continued

PART 4	The Assembler	57
4.1	The Operator, (D:a:1)	58
4.2	The Operator, (D:c:0)	58
4.3	The Operator, (D:d:0)	59
4.4	The Labeling Operator, (O:x:1)	59
4.5	The Operator, (O:v:r)	60
4.6	The Operator, (O:w:2)	61
4.7	The Operator, (O:y:2)	61
4.8	The Operator, (O:e:0)	62

Introduction

To deal with the problem of many problem-oriented languages to be translated to many machine languages, three main lines of attack have been suggested.

(1) That the multiplicity of problem-oriented languages be reduced by the adoption of a universal-algorithmic language, e.g., ALGOL. This legislative manner of abolishing the difficulty does not seem to be a complete solution: such languages as have been proposed lack universality in varying ways. For example ALGOL has no provision for the processing of strings of symbols. In addition, it is not at all clear that present ideas of what constitutes a universal language will be valid in a future with time-sharing and even perhaps self-organising computers.

(2) That a common machine-oriented language be devised. This language (UNCOL for short) is thought of as an intermediary language through which translation will be made. Each problem-oriented language is to be translated to UNCOL by a translator that can be written in UNCOL. An UNCOL to machine-language translation completes the process.

(3) That translators be so constructed that they accept the description of a source language and are thereby converted into translators for that language. For each machine, only one such translator need be built.

This report follows the third approach.

In order to give a degree of universality to a compiler, two things must be done. First, there must be some method of describing the source language; and, second, there must be some way of describing the properties of the machine for which translation is made. In great measure, the first problem was solved by the introduction by Backus of a notation for describing the syntax of ALGOL [1/]. This notation is related to similar notations in linguistics (phrase-structure grammar in substitution form [2/]) and in logic (productions [3/]). The second problem is one of considerable difficulty. Although it is possible to describe the properties of a computing machine, as is done in any reference manual, such descriptions are not in a form which is simple to manipulate mechanically. This report proposes an alternative. - that the description of the source language should not be made independently of the target language but should exploit any properties of the target language that are useful. For example, if the machine has the ability in one instruction to add the absolute value of a number, the source language should be described with that operation as one of its primitives, rather than the two primitives of addition and taking the absolute value.

This report is divided into four sections. The first section proposes a mechanism for scanning a linear text, and performing a syntactic analysis. A pseudo-machine, the Syntax Machine is described, whose programs may be considered to define the language of the text. The output from the Syntax Machine is a string whose evaluation leads to a (partial) translation of the source text.

1/ J.W. Backus et. al.''Report on the Algorithmic Language ALGOL 60''; Communications ACM 3 p.299, May 1960.

2/ N. Chomsky ''Three Models for the Description of Language.'' Tr. IRE, IT-2; No. 3. p.113; Sept, 1956.

3/ M. Davis Computability and Unsolvability, Ch.6; McGraw-Hill, New York; 1958.

The second part of the report discusses, mainly by examples, the application of the Syntax Machine to translation for a particular target-machine language, and shows how the syntax description may be written to exploit the special features of the target machine.

The third sections considers the role of Declarations in the source language and the mechanisms required to effect them in the translation process. The fourth section deals with a supplementary process of assembly which is required to evaluate the strings produced by the Syntax Machine.

1.2. A Notation for Syntax

The notation to be presented is similar to that of Backus, but with an important difference. Whereas the notation of Backus enables texts conformable with the rules of syntax to be derived by substitutions, the present notation is used to express a decision procedure that tests whether an example of text conforms to the rules.

The decision procedure tests the legality of a string by applying one of three types of tests to the string. Let us denote syntactic variables by enclosing the name of the variable within the brackets < > , and denote syntactic constants (i.e., characters of the alphabet) by themselves. The three types of test and their notation are:

(1) Is the string a value of a syntactic variable which is the concatenate of other syntactic variables or constants?

This is expressed by the formula

< A > ::= < B > < C > < D > ... < X >

where juxtaposition in the formula signifies concatenation in the string tested, and the sign ::= means that the variable on the left is defined by the expression on the right.

(2) Is the string a value of a syntactic variable defined as being an alternative of several variables?

$$< A > ::= < B > \mid < C > \mid < D > \mid \quad \ldots \quad \mid < X >$$

where the connective $\mid$ denotes that the variables are alternatives, in the sense that the string is a value of < A > if it is a value of < B >, or of < C > and so on.

(3) Is the string a concatenate of several strings with the last string repeated an indefinite number of times (perhaps none)?

This is expressed by the formula

$$< A > ::= < B > \; < C > \; < D > \; \ldots \; \{< X >\}$$

where $\{ \quad \}$ denotes iterated concatenation, and the definiens has at least one term before the iterated concatenation.

In the foregoing it has been tacitly assumed that tests implied by the right-hand sides of these expressions had been taken in the order of writing. If this is now adopted as a convention of the formalism, then the formulae express algorithms for testing whether strings are values of syntactic variables.

The formulae now have the corresponding interpretations.

(1) The string is an < A > if a head string is found to be a < B > and the head of the remaining part of the string is a < C >, and so on.

(2) The string is an < A > if it is a < B >, or if not that, then a < C >, and so on.

(3) The interpretation is similar to that of the first type, but with the last component iterated.

1.3 Syntax Notation as Program. The Syntax Machine

In this section a pseudo-machine, called the Syntax Machine, will be defined that uses the definitions of the previous sections as programs to decide whether strings are values of syntactic variables.

Consider a machine with an input tape on each consecutive position of which is inscribed one character of a string to be analyzed. The machine obeys program steps of the form F,AT,AF where F specifies the action to be taken, and AT,AF specify the addresses of the next program steps. For each character of the alphabet and for some important subclasses there is a machine instruction of a type called a ''Comparator.'' A Comparator instruction, say for the character ''X'', will read the character presently under the reading head on the input tape. If the character is ''X'', then the input tape is moved by one character position and the next instruction of the program taken from address AT. If the character is not ''X'', then the tape is not moved and the next instruction is taken from address AF. Where the Comparator is for a subset of the characters the action is similar; if the character under the reading head belongs to the subset, the tape is moved and the next instruction is taken from location AT.

In programming for this machine, another type of program step may be used, the Recognizer: it is a subroutine composed out of Comparators and Recognizers. To call a subroutine a special function of the machine, denoted here by S*, AT, AF, is used. Its action is to copy the present position of the input tape on to the current level of the control push-down list, together with the addresses AT, AF, in parallel lists. Then the level of control is increased by 1 and the next program step taken from location S. Two more special instructions provide for exits from subroutines, in case of failure or success of the decision process. These functions, called ''False'' and ''True,'' decrease the level of control by 1 and cause the next program step to be taken from the AF or AT addresses in the control push-down list. In the case of ''False'' the input tape is repositioned to be as it was when the subroutine was entered.

By this means Recognisers can be constructed that act like Comparators, but recognize strings of characters.

With this apparatus it is possible to write programs for the syntax definitions of the previous section.

Examples

1. $< A > ::= \quad a \mid b \mid c$	which recognizes the occurrence of the character a or b or c.
2. $< B > ::= \; < A > \; < A >$	if A is as defined in Ex.1. this recognizes the pairs of characters aa,ab,ac,ba,bb,bc,ca,cb,cc.
3. $< C > ::= \quad x \; \{ y \}$	recognizes x,xy,xyy,xyyy etc.
4. $< I > ::= \; < L > \{ < NL > \}$	recognizes ALGOL identifiers, if L is a recognizer (or comparator) for alphabet letters and NL is a recognizer for letters and numerals.

Programs for these examples may be written in the ''machine'' instruction notation as follows:

	Label	Function	AT	AF
1)	A	C(a)	S4	S1
	S1	C(b)	S4	S2
	S2	C(c)	S4	S3
	S3	False		
	S4	True		
2)	B	A*	S5	S3
	S5	A*	S4	S3
3)	C	C(x)	S6	S3
	S6	C(y)	S6	S4
4)	I	L*	S7	S3
	S7	NL*	S7	S4

Here C(x) denotes the Comparator for x, and similarly.

In this notation, we can write programs for which there is no representation in the algebraic formalism; this will be convenient on occasion. We could define syntax in terms of programs for the syntax machine: this, likewise, may enable us to write some forms of syntax not representable by the algebraic formalism, or if so, only by uneconomical programs.

An additional feature of great power will be to allow subroutines to store bits in a list working in parallel with the push-down list, so that a syntactical property recognized in a subroutine may be tested and cause branching in the routine controlling it. In a binary computer it will be easy to store many bits in the same machine word (usually 30 at least in most binary computers).

Two functions are required: *

(a) M(X) . Copy a bit into bit position X in the k-1 th level of the push-down list: k is the current level of the routine in which M(X) acts. X is specified using the data field of the instruction; the next instruction is taken from the address specified in the AT field.

(b) K(X) . If the pseudo-machine is currently operating on level k, examine the X bit on level k. If it is 1, proceed to the address specified by the AT address; if it is 0 proceed to the address specified by AF.

When a subroutine is entered in level k, from level k-1 the set of bits (or marks as they will sometimes be called) should be set to 0.

---

* There are many ways of doing this. It would be more economical in machine time and storage to allow the M and K functions to set and test many bits. For the simplicity of exposition, we adopt the simplest M and K functions.

1.4 Flow Diagrams for the Syntax Machine

The simplicity of the operations of the syntax machine, makes it possible to write flow diagrams precisely, by the use of the following conventions.

(1) Unless otherwise indicated by arrows, the flow of control is across the page from left to right, or downwards.

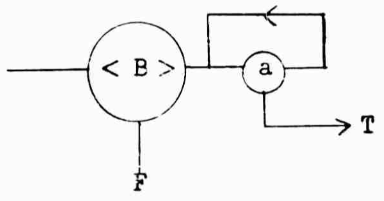
(2) Unless otherwise indicated, true exits from comparators are written horizontally, and false exits vertically.

(3) Comparators are indicated by circles containing the character to be compared: Recognizers are indicated by the name of the recognizer, enclosed in angular brackets.

(4) To indicate the M function that places a mark in the push-down list, write M(X) in the diagram, where X is the mark. For mark comparators, use K(X), with exit conventions as with comparators.

(5) To minimize lines of control, nodes of the flow diagram may be labeled. Recognizer exits will be labeled ''True'' or ''False.''

Example: A ::= < B > $\{$ a $\}$ may be diagramed as



1.5 Recursive Programs for the Syntax Machine

In this section we investigate certain properties of the machine; in particular, we ask for rules for constructing programs that will always provide a decision. An example shows that it is possible for programs to cycle indefinitely. (e.g., single instruction whose AF address is the address of the instruction itself). However, there is one main source of danger in programs using notation of the three standard types, that of the careless use of recursion. The manner of constructing subroutines allows recursive definitions to be used.

Consider the program, $< A > ::= < A > < B >$. In order to test whether the text is a value of $< A >$, the question is asked, ''Is the string of characters starting at this point an example of $< A >$?''

This question is answered if two subsidiary questions are answered in the affirmative. The first question is exactly the same as the original and is asked at exactly the same position of the input tape. However, in the program $< A > ::= < B > < A >$ this circularity does not arise, because the question, ''Is the string an example of $< A >$?'' is never asked twice at the same position of the input string. The tape will have moved because of the application of the program step $< B >$, which must have a successful outcome (and hence the input tape moves) before $< A >$ is applied again. The first example is of a program with an ''infinite loop;'' the second is a finite program, if applied to a text of finite length (and in practice, all texts are finite).

Circularity in programs is not always so easy to discern as in the above example. There is, however, a simple rule whose successive application checks absence of circularity. A program step is non-circular if all program steps in its immediate definition are non-circular when it is defined by a formula of type 2 (i.e., as a set of alternated) or (for formulae of types 1 and 3) if the first step is non-circular. Any program step that is a Comparator is non-circular. For example in the formulae of types 1 and 3

$$\langle A \rangle ::= \langle B \rangle \ \langle C \rangle \ \langle D \rangle \quad \ldots \quad \text{or}$$

$$\langle A \rangle ::= \langle B \rangle \ \langle C \rangle \ \langle D \rangle \quad \ldots \left\{ \langle X \rangle \right\}$$

$\langle A \rangle$ is non-circular if $\langle B \rangle$ is non-circular. In the formula of type 2

$$\langle A \rangle ::= \langle B \rangle \ \Big| \ \langle C \rangle \ \Big| \ \langle D \rangle \quad \ldots$$

$\langle A \rangle$ is non-circular only if $\langle B \rangle$, $\langle C \rangle$, $\langle D \rangle$ ... are all non-circular. *
All steps in a program must finally be non-circular. The proof of this rule follows from the observation that a non-circular program step either exists via the "Fail" exit, or it moves the tape forward.

Recursive definition is permissible subject to this rule.

1.6 The Algorithmic Form of the Syntax Formalism

In this section we explore the difference between the use of the syntax notation to express rules of derivation and rules of string analysis. The discussion of the previous section shows that some forms of recursive definition are invalid as rules of analysis; these forms may be expanded and rearranged into the form

$$\langle A \rangle ::= \langle A \rangle \ \langle B \rangle \ \Big| \ \langle C \rangle$$

which expresses all the possible formulae rendered invalid as rules of analysis. The strings generated by this rule of derivation are of the form CB ... B i.e., those strings which have $n \geq 0$ strings of type B concatenated at the right of a C. The algorithmic form of the definition is $\langle A \rangle ::= \langle C \rangle \left\{ \langle B \rangle \right\}$.
This shows how the invalid recursion may be avoided.

---

* For subprograms written in machine language, the rule is that the program steps that read the heads of strings must be non-circular.

Another difference is in the interpretation of the type 2 formula; in the algorithmic form the order of the terms is important since it is the order in which tests are made. For example, the substitution rule

$< A > ::= \quad b \quad | \quad ba$ generates the two examples ''b'' and ''ba.''

However, if this were taken as an algorithm and applied to the string ''ba'' it would test merely the first character ''b,'' and finding this to be a possible value would accept it, leaving the character ''a'' unscanned.

Consequently the correct algorithmic form would be

$< A > ::= \quad ba \quad | \quad b$

The ordering relation among the alternatives in the definiens of a type 2 formula may be expressed by the rule that if one recognizer A defines strings that are heads of any strings defined by a recognizer B , then B must precede A in the formula. If no ordering is imposed by this rule,then it can be made to minimize cost by testing those strings that are frequent before those that are rare.

Remark

The difference between the two formalisms is that in the case of the algorithmic form a direction of scan is an essential part of the interpretation, whereas in the substitution form no notion of scanning is present. It is suggested that source-language syntax be expressed in algorithmic form to avoid ambiguity; this form may always be interpreted in substitution form (but not vice versa, as we have seen). Two forms of algorithmic syntax are possible, according to the direction of scan; in this note the natural order of scanning, as in reading, is assumed.

1.7 The Syntax Machine with Output

In previous sections we have discussed how to recognize texts that conform to the rules of a syntax; the result produced by the machine has been only an indication of validity.

An output can be generated as follows:

(1) Each comparator instruction (reading a character of the input string) can be modified to write the character on an output tape if it is recognized by the comparator. Such comparators that produce output will be written with underlining. Thus in the recognizer < A > ::= $\underline{a} \mid \underline{b} \mid c$, ''a'' and ''b'' will be written on the output tape but not '' c '' whenever one is recognized by the recognizer < A >.

(2) Whenever a ''True'' return is made from a recognizer there will be the option of writing a pattern of the form (p : q : r) on the current position of the output tape. This pattern may be written in the data portion of the ''True'' return instruction. The elements of this pattern will have the interpretations:

2a. p specifies an instruction or a macro-instruction for subsequent assembly.

2b. q is a type number, specifying the manner in which the pattern (p : q : r) will be treated by an assembler whose input is the present output tape.

2c. r is the number of characters or character groups written on the output tape by the recognizer.

(3) If a recognizer is named by a pattern (p : q : r) the whole output generated by this recognizer will contribute 1 to the character count of any recognizer using it. If a recognizer is not so named, each unit of output generated by it will contribute to the character count of any recognizer using it.

(4) The action of naming will be signified in the algorithmic notation by adding the naming pattern in quotation marks at the end of the corresponding formula. Example:

$$<C> ::= x \quad \{y\} \quad ''(P : Q : 0)''$$

will recognize x, xy, xyy, etc. on the input tape and generate the corresponding patterns on the output tape, viz,

(P:Q:0)

y,(P:Q:1)

y,y,(P:Q:2)

y,y,y,(P:Q:3) and so on.

Note that the naming pattern has r=0 in the program.

In flow diagrams a true return with naming will be indicated by the naming pattern, ''(P:Q:0)''.

(5) When a ''False'' return is made from a recognizer, the output tape is repositioned to the position it had when the subroutine was entered.

1.8 The Syntax of the Output

The language of the output is particularly simple. Its alphabet is formed from the characters of the original alphabet together with the symbols, (P:Q:R) which are written by naming. These latter are ''syntactic operators'' whose operands are either characters of the original alphabet or are expressions formed by syntactic operators.

We define recursively the class of output strings as follows:

1. All characters from the original alphabet are values of syntactic variables.
2. Let V_i denote values of syntactic variables and (Ø;r) denote syntactic operators of order r, $r \geq 0$. Then the expression, V_1, V_2, ... , V_r , (Ø:r) is also a value of a syntactic variable. Examples are

(Ø:0)
V,(Ø:1)
V,V,(Ø:1),(Ø:2)

3. The output string generated by a named recognizer is a value of a syntactic variable.

In the processing of the output string, the values of syntactic variables will be used to construct segments of the target language according to nature of the syntactic operators. The three parts of the syntactic operator have separate purposes. P will be data, Q will tell how the data P and the data from the R operands will be combined. Thus the output string may be viewed as data, with the processing rules combined with it.

The output string is an example of postfix notation, similar to the prefix notation of the logicians, but in reverse order. There is a particularly simple algorithm to evaluate expressions in postfix notation. Let there be a list, the push-down list L, each position of which is capable of holding (directly or by indirect reference) the value of a syntactic variable. Then as the output string is scanned syntactic variables are placed in successive positions of L until a syntactic operator appears. If the syntactic operator is of order r, then its operands are to be found in the current last r positions of L. The result of the evaluation of the expression specified by the operator is then placed in the first of these positions, say position m, and the process continued, with the next syntactic variable being read into position m+1 - or if an operator is next read, its operands will be in the positions M-q+1 through m (q is the order of the operator).

For example, if the string to be processed is V_1, V_2, $(\phi_1 : 1)$, $(\phi_2 : 2)$ the successive configurations of the list L will be

(1) $L_1 = V_1$.

(2) $L_1 = V_1$, $L_2 = V_2$.

(3) $L_1 = V_1$, $L_2 = \phi_1(V_2)$. by application of ϕ_1.

(4) $L_1 = \phi_2(V_1, \phi_1(V_2))$. by application of ϕ_2.

1.9 The Algebra of the Algorithmic Syntax

Let X_i (i= 1, 2 ...) stand in place of the forms a , < A >, $\{< A >\}$, ''(a:b:0)'', i.e., in place of recognizer (or comparator) symbols, iterated recognizer symbols and naming symbols. Then the standard formulae become

$$X_a = X_b \quad X_c \quad X_d \quad \ldots \quad X_z \qquad \text{from type (1) and (3) formulae}$$

$$X_a = X_b \mid X_c \mid X_d \mid \ldots \mid X_y \qquad \text{from the type (2) formulae.}$$

The operations of this algebra are concatenation and $\mid$. It is easily verified that there are no commutative laws, but associative and distributive laws hold, thus

$$X_1X_2 \neq X_2X_1 , \qquad X_1 \mid X_2 \neq X_2 \mid X_1$$

$$(X_1X_2)X_3 = X_1(X_2X_3) , \qquad (X_1 \mid X_2) \mid X_3 = X_1 \mid (X_2 \mid X_3)$$

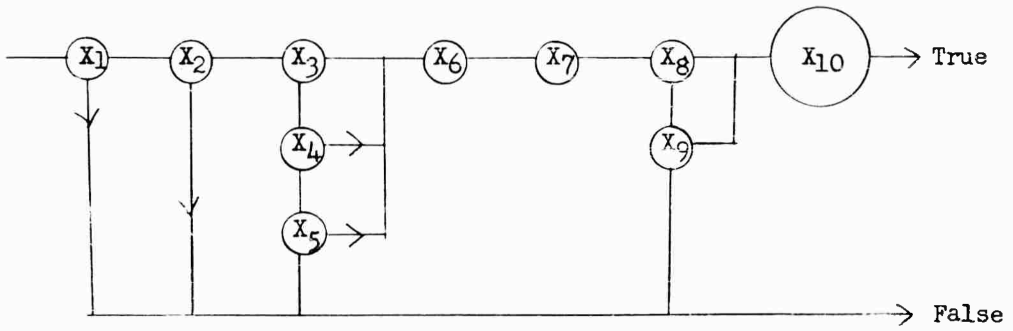
$$(X_1 \mid X_2)\, X_3 = (X_1X_3) \mid (X_2X_3) , \qquad X_1\, (X_2 \mid X_3) = (X_1X_2) \mid (X_1X_3)$$

The distributive laws are important as they provide for possible economization.

One particular form of parenthesized-syntax notation is of importance because (in this case only) the parentheses do not imply an internal subroutine for the bracket. This might be called normal-concatenated form, of which an example is

$$X_1X_2(X_3 \mid X_4 \mid X_5) \quad X_6X_7(X_8 \mid X_9) \quad X_{10}$$

The flow diagram for this is



The general form of the normal concatenated form is $A_1A_2A_3 \ldots A_n$ where the A are either single symbols or are of the form $(B_1 \mid B_2 \mid \ldots \mid B_m)$ where the B are also single symbols.

The other normal form, example $W=X_1 \mid (X_2X_3) \mid X_4 \mid \ldots$ requires all alternates which are concatenates (except for a concatenate in the last position) to be constructed as subroutines. The above form must be programmed as $Z = X_2X_3$, $W = X_1 \mid Z \mid X_4 \ldots$ An exception is made for concatenated pairs, where the second member stands for a naming operation.

All these rules follow from the interpretation of the notation. For example, consider $(X_1X_2) \mid X_3$, where X_2 is not a naming operation. This program tests a string using X_1 ; if this succeeds, the X_2 is applied to the next part of the input string. If X_2 fails, the string must be repositioned so that the alternate test X_3 may be correctly applied. This can be done only by making (X_1X_2) a subroutine (whose False exit will do the repositioning).

Identity and Infinity symbols

The notation may be enriched by the addition of three symbols $\wedge$, $\vee$ and ∞ , corresponding to comparators which have respectively

$\wedge$: no false exit, does not read the input.

$\vee$: no true exit, does not read the input.

∞: no exits at all.

The first two of these symbols are the identity elements for concatenation and alternation. They allow certain transformations to be made in expressions of the notation, according to the rules given at the end of this section. For example,

$$Z=X_1X_2 \mid X_1 = X_1X_2 \mid X_1 \wedge = X_1(X_2 \mid \wedge)$$ by the distribution law.

$X_2 \mid \wedge$ is a recognizer with its false exit joined to its true exit.

The following equations hold

$$\Lambda\, X = X$$

$$X\, \Lambda = X$$

$$V\, X = X$$

$$X \mid V = X$$

$$V\, X = V$$

$$\Lambda \mid X = \Lambda$$

$$\{ V \} = \Lambda$$

$$\{ \Lambda \} = \infty$$

$$\{ X\ V \} = X \mid \Lambda$$

$$\{ X \mid \Lambda \} = \infty$$

$$\{ A \} = (A \mid \Lambda)\ \{ A \}$$

$$\{ \{ A \} \} = \infty$$

1.10 Examples

These examples are for the pseudo-machine with no output. They are comparable with the descriptions of ALGOL 60[1].

(1) Programs in problem-oriented languages are usually written as sequences of statements; there may be several types of statement.

<Program> ::= <Statement> {<Statement>}

<Statement> ::= <Statement 1> | <Statement 2> | <Statement 3>

This states that a program is composed of a sequence of statements, and that there is at least one statement in the sequence. There are 3 types of statement. Each statement type would, of course, be defined in terms of simpler syntactic variables - and in the limit, in terms of the alphabet. The application of <Program> to a string will determine whether the string is an example of a text in the language.

(2) Consider algebraic prefix notation using +, *, / as binary operators and - as an unary operator. Then < E > is the recognizer for the notation, where

< E > ::= < A > | < B > | < C > | < D > | < V >

< A > ::= - < E >

< B > ::= + < E > < E >

< C > ::= * < E > < E >

< D > ::= / < E > < E >

< V > is a recognizer for variables and constants.

This example shows the use of recursive definition, and it is easily

[1] J.W. Backus et.al., "Report on the Algorithmic Language; ALGOL 60, Communications, ACM 3; p.299, May, 1960.

shown to be non-circular. It may be written in normal concatenate form as

```
        < E >  ::=  < A > | < V > | ( + | / | * )  < E >  < E >
  where  < A >  ::=  -  < E >
```

The parentheses are used in this example as characters in the syntax language: it is assumed that they will not occur in the text analyzed. Note that in this example the ordering of the alternates is not important.

(3) Normal Algebraic Notation

We repeat example (2) but now using the more usual infix notation, with the operators as binary connectives.

```
    < E >   ::=  < F >  {< S >}         3.1
    < S >   ::=  < +- >  < T >          3.2
    < F >   ::=  < T > | < S >          3.3
    < +- >  ::=    +  |  -              3.4
    < T >   ::=  < A > | < V >          3.5
    < A >   ::=  < V >  < */ >  < T >   3.6
    < */ >  ::=    *  |  /              3.7
```

The notation may be extended to include parenthetical notation in the text by replacing < V > by < W > in 3.5, 3.6, and adding two more lines.

```
    < W >   ::=  < V > | < (E) >        3.8
  < (E) >   ::=  ( < E > )              3.9
```

Remarks

3.1 says that an algebraic expression is composed of a first part < F >, followed by an indefinite number of subsequent parts < S >, which are additions or subtractions of terms < T >. 3.3 says that the first part is either a signed term < S > or an unsigned term < T >. By 3.5 < T > is either a product-quotient form < A > or merely a single variable < V >; it is important to test < A > before < V >, since < V > occurs as the first element in < A >. Suppose the order of 3.5 had been changed. Then

$$\begin{array}{llll} \langle T \rangle & ::= & \langle V \rangle \mid \langle A \rangle & \\ & ::= & \langle V \rangle \mid \langle V \rangle \ \langle */ \rangle \ \langle T \rangle & \text{from 3.6.} \\ & ::= & \langle V \rangle \left[\Lambda \mid \langle */ \rangle \ \langle T \rangle \right] & \text{by the distribution law.} \\ & ::= & \langle V \rangle & \text{using the laws of the algebra.} \end{array}$$

Clearly something is wrong with this, as was to be expected.

Part 2. Applications of the Syntax Machine

In the first part of this report the syntax machine was defined and is properties discussed. Now we go on to discuss its application, and in so doing we see what are the desirable and necessary properties of an assembly program which can process the output from the syntax machine. The whole translation will be a multi-stage process in which syntax analysis alternates with assembly operations. The assembly operations construct new strings which may then undergo syntactic analysis. How many times this has to be done will depend on the source language. Whether the alternation of syntax analysis and assembly is made over segments of the text or over the whole text depends also on the source language and on the amount of storage that may be available for intermediate strings.

For example, any language that contains declarations will require several alternations between syntax and assembly processes. Consider how names are used for different types of numbers, e.g., fixed-point and floating-point numbers. If the distinction between these classes of numbers is made by a declaration, rather than by properties of the names themselves (e.g., by defining integer variable names to be those that begin with I, J, K) the declarations must be used to form tables of the names of each class. These tables must then be consulted to find the syntactic properties of the objects named, whether they are integer variables, or are functions and so on.

This table lookup feature is not a property of the syntax machine as described; it is proposed that this should be part of the assembly processes. Syntax analysis is, however, usually sufficient to separate names from operator signs, since it is unusual for the syntax of names to change within segments of a program. Thus, the strategy for translation would be

(a) Use the syntax analyser to discover the names and operator signs in segments of the text.

(b) In the output there will be values of syntactic variables corresponding to names, literal constants and other character groups. For example, the name ABC will appear on the output from the syntax analyzer as A, B, C, (Ø:3), where Ø will specify an assembly process, that might replace A, B, C, (Ø:3) by a ''co-ordinate name''I_n, meaning that ABC is the n'th integer-variable name. I_n will be constructed from the position of ABC in the table of integer names, and will be stored as a single character that will be recognized syntactically in a later use of the syntax analyzer as a member of the class I.

(c) The syntax analyzer can then be applied to strings which now consist of operator signs from the original text and co-ordinate names which stand in place of the original names and literal constants.

These semantic considerations shall be deferred to part 3 of this report. They are mentioned here so that it will be possible to use co-ordinate names in this part without implying that these co-ordinate names are written in the original text. We shall also be able to treat words like ''If,'' ''then,'' ''do'' and other such words as single characters of the string analyzed. This will simplify the exposition. We shall therefore, in this part, now ignore the interplay between syntax analysis and assembly.

2.1 <u>Example 1. Addition and Subtraction of Floating-Point Numbers</u>

(a) Source language syntax

$< E > \quad ::= \quad < F > \{< S >\}$ 1.1

$< F > \quad ::= \quad < V_i > \Big| < C_i > \Big| < S >$ 1.2 *

$< S > \quad ::= \quad (+ \Big| -) \ < F >$ 1.3

For example : $V_1 + C_1 - V_2$. 1.4

---

* $< V_i >$, $< C_i >$ are recognizers for floating-point variables and constants.

(b) Syntax Program with annotations

$< E > ::= < F > \{< S >\}$ ''(0:v:0)'' 1.5

$< F > ::= - < T >$ ''(CLS:a:0)'' $| \; (+ | \Lambda) < T >$ ''(CLA:a:0)'' 1.6

$< S > ::= < S1 > | < S2 >$ 1.7

$< S1 > ::= + < T >$ ''(FAD:a:0)'' 1.8

$< S2 > ::= - < T >$ ''(FSB:a:0)'' 1.9

$< T > ::= < \underline{V}_i > | < \underline{C}_i >$ 1.10

Explanation:

The source language syntax defines valid strings to consist of a first signed or unsigned term < F >, followed by an indefinite number of subsequent terms < S >, which are signed. In step 1.5, the naming operation ''(0:v:0)'' represents an assembly operation that will put together the separate terms to form the whole expression. These terms each generate an instruction in the machine language by naming operations such as ''(CLA:a:0)'' where ''a'' specifies an assembly operation to combine the data portion of the naming operation, e.g., CLA, with the name or symbolic address of the variable or constant.

The application of the syntax program to the example 1.4 produces an output string

V_1 , (CLA:a:1), C_1 , (FAD:a:1) , V_2 , (FSB:a:1) , (0:v:3) 1.11

By virtue of the step 1.10 the names of variables and constants are copied from the input to the output strings: these are the only characters so copied. The choice of machine instruction is made in S1 and S2 from the signs + or - on the input string but these signs do not appear in the output, being replaced by the corresponding machine instructions from the naming operations.

When the string 1.11 is assembled, two processes occur

(1) Combination of a symbolic address with a machine instruction.

V , (OP:a:1) $\longrightarrow$ OP V . and

(2) Combination of several segments of code (here 3 separate machine instructions) into one segment.

$$S_1,\ S_2,\ \ldots,\ S_r,\ (0{:}v{:}r) \longrightarrow S_1S_2\ \ldots\ S_r$$

Such assembly operations convert 1.11 into 1.12,

```
CLA V1
FAD C1                    1.12  *
FSB V2
```

which is in the target language.

2.2 Example 2. Extension of Example 1 to Include Storage Operations

Example 1 may be extended to include simple assignment statements, so that statements like $V_3 = V_1 + C_1 - V_2$ may be translated. We give two examples, where only one assignment of a value is made, and where many variables may be assigned the same value, as in $V_1 = V_2 = V_3 + V_4$.

(a) Single assignment.

$< H > ::= < G > < E >$ ''(0:b:0)''   2.1

$< G > ::= < \underline{V}_i >$ = ''(STO:a:0)''   2.2

Here < G > represents the assignment part ''V =''. The ''='' sign is not transmitted to the output string, being replaced by the naming data. The two parts of the assignment statement are < E > which is the < E > of example 1, and < G >. The naming operation ''(0:b:0)'' will combine these so that the assignment follows the calculation : it should always have two arguments which are blocks of code to be interchanged.

* The meanings of the machine functions are:

CLA :	clear the accumulator and place the quantity addressed in the accumulator.
CLS :	clear and subtract.
FAD :	add into the accumulator, using floating-point arithmetic.
FSB :	subtract from the accumulator, floating point.

(b) Multiple assignment

$$< H > ::= < I > < E > \quad \text{''(0:b:0)''} \qquad 2.3$$

$$< I > ::= < G > \left\{ < G > \right\} \quad \text{''(0:v:0)''} \qquad 2.4$$

where < G > is an in 2.2 . Step 2.4 says that there may be one or more assignments, which are grouped by an assembly operation ''v'' before being interchanged, according to 2.3, by ''b.''

2.3 <u>Example 3. Arithmetic expressions using + , - , * , / and parentheses</u>

We use the IBM 709 as the target machine. In this machine, as in many others, there are two registers concerned with multiplication and division. One register, the AC, is concerned with addition and subtraction, and holds the result of a multiplication; in it must be placed the numerator before division. The other register, the MQ, holds the result of a division; in it is also placed one of the factors of a product before multiplication. Consequently, there are certain forms for which it is unnecessary to use intermediate storage; for floating-point arithmetic these are

(a) + X*Y/Z* ... , where multiplication and division alternate.

(b) + X*Y/ ... /U*W $\pm$ S $\pm$ T ... , where multiplication and division alternate in the first term, the last operator in the first term is * and then follows addition or subtraction.

(c) + (+X/Y* ... *Z + A ...) /U ... , where a parenthetic expression will provide a result in the AC, which is the numerator for a division.

For problems of this sort we must use the machine instruction programming for the syntax machine. We shall see here the use of the marking and sensing operations, M(X) and K(X), which allow notes to be kept of where intermediate results are to be found at the various stages in the object program. In devising programs of this sort, it is fruitful to consider the states of the target machine as it would obey the program we wish to generate. There will be states in the syntax program corresponding to the states in the object program being generated; these states in the syntax program will be the states at the commencement of the program steps (or on lines in the flow chart). Sometimes, a state of the syntax machine will also be represented by marks placed by M operations, for later sensing by K operations.

There are four principal states of the target machine called A+, A-, Q+ and Q-, when the AC, MQ, is holding positively (negatively) the result of a partial evaluation of the expression. Correspondingly named states exists in the syntax machine. These four bit-symbols are used by M and K operations, and are also used as labels in the flow diagram. An example of the use of this notion of states in the object machine occurs in the scanning of the expression '' -X*Y+Z''. This is analyzed by the syntax program as - (X*Y-Z) since we can only form products positively in the AC, and may be able to absorb the negative sign on - (X*Y-Z) in a later operation, so that A+ (-X*Y+Z) can be computed as A- (X*Y-Z), for example. The states that occur during the computation (and during the syntax analysis are

text:	-X*	Y	+Z
states:	Q-	A-	A-
output form:	X*	Y	-Z

and since the end state is A- , the object program will produce the negative of the -X*Y+Z . There will be a mark, A-, in the marker part of the push-down list, so that it can be subsequently recognized that a program to evaluate the negative has been constructed. In general, the process brings negation from the inside of parentheses to the outside; at the worst, therefore it will only be necessary to provide a change of sign for any parenthesized expression, and then only for the complete expression and not for any of its parts. Indeed the only occasion when a negated result will be produced may be discovered by the application of the rules:

(1) A variable or constant has parity +1 .

(2) A parenthesized expression has the parity of its first term.

(3) If the first term of an expression is of product-quotient form, its parity is given by rules (4), (5) and (6). Otherwise, the parity is +1 .

(4) If the first of the multiplication or division operators is ''*'', then the parity of the term is the evaluation of the term (including leading + or - signs) using the parities of the components as values.

(5) If division comes first, and the first numerator is a variable or constant, then the parity is the evaluation of the term using parities, taking that part to the right of the first ''/''sign only.

(6) Otherwise, proceed as in rule (4), but with ''/'' instead of ''*'' .

If the parity of the expression is -1, its negative will be produced. The target-machine instructions used are

LDQ	load the MQ register.		
FMP	multiply the number in the MQ by the number in the specified address. The result appears in the AC.		
FDH	divide the AC by the number from storage: the quotient appears in the MQ.		
XCA	interchange the contents of the AC and MQ.		
FAD	add to the AC.		
FSB	subtract from the AC.		
CLA	clear the AC and add.	STO	store the AC.
CLS	clear the AC and subtract.	STQ	store the MQ.

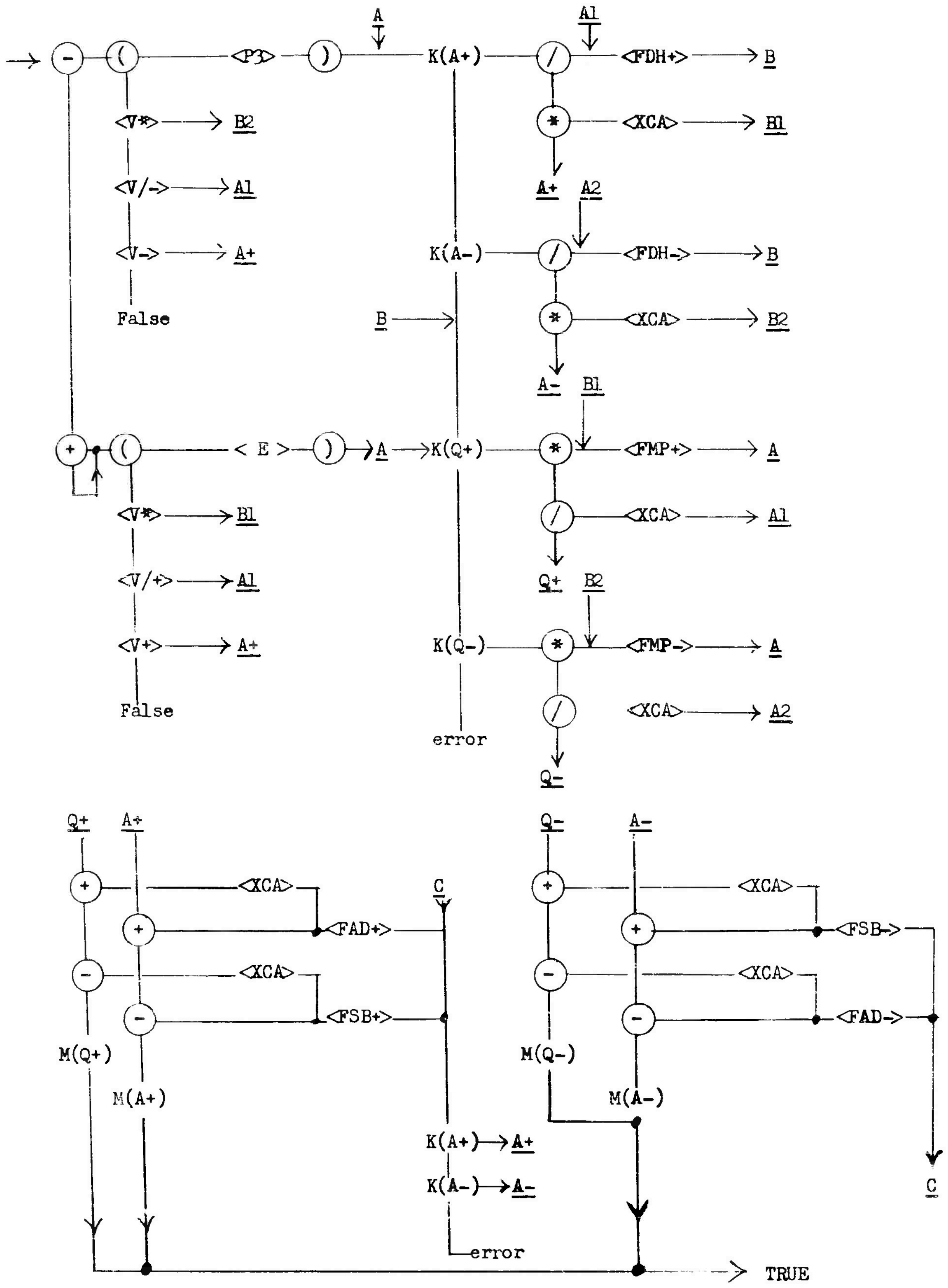
In the course of evaluation it is sometimes necessary to store intermediate results: for this purpose the assembly process following syntactic analysis must be able to generate the address of a working location. The syntactic operator (D:c:0) does this, where D will be the machine instruction required to store the result. If a parenthetical expression, say (A-B) , requires the result to be stored, the corresponding output produced by the syntax analyzer will be

... , A, (CLA:a:1), B, (FSB:a:1), (STO:c:0), (0:v:3), ...

(STO:c:0) will obtain a working space location, say W, and construct the instruction STO W , leaving it in the push-down list of the assembler so that when the operator (0:v:3) is processed it will have as arguments the three assembled single instructions (in this case) CLA A, FSB B, STO W . (0:v:3) assembles this into a block of code, placing the name of the result W in the push-down list. Later W will be combined with a machine instruction by an operator of type (D:a:1), at which point W could be returned to the list of addresses available for use as working space.

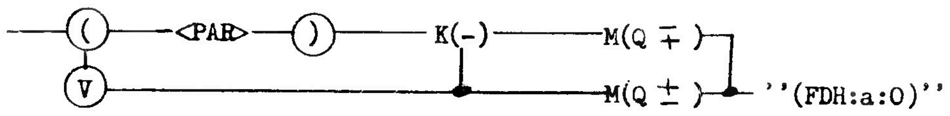
The syntax program flow diagrams follow. < E > is the recognizer for arithmetic expressions; successful outcome will be marked in the syntax machine push-down list by A+, A-, Q+ and Q- according as the result in the object machine would be in the AC (positively or negatively) or the MQ (positively or negatively).

< E > ::=

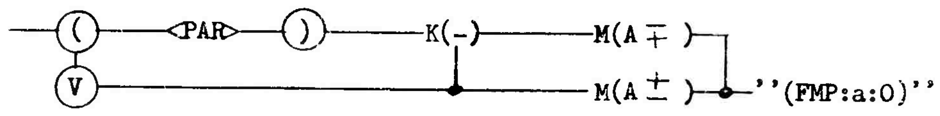


-
(
<P3>
)
A
K(A+)
/
A1
<FDH+>
B
<V*>
B2
*
<XCA>
B1
<V/->
A1
A+
A2
<V->
A+
K(A-)
/
<FDH->
B
False
B
*
<XCA>
B2
A-
B1
+
(
< E >
)
A
K(Q+)
*
<FMP+>
A
<V*>
B1
/
<XCA>
A1
<V/+>
A1
Q+
B2
<V+>
A+
K(Q-)
*
<FMP->
A
False
/
<XCA>
A2
error
Q-
Q+
A+
Q-
A-
+
<XCA>
C
+
<XCA>
+
<FAD+>
+
<FSB->
-
<XCA>
-
<XCA>
-
<FSB+>
-
<FAD->
M(Q+)
M(Q-)
M(A+)
M(A-)
K(A+)
A+
K(A-)
A-
C
error
TRUE

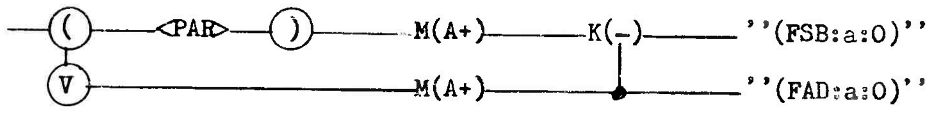
<FDH±> ::=



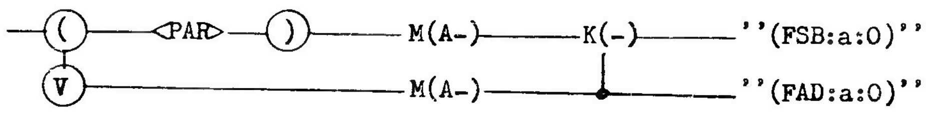
<FMP±> ::=



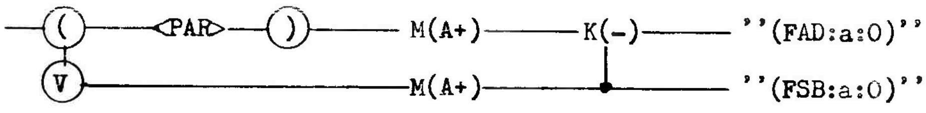
<FAD+> ::=



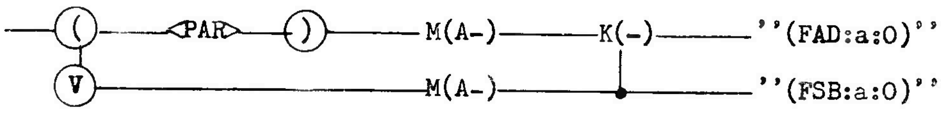
<FAD-> ::=



<FSB+> ::=



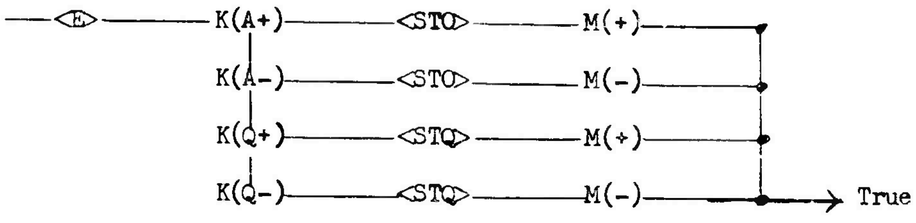
<FSB-> ::=



<V*>	::=	<V> * ''(LDQ:a:0)''		<V->	::=	<V> ''(CLS:a:0)''
<V/->	::=	<V> / ''(CLS:a:0)''		<V+>	::=	<V> ''(CLA:a:0)''
<V/+>	::=	<V> / ''(CLA:a:0)''		<XCA>	::=	''(XCA:a:0)''

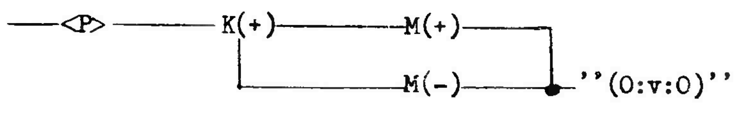
<V> ::= <$\underline{V_i}$> | <$\underline{C_i}$>

<P> ::=

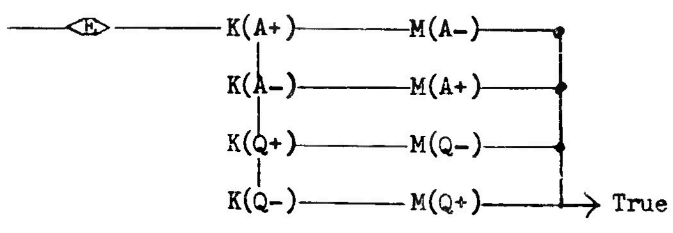


<STO> ::= ''(STO:c:0)'' ------------------ <STQ> ::= ''(STQ:c:0)''

<PAR> ::=



<P3> ::=



2.4 <u>Example 4. Assignment Statements using the Expressions of Ex.3</u>

We treat assignment statements like A=B= ... C= E where E is an expression of the type < E > of the previous example.

The source syntax is < AS > ::= < V_i > = < AS1 >

<AS1 > ::= < E > | < AS >

At this point we could merely treat Ex. 4 in the same manner as Ex. 2. A feature of this type of treatment for this case would be that we have to decide which type of storage instruction to use according to the mode A+, A- or Q+, Q- of the right hand side. In Ex. 2, it was possible to know what type of storage instruction was required as soon as the ''='' was scanned. In Ex. 4 , this is not so. It could be assumed that the mode was A+, say, and scan the right-hand side. If the assumption were correct, the assignment statement could be constructed. If not another assumption could be tried, and the assignment statement re-scanned. This might have to be repeated before a correct assumption is made.

In example 3 the necessity for multiple scanning is largely avoided by the use of state markers: in example 4, to save multiple scanning we require new apparatus, which may be a part of the assembly process rather than the syntax machine. We must have some process of re-ordering so that the names of the variables on the left of the ''='' sign may be combined with functions that can be specified only after the right-hand side of the assignment statement has been scanned. Recall that the symbols copied from the input to the output tapes of the syntax machine are in the same order on both tapes. For the statement A=E we can most simply generate an output A,(E),F where (E) stands in place of the string generated by the right-hand side, and will eventually in the assembly process be represented by a single level of the assembly push-down list. The symbol F stands for a syntactic operator, or set of syntactic operators which, because their generation by the syntax machine follows the generation of (E), can be made to depend on the mode of (E).

The primitive operator that we seek is a sort of interchange operator ''(0:e:0)'' of actual degree 2, but appearing with 0 as its ostensive degree. To use it, and to preserve the well formed nature of the postfix notation at all stages of its processing we require a null syntactic variable Λ_o . The action of this operator is defined by the transformation

$$(D) \ , \ (E) \ , \ (0{:}e{:}0) \quad \rightarrow \quad \Lambda_o \ , \ (E) \ , \ (D) \qquad \ldots \ 2.4.1$$

in the assembler's push-down list. The null symbol Λ_o will not occur as an argument of all syntactic operators; it will occur as an argument of (0:v:0) but not of (0:a:0).

The flow diagram for the assignment statement follows

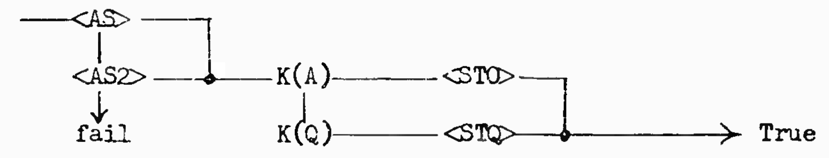
<AS> ::= <V$_i$> = <ASl> ''(0:v:0)''

<INT> ::= ''(0:e:0)''

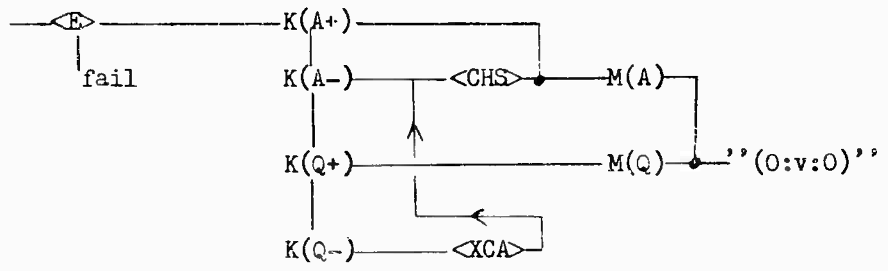
<STO> ::= <INT> ''(STO:a:0)''

<STQ> ::= <INT> ''(STQ:a:0)''

<ASl> ::=



<AS2> ::=



<CHS> ::= ''(CHS:a:0)''

If this program is applied to the assignment statement ''B=C/D'' , the resultant output is

B, C, (CLA:a:1), D, (FDH:a:1), (O:v:2), (O:v:1), (O:e:O), (STQ:a:1), (O:v:3).

When the ''Interchange'' operator comes to be processed by the assembler, the assembler's push-down list contains (or refers to)

Position:	m	m+1	m+2
Contents:	B	CLA C	(O:e:O)
		FDH D	

which changes by the ''interchange'' operator to

Position:	m	m+1	m+2
Contents:	Λ_o	CLA C	B
		FDH D	

at which point B is now available as the argument for the operator (STQ:a:1) which converts position m+2 of the push-down list to STQ B. The last operator then completes the evaluation of the program.

2.5 Example 5

Simple Relational Expressions

Here we consider relational expressions such as $X \geq 0$, $X > Y$ and so on, where the general form is E_1 Op E_2, where E_1, E_2 are expressions which have values which are numbers and Op is a relational infix operator specifying a condition that holds or does not hold between the values of E_1 , E_2 . The result of the operation is a binary value, which we shall take to have the following interpretation.

(a) If the condition of the relation is satisfied, the object program is to branch.

(b) If the condition is not satisfied, the branching operation is to be ineffective.

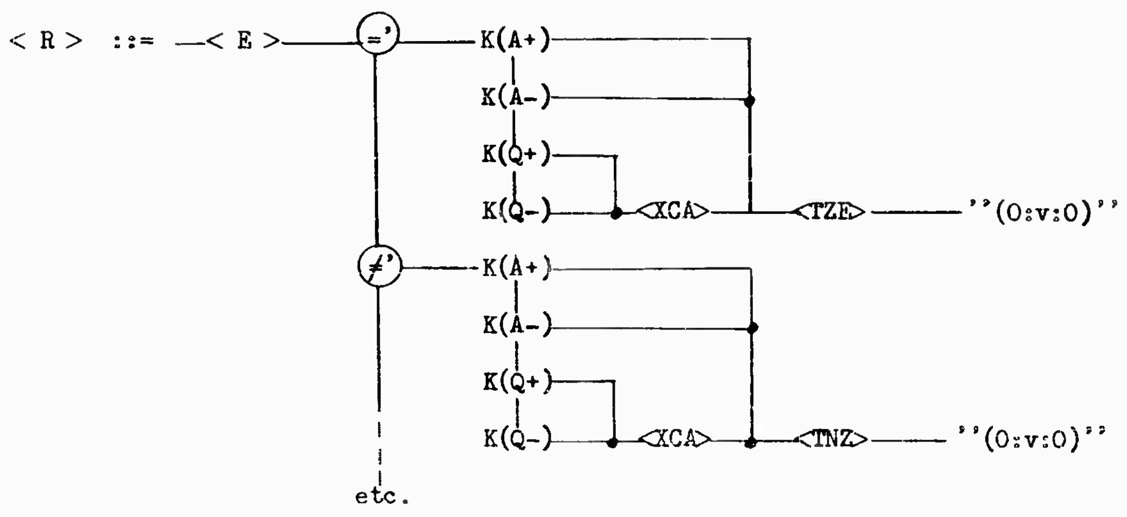
We shall consider only those relations where E_1 Op E_2 is equivalent to $E_1 - E_2$ Op 0, e.g., where Op is the relational operator $=$, $\neq$, $\geq$, $>$ etc. The object program that results will be a computation of $E_1 - E_2$, followed by a branching instruction. The program branches if the test is satisfied.

We shall consider first the case $E_2 = 0$, and then treat the more general case. In anticipation of the next example, we shall provide a means of complementing the relation during syntax analysis, so that, for example, $X = Y$ could be translated as if $X \neq Y$ had been the text.

For the special cases, the initial translation from the original names to the co-ordinate names can be extended to recognize the diagrams $=0$, $\neq 0$ etc., translate them by single characters $='$, $\neq'$ etc. These characters will now distinguish the special cases.

Then the syntax program for the recognition of simple relational expressions is an extension of the program for recognizing arithmetic expressions, which is used to scan the arithmetic expression part of the relational expression. The appearance of the relational operator forces an exit from that recognizer, whereupon the appropriate branching instruction can be added to the output according to the type of relational operator. We shall give an example for translation to the IBM 709 for the operators $='$ and $\neq'$.

The syntax program follows: it uses a new assembly operator (D:d:O) which constructs a branching instruction with machine instruction code D, and notes in the assembler's push-down lists that the constructed instruction lacks a transfer address which must be filled at some later time in the assembly.



TZE	::=	’’(TZE:d:O)’’	*
TNZ	::=	’’(TNZ:d:O)’’	
XCA	::=	’’(XCA:a:O)’’	

The complementary recognizer $< \bar{R} >$ is similar to < R > but with the comparators =’ and ≠’ interchanged; it can therefore be constructed with much in common with < R >.

For the general case E_1 Op E_2, the strategy for constructing a recognizer is to analyze the expression E_1 as in example 3 until the relational operator is encountered. At this point a chain of comparators can be used to test for each relational operator, and make a mark in the syntax machine’s push-down list using an M operation: the state of the recognizer < E > (i.e., A+, A-, Q+, Q-) may then be tested so that < E > may be entered again

* Two IBM 709 machine instructions have been introduced, namely

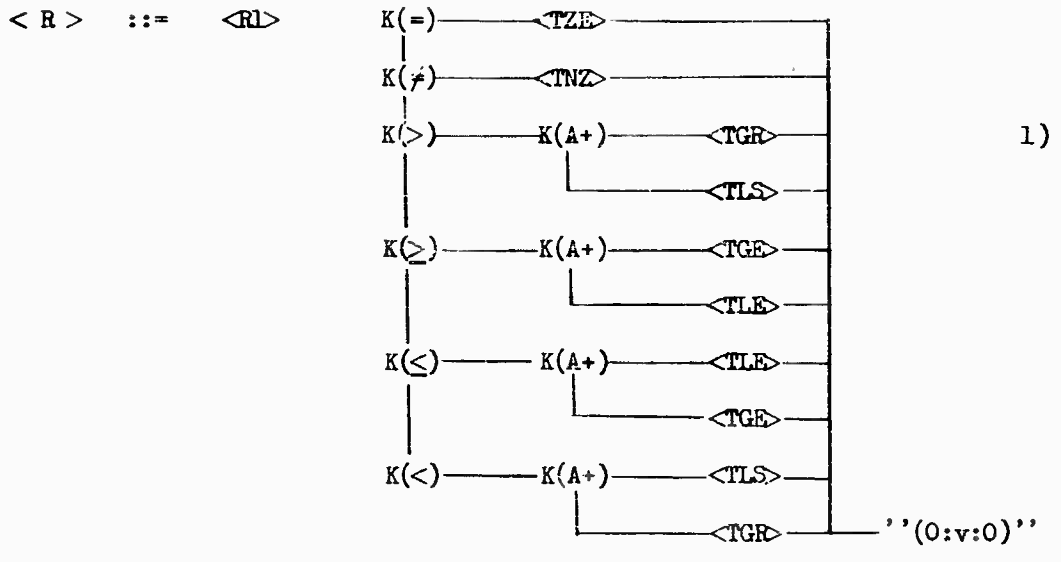
TZE, transfer control if the AC is zero.

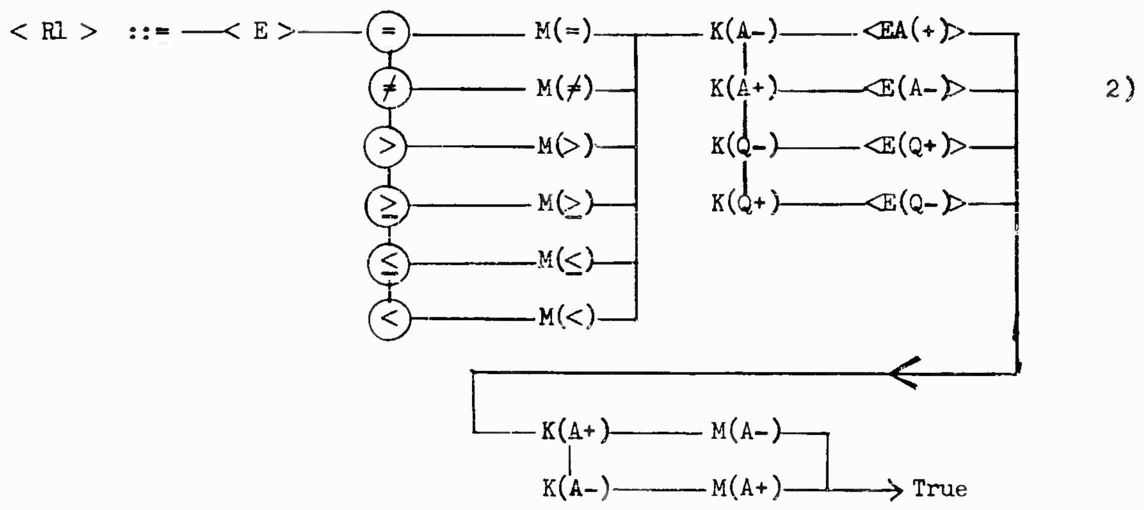
TNZ, transfer control if the AC is not zero.

(but not at its normal entry point) to complete the recognition and corresponding program generation for the expression $-E_1+E_2$ *. That is, the syntax machine is programmed to read E_1 Op E_2 and provide an output as if it had been reading the arithmetic expression $-E_1+E_2$. This is achieved by entering < E > for the second time at the position (in the flow diagram of Example 3) A- (or A+ , Q+ , Q-) if the output state of < E > on its first use had been A+ (or A- , Q- , Q+ respectively). On the exit from < E > for the second time it is possible to add the appropriate branching instruction, since the specification of the relational operator has been preserved by a marking operation.

---

* For this process to be effective, the expression E_2 must be signed; this necessary sign can be added in the preliminary scan, just as the characters =0 were replaced by =' for the simpler case. Thus X=Y should be transformed to X=+Y .





Notes 1) TGE is a subroutine to construct on the output tape an instruction to branch if AC is greater than or equal to zero. To write this we require an assembly operation not yet introduced. In example 7 we return to this matter.

2) The other subroutines have obvious significance. The subroutine <E(A+)> is the subroutine < E > of example 3 entered at the point labeled A+ .
Similarly for the others.

Program for simple relational expressions.

2.6 Example 6. Combinations of Relations

In this example we treat combinations of relational expressions using the Boolean operators ''and,'' ''or'' and ''not.'' In so doing, we introduce a novel algorithm for the analysis of logical expressions by use of the syntax machine.

In examples 1 to 4 we were translating programs which did not have branch points in their control sequencing so that the object program was obeyed sequentially. In example 5, we had object programs with a branching operation. Now we combine programs that have branching.

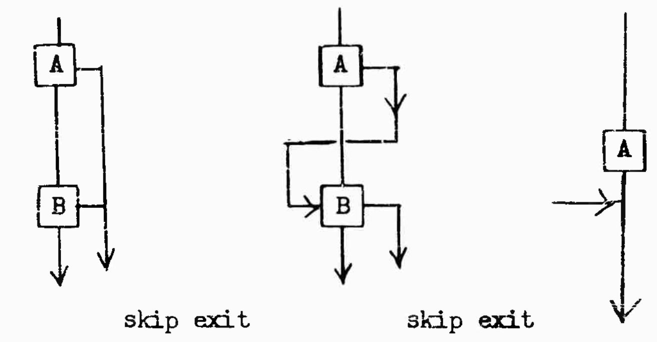
We define a program block as a block of object program which is an assembled single instruction of object code or a block of code assembled from program blocks. Program blocks may be conditional, when they have one skip exit in addition to the exit of normal (sequential) sequencing - or they may be unconditional, lacking the skip exit. Within a conditional program block there may be many branching operations, but the block as a whole has one skip exit. Program blocks may also be labeled, but by one label only.

For example 6, we need three assmebly operators for combining conditional program blocks. These are (0:v:0), (0:w:0) and (0:x:0). The first of these, (0:v:0), has been used before without all its properties being announced; it combines those program blocks which are its arguments into one program block whose skip exit is the common skip exit of the argument blocks. If all the arguments are unconditional, the result is also. At most,one of the arguments may be labeled, which label (if any) is the label of the combination.

The operator (0:w:0) has two operands, which are program blocks. If (A), (B) stand in place of program blocks, the block (A), (B), (0:w:2) is the combination of the blocks (A), (B) (in that sequence) with the skip exit of (A) joined to the label of (B). The conditionality of the result depends on the conditionality of (B): the result is labeled by the label of (A).

The third operator is a labeling operator (0:x:0), which has one operand, which must be an unlabeled program block. It provides a label for the block so that a transfer of control could be made to skip over the block.

The diagrams for these operators are



A, B, (0:v:0)   A, B, (0:w:2)   A, (0:x:1)

They provide the mechanism for realizing conditional expressions. For example, if p(A) is the proposition that the skip exit is the actual exit from A , when program A is run, then

$$p(A,B,(0{:}v{:}2)) = p(A) \lor p(B)$$

$$p(A,B,(0{:}x{:}1),(0{:}w{:}2)) = \overline{p(A)} \land p(B)$$

The operators are chosen so that the normal exit from the first program block is the normal entry to the second program block. Thus the program blocks may be assembled in position before the connecting operators (0:v:0) and (0:w:0) have been reached. Together with negation, these operators enable binary decision programs to be written for any Boolean function. Moreover, if the logical operators $\equiv$ and $\not\equiv$ are not used, the Boolean function can be re-written by changing the operators only, without duplication or change of ordering of the predicates or program blocks.

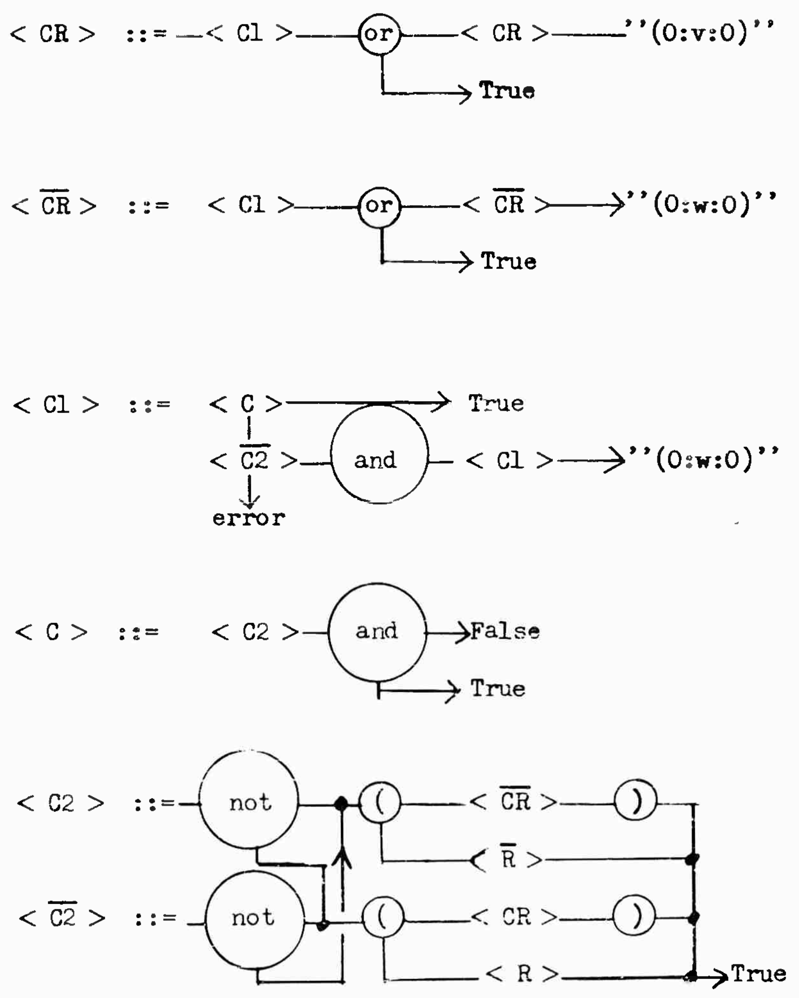
We are now in a position to write a translation algorithm for the source language string defined by

$$\langle CR \rangle ::= \langle C1 \rangle \text{ or } \langle CR \rangle \mid \langle C1 \rangle$$
$$\langle C1 \rangle ::= \langle C2 \rangle \text{ and } \langle C1 \rangle \mid \langle C2 \rangle$$
$$\langle C2 \rangle ::= \langle R \rangle \mid \text{not } \langle R \rangle \mid (\langle CR \rangle)$$

where R are simple relations of the form E_1 R E_2 as treated in the previous example.

The analysis is made in terms of the operators (0:v:0) and (0:w:0), or rather in terms of the corresponding logical operators. Because the input text is written using ''and,'' but the analysis is made in terms of ''w,'' we require complementary pairs of recognizers so that terms like ''R_1 and R_2'' may be translated to ''not R_1 w R_2''. In this example we have to apply the complementary recognizer to the first operand so that ''R_1'' is translated as if ''not R_1'' had appeared on the input string instead of ''R_1''. The use of De Morgan's rules also allows the ''not'' operarions to be passed inside parentheses so that in the translation they apply only to the simple relational expressions.

The syntax program < CR > follows



< CR > ::= < C1 > or < CR > ''(0:v:0)''
True
< $\overline{\text{CR}}$ > ::= < C1 > or < $\overline{\text{CR}}$ > ''(0:w:0)''
True
< C1 > ::= < C > True
< $\overline{\text{C2}}$ > and < C1 > ''(0:w:0)''
error
< C > ::= < C2 > and False
True
< C2 > ::= not (< $\overline{\text{CR}}$ >)
< $\overline{\text{R}}$ >
< $\overline{\text{C2}}$ > ::= not (< CR >)
< R >
True

Syntax Program for Combining Relational Expressions

2.7 Example 7, Simple Branching Instructions

We deferred from example 5 the matter of how to write certain branching instructions which have no counterparts as single instructions of the machine's code. For example, on the IBM 709 to test that the contents of the accumulator is greater than or equal to zero, we must first test for zero and then for positive accumulator. This is because the number representation is by sign and absolute value, and the branching instructions operate on the sign (TPL = transfer on positive or TMI = transfer on minus) or on the absolute value of the accumulator (TZE = transfer on zero or TNZ = transfer if not zero).

Thus to provide a branch on the accumulator being positive or zero we require a TZE instruction followed by a TPL instruction both with the transfer address. The assembly operators introduced in the last example now make it possible to write segments of the output string that correspond to tests for the inequalities in the source language, as follows

Source language	Output string translation
>	(TZE:d:0) , (TPL:d:0) , (0:x:1) , (0:w:2)
<	(TZE:d:0) , (TMI:d:0) , (0:x:1) , (0:w:2)
≥	(TZE:d:0) , (TPL:d:0) , (0:v:2)
≤	(TZE:d:0) , (TMI:d:0) , (0:v:2)

We can now construct subroutines to provide these output strings. For example the TGR subroutine, to test > , in example 5 (second part), may be written

```
<TGR>   ::=   <TZE>   < T1 >   ''(0:w:0)''   , where
<TZE>   ::=   ''(TZE:d:0)''
< T1 >  ::=   <TPL>   ''(0:x:0)''
<TPL>   ::=   ''(TPL:d:0)''
```

The subroutine for providing a greater than or equal test is

<TGE> , where

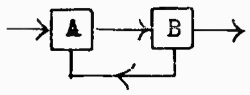
<TGE> ::= <TZE> <TPL> ''(0:v:0)''

and <TZE> and <TPL> are the subroutines described above.

2.8 Example 8. Iteration Statements

The purpose of this example is to introduce another syntactic operator (or assembly operator) of order 2 which will be useful in the construction of program loops. Consider two programs A , B where A and B stand for the syntax machine output for these programs. Program A must be a labeled program and program B must be conditional. Then the operator (0:y:0) applied to A , B forms a combination of A and B in that order with the skip exit of B connected to the labeled entry point of A , as shown below.

C = A, B, (0:y:0)



The result program C may itself be conditional, if A was conditional, or labeled if B was labeled. In other words C has the skip exit (if any) of A, and the label of B (if any).

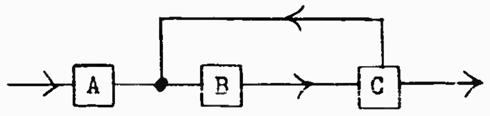
As an example, consider an iteration statement which in the source language consists of three parts concatenated e.g., A B C , where

A represents an initialization of variables (i.e., iterates).

B represents the calculation of new values of the iterates from the old.

C represents an end test for the iteration,

so that the diagram for the program is to be



Clearly C is a conditional program and **A** must be labeled: the postfix representation is either

D= A, (0:x:1), B, (0:v:2), C, (0:y:2) ... 2.8.1 or

D_2= A, (0:x:1), B, C, (0:v:2), (0:y:2) ... 2.8.2

according as B is first combined with **A** or with C. In 2.8.1 B could be a conditional program, but not labeled: in 2.8.2 B must be unconditional but may be labeled.

We refrain from giving further examples, as we now go on to consider the properties of the translations that have been illustrated in the preceding examples.

<u>Remarks on Part 2</u>

In examples 1 to 8 we have shown various examples of translation that the syntax machine and a suitable post-assembler can make. We now gather together some of the salient features.

The principal property of the process is that the ordering of the variables is not changed by the translation, except by the re-ordering of arithmetic expressions by parenthesizing and by the interchanges made by the operators ''b'' and ''e''. Example 4 shows how the role of the interchange operator ''b'' can be taken over by the operator ''e'', so we may consider ''e'' only. The properties of ''e'' depend on the assembler.

The simplest assembler would be one which assembled directly into machine code and placed each instruction into its final position. Thus ''e'' could be used to effect the transformation 2.4.1, i.e.,

$$(D), (E), (0:e:0) \rightarrow \Lambda_0, \ (E), (D)$$

only when (D) stands for the address part of an incomplete machine instruction, where (D) is stored directly in the assembler's push-down list and not merely by reference to an assembled set of machine instructions already located.

We hope to show in part 3 of this report, how this condition on translation may be relaxed by using the mechanism of declarations.

Another property of the object program is that no advantage has been taken of common subexpressions, to economise in the object code. It is the author's opinion that the search for common subexpressions in algebraic formulae is a simple matter for the composers of programs and should be left to them rather than to the mechanical translators if it is desirable to have a quick translation. The same may be said about many other forms of economization which could be made unnecessary by simple rephrasing of the source program. Example 3 shows, however, that economization in the use of arithmetical registers is possible.

The syntax machine can differentiate many special cases of the source-language text where the properties of the target machine allow the use of program tricks. With some of the extensions to be proposed in part 3 of this report, it becomes possible to recognize many special cases in the source language that are of common occurrence, and to provide corresponding segments of machine code (or macro-instructions).

The program combination operators v, w, x, y provide a quite powerful notation for combining programs with branches; in effect they provide a method of writing a wide class of branched programs without using explicitly written labels. For example, in the iteration 2.8.2 of example 8 the iteration part B could be entered from some program other than the initialization program A.

3.1 Part 3. Declarations

Declarations are made about symbols used in the source program and alter their meaning. They are used to specify which names apply to the various classes of objects in the program, e.g., which are names of floating-point variables, fixed-point variables, functions, procedures etc. They may also be used to define new functions in terms of existing functions, or to define symbols which stand in place of whole segments of text. In addition the mechanism of declarations may be used internally in a translator.

We distinguish between two occasions where Declarations affect the translator, when a Declaration is made and when a Declaration is used. For example, if we wish to use the name ''ABC'' as the name of a function, it must be declared to be the name of a function. This declaration may be explicit, when a segment of the source text says explicitly that ABC is a function, or the declaration may be implicit, when ABC appears in such a manner that the syntax shows that a declaration about ABC is being made as a part of another declaration, as for example in

ABC(X,Y) = X sin (Y),

which definition might be given without any explanation in the source language, because this form of expression could only be what it is, a definition of a new function whose name is ABC.

The declaration is used whenever the objects named in the declarations are used elsewhere in the text, as for example if we use the function ABC as part of an arithmetic expression. e.g.,

Z= X + ABC (X+y,w)

We shall discuss three types of declaration

(1) Declarations about the syntactic properties of names.

(2) Declarations which define substitutions, where a declaration is made that a symbol stands in place of a string of symbols.

(3) Declarations about substitutions in which, when substitution is made of a string for a symbol, the string is modified by parameters.

3.2 Declarations about Syntactic Properties

An example of a declaration about syntactic properties would be

Integers, A, B, C1

which delares the names A, B, C1 to be the names of integer variables. We regard the properties of names as syntactic properties, because in the analysis of statements we must distinguish between the various types of variable, and between the names of variables and the names of functions. Our intention is to replace the names like A, B, and C1 by symbols like I_1, I_2 and I_3 which are so constructed that the syntax machine can recognize them as the names of integer variables. The subscripts could have uses in storage allocation.

However, we must first recognize declarations before we can act on them. To recognize such declarations and distinguish them from other forms of statement we assume that the Syntax Machine is analyzing programs statement by statement. Let us suppose that there are several sorts of property for which we wish to make declarations about names. We can start the scan of statements by checking whether any of the leading words are signals for declarations. A chain of comparators will do this. For example, if we have the declarations about integer variables, functions etc., we could use the following syntax program < SD >.

$$< SD > \; ::= \; < DI > \;\big|\; < DF > \;\big|\; \ldots \;\big|\; < ND >$$

where

$$< DI > \; ::= \; \text{i n t e g e r} \; \big[s \,\big|\, \Lambda \big] < ZI > \quad \text{''(0:m:0)''}$$

$$< ZI > \; ::= \; < ZI1 > \{ < ZI1 > \}$$

$$< ZI1 > \; ::= \; < ZI2 > \;\big|\; U$$

$$< ZI2 > \; ::= \; \underline{L} \, \{ \underline{NL} \} \quad \text{''(I:k:0)''}$$

and < DF > is similar to < DI > but begins with a chain of comparators for the word, ''Functions''(or its singular), and the subroutine corresponding to < ZI2 > is named by the operator ''(F:k:0)''. < ND > is the syntax program for statements which are not declarations. L is a comparator for letters of the alphabet and NL is the comparator for letters and numerals. U is a comparator for all characters but the statement ending punctuation.

The syntactic operators are

''(O:m:r)'' Return control to the syntax machine from the assembler, resetting the assembler push-down list so that the next symbol placed there will be in the same position as the first symbol used in this use of the assembler.

''(D:k:r)'' This is a combined table lookup and table constructing operator. It constructs and uses a table of equivalences between external and internal names. A possible definition of this operator might be:

(a) If D=0 and the external name is not already stored, store it in the proper place and generate a corresponding internal identifier, placing it in the corresponding position of the table and in the result position of the push-down list.

(b) If D≠0, find the place in the table for the external name and in the corresponding position for the internal name place a generated symbol D_i, where any name so generated may be recognized by a comparator as an internal identifier of class D. Place D_i in the push-down list.

(c) Otherwise, look up for the external name and place the corresponding internal name in the result position of the push-down list.

In the use of this operator in the making of declarations, only operation (b) would be used. Part (a) of the operator makes it useful for dealing with the class of names about which no declarations are made. A possible method of storing external names is discussed by Williams (Comm. ACM 2, 6. p21 June 1959).

In the program < DI > and corresponding programs, the return from the program must be made in a special way. When the operator ''(O:m:r)'' has been written on the output string from the syntax machine, the assembler is then entered to evaluate the part of the output string generated by the subroutine < DI >; after the evaluation process, control returns to the syntax machine which is set so that further output overwrites the string which the assembler processed.

If the program < DI > is applied to the example at the beginning of this section the syntax machine produces an output

... , A, (I:k:1), B, (I:k:1) C,1,(I:k:2), (O:m:3)

whose evaluation by the assembler will store the external names and generate the corresponding internal names.

In statements which are not declarations, external names must be replaced by their internal name equivalents. This may be done by the program which we shall discuss in the next part where we show how statements may be handled by a similar mechanism to substitution declarations.

3.3. Substitution Declarations

We now consider the type of declaration where a string in the source language is given a name, which may thereafter stand in place of the string. There are two sorts of replacement which we might consider; replacement in the input string, and replacement in the output of the syntax machine. The latter is what we shall consider as the medhanism is useful in dealing with non-declaratory statements.

We take, as an example of this sort of declaration,

Let B1 := A=B+C

by which we define B1 to stand in place of the statement ''A=B+C''. As before we can write a program with a chain of comparators that check the presence of the word ''Let'' before proceeding in the manner particular to this type of statement. The program is called

$$< DL > ::= < IdF > : = \{< DL >\} \quad ''(0:i:0)''$$

$$< IdF1 > ::= \underline{L} \{\underline{NL}\} \quad ''(F:k:0)'' \; , \quad < IdF > ::= < IdF1 > \quad ''(0:j:0)''$$

$$< DL1 > ::= < Id > \mid \underline{U}$$

$$< Id > ::= \underline{L} \{\underline{NL}\} \quad ''(0:k:0)''$$

where U is the comparator for all characters except end of statement punctuation.

When this program is applied to the example the resultant string is

..., B, 1,(F:k:2), (0:j:1), A,(0:k:1), =,B,(0:k:1), +,C,(0:k:1), (0:i:6)

and because of the special treatment of subroutine returns associated with the operator (0:i:0), this string is now evaluated by the assembler. What is to happen is this

(a) The external name B1 is processed by the operator (F:k:2), with the result that the internal identifier F(B1) is placed in the push-down list.

(b) The operator (0:j:1) is next encountered. Its operand is the internal name generated in (a). Its purpose is to set up a table of absolute addresses where the processed string form of the declaration will be stored. This address will be that occupied in the example by the character B. The table of locations of processed strings then contains F(B1) and L(B) where L(B) is the location of the first character of the string in process. The result in the Push-down list is a null symbol Λ_0 .

(c) The k operators replace A, B and C by the corresponding internal names.

(d) The operator (0:i:r) then sets the syntax machine to work on the result in the assembler's push-down list which is now

$$\ldots, \Lambda_0, \phi_A, =, \phi_B, +, \phi_C, (0:i:6)$$

where $\emptyset_A$ is the internal name of A, and, of course, is now a single character by which the declared syntactic properties of A may be recognized. The syntax program starts at $\emptyset_A$. The operator (0:i:6) left in the push-down list now acts as an end of statement mark.

When the syntax machine is applied now it analyzes the string by the type of program of which examples were given in the second part of this report. On completion of its work, the program exits via a special true return to the syntax program that called < DL >. This abnormal return switches the input of the syntax machine back to the original string. This abnormal return situation can be anticipated when control left the syntax machine for the assembler, and the position of the input string stored.

The processing of non-declaratory statements can be done in the same way except for the treatment of the name of the string. The syntax program is < ND >,

< ND > ::= < ND1 > {< DL1 >} ''(0:i:0)''

< ND1 > ::= ''(0:n:0)''

and < DL1 > is as before.

If < ND > is applied to the string ''A=B+C'', the first output string is

..., (0:n:0), A,(0:k:1), =,B,(0:k:1), +,C,(0:k:1), (0:i:6)

The processing proceeds as before except for the action of the operator (0:n:0), which is to generate an internal formula symbol G_r as its result. Otherwise, it acts like the operator (0:j:1) in placing the internal formula symbol in the table of processed string locations. The result is

... , G_r, $\emptyset_A$, =, $\emptyset_B$, +, $\emptyset_C$, (0:i:6)

When this comes to be processed by the syntax machine, processing starts at the second symbol as before, since the result of the operator (0:i:6) placed in the push-down list is merely G_r. As a consequence, the push-down list of the assembler contains a list of symbols, one for each statement processed. For declarations this symbol is the null symbol; for other statements it is the internal formula symbol. When all statements have been read the string in the assembler's push-down list stands in place of the program, which now exists in corresponding order as segments on the output string of the syntax machine. These segments all end with a punctuating symbol that was added by the second pass of the syntax machine.

The assembler also has a second pass, which is an assembly to machine-language code. It is here that the substitution of strings for internal formulae symbols and declared string symbols occurs. Unless a ''Load and Go'' type of assembly is required this second assembly would be done when the compiled program is loaded. Actually the loading process would also include a syntax analysis since it is very easy to incorporate corrections at load time by replacing whole statements. This would be done by writing a declaration for the corrected statement, using the internal formula symbol for the string to be corrected.

The expansion of internal formulae symbols is done by the assembler switching its input. This may be explained as follows.

Suppose that the assembler is reading from a string S1 and finds internal formula symbol. The table of string locations is consulted to find the absolute location of the first symbol of the string. The assembler takes this next, noting that it has to return to the original string when the end of the secondary string is reached. Clearly this process is recursive, if all the return addresses are kept.

3.4 <u>Declarations about Macro-instructions</u>

An important class of declarations is that in which macro-instructions are defined by a declaration such as

Macro F(X,Y,Z) = X(Y+Z)

where the form on the left, namely F(X,Y,Z), is short for the expression on the right. The macro-instruction is different from the closed subroutine in that every time the short form is used in the program a modified copy is placed in the appropriate part of the target program. In the definition, the parameters (i.e., X,Y,Z in the example) are dummy symbols.

There are two ways in which we might approach this problem, by using substitution methods on the input string of the syntax machine or by using substitution on the output as in the previous section. In the first method we would consider macro-instructions to be merely shortened ways of writing parts of the source language with the expansion to full form being made in the input string, so that, for example, writing F(A,B,C) is completely equivalent to writing in its place the expression A(B+C). This method has the advantage that we do not need to make any declarations about the modes of the variables (i.e., whether the variables are integer variables, floating-point variables etc.). The second method is more appropriate for large sections of a program, such as the ALGOL procedures. Here we deal with Method 1.

The macro declaration is processed as follows.

On the first pass of the syntax machine the word ''Macro'' can be recognized and program control switched to the program for processing the rest of the declaration. The program scans the text and produces a string whose evaluation by the assembler will leave the following pattern on the output string. For the example ''Macro F(X,Y,Z) := X(Y+Z)'', the pattern is

Cell address	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10
Contents	0	0	0	$\overline{\not{O}}$	$\overline{n}$	(	$\overline{n+1}$	+	$\overline{n+2}$	)	$\overline{\not{c}}$

The overlined symbols have a special effect on the syntax machine. To distinguish them from normal symbols, they might be negative. The first three cells are to hold the names which will be the parameters of the macro when it is used : the symbols $\overline{\not{O}}$ and $\overline{\not{c}}$ cause switching of the input and output of the syntax machine; the symbols like $\overline{n}$ are address symbols, in the sense that when

$\bar{n}$ is read by the syntax machine, it acts as if it were reading the symbol from the cell whose address is n .

The program for making the declaration is < MD >.

```
< MD > ::= m a c r o < MD1 > (< MD4 >) < MD10 > < MD 7 > ''(0:m:0)''
< MD1 > ::= < MD2 > ''(0:p:0)''
< MD2 > ::= < MD3 > ''(0:j:0)''
< MD3 > ::= L {NL} ''(M:k:0)''
< MD4 > ::= < MD9 > {< MD5 >}
< MD5 > ::= , < MD9 >
< MD6 > ::= L {NL} ''(0:k:0)''
< MD7 > ::= < MD8 > {< MD8 >} ''(∅:s:0)''
< MD8 > ::= < MD6 > ''(0:r:0)''
< MD9 > ::= < MD6 > ''(0:q:0)''
< MD10 > ::= : = ''(∅:s:0)''
```

The application of MD to the example will yield an output string: -

```
..., F, (M:k:1), (0:j:1), (0:p:1), X, (0:k:1), (0:q:1), Y, (0:k:1), (0:q:1)
  , Z, (0:k:1), (0:q:1), (∅:s:0), X, (0:k:1), (0:r:1), (,Y,(0:k:1), (0:r:1), +
  , Z, (0:k:1), (0:r:1), ), (0:s:6), (0:m:3)             *
```

The new assembly operators are: -

(0:p:1) switch the output from the assembler to the output list. This ensures that the coded definition of the macro is placed on the output string. This operator also clears out a temporary table used by the q and r operators.

(0:q:1) In the temporary table mentioned above place the internal name (which is the operand) and the absolute location in which this was stored at the time (0:q:1) was applied to it.

* as with other examples it has been assumed that the input text contains no spaces. This simplifies the exposition.

(0:r:1) The operand is an internal name. Look for it in the temporary table and if it is found there, replace the operand by the absolute address noted against it in the temporary table; otherwise the operator has no effect. The purpose of this operator is to replace the parameters by an address referring to the position in which the actual parameters will be placed when the macro is used.

(0:s:0) No matter what the operand count of this operator , write the character from the data field in the place occupied by the operator.

The evaluation of the output of the first scan of the syntax machine causes

(1) The name of the macro to be written in the table of processed strings together with the address (n in the example) of the processed macro definition.

(2) The p operator then switches the output from the assembler to the output string.

(3) The q operators then take note of the formal parameters in the definition, so that the r operators can replace them in the processed string by the absolute address of the location to which the internal names of the parameters will go when the string is used.

(4) The s operator writes a mark ¢ which will switch the input of the second scan of the syntax machine when the macro is used.

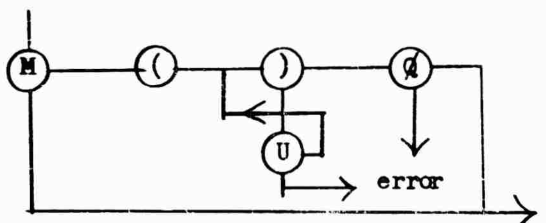
To use such a macro we have to make some extensions to the syntax machine, so that the input can be switched from one text to a subsidiary text and then returned to the original text. The symbols that are special in this respect are symbols of class M denoting internal macro names, the special symbols ¢, a̸ written by the operator s , and the absolute addresses written by r operators in the processed form of the macro definition.

Now consider what happens when the syntax machine scans a test in which the internal symbol M_F appears. This would not be in the original text so we are talking about the second pass of the syntax machine when its input has external names replaced by internal names. Let the original source text contain ''F (A,B,C)'' where F is the macro of our example and A,B,C are names of variables or constants. Then the corresponding string within the input for the second pass of the syntax machine is M_F, (,$\emptyset_A$, $\emptyset_B$, $\emptyset_C$,), where the commas are used to separate the characters of this string, and the characters $\emptyset_A$ etc. are the internal character names of A etc.

A special comparator is used for symbols of class M i.e., names of this type of macro. If such a symbol is recognized by a comparator, the output of the syntax machine is switched to the address where the macro definition begins. The syntax program then fills the parameter cells with the names of the parameters used here, namely $\emptyset_A$, $\emptyset_B$ and $\emptyset_C$. When these have been read the syntax machine uses yet another special comparator to check the presence on the current output position of the symbol $\not{\emptyset}$ and if it is found the input of the syntax machine is switched to the next position of the macro-definition list (cell n+4 in the example), and the output list of the syntax machine reset to its state before the M symbol appeared.

The syntax machine now scans the rest of the macro definition until the symbol $\not{\emptyset}$ appears when the input of the syntax machine is switched back to what it was before the last M symbol appeared. By the usual technique of push-down lists it is simple to make these macros recursive.

The syntax program for the use of macros is



where the comparators M and $\not{\emptyset}$ are the special comparators mentioned in the text. This should be placed in all parts of a syntax program where an M might be under the scrutiny of the syntax machine.

Part 4. The Assembler

In parts 1, 2 and 3 much has already been said about the assembler. We consider now only one part of the assembler, that used to assemble postfix strings to target machine language, using the operators ''a'', ''c'' and ''d'' which form single machine instructions and the operators ''e'', ''v'', ''w'', ''x'' and ''y'', which manipulate program blocks.

The assembler for these operators is best considered separately from the assembler for other operators since the push-down list requires four registers AR_r, BR_r, CR_r and LR_r on each level r. AR_r holds the names and operators from the postfix string being assembled. BR_r holds the assembled forms of single instructions provided by the operators ''a'', ''c'' and ''d''. CR_r holds an address which refers to a conditional machine instruction or a conditional program block. It also holds a negative sign * if the level r is holding a single machine instruction in BR_r. LR_r holds an absolute address which is a transfer point generated by a label. There is also a location counter whose contents L give the address where the assembled instructions of the program go when transferred from the push-down list.

For this assembly it is assumed that the ''k'' operator which provided internal names generated the subscripts on these names by incrementing a counter so that the subscript is a relative address for each variable in the block for variables of that type. The final values of these counters (one for each class of variable) can be used to provide base addresses for each block, from which the absolute addresses of any variable can be constructed by the operator ''a''.

* We assume that each register of the push-down list has a sign position and a value position, so that representation is by sign (+ or −) and value.

4.1 The Operator, (D:a:1)

When this operator appears in the position AR_n the internal name which is its operand is in AR_{n-1} . The absolute location corresponding to the internal name is combined with the machine instruction specified by D and the result placed in BR_{n-1}. CR_{n-1} is made negative to show that BR_{n-1} contains a single machine instruction. The push-down list level counter is then set to n, so that the next item is brought into AR_n. If the operand was the name of a working location send it back to the list of working spaces (see below).

4.2 The Operator, (D:c:0)

This operator combines a function specified by D with a working-space location. Associated with this operator and with operator ''a''is a list of used working spaces. If this list is empty then ''c'' must construct the name of a working location which it can do by incrementing a counter whose initial contents was the address of the beginning of a block of storage allocated for working space. If WS is internal name of this working-space variable, (selected from the list, or constructed) then the result in the push-down list is

AR_n = WS

BR_n = D:L(WS) i.e., the machine instruction with function D and address L(WS) which is the absolute location corresponding to the working space name WS.

CR_n is negative.

where the operator (D:c:0) was in AR_n . Note that the operator ''c'' acts like the operator ''k'' in the production of an internal name. We want an internal name to appear in AR_n because it will subsequently be used as the operand of an ''a'' operator. The next item to be read into the push-down list must enter level n+1.

4.3 The Operator, (D:d:0)

If this appears on level n in AR_n then we have in the result

BR_n = D:0 the machine instruction with zero address.

CR_n = - n to show that there is a machine instruction in BR_n.

The next item to be placed in the push-down list must be placed on level n+1 .

4.4 The Labeling Operator , (0:x:1).

This has two cases according as the operand is a machine instruction within the push-down list or is a block of code assembled in its final position.

Case 1 : The initial configuration of the push-down list is

AR_n	
BR_n	holds a machine instruction
CR_n	is negative
LR_n	should be positive
AR_{n+1}	(0:x:1)

This case is recognized by CR_n negative. LR_n should also be positive, indicating an unlabeled instruction. The action of the labeling opertor in this case is to mark level n on the push-down list by making LR_n negative.

Case 2. In this case CR_n is positive, LR_n is positive and contains the address which will be the value of the label if one is required. This is furnished by the the operators ''v'', ''w'' or ''y''. The action is merely to make LR_n negative.

In both cases the next item is read into position AR_{n+1} .

4.5 The Operator, (0:v:r)

Suppose that this operator appears in AR_{n+r} ; then its operands are in levels n through n+r-1 of the push-down list; they may be machine instructions still in the push-down list (recognized by the CR part of the level being negative) or they may be blocks of machine code already stored.

The first action of the operator is to check that there is at most one labeled operand, by testing all the LR positions of the operands; those levels that are labeled will have negative LR .

Then the operands are taken in order and process A applied to those that are single instructions still within the push-down list. Process A is common to operators ''v'', ''w'' and ''y'' ; it places the single instructions on their final positions in store, using L which is incremented by 1 whenever single instructions go to the store. If a conditional instruction is stored from the push-down list in location L then L is copied into the CR position and L+1 is copied into the LR position. In both instances the signs describing conditionality and labeling are preserved. At this point all the operands have been stored in their final positions.

Now we must connect any skip exits from the operands. A single machine skip instruction will reside in its final position with its transfer address zero, and the corresponding CR position will point to the location of the instruction. For program blocks the CR position will point to a location holding one of the conditional instructions in the block. If the transfer address here is zero, then this is the only conditional instruction in the block that contributes to the skip exit. If the address of the conditional instruction is non-zero it is pointing to another conditional instruction contributing to the skip exit. Thus, the CR contents is the first of a chain of addresses ending with address 0 , which specify locations of instructions contributing to the skip exit, (except the last, 0).

In the ''v'' operator, these chains are linked together into a single chain, which now shows which are the conditional instructions requiring transfer addresses. The first member of this chain is stored in CR_n . In LR_n is stored the absolute value of the label if any of the operands were labeled. As usual LR_n shows labeling.

4.6 The Operator, (0:w:2)

If any of the operands are single instructions then process A is applied to them, reducing the operands to refer to program blocks in their final position. The first and second operands are then checked for conditionality and labeling respectively. Then process B is applied to link the skip exits from the first operand with the label of the second operand, by proceeding down the chain of locations in which are to be inserted the address value of the label. Finally, the conditional information CR_{n+1} for the second operand replaces CR_n to form the result on level n. The next item is to be read into level n+1.

4.7 The Operator, (0:y:2)

The action of this operator is almost identical with that of (0:w:2) but the label from the first operand is used with the chain of locations of conditional instructions of the second operand.

In the operators ''v'', ''w'' and ''y'' it may be necessary to provide a label value in anticipation of the use of ''x'' to label the result. If the result of the operation is an unlabeled block and process A has been used to insert single instructions in their final locations then the LR position of the result should hold +,L+1 , where L was the address of the location last used by process A.

4.8 The Operator, (O:e:O)

If this operator appears in AR_n , then the data on level $n-2$ of the push-down list is placed on level n , and the registers on level $n-2$ set to zero to indicate nullity. The next item to be read to the push-down list goes to AR_{n+1}.

Conclusion

This report has outlined a method by which a compiler can be programmed (by syntax machine programs) to accept various source languages. Apart from the final assembly of the postfix string to target-machine code the method is not particularly dependent on the computer making the translation, since the compiler is constructed to perform interpretively on the syntax program and on the syntactic operators in the postfix strings.

The syntax program will not be lengthy, as is demonstrated by the examples of Part 2. Perhaps 300 - 400 instructions in the syntax program are sufficient.

The quality of the translation will be variable, since no method of economization of subexpressions is included, nor is any method of economization of index register proposed. Methods for these could be developed, for example, by modifying syntax machine so that it could

(1) Analyze arithmetic expressions to produce the so called three-address form (this might require a right to left scan) and search for common subexpressions among the output.

(2) Abstract from the source language some parts, e.g., subscripts and loop control statements, for analysis by a more powerful symbol manipulator with re-insertion in the program by methods like those of Part 3. This would require extensions to the syntax machine so that its subprograms (recognizers) could be written with parameters.

The speed of translation is likely to be high; it is estimated that it would take 1000 instructions in the computer making the translation to produce one machine instruction of the translation. On the IBM 704 for example, this means that translation is at the rate of 40 instructions per second.

The major part of the syntax machine has been simulated on the IBM 650. This interpretive simulation program required 60 instructions and simulated comparators for single characters and the subroutine facilities described in Part 1; the output mechanism was also simulated. Each pseudo-instruction required two cells of storage. Some coding experiments indicate that the assembler will require not more than about 400 instructions. Thus, it seems possible to write a quite powerful compiler in 500 instructions plus the syntax program.

C O N T R A C T O R

Carnegie Institute of Technology
Computation Center
Pittsburgh 13, Pennsylvania
Attn. Professor Alan J. Perlis

L I S T   O F   R E C I P I E N T S

Assistant Sec. of Def. for Res. and Eng.
Information Office Library Branch
Pentagon Building
Washington 25, D. C.

Chief of Naval Research
Department of the Navy
Washington 25, D. C.
Attn. Code 437, Information Systems Br.

Chief of Naval Research
Department of the Navy
Washington 25, D. C.
Attn. Code 923

Director, Naval Research Laboratory
Tech. Information Officer /Code 2000/
Washington 25, D. C.

Commanding Officer , Office of Naval Res.
Navy No. 100, Fleet Post Office
New York, New York

Commanding Officer, ONR Branch Office
346 Broadway
New York 13, New York

Commanding Officer, ONR Branch Office
495 Summer Street
Boston 10, Massachusetts

Office of Technical Services
Technical Reports Section
Department of Commerce
Washington 25, D. C.

Chief, Bureau of Ships
Department of the Navy
Washington 25, D. C. (Attn. Code 280)

Chief, Bureau of Ships
Department of the Navy
Washington 25, D.C. (Attn. Code 677)

Armed Services Technical Information Agency
Arlington Hall Station
Arlington 12, Virginia

Chief, Bureau of Ships
Department of the Navy
Washington 25, D.C. (Attn. Code 684)

Chief, Bureau of Ships
Department of the Navy
Washington 25, D.C. (Attn. Code 686)

Chief, Bureau of Ships
Department of the Navy
Washington 25, D.C. (Attn. Code 687E)

Chief, Bureau of Ships
Department of the Navy
Washington 25, D.C. (Attn. Code 687F)

Chief, Bureau of Ships
Department of the Navy
Washington 25, D.C. (Attn Code 687G)

Naval Ordnance Laboratory
White Oaks
Silver Spring 19, Maryland
Attn. Technical Library

David Taylor Model Basin
Washington 7, D.C.
Attn. Technical Library

Chief, Bureau of Ordnance
Department of the Navy
Washington 25, D.C.

Naval Electronics Laboratory
San Diego 52, California
Attn. Technical Library

Naval Ordnance Laboratory
Corona, California
Attn. Robert Conger, Electr. & Mag. Div.

LIST OF RECIPIENTS Continued:

University of Illinois
Control Systems Laboratory
Urbana, Illinois (Attn. D. Alpert)

University of Illinois
Digital Computer Laboratory
Urbana, Illinois (Attn. Dr. R.E. Meagher)

Air Force Office of Scientific Research
Washington 25, D.C.

Air Force Cambridge Research Center
Laurence C. Hanscom Field
Bedford, Massachusetts
Attn. Electronic Res. Directorate Library

Technical Information Officer
US Army Signal Research & Dev. Lab.
Fort Monmouth, N.J. (Attn. Data Equip. Branch)

Commanding Officer
Diamond Ordnance Fuze Laboratories
Washington 25, D. C.
Attn. Tech. Ref. Section, /ORDTL 06.33/

Office of Ordnance Research
Box CM, Duke Station
Durham, North Carolina

Director, National Security Agency
Fort Geo. G. Meade, Maryland
Attn. Chief, Remp.

Naval Proving Ground
Dahlgren, Virginia
Attn. Naval Ordn. Computation Center

National Bureau of Standards
Washington 25, D. C.
Attn. Dr. S.N. Alexander

Aberdeen Proving Ground, BRL
Aberdeen Proving Ground, Maryland
Attn. Chief, Computation Lab.

Office of Naval Research
Resident Representative
University of Pittsburgh
Room 107 Salf Hall
Pittsburgh 13, Pennsylvania

National Bureau of Standards
Washington 25, D.C. (Attn. Mr. R.D. Elbourn)

Dynamic Analysis and Control Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts
Attn. D.W. Baumann

Syracuse University
Electrical Eng. Dpt.
Syracuse 10, New York
Attn. Dr. Stanford Goldman

Princeton University
Electrical Engrg. Dept.
Princeton, New Jersey
Attn. Professor F.S. Acton

Burroughs Corporation
Research Center
Paoli, Pennsylvania
Attn. A.J. Meyerhoff

Hycon Eastern, Inc.
75 Cambridge Parkway
Cambridge 42, Massachusetts
Attn. Mr. J.E. Deturk

Cornell Aeronautical Laboratory
4455 Genesee Street
Buffalo 21, New York
Attn. Systems Requirements Dept.
Dr. Frank Rosenblatt

Lockheed Missile Systems Division
Sunnyvale, California
Attn. J. P. Nash

University of Michigan
Ann Arbor, Michigan
Attn. Dept. of Speech, Director
Speech Research Laboratory, Gordon Peterson

University of Michigan
Ann Arbor, Michigan
Attn. Dept. of Philosophy
Prof. A.W. Burks

Census Bureau
Washington 25, D. C.
Attn. Office of Asst. Director for
Statistical Services, Mr. J.L.McPherson

University of Illinois
Urbana, Illinois
Attn. Electrical Engr. Dept.,
Prof. H. Von Foerster

National Science Foundation
Washington 25, D.C.
Attn. Res. Info. Center & Advisory
Serv. Of Info. Processing

Wayne State University
Detroit, Michigan
Attn. Dept. of Slavic Languages,
Prof. Harry H. Josselson

University of California - LA
Los Angeles 24, California
Attn. Dept. of Engineering,
Prof. Gerald Estrin

Columbia University
New York 27, New York
Attn. Dept. of Electrical Eng.,
Prof. Ralph J. Schwartz

Hebrew University
Jerusalem, Israel
Attn. Prof. Y. Bar-Hillel

Benson-Lehner Corporation
11930 Olympic Blvd.
Los Angeles 64, California
Attn. Mr. Bernard Benson

Atomic Energy Commission
Washington 25, D.C.
Attn. Div. of Research

Naval Research Laboratory
Washington 25, D.C.
Attn. Solid State Electronics,
Code 5210, Mr. G. Abraham

Zator Company
140.5 Mt. Auburn
Cambridge 38, Massachusetts
Attn. R. J. Solomonoff

David Taylor Model Basin
Washington 7, D.C.
Attn. Arthur Shapiro

U.S. Naval TRaining Device Center
Port Washington
Long Island, New York
Attn. Mr. Arthur Miller, Code 3144

The University of Chicago
Institute for Computer Research
Chicago 37, Illinois
Attn. Mr. Nicholas C. Metropolis, Director

Stanford Research Institute
Computer Laboratory
Menlo Park, California
Attn. W.H. Kautz, Senior Research Engineer

Commander
Wright Air Development Center
Wright Patterson Air Force Base, Ohio
Attn. WCLJR, Maj. L.M. Butsch

US Army Biological Warfare Laboratories
Fort Detrick, Frederick, Maryland
Attn. Clifford J. Maloney,
Chief, Statistics Branch

Zator Company
140.5 Mt. Auburn St.
Cambridge 38, Massachusetts
Attn. Calvin N. Mooers

Air Force Office of Scientific Research
Washington 25, D.C.
Attn. Dr. Harold Wooster

National Bureau of Standards
Washington 25, D.C.
Attn. Miss Ida Rhodes, 120 Far West

Post Office Department
Office of Research and Engineering
12th and Pennsylvania Avenue
Washington 25, D.C.
Attn. Mr. R. Kopp, Research & Development Div.

Air Force Cambridge Research Center
L.G. Hanscom Field, Bedford, Massachusetts
Attn. Chief, CRRB

Research Air Development Center
Griffiss Air Force Base, New York
Attn. RCWID, Capt. B. J. Long