Z77-4287

TECHNICAL INFORMATION EXCHANGE

November 15, 1966

THE EVOLUTION OF COMPLIERS

Miss Marilyn M. Jensen
IBM Corporation
3223 Wilshire Boulevard
Santa Monica, California 90406

A paper covering the evolution of compliers, with emphasis on techniques for machine independent languages including Polish notation, operations for handling strings, use of pushdowns, and scanning techniques. Introductions to the topics of recursive subroutines, ALGOL, JOVIAL, SNOBOL, XTRAN, and the heuristic complier is given. An extensive Bibliography is included.

For IBM Internal Use Only

Z77-4287

IBM Corporation. Technical Publications Dept., 112 E. Post Road, White Plains, N.Y. 10601

TABLE OF CONTENTS

I. INTRODUCTION............................Page 1

II. XTRAN....................................Page 4

A. INTRODUCTION.......................Page 5

B. PUSHDOWN STACKS.................. Page 8

C. STRING AND SYMBOL MANIPULATION Page 12

D. LANGUAGE 1........................ Page 18

E. LANGUAGE 2........................ Page 26

F. ALGORITHMS FOR POLISH NOTATION Page 32

G. LANGUAGE AMBIGUITY.............. Page 34

H. COMPILER LOGIC................... Page 44

I. SWITCH METHOD/SINGLE ADDRESS OUTPUT............................ Page 50

J. FORCING CODES AND ROUTINES...... Page 52

K. BACKUS NORMAL FORM.............. Page 62

III. HUERISTIC COMPILERS.................... Page 63

A. INTRODUCTION...................... Page 64

B. INFORMATION PROCESSING LANGUAGE-V........................ Page 67

C. COMIT..............................Page 70

D. SAINT..............................Page 71

E. GIT................................Page 72

F. LINE BALANCING.................... Page 73

IV. SYMBOL MANIPULATIVE COMPILERS...... Page 75

A. ALGOL............................. Page 76

B. JOVIAL............................ Page 82

C. SNOBOL............................ Page 89

D. NELIAC............................ Page 93

V. BIBLIOGRAPHY...........................Page 94

A. SUGGESTED READINGS............... Page 95

B. REFERENCES........................Page 110

INTRODUCTION

Definition

The routine which accepts a set of symbolically coded instructions, translates them into a machine language, and at the same time also assigns either symbolic or regional addresses to absolute machine addresses is called an assembly routine. It may or may not allow for macro instructions. The assembler helps with the symbology problem confronting the programmer, expands the number of apparent operations available from the machine and eases the chore of assigning storage locations.

An extension of the assembly technique is the compiling routine. A compiler permits more complex macro instructions than an assembler, and often excludes machine instructions, even in symbolic form from the language which it can accept. While the assembler generally deals with each instruction independently of all others, the compiler attempts to capitalize on the information which is contained in the structure or logic of the problem. The context in which each instruction is nested is important. Commonly, a compiler is language or problem-oriented in that it accepts as input the language and operations or a particular class of problems.

List Processing Languages

The allocation of storage space even with sophisticated automatic coding languages can be a major difficulty, because many times it is impossible to foresee how much information will be produced or have to be stored at each phase of the routine. The concept of list stores and list processing languages was developed in order to surmount such difficulties. "List" is used in the conventional sense to designate a linear sequence of pieces of information which for some reason are to be associated. The length of a list is not necessarily fixed. This implies a capability for inserting or deleting an entry anywhere along the list. Such a list is frequently called pushdown. If the only entry point is at the top, an entry on a list may be the name of a sub-list. The sub-list in turn may reference another sub-list. Such a collection of lists and sub-lists is called a list structure.

List processing languages permit such operations as 1) Insert an entry on a list, 2) Delete an entry on a list, 3) Create a list, 4) Destroy a list, 5) Coalesce or concatenate a list, and 6) Search a list for a given symbol. They are sometimes called symbol manipulating languages, and include LISP, and the family of Information Processing Languages, IPL-V being the best known.

Such languages facilitate storage allocation and give complete freedom for development and use of recursive subroutines.

All of the List Processing Languages employ pushdown stores or stacks to hold the return address and the parameters used by a subroutine. If, during the execution of a first subroutine, a second subroutine is entered, the return address and parameters of the second are simply pushed down on those of the first. In this manner, the latter are preserved and pop up upon completion of the second routine and return to the first. When the stack has sufficient capacity, subroutines can be nested to any depth and used recursively.

Complier Concepts

One of the basic concepts employed when writing a compiler is that information which is phased for further use when translating can be conveniently kept in a stack. A stack is distinguished from other types of tables by the fact that only the item at the top of the stack, the youngest item, is important at any given time. A second concept implemented is that comparison of adjacent operator properties provides a valuable criterion, and no more than this is needed to correctly interpret any formula. The third technique is that parentheses can be treated as operators with priorities, thus enhancing the algorithm.

Roughly, a compiler will scan a long expression, left to right or right to left, until some operation is found which can be performed regardless of what will occur in the remainder of the expression. This operation is discovered through a force table or force codes. As soon as an operation is found which can be performed regardless of future input, it is accomplished. Meanwhile, the unused portion of the formula is retained in the stack. The compiler must now be able to switch back and forth discover which operators can be forced and what is available within the stack.

Compiler Construction

Compilers allow the programmer to write the problem solution in broad source statements, i. e. macro instructions. These statements are analyzed by the compiler which in turn generates the necessary symbolic instructions. The segment of a compiler which interprets a macro instruction and develops the required symbolic instruction sequence is called the macro generator. For each macro instruction included in a given compiler language, there is a separate macro generator. The entire complex of macro generators provided in a compiler is in only one section of that compiler. The generators are used in a specific phase of the entire translation process. From this point the final portion of a compiler performs the same functions as any other assembly program. Every compiler includes an assembly process to affect final conversion to machine language.

II. XTRAN

A. INTRODUCTION

History

As the complexity and quantity of machine applications have increased, problem oriented programming languages have become more popular. FORTRAN and ALGOL for scientific formula oriented programs, and COBOL for business type problems are being used more widely than ever before. It is only natural then that attempts have been made toward a problem oriented approach to compiler writing, especially when we consider the magnitude of the job of efficiently writing a compiler. XTRAN is a language that was developed by the IBM Corporation for use by the Programming Systems Department. The XTRAN language exists in a slightly different form for different machines. Therefore, the following discussion will not deal with an official version of the language, but instead, general concepts and techniques used.

String Techniques

In FORTRAN, the basic element dealt with is the formula. In XTRAN we are concerned with the sequence of the characters called the string, which must be broken down and analyzed to such an extent that linkages to closed subroutines can be generated by a subsequent program called the macro expander program. For the 7090 the output of XTRAN must be such that the macro expander will be able to generate a TSX followed by an appropriate number of PZE instructions which would cause the desired subroutine linkages to be formed.

What types of operations can we expect XTRAN to perform on strings? One useful bit of information about a string during analysis would be its length or number of symbols called the NORM. Consequently one operation found in XTRAN is NORM (ST, 1) which would compute the NORM of the string named ST and store the result in location 1. Another operation is used to isolate the end symbol of the string in order to analyze it. Further operations remove a symbol from the string, insert a symbol in a string, join two strings together or find the first occurence of a symbol in a string.

Compilation

One of the most important aspects of XTRAN is that it is a language that can be used to compile itself. The approach taken can be to write a few basic XTRAN operations in the machine symbolic language and after these are assembled and debugged write more powerful operations in XTRAN. The language will generate a set of symbolic instructions which can be reassembled with the prior

version of the language and included in the new language. XTRAN has the ability to scan source statements and compile itself in FORTRAN in say the 7040 -- then, if someone wants a FORTRAN compiler for the 1440, XTRAN does not have to be rewritten for it. Only the macro expander for the 1440 to take the output of XTRAN must be written in order to generate the desired linkages. What we have in XTRAN then is a truly general purpose compiler writing program.

High-and Low-Level Languages

Before commencing with a definition of XTRAN some terminology should be clarified. When considering high-and low-level languages, the low-level language is close to the machine language and would be similar to 1401 or 1440 SPS. A high level (or higher level) language is more removed from machine language and would encompass such languages as COBOL, FORTRAN, and ALGOL.

Functions and Procedures

Consider the mathematical computation: XNEW = 1/2 (A/XOLD + XOLD). This is the appropriate mathematical format for expressing the function XNEW. The proper format for the function XNEW as stated in the FORTRAN language would be XNEW =1./2.* (A/XOLD + XOLD). A language which has only procedures will express this same function as TEMP 1 = A/XOLD; TEMP 2 = TEMP 1 + XOLD; XNEW = TEMP 2/2. In the latter procedural language as soon as the value was produced, a place had to be generated for it to be stored.

Functions are rules for producing a particular value. 1/2 (A/XOLD + XOLD) was a function of XNEW expressed in the correct mathematical notation. A procedure differs from a function in that no value is produced. A typical example of a procedure would be R = S. In FORTRAN the value of S is placed in a location that is labeled R. No new value has been produced and the old value of S has not changed.

Prefix, Index, and Suffix Notation

All of the notations that we have used to this point, have been of the type infix. Infix notation means that the operator is included between or inside the two operands. Infix notation would include the value A + B. Prefix notation has its operator preceding the two values, or operands. An example of addition of two values expressed in prefix notation would be + AB. The third type of notation that is used is suffix. The suffix notation has the operator following the two operands. An example would be AB +.

Further examples of infix notation would be TEMP 1 = A/XOLD and

TEMP 2 = TEMP 1 + XOLD. Prefix notation is similar to a three address machine language. The earlier two infix statements would be expressed in the following manner: /A, XOLD, TEMP 1 states that A should be divided by XOLD and placed in TEMP storage location 1; and + TEMP 1, XOLD, TEMP 2 states that TEMP storage 1 should be added to XOLD and the result placed in TEMP storage location 2.

Language Ambiguity

For procedures, there is little difference between prefix and infix notation, however, this is not true of functions. Since infix is not completely specified it requires rules of precedence. Prefix notation is specified, and this is called Polish or Parenthesis-free notation.

Functional notation is found many times in the form of a prefix, for instance: F(A, B); G(X, Y, Z); or R(S, T(M, N), T). With functional notation operators may not be binary and they do not refer necessarily to functions or procedures.

Polish Notation

Let us consider a further example in prefix notation, the math function: Y=S/T(A/X+X). A mathematician working with this formula would know what rules of precedence are needed in order to compute the correct result. However, the computer must be given additional information. Therefore, Polish notation or parenthesis-free notation is used. The one rule in using Polish notation is that the operand always has its two addresses immediately to the right. The preceding math statement would be revised to: Y=*/ST+/AXX. Since the operand always has its two addresses immediately to the right, the first operand in the scan from left to right to be performed would be /ST. Here S would be divided by T and a temporary value would be placed in this area, TEMP 1. Scanning again from left to right the next group of values that has an operand with two addresses to the right is /AX therefore, X divided by A will be the next computation performed. After /AX has been accomplished, two addresses now follow the +, i.e., +TEMP 2 X. The third operation would be to add TEMP 2 to X. Two addresses, TEMP 1 and TEMP 2 now follow the original asterisk. TEMP 1 and TEMP 2 are multiplied in order to compute the value Y.

B. PUSHDOWN STACKS

Consider the function (A + B) and infix notation (+ AB) in Polish notation. Within infix notation equally simple examples can be given such as (A + B * C) which in Polish notation would be expressed as (+A * BC). Again in infix (Y=Z + R) would be expressed in Polish as (=Y + ZR). It becomes more complicated when taking the example: [(B-C)*A] * [X-Y↑T] . This would have to be expressed in Polish notation as **=BCA-X↑YT. The scan may be performed from right to left. (See Figure 1). When the operator is seen, it is combined with the two names on the right, producing a new name which is the name of the result, or the scan can be made from left to right, where after finding two names without an intervening OP Code the combination would be made with the preceding OP Code. In implementing a right to left scan of the example **-BCA-X↑YT with one pushdown and output on every operator, we would have TEMP 1 containing the calculation of Y to the power T; TEMP 2 containing the value X - TEMP 1; TEMP 3 containing the value B - C; and TEMP 4 containing TEMP 3 multiplied by A; and TEMP 5 containing the product of TEMP 4 (which contains the results of B - C multiplied by A) and of TEMP 2 (which contains the results of X - Y to the T power).

[(B-C)*A] * [X-Y↑T] or **-BCA-X↑YT

Stack	Output
Y, T	↑Y, T, T_1
X, T_1	-X, T_1, T_2
B, C, A, T_2	-B, C, T_3
T3, A, T_2	*T_3, A, T_4
T_4, T_2	*T_4, T_2, T_5

Figure 1

If the implementation was from left to right TEMP 1 would include the result of B-C; TEMP 2, the product of TEMP 1 and the value A; TEMP 3, the value of Y to the T power; and TEMP 4 would contain the value X-TEMP 3. (See Figure 2). In the first phase of the operation, from bottom to top of the list, an **- would have been pulled into our operator list. In the corresponding operand list reading from bottom to top, appear (((which are the separators for the operators, followed by B and C. Once two operators are adjacent the last operator in the list which was a minus can be performed, thus causing the operation of B-C stored in TEMP 1.

Phase 2 has the operator stack containing ** with the operand stack containing ((TEMP 1 and A. Once again, two operands have been entered into the operand stack without separators. Therefore, the last operator in the operator stack is associated with the last two operands and the calculation of multiplying A by TEMP 1 is performed. TEMP 2 is created and put into the operand stack.

In Phase 3, reading from bottom to top, the operator stack contains *-↑, and the operand stack has (T2(X(YT. Since two operands are not separated, the operation of Y raised to the T power can be performed.

The operator stack has in the fourth phase an *- with the operand stack of (T2(XT3. X-T3 is performed to create T4.

Only the * remains in the operator stack in phase 5, with (T2 followed by T4 in the operand stack. T2 can then be multiplied by T4 and in the final phase, the operator stack would be empty and the operand stack holding T5.

**-BCA-X↑YT

Figure 2

**-BCA-X↑YT

Phase 1		Phase 2		Phase 3		Phase 4		Phase 5		Phase 6	
Op.	Operand	Op.	Operand	Op.	Operand	Op.	Operand	Op.	Operand	Op.	Operand
					T						
					Y						
	C				(		T_3				
	B		A		X		X				
-	(		T_1	↑	(		(		T_4		
*	(	*	(	-	T_2	-	T_2		T_2		
*	(	*	(	*	(	*	(	*	(		T_5

Right to Left and Left to Right Scans

It should be noted at this point that in the left to right scan of Figure 2 the output number of ('s in the stack is equal to the order of the operator. In the right to left scan the order of the operators are kept with the operators in the operator stack. Where the operand is added, a check must be made to see whether or not there are enough to satisfy the top operator.

C. STRING AND SYMBOL MANIPULATION

Definition

By the term symbol we mean a character, an atom, a basic element, such as A,+, or $. A string is an ordered sequence of symbols such as A+B*X-(Y+Z). String literals tend to be notative, such as F(ABC) or S(3,ABC) or JOE S (3,ABC).

REMNS

One requirement of any string manipulative program would be to remove and insert operations. Let us take the following example:

A*B-R/S

T1 - R/S

T1 = T2

T3

In order to remove and insert operations it is convenient to express this with one statement in a compiler language. In XTRAN the statement is in the following format: REMNS(STA,N,SYA).

REMNS represents the phrase REmove N Symbol; STA, the name of' the STring; N, the position in the string; and SYA where it is to be stored after removal. This is termed functional notation.

A string is expressed in XTRAN with format: S(N,S1,S2,S3...SK*). All N symbols are unconditionally a part of the string and will continue to be until the next) is met. As an example, the string (93,RX)ABCDE). This string includes N which is 3 symbols, being RX and) and, according to the definition, the values through E. This string is terminated by the right parenthesis following the letter E.

Strings are assembled and held in main memory through dynamic storage allocation. (See Figure 3). Main memory might contain, as an example, two words JOE and FREE, with an additional area called the string area. The string header is JOE. In the address portion of JOE is the location of the first piece of string which is 2,000. FREE has all locations that are not being used by an active string.

The string words are also divided into two portions. At location 2,000 exists a value X with an address of the next value of the string, which is 1,000.

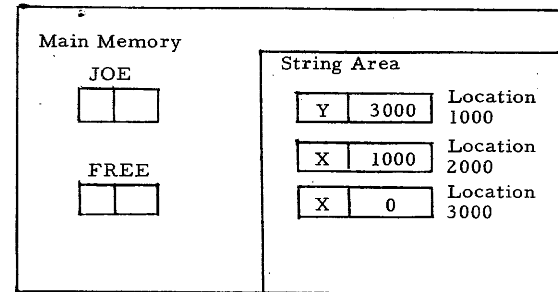
At location 1000 the string value Y has its pointer at 3000, and at 3000 value Z, pointer of zero. The string might be expressed as S(0,XYZ).

NORM

An additional value that would be required for string manipulative procedures would be the NORM of a string, which is the number of symbols in the string. A NORM word is always present in any string definition. The NORM word is divided into two sections. The right half is the pointer to the end of the string, and the left half contains the NORM value. The address of the NORM word of the string is in the string header.

Figure 3

Memory Map



Unary and Binary Operators

Only unary operators, i. e., one operator per value have been discussed. However, in many calculations, binary operators are also present. An example would be: A*(-B). Here two operations are required for B. First of all B must be set to a negative value and secondly this negative value of B must be multiplied with A.

Consider another example: (A*(-B))*(-(X-Y)). Expressed in Polish notation, the formula would be: **A-B--XY. The first level of operations would be to find the value X-Y. Within the same operational level, B must be set to a negative value. The second level of operations would multiply A times the negative value of B and set the value of X-Y to its negative form. The third operation

would be to multiply the first value by the second value, i. e., multiplying A-B and the negative value of X-Y.

Using parentheses, the formula would be expressed as:
((A-B))-(-(XY))). Or, in a slightly different manner:
((A, -(B)), -(-(X, Y))).

A third way of expressing the same equation would be to substitute the letter F for the *, G for the binary minus, and H for the unary minus. Completing the substitution we would have the formula:
F(F(A,H(B)),H(G(X, Y))).

Functions and Procedures

Symbol manipulative procedures or routines may be divided into two types, those of functions and procedures. As defined earlier, functions are rules for producing a particular value, and a procedure differs from a function in that no value is produced. (See Figure 4). Functions in the symbol manipulation routines are contatenate, norm, get n-th symbol, first occurrence of symbol, and strings identical. Procedures are free string, remove n-th symbol, insert, add symbol to list, push, replace n-th symbol, replace symbol by symbol, string assign, set pointer, and sequence.

Figure 4

Symbol Manipulation Routines

The following prefixes will distinguish the type of value which a name represents:

ST	string name
SY	symbol name
I or N	integer name
L	statement name (label)

FUNCTIONS

Representation	Name	Type of Value Produced	Description	Example
CONCAT (STA, STB)	Concatenate	String	Produces a string which is STB concatenated to the end of STA	JOE = S(O, AB) SAM= S(O, XYZ) CONCAT(JOE, SAM) produces the string S(O, ABXYZ)
NORM (ST)	Norm	Integer	Produces an integer which is equal to the number of symbols in ST	ANN = S(O, RST) NORM(ANN) produces the integer 3.
GETNS (ST, N)	Get n-th symbol	Symbol	Produces a symbol which is the n-th symbol of ST	BOB = S(O, MNXY) GETNS(BOB, TWO) produces the symbol N
FOS (ST, SY)	First occurrence of symbol	Integer	If SY is in ST, the value produced is an integer equal to the position which the fir st SY occupies. Otherwise the value is zero	JOE = S(O, RMQZMY) SYM is a location containing the symbol M. FOS(JOE, SYM) produces the integer 2.
STID (STA, STB)	Strings identical	Boolean	If STA and STB are identical a true value is produced otherwise a false value.	

PROCEDURES

Call	Name	Description	Example
FREE (ST)	Free string	Free ST.	
REMNS (ST, N, SY)	Remove n-th symbol	Set SY equal to the n-th symbol of ST. Remove the n-th symbol of ST.	BOB = S(O, MNXY). REMNS(BOB, TWO, SYA) will cause the symbol N to be stored in SYA and will change BOB to S(O, MXY).
INSRT (ST, I, SY)	Insert	Insert SY in ST between the i-th and (i+1)th symbols, so that SY becomes the (i+1)st symbol of ST.	ANN = S(O, RSTW), SYL contains the symbol L. INSRT (ANN, THREE, SYL) will change ANN to S(O, RSTLW).
ADDSL (ST, SY)	Add symbol to list	Add SY to the end of ST	SAM = S(O, XYZ). SYA contains the symbol A. ADDSL(SAM, SYA) will change SAM to S(O, XYZA).
PUSH (ST, SY)	Push	Add SY to the beginning of ST.	SAM = S(O, XYZ). SYA contains the symbol A. PUSH (SAM, SYA) will change SAM to S(O, AXYZ).
REPNS (ST, N, SY)	Replace n-th symbol	Replace the n-th symbol of ST by SY.	JOE = S(O, BXRS). SYQ contains the symbol Q. REPNS(JOE, TWO, SYQ) will change JOE to S(O, BQRS).
REPLS (ST, SYA, SYB)	Replace symbol by symbol.	Wherever SYA occurs in ST, replace it by SYB.	BOB = S(O, RMRSTRQ). SYR contains the symbol R. SYM contains the symbol M.REPLS(BOB, SYR, SYM) will change BOB to S(O, MMMSTMQ).
STASN (STA, STB)	String assign	Set STB equal to STA	JOE = S(O, XYZ). SAM = S(O, AB). STASN (JOE, SAM) will change SAM to S(O, XYZ) and leave JOE unchanged.

Call	Name	Description	Example
POINT (ST, I, PT)	Set pointer	Set pointer PT to point to the i-th symbol of ST.	
SEQ (PT, SY, L)	Sequence	Set SY equal to the symbol that PT is pointing at. Advance PT to the next symbol in the string. If PT was past the last symbol in the string, transfer to L.	

D. LANGUAGE ONE

Introduction

A very simple language such as an SPS assembly, where there are no macros, is certainly efficient and straight forward. By having only procedures available a one-for-one assembly can be assumed. For instance, to express the function NORM(ST) in Language One would require an OP Code of FUNCL with an address of norm, followed by an OP Code of PAR with an address of ST, and an additional OP Code PAR followed by the address N, as in Figure 5.

Figure 5

Operation	Address
FUNCL	NORM
PAR	ST
PAR	N

If we wish to express the function, REPlace N Symbol, the OP Code of FUNCL would appear with an address of REPNS followed by three parameters for OP Codes with the addresses being respectively ST, N, and SY. (See Figure 6).

Figure 6

Operation	Address
FUNCL	REPNS
PAR	ST
PAR	N
PAR	SY

Programs in Language One

Examples of programs written in Language One, the scan, and a description of the boot strap procedure appear in Figures 7, 8 and 9. The symbols mean following: colon, a label; comma, a parameter;), a parameter; (, a procedure; a semi-colon, noise; and , and) are interchangeable as several labels are possible.

Figure 7

Examples of Programs in Language 1

FOS (ST, SY, I)

NORM (ST, IA); ASSIGN (ONE, I); LC: GETNS (ST, I, SYA);

EQUAL (SYA, SY, B); CONDTRA (B, LA); EQUAL (IA, I, B);

CONDTRA (B, LB); ADD (ONE, I, I,); GO TO (LC);

LB: ASSIGN (ZERO, I); LA: RETURN (DUMMY);

SQUARE ROOT

ASSIGN (A, XOLD); LB: DIV (A, XOLD, TA); ADD (TA, XOLD, TB);

DIV (TB, TWO, XNEW); SUB (XOLD, XNEW, TC); ABS (TC, TD);

LESS (TD, EPSILON, TE); CONDTRA (TE, LA); ASSIGN (XNEW, XOLD);

GO TO (LB); LA: STOP (IDENT);

Figure 8

Scan for Language 1

```
START    : FREE (NAMST);
LA       : INPUT (SRCST, LEND);
LB       : NORM (SRCST, TA);
           TRAEQ (TA, ZERO, LA);
           REMNS (SRCST, ONE, CURSY);
           TRAEQ (CURSY, BLKSY, LB);
           TRAEQ (CURSY, SCLSY, LB);
           TRAEQ (CURSY, COLSY, COLSR);
           TRAEQ (CURSY, LPRSY, LPRSR);
           TRAEQ (CURSY, COMSY, COMSR);
           TRAEQ (CURSY, RPRSY, COMSR);
           ADDSL (NAMST, CURSY);
           GOTO (LB);
COLSR    : CONCAT (COLST, NAMST, NAMST; GO TO (LC);
LPRSR    : CONCAT (LINKST, NAMST, NAMST); GO TO (LC);
COMSR    : CONCAT (COMST, NAMST, NAMST);
LC       : OUTPUT (NAMST);
           FREE (NAMST);
           GO TO (LB);
LEND     : STOP (IDENT);
```


Where:

SRCST - Source string

LEND = Label, end of file routine

CURSY = Current symbol

BLKSY = Blank symbol

SCLSY = Semi-colon

COLSY - Colon

LPRSY = Left parenthesis

RPRSY = Right parenthesis

Figure 9

Bootstrap Procedure

1. Write "basic' subroutine in machine language.
2. Create macros which are calling sequences to these subroutines.
3. Additional subroutines can now be written using these macros. As each subroutine is written, a macro can be created for it.
4. Scan can now be written with macros, one per card. Assembly deck must include the macro definitions, and the subroutines. Output of this assembly will be the compiler in machine language.

At this point the following problem could be worked: write replace symbol REPLS (ST, SYA, SYB) using the functions of NORM, REMNS, ADDSL, PUSH, FREE, CONCAT, POINT, SEQ, and INSRT. (See Figure 9A).

Figure 9A

REPLS (ST, SYA, SYB)

```
LA      SET STR=ST
        POINT (ST, (FOS(STR, SYA)), PT) SET POINT TO FIRST OCCUR
        SEQ (PT, SY, LB)
        INSRT (ST, (FOS(STR, SYA)), SYB) INSERT B
        REMNS (ST, (FOS(STR, SYA)), SYA) REMOVE A
        SET STR=PT
        TRA LA
LB      END
```

Balancing Parenthesis

Consider the problem of balancing parentheses on input. Add to the beginning and to the end of the string any required parentheses to make matched pairs with the minimum number of additions. Figure 10 illustrates one solution, with the block diagram of Figure 11.

Figure 10

Insert Parenthesis

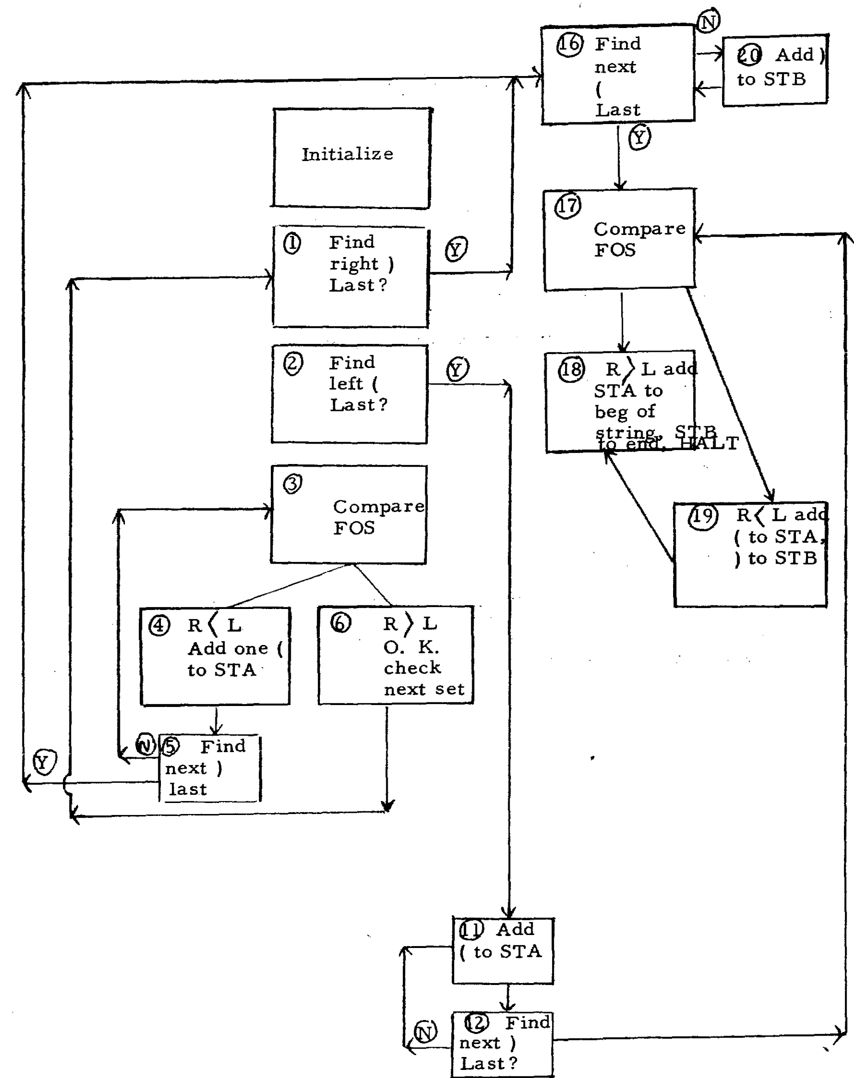
```
        SET         ST1+ST
        SET         ST2+ST
        SET         STR=ST1
        SET         STL=ST2
L1      POINT       (ST1, FOS(STR, SYRP), PTR)      1
        SEQ         (PTR, SYR, L16)                 1
        SET         STR=PTR                         1
L2      POINT       (ST2, FOS(STR, SYLP), PTL)      2
        SEQ         (PTL, SYL, L11)                 2
        SET         STL=PTL                         2
L3      SET         L=PTL                           3
        SET         R=PTR                           3
        SET         I=R-L                           3
        POS         I, B                            3
        TRATRUE     B, L6                           3
        NEG         I, B                            3
        TRATRUE     B, L4                           3
L4      ADDSYL      (STA, SYLP)                     4
        POINT       (ST1, FOS(STR, SYRP), PTR)      5
        SEQ         (PTR, SYR, L16)                 5
        SET         STR=PTR                         5
        TRA         L3                              5
```

```
L6      TRA        L1                          6
L11     ADDSYL     (STA,SYLP)                 11
        POINT      (ST1,FOS(STR,SYRP,PTR)     12
        SEQ        (PTR,SYR,L17)              12
        SET        STR=PTR                    12
        TRA        L11                        12
L16     POINT      (ST2,FOS(STL,SYLP,PTL)     16
        SEQ        (PTL,SYL, L17)             16
        SET        STL=PTL                    16
        ADDSYL     (STB,SYRP)                 20
        TRA        L16                        20
L17     SET        L=PTL                      17
        SET        R=PTR                      17
        SET        I=R-L                      17
        POS        I,B                        17
        TRATRUE    B,L18                      17
        NEG        I,B                        17
        TRATRUE    B,L19                      17


L19     ADDSYL     (STA,SYLP)                 19
        ADDSYL     (STB,SYRP)                 19
L18     CONCAT     (STA,ST)                   18
        CONCAT     (STA,STB)                  18
        SET        ST=STA                     18
        HALT                                  18
```


Figure 11

Insert Required Parenthesis



E. LANGUAGE TWO

While Language One has only procedures with the functional notation, Language Two has both functions and procedures with functional notation. Figure 12 is an example of a program written in Language Two.

Figure 12

Examples of Programs in Language 2

```
FOS (ST,SY,I)

ASSIGN (ONE,I);

LC:  CONDTRA (EQUAL(GETNS(ST,I),SY),LA);

CONDTRA (EQUAL(NORM(ST),I),LB);

ASSIGN (ADD(ONE,I),I); GO TO (LC);

LB:  ASSIGN (ZERO,I); LA:  RETURN (DUMMY);

SQUARE ROOT

ASSIGN (A,XOLD);

LB:  ASSIGN (DIV(ADD(DIV(A,XOLD),XOLD),TWO),XNEW);

CONDTRA (LESS(ABS(SUB(XNEW,XOLD)),EPSILON),LA);

ASSIGN (XNEW,XOLD); GO TO (LB); LA:   STOP (IDENT);
```

Polish Notation

All examples in this section will be done in Polish or (Parentheses-free) notation which, by the way, is used for the B5000. Polish notation is a method of expression which was developed by Jan Lukasiewicz, a Polish mathematician. Since "Polish" is much easier to say than the mathematician's name, his method of notation has been dubbed "Polish".

Instead of writing A+B, the notation would be +AB. A+B+C would be transformed to ++ABC, where A+B is executed first and the quantity A+B is added to C as a second operation.

The hierarchy of operations is: exponentiation, denoted as an arrow pointing upward; multiply and divide; and add and substract. The example A+(B*C) would be written as +A*BC.

Consider the formula: ((B-C)*A)*(X-(Y↑T)). This would be expressed in Polish notation as **-BCA-X↑YT. Performing a right to left scan to find the order of operations, the first operation the machine would perform is raise Y to the T power and place it in TEMP 1.

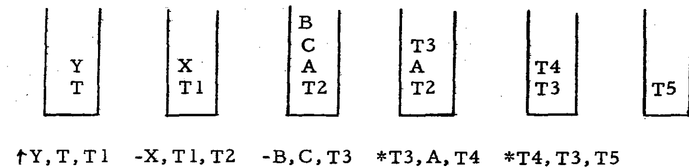
The second is to substract TEMP 1 from X and store it in TEMP 2. The third operation substracts C from B and places it in TEMP 3. The fourth phase will multiply A and TEMP 3 and place the product in TEMP 4. The fifth is to multiply TEMP 3 and TEMP 4 and place the product in TEMP 5. The output as expressed in machine language is shown in Figure 13.

Figure 13

```
↑   Y,T,T1
-   X,T1,T2
-   B,C,T3
*   T3,A,T4
*   T4,T3,T5
```

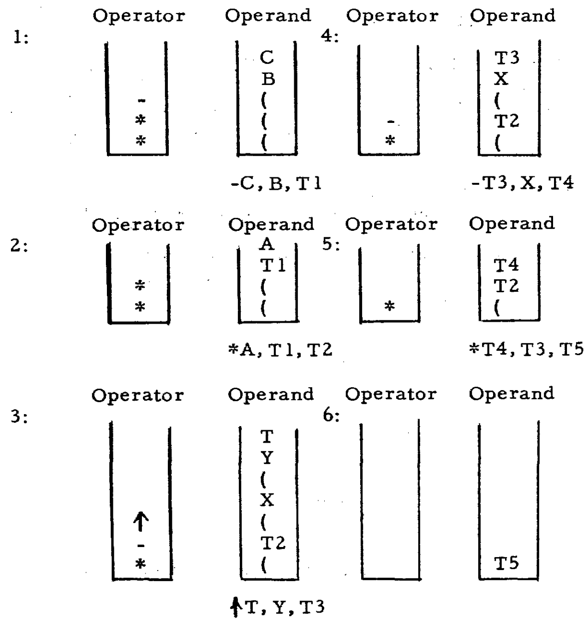
Figure 14 illustrates the same solution, with the stack activity.

Figure 14



In using a left to right scan, the pushdown storage is represented in Figure 15 with two pushdown stacks.

Figure 15

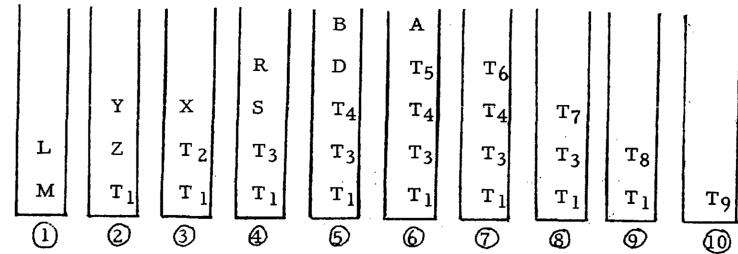


Using a more complex problem a transformation into Polish will be accomplished (Figure 16) and the two types of stack analysis will be performed. They are: 1) the right to left analysis as shown in Figure 17, and 2) the left to right analysis, using two pushdown stacks, in Figure 18. The problem is: {[[A*(B-D)]* [R-S]] - [x/(y-z)]} *(L-M)

Figure 16

*-**A-BD-RS /x-yz-LM

Figure 17



① $L-M=T_1$ $(-L, M, T_1)$

② $Y-Z=T_2$ $(-Y, Z, T_2)$

③ $X/T_2=T_3$ $(/X, T_2, T_3)$

④ $R-S=T_4$ $(-R, S, T_4)$

⑤ $B-D=T_5$ $(-B, D, T_5)$

⑥ $A*T_5=T_6$ $(*A, T_5, T_6)$

⑦ $T_6*T_4=T_7$ $(*T_6, T_4, T_7)$

⑧ $T_7-T_3=T_8$ $(-T_7, T_3, T_8)$

⑨ $T_8*T_1=T_9$ Answer $(*T_8, T_1, T_9)$

Figure 18

*-**A-BD-RS /x-yz-LM

Left to right (2 pushdowns).

1	2	3	4	5	6	7	8	9	10
D				Z					
B		S		Y					
(	T_1	R		(	T_5				
A	A	(	T_3	X	X		M		
(	(	T_2	T_2	(	(	T_6	L		
(	(	(	(	T_4	T_4	T_4	(	T_8	
(	(	(	(	(	(	(	T_7	T_7	
(	(	(	(	(	(	(	(	(	T_9

1	2	3	4	5	6	7	8	9	10
-									
*	*	-		-					
*	*	*	*	/	/				
-	-	-	-	-	-	-	-		
*	*	*	*	*	*	*	*	*	

①	$B-D=T_1$	$(-D, B, T_1)$
②	$T_1*A=T_2$	$(*T_1, A, T_2)$
③	$R-S=T_3$	$(-S, R, T_3)$
④	$T_3*T_2=T_4$	$(*T_3, T_2, T_4)$
⑤	$Y-Z=T_5$	$(-Z, T, T_5)$
⑥	$X/T_5=T_6$	$(/T_5, X, T_6)$
⑦	$T_4-T_6=T_7$	$(-T_6, T_4, T_7)$
⑧	$L-M=T_8$	$(-M, L, T_8)$
⑨	$T_8*T_7=T_9$	$(*T_8, T_7, T_9)$

From the preceding example it is seen that two types of algorithms are necessary: One is an algorithm for scanning algebraic expressions in Polish notation and the other an algorithm for scanning functional notation.

F. ALGORITHMS FOR SCANNING

Right to Left Scan - Algebraic Expression

A right to left scan must:

1) Add operands to stack;

2) When an operator is encountered, output the operator, the two operands on top of the stack and a generated temporary, remove the two operands from the top of the stack, and put the GT on the stack.

Left to Right Scan - Two Pushdowns - Algebraic Expression

When an operator is encountered, it is added to the operator stack and a separator placed on the operand stack. When an operand is encountered, it is entered in the operand stack. Whenever an operand is added to the operand stack (this would happen when a generated temporary is added to the stack, as well as when an operand is encountered in the source program), the stack should be checked for TWO adjacent operands on top. If there are none, the scan is continued. If there are, the operator on top of the operator stack is outputted along with the two operands on the top of the operand stack and a generated temporary. The operator and the operands that were output from the stacks, and the separator are removed. The generated temporary is added to the operand stack.

Left to Right Scan - One Pushdown - Algebraic Expression

Add the operators and operands to stack. Whenever an operand is added, a check is made for TWO adjacent operands on top; if not, the scan is continued. If there are, the TWO operands and the operator on top of the stack and a generated temporary are outputted. The two operands and operator are removed and the generated temporary added to the stack.

Right to Left Scan - Functional Notation

Place)s and operands in stack. When a (is encountered, this means the name in front of it is an operator or a function. This operator should be outputted, along with all the operands down to the first) in the stack. A generated temporary should also be outputted. The operands and the) should be removed from the stack, and the generated temporary added to it.

Left to Right Scan - One Pushdown - Functional Notation

Add all operators, (s, and operands to the stack as they are encountered. When a) is encountered, output the operator which is below the highest (in the stack and all the operands in the stack above it. Also output a generated temporary. Remove the operator and all the operands that were outputted from the stack, as well as the (that separated them. Add the generated temporary to the stack.

Left to Right Scan - Two Pushdowns - Functional Notation

When an operator is encountered, recognized by the fact that it is followed by a left parenthesis, add it to the operator stack and place a (separator in the operand stack. When an operand is encountered, add it to the operand stack. When a) is encountered, output the operator that is on top of the operator stack, all the operands from the operand stack down to the first and a generated temporary. Remove all of these items outputted from the stacks as well as the top-most (from the operand stack. Add the generated temporary to the operand stack.

The Burroughs' B5000 uses Polish notation with a right operator instead of a left, i. e., suffix notation. For instance, externally a value might be expressed as ((b-c)*a)*(y↑t)). Expressed in Polish notation with a suffix operator, this value would be: bc-a*xyt↑-*.

G. LANGUAGE AMBIGUITY

Introduction

Consider each of the following three examples. Examine for ambiguity.

For a*(-b) there are two operators and two operands. The hierarchy is not clear as to which operation should take precedence. a*(-b+r) is a value which may also be expressed as *a-br. Here no ambiguity exists, for b can very easily be subtracted from r and "a" multiplied by the product. The last example is a*(-b-d). In Polish notation, this would be: *a--bd, and once again a hierarchial ambiguity exists.

Subscripts

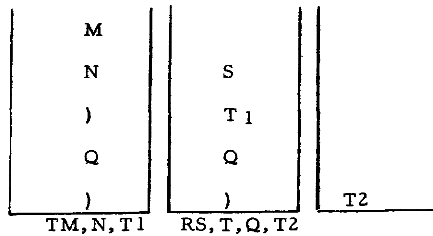
One possibility for eliminating ambiguity would be to add subscripts to the operators. Our last example would be expressed then as: *2,A-2-1 BD

To use a more complex example, consider the following: (A*(-B))*(-(X-Y)). In Polish notation the formula would be expressed with subscripts as **A-1B-1-XY. If then, by simple substitution, an F represented the *, a G the -, and H the -1, the following would result: F(F(A,H(B)),H(G(X,Y))). If the normal operands of +, -, *, /, as well as subscripted operators are eliminated, an input statement would be similar to R(S,T(M,N),Q).

R(S,T(M,N),Q) as an input string could be processed in a manner similar to Figure 19.

Figure 19

Right To Left Scan

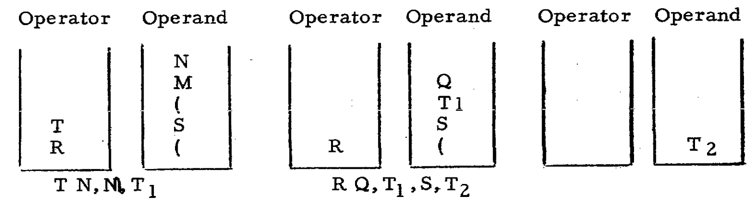


With a right to left scan,)Q)N and M would appear in the stack. At this point, T M, N, T_1 would be outputed. During the second phase the stacker would contain)QT_1S, and would require outputing of R S, T_1, Q, T_2. In the third phase T_2 would be alone in the stack.

Figure 20 illustrates the same problem with a left to right scan with two pushdown stacks.

Figure 20

Left to Right Scan

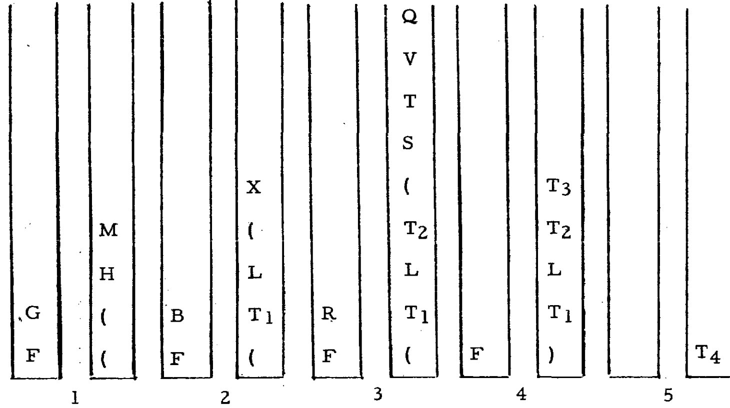


In phase one the operator stacker would contain RT, and the operand stacker (S(MN, with an output of T N, M, T_1. The second phase would have an R in the operator stack, and an operand stack of (S T_1 and Q, and output of R Q, T_1, S, T_2. In the last phase the operator stack is empty and the operand stack contains T_2.

A further example of elimination of all normal operators such as +, -, / and * would be: F(G(H,M),L,B(X),R(F,T,V,Q)). The two solutions to this problem include both types of scans; left to right with the two pushdown stacks , one for the operator and one for the operand (Figure 21) and the right to left scan (Figure 22). For both the output is written and included in functional notation.

Figure 21

Left to Right Scan



1 G M, H, T_1

2 B X, T_2

3 R S, T, V, Q, T_3

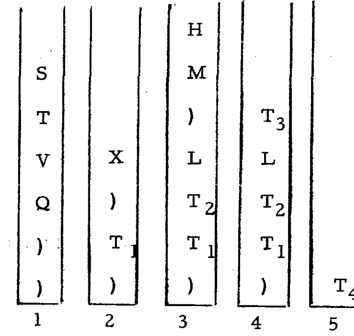
4 F T_1, L, T_2, T_3, T_4

Figure 22

Right to Left Scan

F(G(H, M), L, B(X), R(S, T, V, Q))

R to L



1 R S, T, V, Q, T_1

2 B X, T_2

3 G H, M, T_3

4 F T_3, L, T_2, T_1, T_4

Review of Polish Notation

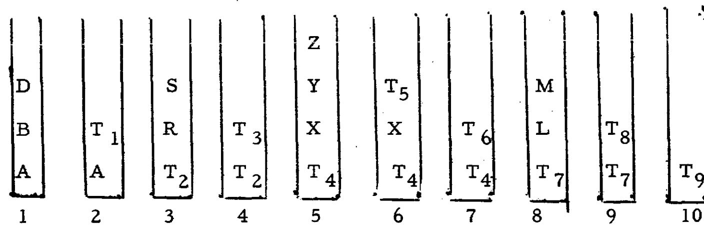
Polish notation with left to right scan will be implemented in the following example: (((A*(B-D))*(R-S))-(X/(Y-Z))*(L-M)).
The pushdown at the end of each output is shown along with the outputed functional notation required in Figure 23.

Figure 23

{[[A*(B-D)] * [R-S]] - [X/(y-z)] *(L-M)}

Right Polish, L→R scan.

ABD-* RS-* XYZ-/ - LM-*



1 -D, B, T_1

2 *T_1, A, T_2

3 -S, R, T_3

4 *T_2, T_3, T_4

5 -Z, Y, T_5

6 /T_5, X, T_6

7 -T_6, T_4, T_7

8 -M, L, T_8

9 *T_8, T_7, T_9

Scan for Language Two

Since Language Two is a higher level language than Language One, a scan for the former can be readily implemented in the latter. Figure 24 is an example of such a scan.

Figure 23

Scan For Language Two
(written in Language One)

```
START:    FREE (NAMST);
LA:       INPUT (SCRST, LEND);
LB:       NORM (SCRST, TA);
          TRAEQ (TA, ZERO, LA);
          REMNS (SCRST, ONE, CURSY);
          TRAEQ (CURSY, BLKSY, LB);
          TRAEQ (CURSY, SCLSY, LB);
          TRAEQ (CURSY, COLSY, COLSR);
          TRAEQ (CURSY, LPRSY, LPARSR);
          TRAEQ (CURSY, COMSY, COMSR);
          TRAEQ (CURSY, RPRSY, RPARSR);
          ADDSL (NAMST, CURSY);
          GO TO (LB);
COLSR:    CONCAT (COLST, NAMST, NAMST);
          OUTPUT (NAMST); FREE (NAMST);
          GO TO (LB);
LPARSR:   PUSH (FUNCST, LPRSY);
          PUSH (PARST, LPRSY);
          CONCAT (NAMST, FUNCST, FUNCST);
```

```
            FREE (NAMST; GO TO (LB);
COMSR:      NORM (NAMST, TA);
            TRAEQ (TA, ZERO, LB);
            CONCAT (NAMST, PARST, PARST);
            PUSH (PARST, COMSY);
            FREE (NAMST);
            GO TO (LB);
RPARSR:     NORM (NAMST, TA);
            TRAEQ (TA, ZERO, LD);
            CONCAT (NAMST, PARST, PARST);
            PUSH (PARST, COMSY);
            FREE (NAMST);
LD:         FREE (WSTA);
LF:         REMNS (FUNCST, ONE, SYA);
            TRAEQ (SYA, LPRSY, LE);
            ADDSL (WSTA, SYA);
            GO TO (LF);
LE:         CONCAT (LINKST, WSTA, WSTA);
            OUTPUT (WSTA);
            FREE (WSTA);
            REMNS (PARST, ONE, SYA);
LI:         REMNS (PARST, ONE, SYA);
            TRAEQ (SYA, LPRSY, LG);
            TRAEQ (SYA, COMSY, LH);
            ADDSL (WSTA, SYA);
```



```
            GO TO (LI);
LH:         CONCAT (COMST, WSTA, WSTA);
            OUTPUT (WSTA);
            FREE (WSTA);
            GO TO (LI);
LG:         CONCAT (COMST, WSTA, WSTA);
            OUTPUT (WSTA);
            FREE (WSTA);
            NORM (FUNCST, TA);
            TRAEQ (TA, ZERO, LJ);
            GENT (SYA);
            PUSH (PARST, SYA);
            PUSH (PARST, COMSY);
            ADDSL (WSTA, SYA);
            CONCAT (RESST, WSTA, WSTA);
            OUTPUT (WSTA);
            FREE (WSTA);
            GO TO (LB);
LJ:         RESTEMP (DUMMY);
            GO TO (LB);
LEND        STOP (IDENT);
```

In order to further clarify Language Two, it will be used in the last example of this section showing the contents of the main string, the functional strings, the parameter strings, and the output at each of the five stages in Figure 25. The example is:
L: $AB(DE(M, NT_1)_2, FG(RQ_3)_4)_5$;

Figure 25

L: AB(DE(M, NT)-FG(RQ));

Name String		Func. String	Parameter Str.		Output
1	NT	D E (A B (	, M ((		⊣LABEL⊢ L
2	FREE	D E (A B (	, N T , M (	, T_1	⊣FUNC L⊢DE ⊣PAR⊢ NT ⊣PAR⊢ M ⊣RES⊢ T_1
3	RQ	F G (A B (	(, T1 (		
4	FREE	F G (A B (	, R Q (, T1	(, T_2 , T_1 (	⊣FUNC L⊢FG ⊣PAR⊢ RQ ⊣RES⊢ T_2
5	FREE	FREE	FREE		⊣FUNC L⊢AB ⊣PAR⊢ T_2 ⊣PAR⊢ T_1

Language Three

Language Three uses an infix notation. Having the statement X = A*(B+R)/Q, in XTRAN, the equal sign is replaced by a ←. This is the type of notation that is used for XTRAN on the 7090, 1620, 1401 and 1410. In ALGOL a statement similar to the following three will exist:

1. L:X←A*(X-Y):
2. IF A > B, THEN X←Y; S←T.
3. IF B THEN E_1, ELSE E_2; E_3.

The last statement will be executing either E_1 and E_3 or E_2, E_3.

At this point, a similarity between the statements used in XTRAN and those used in ALGOL can be recognized.

XTRAN is a dialect of ALGOL. Almost all languages, FORTRAN, COBOL and ALGOL use an infix notation so that each operator is unique, thus eliminating ambiguities.

H. COMPILER LOGIC

Generally, there are five phases to compilers. They are:

Phase 1 - HTR - Hardware To Reference

Phase 2 - Pre-scan

Phase 3 - Scan

Phase 4 - Macro expander

Phase 5 - Assembly program

Phase 1 - Hardware To Reference, converts the external representation to the internal representation. With the example A*B, referring the Dictionary of Internal Identifiers for Symbols (Figure 26) a transformation is made into 178 52 180. The internal representation has 0 through 177 as operators, 178 through 239 as alphabetic characters, 240 through 254 as integers, and 1,000 through 4,999 as names. Before leaving the HTR phase, consider a more complex example that can be followed through the subsequent phases. The example: RSA. LA. RB. UP. RSA would be translated as: 214 218 178. LA would be translated as 80; RB as 214 180; and UP as 35. Therefore, leaving the hardware to reference phase would be the string of numbers: 214 218 178 80 214 180 35.

Phase 2 - The pre-scan converts the names to single integers and also eliminates ambiguity within parenthesis. 214 218 178 would, in this phase, be grouped together and given an arbitrary number, such as 1,000. 214 180 would also be grouped together and given another arbitrary number such as 1001. The string 1000 80 1001 35 would have been developed.

Phase 3 initiates another scan; Phase 4 is the macro expander which includes such things as multiply; and Phase 5, the final assembly.

Figure 26

Dictionary of Internal Identifiers for Symbols - 7090 XTRAN

I for Internal Symbol	Internal Representation	External Representation	Comment
	000	. NUL.	Null symbol
	001	. COMMENT.	Comment symbol
	001	'	Comment symbol
	002	. ARRAY.	Declaration
	003	. SWITCH.	Declaration
	004	. INIT.	Declaration
	004	. INITIAL.	Declaration
	004	. INITIALIZE.	Declaration
	005	. SWITCHVAR.	Type declaration
	008	. REAL.	Type declaration
	009	. INTEGER.	Type declaration
	009	. SYMBOL.	Type declaration
	010	. LOGICAL.	Type declaration
	010	. BOOL	
	011	. STRING.	Type declaration
	016	. SC.	Semicolon
	019	. COL.	Colon
	019	. LAB.	Colon
I	022	. TLAB.	
	024	. THEN.	
	027	. STEP.	
	028	. WHILE.	
	029	. TILL.	Until
	030	. REPEAT.	
	032	(	Left parenthesis
	034	. LB.	Left bracket
	035	. UP.	Exponentiation
	036	. BEGIN.	Left brace
I	045	. UM.	Unary minus
	045	. UMIN.	
	046	. AB.	Absolute value
	048	+	
	050	. DEFINE.	
	051	. MD.	Modulo
	052	*	
	054	-	
	055	/	
	064	,	
	065	. RB.	Right bracket
	066	)	Right parenthesis
	069	. END.	Right brace
	070	. XB.	

I for Internal Symbol	Internal Representation	External Representation	Comment
	073	. BLANK.	Blank
	073		Blank
	073	. AN.	Blank
	074	. PROCEDURE.	
	075	. OP.	Operation
	075	. OPOP.	
	076	. LC.	Location
	076	. LCOP.	
	077	. RETURN.	Return
	077	. RTN.	Return
	080	. LA.	Left assign
	080	. AS.	Left assign
	082	=	Relational
	083	. UE.	Relational
	084	. GT.	Relational
	085	. GE.	Relational
	086	. LT.	Relational
	087	. LE.	Relational
	096	. NOT.	Logical
	097	. OR.	Logical
	098	. LOG*.	Logical
	098	. AND.	Logical
	099	. IDENTICAL.	Logical
	099	. IDENT.	
	099	. IDT.	Logical
	102	. IMPLIES.	Logical
	102	. IMP.	Logical
I	104	. COMPA.	
	105	. COMPUTE.	
	112	. GOTO.	
	112	. GO TO.	
	112	. GO.	
	114	. FOR.	
	115	. IF.	
	116	. PARSEP.	Parameter Separator
	116	. PSEP.	Parameter Separator
	117	. ELSE.	
	119	. LOOP.	
	119	. DO.	
	121	. RA.	Right assign
	121	$	Right assign
	121	. EQCO.	Right assign
	125	. BSOS.	Begin SOS
	126	. ESOS	End SOS
	127	. MACSEP.	Macro Separator
	127	. MSEP.	Macro Separator

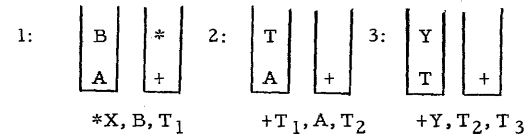
I for Internal Symbol	Internal Representation	External Representation	Comment
I	128	. THENA.	
I	129	. CM.	
I	130	. F.	
I	131	. P.	
	132	. N.	
	133	. X.	
I	134	. XB.	
	135	. A.	
	136	. C.	
	137	. YIELDS.	
I	138	. YIELDSA.	
	139	. OTHERW.	
	139	. OTHERWISE.	
I	140	. FORA.	
I	141	. FORB.	
I	142	. FORC.	
I	143	. FWHILE.	
	144	. INFOR.	
I	145	. LOOPE.	
I	146	. PAR.	
I	147	. RES.	
I	148	. PL.	
I	149	. FL.	
I	150	. SWIL.	
I	151	. SWM.	
I	152	. AST.	
I	153	. TATS.	
I	154	. SWAIL.	
I	155	. ATL.	
I	156	. DLAB.	
I	157	. YLB.	
I	158	. YXB.	
I	159	. YF.	
I	160	. YP.	
I	161	. DLA.	
I	162	. ILA.	
I	163	. SDLA.	
I	164	. SILA.	
	178	A	
	180	B	
	182	C	
	184	D	
	186	E	
	188	F	
	190	G	
	192	H	
	194	I	
	198	J	

I for Internal Symbol	Internal Representation	External Representation	Comment
	200	K	
	202	L	
	204	M	
	206	N	
	208	O	
	210	P	
	212	Q	
	214	R	
	218	S	
	220	T	
	222	U	
	224	V	
	226	W	
	228	X	
	230	Y	
	232	Z	
	240	0	
	241	1	
	242	2	
	243	3	
	244	4	
	245	5	
	246	6	
	247	7	
	247	8	
	249	9	
	255	. EOP.	(End of Program)

1620 Scan

The 1620 scan is from left to right. The formula A+B*X+Y appears in the first phase with the operand stack containing AB, the operator stack +*. This will output the statement: *X, B, T_1. The termination of the second phase finds the operator stack with +, and the operand stack A T_1, with output of + T_1, A, T_2. The third phase would have in the operator stack + and the operand stack T_2Y, and would have outputed +Y, T_2, T_3.

Figure 27



I. SWITCH METHOD/SINGLE ADDRESS OUTPUT

Taking the formula X+Y+Z+R*S+B; the following output would be derived, turning a switch on if there is something in the accumulator, and a switch off if there is nothing in the accumulator. Initially, nothing is in the accumulator (See Figure 28). The operand stack would contain XY with a + in the operator stack. Output from this phase would be CLAX, ADD Y.

The second phase would hold Z in the operand stack, + in the operator stack with the switch ON. The output is ADD Z.

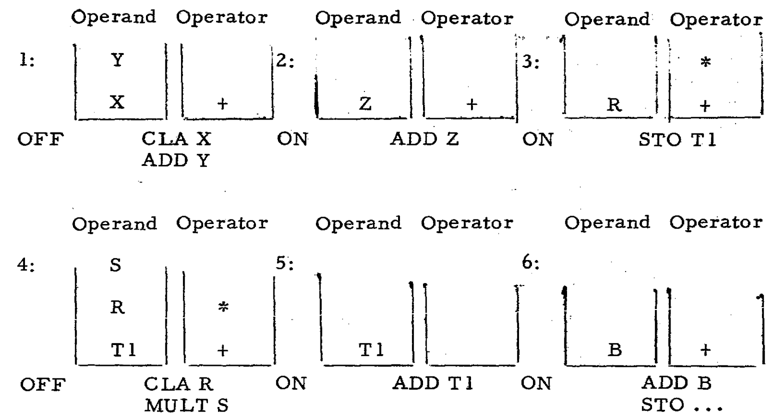
The third phase contains R in the operand stack with + and * in the operator stack. Since the switch is ON, the output would be STO T_1 which stores T_1 in the bottom of the operand stack.

During phase 4 the operand stack holds T_1 RS and in the operator stack +* with the accumulator switch OFF. The output is CLA R MULT S and the switch is turned ON.

For phase 5 only T_1 will appear in the operand stack and + in the operator stack. Output would be ADD T_1 with the switch turned ON.

The last phase would have B in the operand stack and + in the operator stack. Output is ADD B with the switch ON, STO.

Figure 28



J. FORCING CODES AND ROUTINES

Introduction

The 1620 XTRAN Compiler uses the technique of forcing tables for compilation purposes. Since the hierarchy of operators must be free of all ambiguities, one method of elimination would be placing a numeric value on each operator and devising a rule for forcing the operator. For instance, assigning the value of 10 to + and 5 to * for the example X+Y*Z with the pointer at *, the multiplication would not be forced.

In the example X * Y + Z the multiplication should be forced and in X + Y + Z all should be forced.

These rules could be condensed into the single statement of: force the operator if the right value of ϕ_R is greater than or equal to the less value of ϕ_L. By consulting a predetermined forcing table similar to Figure 29, ambiguities would be eliminated from the language.

Forcing routines are included in Figure 30.

Figure 29

Forcing Table

Operator	Left Value	Right Value	Force Code	Operator	Left Value	Right Value	Force Code
+	10	10	1	.COMPUTE.			
-	10	10	1	.COMPA.			
*	5	5	1	.WHILE.			
/	5	5	1	.IF.	0	0	11
↑	5	4	1	.THEN.	0	60	12
=	12	12	1	.THENA.	60	0	13
≠	12	12	1	.ELSE.	60	60	14
<	12	12	1	.N.	2	0	16
≥	12	12	1	[	49	0	18
>	12	12	1	]	0	49	0
≤	12	12	1	.A.	2	0	2
.OR.	20	20	1	.C.	2	0	2
.AND.	15	15	1	.F.	50	0	21
U	4	0	2	.P.	50	0	22
.NOT.	14	0	2	.YF.	50	0	23
←	60	59	3	.YP.	50	0	24
:				,	50	49	25
.GOTO.				.CM.	0	50	0
(	50	1	6	.FOR.	0	0	36
)	1	50	0	.FORA.	59	0	37
{	65	0	6	.FORB.	49	0	38

Operator	Left Value	Right Value	Force Code	Operator	Left Value	Right Value	Force Code
}	0	65	0	. FORC.	60	0	41
;	60	60	7	. LOOP.	60	60	45
				name	0	0	0

Figure 30

Forcing Routines

Force Code 1 (for binary operators)

STO: $\alpha_1 N_1 \varphi_L N_2 \varphi_{\Delta R} \alpha_2$ (Original string)

STN: $\alpha_1 T_i \varphi_{\Delta R} \alpha_2$ (New string)

STP: $\dashv \varphi_L \vdash N_1, N_2, T_i$ (Output string)

Replace $N_1 \varphi_L$ by T_i in the string.

Force Code 2 (for unary operators)

STO: $\alpha_1 \varphi_L N_1 \varphi_{\Delta R} \alpha_2$

STN: $\alpha_1 T_i \varphi_{\Delta R} \alpha_2$

STP: $\dashv \varphi_L \vdash N_1, T_i$

Replace $\varphi_L N_1$ by T_i in the string.

Force Code 3 (for $\leftarrow$)

STO: $\alpha_1 N_1 \varphi_L N_2 \varphi_{\Delta R} \alpha_2$

STN: $\alpha_1 N_2 \varphi_{\Delta R} \alpha_2$

STP: $\dashv \varphi_L \vdash N_1, N_2$

Remove $N_1 \varphi_L$ from the string.

Force Code 4 (for :)

STO:

STN:

STP:

Force Code 5 (for GOTO)

STO:

STN:

STP:

Force Code 6 (for (and {)

STO: $\alpha_1 \varphi_L N_1 \varphi_{\Delta R} \alpha_2$

STN: $\alpha_1 N_{\Delta 1} \alpha_2$

STP: nothing

Remove φ_L and φ_R from the string, outputs nothing; moves pointer back one.

Force Code 7 (for ;)

STO: $\alpha_1 N_1 \varphi_L \alpha_2 \varphi_{\Delta R} \alpha_3$

STN: $\alpha_1 \alpha_2 \varphi_{\Delta R} \alpha_3$

STP: nothing

Remove $N_1 \varphi_L$ from the string.

Force Code 8 (for COMPUTE)

STO:

STN:

STP:

Force Code 9 (for COMPA)

STO:

STN:

STP:

Force Code 10 (for WHILE)

STO:

STN:

STP:

Force Code 11 (for IF)

STO: $\alpha_1 \varphi_L \alpha_2 \varphi_{\Delta R} \alpha_3$

STN: $\alpha_1 \alpha_2 \varphi_{\Delta R} \alpha_3$

STP: nothing

Remove φ_L from string.

Force Code 12 (for THEN)

STO: $\alpha_1 N_1 \varphi_L \alpha_2 \varphi_{\Delta R} \alpha_3$

STN: $\alpha_1 GL_i$ THENA $\alpha_2 \varphi_{\Delta R} \alpha_3$

STP: $\dashv \varphi_L \vdash N_1, GL_i$

$\dashv$ THEN $\vdash$ is TRA to GL_i if N_1 is false.

Replace $N_1 \varphi_L$ by GL_i THENA in string.

Force Code 13a (for THENA when φ_R is ELSE)

STO: $\alpha_1 GL_i \varphi_L N_1 \varphi_{\Delta R} \alpha_2$

STN: $\alpha_1 GL_j \varphi_{\Delta R} \alpha_2$

STP: $\dashv$ TLAB $\vdash GL_i, GL_j$

$\dashv$ TLAB $\vdash$ means TRA to GL_j with the label GL_i attached to the location following the TRA instruction.

Replace $GL_i \varphi_L N_1$ by GL_j in the string.

Force Code 13b (for THENA when φ_R is not ELSE but ;, }, etc.)

STO: $\alpha_1 GL_i N_1 \varphi_{\Delta R} \alpha_2$

STN: $\alpha_1 N_1 \varphi_{\Delta R} \alpha_2$

STP: $\dashv$ LABEL $\vdash GL_i$

Remove $GL_i \varphi_L$ from string.

Force Code 14b (for ELSE when φ_R is not THEN)

STO: $\alpha_1 GL_i \varphi_L N_1 \varphi_{\Delta R} \alpha_2$

STN: $\alpha_1 N_1 \varphi_{\Delta R} \alpha_2$

STP: $\dashv$ LABEL $\vdash GL_i$

Remove $GL_i \varphi_L$ from the string.

Illustrative Solution

Consider the problem of:

COMPUTE A←A+B↑2 WHILE A $<$ 100

with a general form of:

COMPUTE Σ WHILE C

A folution for the problem by developing a forcing table (Figure 31), applicable codes (Figure 32) follows. An example for purposes of proof is also included as Figure 33.

Figure 31

Force Table

	LV	RV	FC
COMPUTE	0	60	8
WHILE	60	59	10
COMPA	60	60	9

Figure 32

Force Codes

Force Code 8 - COMPUTE

STO: $\alpha_1\ f_L\ N_1\ f_{\uparrow R}\ \alpha_2$

STN: $\alpha_1\ GL_1\ COMPA\ N_1\ f_{\uparrow R}\ \alpha_2$

STP: ⊣ LABEL ⊢ GL_1

Replace f_L with GL_1 COMPA

Force Code 9 - COMPA

STO: $\alpha_1\ N_1\ f_L\ N_2\ f_{\uparrow R}\ \alpha_2$

STN: $\alpha_1\ f_{\uparrow R}\ \alpha_2$

STP: ⊣ COMPA ⊢

STP: ⊣ f_L ⊢ N_2, N_1

Remove $f_L\ N_1\ N_2$ from string.

Force Code 10 - WHILE

STO: $\alpha_1\ N_1\ f_L\ N_2\ f_R\ \alpha_2$

STN: $\alpha_1\ N_{\uparrow 2}\ f_R\ \alpha_2$

STP: none

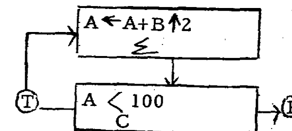
Removes $N_1\ f_L$ from string.

Figure 33

Proof

COMPUTE A← A+B↑2 WHILE A$<$100;

General form: COMPUTE Σ WHILE C



```
COMPUTE A←A+B↑2 WHILE A < 100;          ⊣LABEL⊢GL1
GL1 COMPA A←A+B↑2 WHILE A < 100;        ⊣↑⊢ B,2,T1
GL1 COMPA A←A+T1 WHILE A < 100;         ⊣+⊢ A,T1,T2
GL1 COMPA A←T2 WHILE A < 100;           ⊣←⊢ A,T2
GL1 COMPA T2 WHILE A < 100;             ⊣<⊢ A,100,T3
GL1 COMPA T2 WHILE T3;   ⊣              ⊣COMPA⊢ T3,GL1
```

⊣COMPA⊢ will generate coding to test T_3 and branch to GL_1 if true, i.e. TRATRUE T_3, GL_1.

XTRAN Compiler

With this background, an exerpt from an EXTRAN compiler may be examined in Figure 34.

Figure 34

Excerpt From XTRAN Compiler

```
        SWITCH BRANCH ← (LCV01, LCV02, LCV03, LCV04, LCV05,...,
        LCV50);
L1      GETXT (SR, LA);
        ADDSL (WST, SR);
L5      NWST ← NORM (WST);
        IF NWST < 3 THEN GOTO L1;
        IF PTR < 3 THEN PTR ← 3;
        IF PTR > NWST THEN GOTO L1;
L6      SL ← GETNS (WST, PTR-2);
        SR ← GETNS (WST, PTR);
```



```
IF IMIN ≤ SL
THEN GOTO LCV00 $ELSE LVSL ← LV [SL] ;
IF IMIN ≤ SR
THEN RVSR ← 0 ELSE RVSR ← RV [SR] ;
IF LVSL ≤ RVSR
THEN CVSL ← CV [SL]
ELSE CVSL ← 0;
IF CVSL = 0 THEN GOTO LCV00;
GOTO BRANCH [CVSL] ;
```

K. BACKUS NORMAL FORM

John Backus, who wrote the first FORTRAN, developed a method of expression called Backus Normal Form. This mode of expression, or alphabet, is used in ALGOL. The alphabet BNF is probably the most exciting development in metalanguages today — (a metalanguage is a language to describe another language).

Since the syntax of ALGOL is defined in BNF, the symbol definition should be examined. The alphabet of BNF consists of:

< > class of objects

: : = defined

| or

So that a statement would take the following formats:

<letter> : : = A|B|C|D . . . |Z

<operator> : : = +|-|< |/|↑

<operand> : : = <letter>|<operator> <operand> <operand>

<expression> : : = <operand>

The syntax of ALGOL, expressed in BNF, would be of the format:

<name> : : = <letter>|<name>[<letter>|<digit>]

<legit> : : = <letter>|<digit>

<name> : : = <letter>| <name> digit>

<operand> : : = <letter> | <function>

<function> : : = <letter> (<list>)

<list> : : = <operand>|<list> , <operand>

III. HEURISTIC COMPILERS

A. INTRODUCTION

Need

During the past decade there has been enormous progress in the development of higher level programming languages for instructing computers. Through the development of FORTRAN and ALGOL, COBOL, and the list processing languages of IPL and COMIT, the labor of programming has been reduced several orders of magnitude.

Yet, programming a computer to perform a complex task, is still very much more intricate and tedious than instructing an intelligent and trained human being for the same task. The intelligent and knowledgeable human does not have the literalness of mind that is so characteristic of the computer, and so exasperating in a programmers interaction with it. He supplies facts from his own store of knowledge that his instructor neglected to give him. Given statements of objectives and broad functional terms he can apply to his problem solving powers to filling in the details of method. Meaning and intent are interpreted when he is confronted with the vagueness and informality of natural language.

Experiments in heuristic compilers are aimed at further bridging the gap between the explicitness of existing computer programming languages and the freedom and flexibility of human communication.

Definition

Intelligent problem solving, whether by man or by machine, implies selective rather than just rapid behavior. Humans achieve this selectivity through heuristics-principals that, on the average contribute to a reduction of search for problem solving. Heuristic programming then, is the construction of computer problem solving programs whose behavior is similarly organized.

Implications

There are several implications of the definition of heuristic programming that should be noted: a concern with exploiting partial information in a problem situation where there is no guaranteed way of using that information to find a best solution; preparing a procedure (expressed as a computer program) to make effective use of this partial information; and it is a body of knowledge built up through experience with specific examples lacking as yet an underlying analytical framework.

Using these implications as a guide, a definition of heuristic programming could be: many decisions which are made in an environment which may be characterized by: 1) lack of a feasible

guaranteed method of reaching the best solution; 2) a description of the solution in terms of acceptability of characteristics; and 3) sufficiently many possibilities to make complete trial and error infeasible. Heuristic programming is concerned with constructing decision paradigms (computer programs) emphasizing the use of selectivity rather than computational speed.

Applications

A heuristic program has been prepared to balance production of assembly lines in a factory. Balancing the line is the act of assigning jobs to workers, so that a given production rate can be met with minimum men. The Behavioral Theory of the Firm Project at Carnegie Institute of Technology is developing a simulation of the price and quantity decisions of a department store buyer; a heuristic program is now being prepared to simulate investment activities under a trust fund; and while no formal result has yet been published, two separate attempts to construct heuristic programs for the job-shop scheduling problem are under way.

Design Representation

One distinction between the restricted, relatively simple task of "coding" and the more difficult task of "programming" is that the latter may encompass the selection or design of an appropriate problem representation, and the former does not. Consideration must be given to the requirements in the design or selection of a representation, and what is needed to give the heuristic compiler the capability and capacity to grapple with such design and selection tasks.

Designing a suitable representation, is in effect finding an isomorphism. A description was defined in terms of certain elements, relations between elements and processes. The programmer has to find a set of elements, relations and processes defined in a heuristic compiler that are isomorphic with the required elements, relations and processes (i. e., the proper sub-set).

Ill-structured Problems

An ill-structured problem has generally three characteristics; (1) many of the essential variables are not numerical at all, but symbolic (2) the goal is vague and non-quantative, and (3) computational algorithms are not available.

Allocating marketing expenditures among sales and promotional efforts, preparing a compiler for business applications, playing chess, are ill-structured, if not perverse, problems.

Currently researchers are studying human methods of dealing with ill-structured problems. Theorm proving in geometry, chess, human discrimination, learning, human concept formation and language translation are all being observed. These problem solving activities have been called "heuristic problem solving", to emphasize the importance of principles or rules of thumb that tend to discover acceptable solutions more efficiently in most cases than do exhaustive methods.

Pre-structuring

Present digital computers require an explicit statement of the procedure to be carried out and of the data to be processed. But at any state in developing a heuristic procedure, knowledge of the problem itself is incomplete. Methods of describing the problem and procedures for solving it demand easy revision of both procedures and data.

The common answer to this dilemma is to pre-structure the problem environment, and then within the resulting and often stringent requirements, produce a solution procedure. The danger of this, however, lies in the loss of flexibility thus introduced.

Language Characteristics

Certain requirements for a language to express ill-structured problems are: 1) freedom from memory allocation problems; 2) the ability to define and redefine complex concepts; 3) to make use of these concepts by name in defining further concepts; 4) the ability to express concepts which are not meaningful until some problem solving has occurred; 5) the ability to introduce useful notations; 6) the ability to associate information in an easily recoverable manner; and 7) the ability to change sections of the decision process independently without problems of interrelations with other sections.

A number of computer languages have been developed which begin to provide some of this desired freedom. The IPL series, LISP, FORTRAN, and COMIT, exemplify this approach to computer utilization.

The specific features of the IPL series include: (1) organization of storage into list structures; (2) use of the description processes to associate with structures new information or to delete previously associated information; and (3) hierarchial nature of control which allows for both a natural hierarchial organization and specification of the processes, and for recursive definition.

B. INFORMATION PROCESSING LANGUAGE V

Introduction

IPL-V is designed for list processing and symbol manipulation, the fifth of a series of languages. Its heavy use is among scientists in the fields of artificial intelligence and simulation of cognitive processes.

Compiler Capabilities

The compiling routines of IPL-V accepts the task of writing heuristic programs on the basis of certain information provided to it. The routines differ with respect to their methods of formulating or representing the problem, for each uses a different state language. At present, three compiling routines exist: 1. the SDSC (U140), DSCN (U134), and General Compiler (U135).

The SDSC Compiler (State DeSCription) accepts as an input a description of the contents of the relevant computer cells before and after the routine to be completed has been executed. It produces an IPL-V routine that will transform the input state description into the output state description.

The DSCN Compiler (DeSCriptive Name) employs as its input an imperative sentence definition of the routine to be compiled. DSCN produces an IPL-V routine that is a translation, in the interpretive language, of that definition.

The third compiler, the General Compiler, is an executive routine that can use the SDSC, the DSCN and other sub-routines. The input information can be stated in any form of several representations, i. e., those appropriate to SDSC or DSCN, and selects sub-routines to produce the desired IPL-V code.

From a logical standpoint the Heuristic Coder could be described as a single program whose executive routine is the General Compiler and which contains the SCSC compiler and the DSCN compiler as sub-routines.

IPL-V Features

In the IPL-V language cells may have lists associated with them, and the primitive processes find their operands in a communication cell and its related list. The communication cell is also called an accumulator because it has many of the functions of the accumulator in a standard computer. Processes with the exception of tests, place outputs in the accumulator and in its pushdown lists. Tests in

IPL-V record their result by placing a plus or minus in a special cell called the Signal Cell.

Lists in the IPL-V memory may have description lists associated with them. A description list consists of pairs of symbols in which the first symbol of each pair designates an attribute, the second symbol designates the value of that same attribute. The value may be a simple symbol, or it may be itself a list.

Values of attributes of objects may themselves have descriptions. Representations of routines or programs in memory will take the form of description lists, each routine having one or more of the attributes of IPL name, IPL definition, functional description, state description and flow diagram. The values of these attributes will themselves be described, or have description lists associated with them.

The SDSC Compiler

A computer routine can be defined by specifying the changes it produces in the storage location it affects, or by specifying the before and after conditions of these storage registers. A definition of this kind is not univocal. There will generally be many programs not all equally efficient or elegant that will do the same work, and the SDSC Compiler attempts to find one to accomplish a given task.

The DSCN Compiler

Instead of specifying the before and after condition of the computer cells a routine is designed in terms of a function it performs. The DSCN Compiler searchs a list of available (compiled) routines to find one whose DSCN in as similar as possible to the DSCN of the routine to be compiled. Secondly, an analysis is performed to transfer the compiled routine that has been found into the new routine. When the compiler finds differences, it searches for an operator relevant to removing the differences, and the resulting program will be identical with that obtained by the SDSC Compiler.

The General Compiler

The General Compiler is an executive routine whose task is to compile a routine from information in any of the forms already described (SDSC and DCSN). It takes as its input the name of the routine to be compiled. Associated with this name (on its description list) is the information to be used on the compiliation.

A ROUTINE is a description list containing values of some subset of the following attributes:

1. IPLN, IPL Name (X25). The value of this attribute is a description list that names a region and a location in the region.

2. JDEF, IPL-V definition (X22). The value of this attribute is list of IPL-V instructions. Each is in the form of a description list which describes the corresponding IPL-V word that defines the IPL-V routine with specified names.

3. DSCN, DeSCriptive Name (X20). This is an imperative sentence encoded as a list structure that describes the process assigned by JDEF.

4. SDSC State DeSCription (X240). The value of this attribute is a list structure that describes the state of the IPL computer before and after the routine in question has been executed. Only changes are mentioned explicitly.

5. FLWD FLoW Diagram (X267). This is a list structure that gives a flow diagram corresponding to the JDEF, or job definition.

6. ASOJ ASsOciated J Definition (X23). The value of this attribute is the IPL name of a routine associated in a manner to be described later with a given routine.

A compiled routine is a routine that has its JDEF, or an IPL-V definition. The problem of compiling a routine can be stated as follows: Given a routine without a JDEF (the present object or definition) find the corresponding routine with the JDEF (the goal' object), where "corresponding" means that the compiled routine has the same SDSC, or DSCN, as the given routine.

C. COMIT

COMIT is a problem oriented language for symbol manipulation, especially designed for dealing with strings of characters. Facilities are available for easy re-arrangement, insertion or deletion of characters of strings, and complex searches involving pattern-matching in terms of incomplete descriptions of in terms of context. A facility for dictionary lookup and flexibility input and output facilities are provided.

COMIT was originally designed for mechanical translation and linguistic research to provide the linguist who had no previous programming training with immediate access to a computer. It is based on Chomsky notation that is familiar to linguists, with a number of additional features added making it a useful and convenient programming language. COMIT is not a language in which one "programs in English"--it is a highly abbreviated notation.

The COMIT system, as implemented for the IBM 709 and 7090, consists of about 16,000 instructions. It compiles the source program into an internal language which then runs interpretively at a high level. This program was SHARE distributed in September 1961, and was programmed at the Massachusetts Institute of Technology by a joint effort of the Research Laboratory of Electronics Mechanical Translation Group and the Computation Center.

D. SAINT

Introduction

The IBM 7090 was programmed to solve elementary symbolic integration problems at approximately the level of a good college freshman. "SAINT", an acronym for Symbolic Automatic INTegrator, performs indefinite integration, and definite and multiple integration when these are trival extensions of indefinite integration. It uses many of the methods and hueristics of students attacking the same problems.

Pattern Recognition

Pattern recognition plays an important role, for it consumes much of the program and programming effort. It is used frequently and with great variety in determinations involving standard forms, algorithm-like and hueristic transformations, and relative cost estimates. Finally, it consumes much of the time in solving integration problems.

E. GIT

GIT, an acronym for Graph Isomophism Tester, has been written in the COMIT language described in Section C. The graph isomorphism problem may be stated as follows: Given two direct line graphs determine whether or not they are isomorphic, and if they are, specify a transformation carrying the first graph into the second.

The problem of ascertaining whether a pair of directive line graphs are isomorphic is one for which no efficient algorithmic solution is known. Because a straight forward enumerative algorithm might require 40 years of running time on a high speed computer to compare 2-15 node graphs, a more sophisticated approach is required. The Graph Isomorphism Tester incorporates a variety of processes that attempt to narrow down the search for an isomorphism with no one scheme relied upon exclusively for a solution. The problem is designed to avoid excessive computation along fruitless lines.

Another program of the same type has been written at the Harvard University Computation Lab and is a more generalized version of the same graph isomorphism problem. Its' quite similar in approach to GIT and it has been implemented with the 7090 assembly language rather than COMIT.

F. LINE BALANCING

Introduction

A hueristic program for assembly line balancing has been developed. The assembly line balancing problem, like many combinatorial problems, has not been solved in a practical sense by advanced mathematical techniques, and this approach does not guarantee an optimum solution. The ultimate measure of a hueristic program is whether it provides better solutions more quickly and/or less expensively than other methods.

Problem Definition

Given a production rate (or equivalently, a cycle time) the minimum number of work stations (or operators) consistent with the time in ordering, constraints of the product should be developed. The assembly line balancing problem concerns a set of elemental tasks where each requires a known operation time per unit of product independent of when performed, and where partial ordering exists.

A optimum of solution of the problem consists of an assignment of elemental tasks to work stations so that each task is assigned to one and only one work station; the sum of the times of all tasks assigned to anyone station does not exceed some pre-set maximum of cycle time; the generated stations can be ordered such that the partial orderings among tasks are not violated; and the number of work stations is minimized.

Some difficulties associated with line balancing are determination before solution the minimum number of operators, minimization of the variation in work load among stations in evaluating possible solutions; and juxtaposition of a zoning constraint. (Zoning is the division of the set of elemental tasks into overlapping subset corresponding to physical constraints on the assembly operation).

Zoning of an assembly line may be determined by the position of the product on the conveyer, the layout of the production facility, or both.

Hueristic Procedure

The hueristic procedure for assembly line balancing is:

a) Repeated simplification of the initial problem. This is accomplished by grouping adjacent elemental tasks into compound tasks.

b) Solution of the simpler problems created by assigning tasks to work stations at the least complex level possible. The compound tasks are broken into their elements only when required for solution.

c) Smoothing the resulting balance. Tasks are transferred among work stations until the distribution of assigned time is relatively even.

Regrouping Procedures

Five regrouping procedures were used in the hueristic line balancing procedure: direct transfer, trading, sequential grouping, completing grouping and exhaustive grouping. Direct transfer was limited to two component involvements. This method transferred elements from one component to the other and by this means reduced the number of men required by a straightforward totaling of whole men. Trading also applied only to two components and assumed that direct transfer had been attempted without success. Trading regrouped by shifting an element larger than the acceptable limit from one component in exchange for smaller elements (in a set relationship with the first element shifted) from the other component. Sequential grouping is exploited for several components, and its procedure is to construct an acceptable work station from the front of the given group of components.

The remaining two regrouping procedures, complete grouping and exhaustive grouping attempt to completely solve the remaining sub-problems, and are "last ditch" methods at any particular level. Complete grouping attempts to construct work stations until all task elements are grouped from the front and (if required) the back of the component group. If at any time the method cannot construct the station (i. e., the remaining elements total less than can be handled by the remaining men) the method fails. Exhaustive grouping generates all possible first work stations, then all possible work stations following for each of the first stations. The comparatively large amount of effort required to do an exhaustive grouping dictates that this procedure be used when only two men are assigned.

Conclusion

The hueristic line balancing procedure is not economically competitive when measured against the dollar-per-hour cost of line balancing by the industrial engineer, but a true evaluation of the method should consider: the possibility of averaging fewer men required along the line, the value of quick production of balances at a large number of production rates; and the value of releasing industrial engineers to do other, more creative analytic work.

IV. SYMBOL MANIPULATIVE LANGUAGES

A. ALGOL

ALGOL was designed by an international committee. It is a language for use in scientific type problems hence would compete with FORTRAN. It was designed to have all the features that a user might want, and practical considerations of implementation were not given much weight. This contrasts with FORTRAN, which was designed with the IBM 704 in mind, and which had artificial restrictions so that efficient object code could be produced. (ALGOL was designed with fast compilation in mind, but with little emphasis on efficient object programs.)

The first version of ALGOL was produced in 1958 and the second in 1960. The 1960 version had the more extreme features, and it is this version that will be discussed.

Although IBM is not producing any ALGOLs, it is of concern in competitive situations. Burroughs, CDC, UNIVAC, and RCA are all producing ALGOLS (none of these contain all the features of ALGOL). ALGOL is very popular at universities, with Duke, Michigan, and Princeton all having ALGOLS. The languages MAD , JOVIAL, NELIAC, and XTRAN are all similar to 58 ALGOL. The "intellectuals" support ALGOL. Many articles concerning it appear in the literature. It is especially popular in Europe, since the universities are more influential there. McCracken is publishing an "Introduction to ALGOL" which should increase its popularity here. Much of the published literature is difficult to follow, but this probably will not be true of McCracken's book.

One of the reasons that IBM is not implementing ALGOL is that SHARE seems content with FORTRAN, which is being constantly improved. (IBM did cooperate with SHARE in producing a large part of an ALGOL compiler for the 709/7090). Both IBM and its customers have a large vested interest in FORTRAN because of existing compilers and library programs. (Our competitors pruduce FORTRAN compilers also). However, pressure will probably continue because ALGOL is more powerful than any existing or proposed FORTRAN.

In looking at ALGOL, two aspects should be borne in mind. The first concerns its more standard features, which in many cases are extremely desirable, and the second its rather extreme features, which tend to make it impractical.

There are three forms of the ALGOL language. The reference language is the one used in most publications, and the one we will use. The hardware language is one that each manufacturer determines is to be used on his equipment. The publication language allows things

like putting subscripts below the line and exponents above the line. We will not discuss this.

ALGOL uses a number of special operators such as GOTO, IF, THEN, DO. These are to be considered as though they were single characters. A hardware implementation might be to use an "escape symbol" such as a period, which would be keypunched as .GOTO. .IF. .THEN. .DO. This paper will use underlines, although in the journals they are indicated by boldface. Similar to these are symbols such as which might be keypunched as .GE. .LE. .GT. This is different than the reserved word idea used in COBOL. With the scheme mentioned in THEN, GE, etc. can be used freely as names in the program. The only restrictions on names are for a group of eight commonly used mathematical functions such as sin, sqrt, etc.

Operators are in general not ambiguous in ALGOL. For assignment the := is used, reserving the symbol = for equality. For example, A:=B means to take the value of B and store it in A, while IF A=B has the obvious meaning. F(X) means to execute the function F using X as an argument. On the other hand, the Ith element of an array A is indicated by A [I]. FORTRAN has ambiguities in it, and consequently the compiler is slowed down by it.

Statements are punched in free form in ALGOL. There is no concern with card columns, blanks, or continuation cards. A semicolon is used to indicate the end of a statement. Also statements may be grouped to form compound statements by preceding them by BEGIN and following them by END. An example might be BEGIN A:=B+C; X: =Y; M: =N-1 END. BEGIN can be thought of as { and END as } . Any number of statements can be included in a compound statement, and a compound statement can be placed anywhere in the program that a statement is called for. Hence compound statements can be nested inside compound statements without any restriction as to depth of nesting.

There is an IF THEN ELSE statement. It has the form: IF Boolean expression THEN Statement ELSE Statement. If the Boolean expression is true, the statement following the THEN is executed; if it is false, the statement following the ELSE is executed. The program, in either case, proceeds to the next statement. An example follows:
IF A=B THEN BEGIN X: =R; S:=T END ELSE Q:=L;

The iterative type statement in ALGOL is the FOR statement. Examples will illustrate it.

FOR I: = 1, 5, 12, 13, 152 DO Statement. The statement following the DO will be executed five times, the first time I will have the value 1, the second time 5, the third time 12, etc.

FOR I: = 1 STEP 2 UNTIL 50 DO Statement. The statement following the DO will be executed repeatedly with I taking on the values 1, 3, 5, 7, ... 49. (This is quite similar to the FORTRAN DO.)

FOR I: = I+1 WHILE A > O DO Statement. This statement will be executed repeatedly until the condition A > O becomes false, each time it is executed the value of I will be increased by 1. (For this example to make sense, I should have been assigned a value before this statement is reached in the execution of the program.) Finally, all three of these types can be combined into one FOR statement. The following example illustrates:

FOR I: = 1, 4, 6 STEP 1 UNTIL 9, 15 WHILE A=B, 22 DO Statement. The statement will be executed with I taking on the values 1, 4, 6, 7, 8, 9 and repeatedly for the value 15 until A=B becomes false, and then finally for the value 22.

The following FOR statement is permitted:

FOR I: = 10 STEP -1 UNTIL 5 DO Statement. This statement will be executed for I = 10, 9, 8, 7, 6, 5, in that order.

All of these examples can be generalized by replacing any of the numbers used by any arithmetic expression. The index does not have to be an integer quantity. As an example:

FOR I: = M STEP J-K UNTIL N DO Statement

There are several comments to be made about this example. It is permissible for the statement following the DO to change the values of J and K so that the incrementing constant would have to be recomputed each time through the loop. It may happen that J-K may change sign as the statement is executed. In view of the previous examples, this makes for a rather involved situation for determining when the program is to leave the loop. This is carefully defined in ALGOL, but it makes a quite unusual loop.

Variables are declared in ALGOL in a manner similar to that used in FORTRAN IV. For example:

REAL, X, Y, Z

INTEGER A, B, I, Q

This states that the names X, Y, and Z refer to quantities which can take on as values any real number; these would normally be represented in floating point in the object program. The names A, B, I, and Q would take on only integer values.

ALGOL also allows Boolean variables; these are variables that can take on only two values, true or false, and can then be used in an IF statement. An example:

BOOLEAN D

D: = A=B

IF D THEN ...

D will have the value true if A=B, otherwise it will have the value false. The IF statement will take the appropriate branch depending on the value of D.

Arrays have few restrictions in ALGOL. Any number of subscripts are allowed, each subscript can be any integer expression, and subscripts can themselves be subscripted. An example:

A [B+J, M [5, 7] +T] will pick out an element from a two dimensional array named A in the following manner. It will compute B+J for the first subscript. For the second subscript it will find the appropriate number from the M array and add the value of T to it.

Arrays are declared in the following manner:

ARRAY A [-2:7, 4:8] This states that the first subscript of the A array may take on values -2, -1, 0, 1, 2, 3, 4, 5, 6, 7 and the second subscript may take on values 4, 5, 6, 7, and 8. Hence fifty locations will be reserved for this array. Note that negative and 0 values for the subscript are permitted, unlike FORTRAN.

A switch in ALGOL is a collection of labels the programmer may wish to branch to. It is declared as illustrated in the following example:

SWITCH x:=A, B, C, D

______

______

______

GO TO X [3];

The GO TO will transfer to C, which is the third label in the declaration.

Certain problems arise when a program is segmented, and different sections are coded by different programmers. The concept of blocks is used by ALGOL to handle this. A block is any compound statement which contains declarations. These must occur at the beginning of the block. The following example will illustrate some of the ideas used:

```
P: BEGIN REAL A, B
   ______
   ______
   ______
Q: BEGIN REAL A,C, OWN REAL D
   ______
   ______
     END
   ______
     END
```

We have a block within a block. The variable C has meaning here only inside the inner block. In addition, if the program leaves this block and later returns, the value of C will not have been preserved. The storage location that is used could have been used by other parts of the program. C is called a local variable in the inner block. The same statements apply to B except that they apply to the outer block, which, if course, includes the inner block. B is called global in the inner block. There really are two different variables represented by the name A in this example. The A in the outer block cannot be referred to in the inner block because A in the inner block refers to the A declared there. D is defined as an OWN variable. This means that its value will be preserved after leaving the inner block, and will have this value when the program later returns to the inner block.

Subroutines are called procedures in ALGOL. They are normally written in the program that uses them. The following example illustrates:

```
PROCEDURE Q(X, Y, Z); X:=Y+Z;
   ______
   ______
   ______
X: = R; Q(A, B, C); Y:=T;
```

The procedure declaration defines the procedure. Later in the program, the procedure is called as indicated in the second statement on the last line. This is executed as A: = B+C;

One of the rather extreme features in ALGOL, which is very costly to implement is called the copy rule. Normally when a subroutine is executed the values of the parameters are computed, and these are furnished as input to the subroutine. The copy rule states that the procedure should be executed as though the instructions to produce the parameter were copied into the procedure. As example illustrates:

```
PROCEDURE P(K, M); FOR I STEP 1 UNTIL 10 DO K: = K+M;
```

If this procedure is called by the statement P(J, I↑2), the compiler must produce instructions to execute the procedure as though the statement following the DO read J: = J+I↑2.

If this procedure rule is not desired, variables may be defined with the declaration VALUE which means the arguments will be computed and the results of the computation will be furnished to the subroutine.

Recursive procedures are permitted in ALGOL. This means that the definition of a procedure may call the procedure that is being defined. An example follows:

```
PROCEDURE X(A, B); A:=B+X: X(R, S);
```

ALGOL allows dynamic storage allocation. The following example illustrates:

```
BEGIN REAL A, B, C, INTEGER M, N
J: = 1; I: =1;
L: M: = J+3; N:=I+2;
BEGIN ARRAY A [1:M, 1:N]
   ______
   ______
   ______
END
J: = J+3; I: = I+7; GO TO L;
```

Note that every time the inner block is entered, the program must reserve a different amount of storage for the A array. This feature is implemented on the compiler IBM produced for SHARE.

B. JOVIAL

Introduction

JOVIAL, Jules Own Version of the International Algebraic Language, is based upon ALGOL-58. It was designed by the Systems Development Corporation to provide a flexible and readily understandable language for programming large scale computer-based command control systems. It is a procedure oriented language and at the present time there are compilers written for the IBM 7090, the IBM AN/FSQ31 and the closely related AN/FSQ-32, the IBM AN/FSQ-7, the Philco 2000 and the CDC 1604. The 7090, 2000 and 1604 compilers have been made available through the computer users of SHARE, TUG, and CO-OP. The JOVIAL compilers are written and maintained almost entirely in JOVIAL. They consist of two main parts: first, a Generator which transforms JOVIAL programs into Intermediate Language; second, a Translator which further transforms them into machine language. Translators ordinarily incorporate a completed symbolic assembly phase. Compilers for the Q-7, the 2000 and the 1604 use essentially the same common Generator and Intermediate Language, thus one Generator and three Translators were produced. JOVIAL compilers range in size from 50 to 60 thousand machine instructions, require about 5 man years of work to write a new translator and get a compiler running on a new machine.

Statements

It is convenient to recognize three classes of statements in JOVIAL: Simple statements which express primitive data processing action, complex statements which incorporate simple or compound statements within them, and compound statements which group together while strings of statements - simple, complex or compound. The compound statement is made up of a series of simple statements and enclosed in between the BEGIN and END brackets.

The NAME statement is a statement, label the same as the label in an SPS program. In write-ups of the JOVIAL language, IDENTIFIERS and NAMES are synonymous. The JOVIAL NAME is an arbitrary though usually mnemonic alphanumeric symbol, at least two characters long, which may be punctuated for readibility by the ' mark. NAMEs serve to identify the elements of the JOVIAL program information environment - that is, statements, switches, procedures, items, array items, tables, string items and files. Except for context designed statement names all JOVIAL names must be declared, either explicitly in the program or implicitly in the Compool or in the procedure library. (A Compool is a library of system environment declarations and storage allocation parameters). A NAME is needed only when the statement is to be executed out of sequence.

The ASSIGNMENT statement assigns the values specified by a formula to be value designated by the variable. The two functions of this statement are data manipulation and arithmetic. It takes the form a = b $ where = sign is called the assignment separator.

The EXCHANGE statement exchanges the value designated by a pair of variables. The effect of an EXCHANGE statement on either of the variables involved is as if each has been assigned the value designated by the other. Consequently, the rules of ASSIGNMENT pertain and both variables must be of the same type: numeric, literal, status or Boolean. The statement is of the form a = = b $ where = = is called the exchange separator. As with all other statements it must be followed by a dollar sign.

There are two types of TRANSFER statements: the unconditional statement is a GOTO x $. There are no spaces between the two words and the translator will substitute an unconditional branch. The conditional transfer statement is an IF where the value to be tested appears after the statement. If the IF statement is true, the next statement is executed. If the statement is false, the next statement either compound or simple is bypassed. An example is:

```
        IF ALFA EQ BETA $
BRCH.   GOTO STEP2 $
STEP1.  ALFA=GAMMA+DELTA
          :
STEP2.  DIFF=GAMMA-DELTA
```

A LOOP statement is a complex statement consisting of a list of FOR - clauses which establish the LOOP counters, and a simple or compound statement, which forms the repeatedly - executed body of the loop. The LOOP statement may contain from one to three FOR statements. The one factor FOR statement defines a constant and gives it a value. Example: FOR I = 23 $. The statement would be initialized by setting I = to 0, and after each repetition I must be set to I + 1. The test for the maximum I would be made by the programmer. The two factor FOR statement sets the initial value of I and specifies the increment value, for example FOR I = 0, 1 $. In this case, I will initially be 0 and incremented by 1 each time. The test for the final value of I would be included by the programmer. The three factor FOR statement specifies the initial value, the increment value and the test value, i. e., FOR I = 0, 1, 9 $. The initial value is 0, the increment value is + 1 and the loop would be terminated when I exceeds 9. The test is automatically included by the processor. Decrementation as well as incrementation is possible. Example: FOR I = 9, - 1, 0.

OPEN OUTPUT or OPEN INPUT statement puts the particular device being called into ready status. After this statement a number of output statements or input statements can be executed, bringing in or reading out data. The last statement to be executed in the sequence of OUTPUT and INPUT statements must be SHUT OUTPUT or SHUT INPUT.

A ARITHMETIC statement will contain any combination of the normal arithmetic functions. Parenthesis are used to determine order of operation. JOVIAL and FORTRAN both use the same set of symbols to denote arithmetic operations.

The CONTROL statement has four variations. The unconditional branch is a GOTO n, where n is a statement number. In JOVIAL the n would refer to a NAME statement. The computed GOTO would branch to some statement depending upon the value of the constant being tested. It takes the form GOTO (n_1, n_2, ..., n_n), I. If I were 1 the program would branch to n_1, if I were 4, the program would branch to the fourth value in the parenthesis. The IF statement tests the value of an expression or a variable and depending upon the value, will branch to one of three places that was specified in the statement. The form of a statement is IF (a) n_1, n_2, n_3 where n_1 is the statement branch to if (a) is negative, n_2 is branched to, if (a) is = to 0, and n_3 is branched to if (a) is positive.

The DO statement is the one used in looping. The starting value, final value and increment values are given in the statement, similar to the three factor FOR statement. Decrementation, however, is not permitted. The format is DO - I = n_1, n_2, n_3. n_3 may be omitted if the value is 1. The statement number after the DO is the last statement of the DO to be executed. The tests for the increment is automatically inserted by the processor after the last statement.

The SPECIFICATION is a non-executable statement, for it just reserves space and does not generate any machine instructions. An example of this is the DIMENSION statement which sets aside core storage, where DIMENSION A (20) will reserve enough storage to contain 20 floating point data words.

The I/O statements are of the form of what actually is to be accomplished. To read card, READ is specified. The unit does not have to be placed in ready status first.

Constants

A fixed point number in FORTRAN is so designated because it is an integer (no decimal point may appear) and the variable name begins with the letters I - N. In JOVIAL there must be a space followed by an A after the item name. After the A the number which specifies the number of positions in the constant is given and then an S or U for

Signed or Unsigned. An example would be SUM A7S.

A floating point constant in FORTRAN is designated by the letter with which the constant begins. It may not begin with the letters I through N for they are reserved for fixed point numbers unless, of course, a TYPE statement is used within the FORTRAN IV program. The number must contain a decimal point and may be in the E notation (.32E12). In JOVIAL a floating point number has an F after the item name - example SUM F.

Dual constants appear in JOVIAL and the same value is set up in each half of the item. A "D" comes after the item name, n which specifies the sum of the positions in each constant, S or U for signed or unsigned and + or =n for the number of fractional bits, example: SUM D6 S+2.

Status constant is in the mnemonic label which denotes one of the values of the status item. The form is V (GOOD), where GOOD refers to the status of sum item.

The literal constant is either in Hollerith in which case it is preceded by an H or in standard transmission code in which case it is preceded by a "T". Standard transmission code is the octal representation of the constant in the machine. Example: 44H (THIS IS A LITERAL CONSTANT IN HOLLERITH CODE) 56P (THIS IS A LITERAL CONSTANT IN STANDARD TRANSMISSION CODE).

Figure 35
Sample of a JOVIAL Program

Summing Ten Floating Point Numbers

```
START      JOK 1.
TABLE      NUMBR R 10 $
           BEGIN
           ITEM BB F $
           BEGIN 3.0 6. 10.2 0.0 20.0 1.23 .08 0.32 12.0
                 IE-2 5.0 END
           END
```

```
ITEM        SUM F $
STEP1.      SUM = 0.0 $ ''INITIALIZE SUM''
STEP2.      FOR I = ALL (BB) $
            SUM + BB ($I$) $ ''COMPUTE SUM''
TERM        $
```

Explanation of Jovial Program

The program must begin with a START card and end with a TERM $ card. TERM must have a $ termination separator following it. The program name which is optional in the START statement must be followed by a (.) which is a statement name separator.

Table declaration encompasses NUMBR which is the name of the table. R signifies that this is a rigid (fixed) length ("V" would signify variable), and 10 is the actual length of the table. The table declaration must be followed by a $ which terminates the declaration.

The composition of the table must be defined between BEGIN and END statements.

Item declaration declares item comprising the entry in the table. In this case the item is identified as BBF. This means that it is a floating point. Once again the $ terminates the declaration.

The BEGIN and END statements define the parameter list which comprise the table and the first ten entries of the item BBF.

Statement name - STEP 1. must be followed by a (.) which is the statement name separator. The statement sets SUM = to 0.0. The "COMMENT" tells the processor to ignore what comes next but print it on the program listing.

STEP 2. I is the index. The loop will be accomplished ten times since the statement says FOR ALL (BB). Since the order of taking the sum is unimportant ALL may be used rather than another type of FOR statement. Included with the example is an example of the same program written in FORTRAN. This can be used as a comparative tool in evaluating the language of JOVIAL.

Figure 36

Sample of FORTRAN Program

```
C           PROGRAM TO SUM 10 FLOATING POINT NUMBERS
            DIMENSION BB(10)
            SUM = 0.0
            DO 103, I =1, 10
      103   SUM = SUM+BB(I)
            PAUSE
            END
```

Additional Language Specifications

The statements to this point have been kept on a fairly simple level. In order to evaluate all of the ramifications of JOVIAL, however, additional specifications of the language should be perused.

Regarding the processing of clauses, strings of JOVIAL symbols, (i.e., delimiters, identifiers and constants) form clauses such as item descriptions which describe values; variables which designate values; and formulas which specify values. In general, symbols may be separated by comments or by an arbitrary number of blanks. However, no separation is needed when one or both of the signs so joined is a mark.

Clauses are combined with certain delimiters to form declarations and statements which are classified as sentences of JOVIAL. Statements assert actions that the program is to perform (normally in the sequence in which they are listed) and declarations describe the information environment in which the actions are to occur.

In JOVIAL, values other than those denoted by constants or used only as intermediate results, or for controlling loops must be formally declared as items: simple items, array items, table items or string before they can be referenced. When not a part of a table declaration, an item declaration defines a simple item with a single value. A mode declaration starts a new normal mode for the implicit declaration of all subsequently referenced (and otherwise undefined) simple items. An array declaration describes the structure of a

collection of similar item values and also provides a means of identifying this collection with the single item named. In this manner, arrays of any number of dimensions may be declared.

Functional modifiers facilitate the manipulation both of larger data elements (entries and tables) and the smaller data elements (segments of machine language symbols representing item values). A brief description of each modifier will suffice. The NENT modifier is a vital parameter in table processing of Number of ENTries. The functional modifier NENT allows this unsigned integral value to be designated for rigid length table. The NWDSEN modifier is another parameter in table processing and this is the amount of storage allocated to a table entry and thus to the entire table. This unsigned integral value which is constant throughout the execution of the program is expressed in the Number of WorDS or registers per ENtry. NWDSEN is needed in executive programs that do dynamic storage allocation. The ALL modifier creates a loop with an undefined direction of processing. The ENTRY modifier allows an entry to be treated as a single value represented by a single composite machine language symbol.

The BIT and BYTE modifiers are worthy of mention because of System 360. The BIT modifier allows any segment of the BIT string representing the value of any item to be designated as an unsigned integral variable. Similarly, the BYTE modifier allows any segment of the BYTE string representing the value of any literal item to be designated as a literal variable. The MANT and CHAR modifiers are involved with floating point machine language symbol representation, and by using them, either component of any floating point item can be designated as a fixed point variable. The least significant bit of any loop counter or floating-or-fixed-point item can be designated as a Boolean variable: True, if it represents a magnitude of one and false if it represents a magnitude of zero by the ODD modifiers.

Summary

JOVIAL is a general purpose procedure-oriented are largely computer independent programming language, derived from ALGOL 58 with the major extensions of: input-output notation, a more elaborate description capability, the ability to manipulate fixed-point numeric values, and the ability to manipulate symbolic and other non-numeric values including machine language symbol segments.

Some thirty computer installations have received JOVIAL compilers through the users group of SHARE, TUG and CO-OP, and has been adopted by the Navy Command Systems Support Activity as the interim standard programming language for Navy Strategic Command Systems.

C. SNOBOL

Strings and String Names

The basic data structure in SNOBOL is a string of symbols, where names (a string of numerals and/or letters with medial periods) are assigned to strings for reference. The string named LINE. 1 may have the contents: TIGER, TIGER, BY MY BED.

String Formation

The string name LINE. 1 with the above contents is formed by the rule: LINE. 1 = "TIGER, TIGER, BY MY BED" where the quotation marks specifies the literal contents. Any symbols except quotation marks can be placed within the quotation marks.

Strings can also be formed by concatenation. The rule: "TIGER, TIGER, " "BY MY BED" will produce the same results as the earlier example.

Previously named strings can be used to form new ones, using the rule, EXAMPLE = LINE. 1 forms a string, EXAMPLE, with the same contents as the string LINE. 1. Literals and named strings can be used in formation, similar to:

```
LINE. 1 = "TIGER, TIGER, BY MY BED"
LINE. 2 = "IN YOUR SHADES OF BLACK AND RED"
LINE. 3 = "NEVER WILL I SLUMBER THROUGH, "
LINE. 4 = "WHEN I TURN MY GAZE ON YOU. "
TEXT    = LINE. 1 "/" LINE .2 "/"
          LINE .3 "/" LINE. 4 will form a composite
          string with slashes separating the lines in a
          conventional manner.
```

The spaces between string names and literals serve as break characters for distinguishing the elements to be concatenated, with one space required for separation.

In forming the string, the string itself may be used. After performing the two rules:

NUMBER = "6"

NUMBER = NUMBER NUMBER "0";

the string, NUMBER, will contain the literal "660".

Pattern Matching

To determine whether the string, LINE. 1 contains the literal "TIGER" the rule would be: LINE. 1 "TIGER". This is similar to a formation rule, but without the equal sign. LINE. 1 is scanned from left for an occurence of the five literals "TIGER" in succession. A pattern matching rule may succeed or fail. If LINE. 1 is formed in the previous example, the scan would be successful, and scanned string is not altered. The pattern may be specified by the concatenation of a number of literal and string names: TEXT LINE. 1 "/" LINE. 2 specifies the scan of a string named TEXT for an occurence of the contents of the LINE. 1, immediately followed by the literal "/" and in turn, immediately followed by the contents of the string LINE. 2.

String Variables

If it is required to know whether a string contains one sub-string followed by another, but with the second sub-string not necessarily immediately after the first, a string variable is introduced to permit this. The rule: LINE. 1 "TIGER" *FILLER* "BED" is of this kind. The question is whether LINE. 1 contains "TIGER" followed by "BED" with perhaps something between. The symbols *FILLER* represents a string variable which takes care of this "something". If LINE. 1 is formed as we have previously shown, the scan would be successful.

A string variable may be any string name bounded by asterisks. A by-product of matching a pattern containing a string variable is the formation of a new string that has the name furnished between the asterisks of the string variable.

Replacement

In this string LINE. 2 it is desired to replace "HUES" by "SHADES". This would be accomplished by: LINE. 2 "HUES" "SHADES". This scans LINE. 2 for the occurence "HUES", and if the scan is successful, "HUES" is replaced by "SHADES". LINE. 2 will then become "IN YOUR HUES OF BLACK AND RED".

If the scan fails, the string being scanned is not altered. Any string formed is a result of a successful pattern match of a string variable on the left side of the equal sign and can be used in the replacement on the right side. Thus: LINE. 1 "BY" *FILLER* "BED" = FILLER, would result in the deletion of "BY" and "BED" from LINE. 1.

Rules

A rule may consist of four parts, separated by a blank in the following order:

1. A STRING to be manipulated, i. e. STRING REFERENCE:
2. A LEFT SIDE specifying a pattern;
3. An EQUAL SIGN;
4. A RIGHT SIDE specifying a replacement.

The string reference is mandatory, while the rest of the rule parts may be absent, depending upon the particular rule. "GO TO" consists of a slash followed by one or more of the following parts:

1. An unconditional transfer which has the form (BA). Upon the completion of the statement the next statement to be executed is the statement with the label BA.
2. A conditional transfer on failure has the form F(BB). If the statement fails the statement with the label BB is to be executed next.
3. A conditional transfer on success has the form S(BC). Transfer is made to BC on successl.

Arithmetic

Simple Arithmetic may be performed on strings whose contents are integers. L = C+X would form the string named L containing the arithmetic sum of the contents of strings C and X.

Indirectness

Indirectness is accomplished in SNOBOL by writing $ sign in front of the string name. If the string FACTOR contains the literals "SUM" writing $ FACTOR is the same as writing "SUM".

Input/Output

Input and Output are accomplished by the use of the two commands, READ and PRINT following the string references SYS.

Language Extension

Additional features are being planned for in SNOBOL in the near future, including extended input/output facilities consistent with the string orientation.

D. NELIAC

Introduction

The NELIAC compiler, sponsored primarily by the Navy Electronic Laboratory, was started in July 1958 and completed within the following six months. Three special characteristics of NELIAC should be recognized: NELIAC compilers are self-compilers; most NELIAC compilers have been kept relatively short and simple; most NELIAC compilers have compiling speeds of many thousands of computing words per minute. A programmer familiar with the language can read, understand and improve any given compiler, and can recompile a true version of a compiler quite cheaply, since some recompile in less than a minute.

Operators

The NELIAC language is based upon the use of 25 symbolic operators, including punctuation, arithmetic, and relational symbols. Meanings are ascribed to these operator symbols on the basis of Current Operator-Operand-Next Operator combination. The use of symbolic operators reduces the number and complexity of rules which must be kept in mind reduces the problems of documentation. The language includes the ability to handle bits within computer words, to treat input/output without direct format statements, to insert machine language and to address computer language memory directly.

V. BIBLIOGRAPHY

A. SUGGESTED READINGS

The Computer Journal - British Computer Society Ltd.

April 1958	"The Autocode Programs Developed for the Manchester University Computers - Brooker pp. 15-21
October 1958	"Further Autocode Facilities for the Manchester (Mercury) Computer," pp. 124-127
January 1959	"Deuce Interpretive Programs" - Robinson, pp. 172-175
	"The Pegasus Autocode" - Clarke & Felton, pp 192-195
July 1959	"Algorithms for Formula Translation" - Cleave, pp. 53-54
	"Intercode; a Simplified Coding Scheme for Amos" - Berry, pp. 55-58
	"A Translation Routine for the Deuce Computer," pp. 76-84
January 1960	"A Function Interpretive Scheme for Pegasus" Hornsby, p. 174
July 1960	"An Introduction to Algol 60" - pp. 67-74
	"The Deuce Alphacode Translator," pp. 98-106
October 1960	"An Assembly Program for a Phrase Structure Language," pp. 168-174 (List Structures - Basis for Syntax-Directed Compilers) - Brooker & Morris
January 1961	"Some Proposals for Realization of a Certain Assembly Program" Brooker & Morris, pp. 220-231
April 1961	"Compiling Techniques for Algebraic Expressions" - Huskey, pp. 10-19
	"Atoms and Lists" - Woodward & Jenkins, pp. 47-53 (LISP)

October 1961 — "Nebula: A Programming Language for Data Processing" - Braunhultz, Fraser & Hunt, pp. 197-211

"Improving Problem-Oriented Language by Stratifying It" - Basley, pp. 217-221

January 1962 — "ABS12 Algol: An Extension to Algol 60 for Industrial Use" - Hockney, pp. 292-300

"Principles & Problems of a Universal Computer-Oriented Language" - Bagley, pp. 305-312

April 1962 — "ALP: An Autocode List - Processing Language," pp. 28-32

"Trees and Routines" - Brooker, Morris & Rohl, pp. 33-47

July 1962 — "Current Developments in Commercial Automatic Programming" d"Agapeyeff, pp. 107-111

"FACT - Clippinger, pp. 112-118

"Operating Experience with Algol 60" - Dijkstra, pp. 125-126

"Report on the Elliott Algol Translator" - Hoare, pp. 127-129

"Implementation of Algol 60 for the English Electric KDF9" - Duncan, pp. 130-131

"Operating Experience with Fortran" - Glennie, pp. 132-134

"A Proposed Target Language for Compilers on Atlas" - Curtis & Pyle, pp. 100-106

January 1963 — "The Realization of Algol Procedures and Designational Expressions" - Watt, pp. 332-337

"A Hardware Representation for Algol 60 Using Creed Teleprinter Equipment" - Gerard & Sambles, pp. 338-340

"Input and Output for Algol 60 on KDF9" - Duncan, pp. 341-344

"The Elliott Algol Input/Output System" - Hoard, pp. 345-348

Journal of the ACM

1960 — "Input Output Buffering and Fortran" - Ferguson, pp. 1-9

"A Fortran Compiled List-Processing Language" - Gelerntner, Hansen, Gerberich, pp. 87-101

1959 — "Three Levels of Linguistic Analysis in Machine Translation", Zarechnak, pp. 24-32

"The Share 709 System"
- Introduction -- pp. 123-127
- Program & Modification -- pp. 128-133
- Machine Implementation of Symbolic Programming -- pp. 134-140
- I/O Translation -- pp. 141-144
- I/O Buffering -- pp. 145-151
- Supervisory Control -- pp. 152-155

1958 — "Language Translation" - Brown, pp. 1-8

1957 — "Standardized Programming Methods and Universal Coding" - Saul Gorn, pp. 254-273

April 1961 — "System Handling of Functional Operators" - Lombardi, pp. 168-185

January 1962 — "A General Translation Program for Phrase Structure Languages," pp. 1-10

"Mathematical Structure of Non Arithmetic Data Processing Procedures" - Lombardi, pp. 136-159

April 1962 — "Structure and Use of Algol 60" - Bottenbruch, pp. 165-221

"An Algorithm for Translating Boolean Expressions" - Arden, Galler, Graham, pp. 222-239

July 1962 — "Two Families of Languages Related to Algol" - Ginsburg & Rice, pp. 350-371

"A Method for Obtaining Specific Values of Compiling Parameter Functions" - Peterica, pp. 379-386

October 1962 — "A Translator-Oriented Symbolic Language Programming Language" - A. A. Grau, pp. 480-487

"On the Ambiguity of Backus Systems" - Cantor, pp. 477-479

January 1963 — "Some Recursively Unsolvable Problems in Algol-Like Languages" - Ginsburg & Rose

April 1963 — "Theorem-Proving on the Computer" - Robinson, pp. 163-174

"Operations Which Preserve Definability in Languages" - Ginsburg & Rose, pp. 175-195

"Detection of Generative Ambiguities in Context-Free Mechanical Languages" - Gorn, pp. 196-208

Communications of the ACM

August 1958 — "The Problem of Programming Communication with Changing Machines" - Ad Hoc Share Committee on Univ. Lang., pp. 12-18 (A Proposed Solution)

September 1958 — Part II of above, pp. 9-16

October 1958 — "Proposal for an Uncol" - Conway, pp. 5-8

February 1959 — "Recursive Subscriptions Compilers and List-Type Memories" - Carr, pp. 4-5

"Possible Modifications to the International Algorithmic Language" - Green, pp. 6=8

"The Arithmetic Translator Compiler for the IBM Fortran Automatic Coding System" - Sheridan, pp. 9-21 (704)

"Signal-Corps Research and Development on Automatic Programming of Digital Computers" - Luebbert & Collom, pp. 22-27

March 1959 — "From Formulas to Computer Oriented Language" - Wegstein, pp. 6-7

"A Checklist of Intelligence for Programming Systems" - Bemer, pp. 8-13 (very good)

June 1959 — "Handling Identifiers as Internal Symbols in Language Processors" - Williams, pp. 21-24

July 1959 — "On Gat and the Construction of Translators" - Arden & Graham, pp. 24-26 (650)

August 1959 — "Proposal for a Feasible Programming System" - Phil Basley, pp. 7-9

September 1959 — "Remarks on Algol and Symbol Manipulation" - Green, pp. 25-27

October 1959 — "An Algebraic Translator" - Kanner, pp. 19-21

"Sale, A Simple Algebraic Translator for Engineers" - Brittenham, Clark, Koss & Thompson, pp. 22-24

November 1959 — "Runcible - Algebraic Translation on a Limited Computer" - Knoth, pp. 18-20

	"A Technique for Handling Macro-Instructions" - Greenwald, pp. 21-22
December 1959	"A Proposed Interpretation in Algol" - Irons & Acton, pp. 16-19
February 1960	"Sequential Formula Translation" - Samelson & Bauer, pp. 76-82 (Concept of a pushdown list in compiling)
	"Coding Isomorphisms" - Lynch, p. 84)
March 1960	"An Algorithm Defining Algol Assignment Statement" - Floyd, pp. 170-171
April 1960	Papers from ACM Conference on Symbol Manipulation
	"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" - John McCarthy, p. 184
	"Symbol Manipulation by Threaded Lists" - Alan J. Perlis & Charles Thornton, p. 195
	"An Introduction to Information Processing Language V" Allen Newell & F. Tonge, p. 205
	"Syntactic and Semantic Augments to Algol" - Joseph W. Smith p. 211
	"Symbol Manipulation in Xtran" - Julien Green, p. 213
	"Macro Instruction Extensions of Compiler Languages" M. Douglas McIlroy, P. 214
	"Proving Theorems by Pattern Recognition, I" - Hao Wan p. 220
May 1960	"Report on the Algorithmic Language Algol," pp. 299-314
June 1960	"Compiling Connectives" - Swift, pp. 345-346
	"Conversion Between Floating Point Representations" - Perry, p. 352
July 1960	"Combining Algol Statement Analysis with Validity Checking pp. 418-419
	"Programming Compatibility in a Family of Closely Related Digital Computers" - Leubbert, pp. 420-429

August 1960	"Neliac - A Dialect of Algol" - Huskey, Halstead & MacArthur, pp. 463-467
	"A Short Study of Notation Efficiency" - Smith, pp. 468-474 (Notes on bit-code to accommodate Algol set)
September 1960	"An Introductory Problem in Symbol Manipulation for the Student" - Rosin, pp. 488-489 (Uses Mad)
	"Comments from a Fortran User" - Blatt, pp. 501-506 (Critical comments on original 704 Fortran)
	"Trie Memory" - Fredkin, pp. 490-499 (special memory arramgenent (tree-structure) for storing list data)
November 1960	"Compilation for Two Computers with Neliac" - Masterson, pp. 607-610
December 1960	"A Method for (the) Overlapping and Erasure of Lists" - Collins, pp. 655-658
January 1961	Papers Presented at the ACM Compiler Symposium
	"A Basic Compiler for Arithmetic Expressions" H. D. Huskey and W. H. Wattenburg, p. 3
	"Recursive Processes and Algol Translation" - A. A. Grau, p. 10
	"Use of Magnetic Tape for Data Storage in the Oracle-Algol Translator" - H. Bottenbruch, p. 15
	"The Clip Translator" - Donald Englund & Ellen Clark, p. 19
	"CL-1, An Environment for a Compiler" - T. E. Cheatham, Jr., G. O. Collins, Jr., & G. F. Leonard, p. 23
	"The Internal Organization of the Mad Translator" - B. W. Arden, B. A. Galler, R. M. Graham, p. 28
	"Madcap: A Scientific Compiler for a Displayed Formula Textbook Language" - Mark B. Wells, p. 31

"The Use of Threaded Lists in Constructing a Combined Algon and Machine-Like Assembly Processor" - A. Evans, Jr., A. J. Perlis, and H. Van Zoeren, p. 36

"An Algorithm for Coding Efficient Arithmetic Operations" Robert W. Floyd, p. 42

"A syntax Directed Compiler for Algol 60" - Edgar T. Irons, p. 51

"Thunks" - P. Z. Ingerman, p. 55

"Dynamic Declarations" - P. Z. Ingerman, p. 59

"Allocation of Storage for Arrays in Algol 60" - Kirk Sattley, p. 60

"Comments on the Implementation of Recursive Procedures and Blocks in Algol 60" - E. T. Irons & W. Feurzeig, o. 65

"Compiling Techniques for Boolean Expressions and Conditional Statements in Algol 60" - H. D. Huskey & W. H. Wattenburg, p. 70

"The Slang System" - R. A. Sibley, p. 75

February 1961 — "Two Aubroutines for Symbol Manipulation with an Algebraic Compiler" - Carr & Hanson, pp. 102-103 (It & Gat 650)

March 1961 — "An Alternate Form of the 'Uncol' Diagram" - Harvey Bratman, p. 142

"A Generalized Technique for Symbol Manipulation and Numerical Calculation" - D. T. Ross (MIT), pp. 147-150 (Lists)

April 1961 — "On the Compilation of Subscripted Variables" - Nather, pp. 169-170 (Fortran)

June 1961 — "Algol Confidential" - Knoth & Merner, pp. 268-271

"Logic Structure Tables" - Cantrell, King & King, pp. 271-275

July 1961 — "An Algorithm for Equivalence Declarations" (in Mad) - Bruce V. Arden, pp. 310-313

August 1961 — "Some Basic Terminology Connected with Mechanical Languages and Their Processors" - Saul Gorn, pp. 336-339

September 1961 — "An IR Language for Legal Retrieval" - Kehl, Horty, Bacon & Mitchell, pp. 380-388

"Use of MOBL in Preparing Retrieval Programs" - Hoffman & Opler, pp. 389-391

"A Syntactical Chart of Algol 1960," . 393

"Manipulation of Algebraic Expressions" - Rom, pp. 396-398 (Subroutines written in FAP as Fortran subroutines)

October 1961 — Papers Presented at the ACM Storage Allocation Symposium

- "Discussion One: Toward a Definition of the Storage Allocation Problem, p. 416
- "Discussion Two: Preplanned vs. Dynamic Storage Allocation Techniques, p. 416
  - "A Preplanned approach to a Storage Allocation Compiler" - Robert W. O'Neill, p. 417
  - "The Case for Dynamic Storage Allocation" - Burnett H. Sams, p. 417
- "A General Formulation of Storage Allocation" - A. E. Roberts, Jr., p. 419
- "Problems of Storage Allocation in a Multiprocessor Multiprogrammed System" - R. J. Maher, p. 421
- "Program Organization and Record Keeping for Dynamic Storage Allocation" - Anatol W. Holt, p. 422
- "Dynamic Storage Allocation for an Information Retrieval System" - Burnett H. Sams, p. 431

"Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store" - John Fothingham, p. 435

"Experience in Automatic Storage Allocation" - George O. Collins, Jr., p. 436

"A Storage Allocation Scheme for Algol 60" M. Jensen, P. Mandrup, P. Naur, p. 441

A Semi-Automatic Storage Allocation System at Loading Time" - William P. Heising & Ray A. Larner, p. 446

"Techniques for Storage Allocation Algorithms" - J. E. Kelley, Jr., p. 449

"Core Allocation Based on Probability" - Bernard N. Riskin, p. 454

"Stochastic Evaluation of a Static Storage Allocation" - Leo J. Cohen, p. 460

November 1961

"Some Proposals for Improving the Efficiency of Algol 60" - Stachy & Wilkes, pp. 488-496

"Low Level Language Subroutines for Use Within Fortran" - Barnett, pp. 492-499 (symbol manipulation, etc.)

"Smalgol-61," pp. 499-503 (An Algol for small computers)

"An Engineering Application of Logic Structure Tables," pp. 516-520 (With reference to a program for translation of table to a machine language prog).

December 1961

"Specification Languages for Mechanical Languages and Their Processors - A Baker's Dozen" - Saul Gorn, pp. 532-541 (Includes a large bibliography)

February 1962

"Algol Primer: An Introduction to Algol" - Schwarz, pp. 82-95

"Surge" A Recoding of the Cobol Merchandise Control Algorithm" - Longo, pp. 98-100 (A different type of commercial compiler)

"A Neliac-Generated 7090-1401 Compiler" - Watt & Wattenburg, p. 101

March 1962

"Automatic Programming Language Translation Through Syntactical Analysis" - Ledley & Wilson, pp. 145-155

"An Evaluation of Autocode Reliability" - Ellis, pp. 156-158 (Rapid write)

"On a Floating Point Number Representation for Use with Algorithmic Languages" - Grau, p. 160

"Knotted List Structures" - Weizenbaum, pp. 161-164

May 1962

"Initial Experience With an Operating Multiprogramming System" - Landis, Manos & Turner, p. 282

Cobol Papers

"Why Cobol?" - Joseph F. Cunningham, p. 236

"Basic Elements of Cobol 61" - Jean E. Sammet, p. 237

"Cobol and Compatibility" - A. Lippitt, p. 254

"Interim Report on Bureau of Ships Cobol Evaluation Program" - Milton Siegel and Albert F. Smith, p. 256

"Syntactical Charts of Cobol 61" - Richard Berman, Joseph Sharp and Lawrence Sturges, p. 260

"A Report Writer for Cobol" - W. L. Donally, p. 261

"The Cobol Librarian" - W. Hicks, p. 262

"Modular Data Processing Systems Written in Cobol" - J. C. Emery, p. 263

"Floating Point Arithmetic in Cobol" - O. Kesner, p. 269

"Guides to Teaching Cobol" - I. Greene, p. 272

"An Advanced Input-Output System for a Cobol Compiler" - C. A. Bouman, p. 273

"An Introduction to a Machine-Independent Data Division" - J. P. Mullin, p. 277

"Cobol Matching Problems" - J. W. Mullen, p. 278

June 1962 — "Report on the Algorithmic Language Fortran II" - Rabinowitz, pp. 327-336

"A Redundancy Check for Algol Programs" - Thacher, pp. 337-342

"Analytic Differentiation by Computer" - Hanson, Caviness, Joseph, pp. 349-355

July 1962 — "Communication Between Independently Translated Blocks" - Wegner, pp. 376-380

"On Translation of Boolean Expressions" - Bettenbruch & Grau, pp. 384-386

"A Machine Program for Theorem Proving" - Davis, Logerann & Loveland, pp. 394-396

"Fortran for Business Data Processing" - Robbins, pp. 412-414 (Uses character & bit manipulation subroutines)

September 1962 — "Current Status of IPL-V for the Philco 2000" - Shaffer, p. 479

"Tall, A List Processor for the Philco 2000 Computer" - Feldman, pp. 484-485

"On the Nonexistence of a Phrase Structure Grammer for Algol 60," p. 483

October 1962 — "Implementing a Stack" - Baecker, pp. 505-506

"Input Data Organization in Fortran" - Yarbrough, pp. 508-509

"Syntactic Analysis by Digital Computer" - Barnett & Futrelle, pp. 515-525 (Shadow)

"On the Ambiguity of Phrase Structure Languages" - Floyd, p. 526

December 1962 — "Mechanical Pragmatics: A Time-Motion Study of a Miniature Mechanical Linguistic System" - Saul Gorn, pp. 576-590

"Compiling Matrix Operation" - Galler & Perlis, pp. 590-599

"Pracniques - Character Manipulation in 1620 Fortran II" - Vasilakos, p. 602

"Fixed-Word Length Arrays in Variable Word-Length Computers" - Sonquist, p. 602 (1401)

January 1962 — IR-Oriented Languages

- "Discussion - The Pros and Cons of a Special IR Language, p. 8
  - "Comments" - Jean E. Sammett, p. 8
  - "Pro A Special IR Language" - Herbert Ohlman, p. 8
  - "Comments" - H. G. Bohnert, p. 10
- "Information Structure for Processing and Retrieving" - Robert A. Colilla and Burnett H. Sams, p. 11
- "An Information System With the Ability to Extract Intelligence from Data" - T. L. Wang, p. 16
- "Comit as an IR Language" - Victor H. Yngve, p. 19
- "Language Problems Posed by Heavily Structured Data" - Robert F. Barnes, p. 28
- "Translation of Retrieval Requests Couched in a 'Semi-formal' English-Like Language" - T. E. Cheatham, Jr., and S. Warshall, p. 34
- "Use a Semantic Structure in Information Systems" - J. D. Sable, p. 40
- "A Survey of Languages and Systems for Information Retrieval" - Assembled by Mandalay Grems, p. 43

"Machine Language Paper: A String Language for Symbol Manipulation Based on Algol 60" - J. H. Wegstein and W. W. Youden, p. 54

January 1963	"Algol Revised Report - Supplement to the Algol 60 Report," pp. 1-17
	"Suggestions on Algol 60 (Rome)," pp. 20-23
February 1963	"Character Manipulation in Fortran" - Lewis, p. 65
March 1963	"Toward Better Documentation of Programming Languages": Intro. - Yngve & Sammett, p. 76 Algol - Naur, p. 77 Cobol - Cunningham, p. 79 Comit - Yngve, p. 83 Fortran - Heising, p. 85 IPL-V - Newell, p. 86 Jovial - Shaw, p. 89 Neliac - Halstead, p. 91
	"Survey of Programming Languages and Processors," p. 93
	"Note on the Nonexistence of a Phrase Structure Grammar for Algol 60" - Brown, p. 105
	"Recol - A Retrieval Command Language," pp. 117-122
April 1963	"A Suggested Method of Making Fuller Use of Strings in Algol 60" - Shoffner & Brown, pp. 169-171
May 1963	Full issue on Sorts
June 1963	"Corc - The Cornell Computing Language" - Conway & Maxwell, pp. 317-320
	"The External Language Klipa for Ural-2 Digital Computer" - Greniewski & Turski, pp. 321-324
	"Description - Automated Descriptive Geometry" - Kliphardt, pp. 336-339 (Fortran-Coded Subroutines)
July 1963	"A Syntactic Description of BC Neliac" - Huskey, Love, Wirth, pp. 367-374
	"The Linking Segment Subprogram and Liking Loader" - McCarthy, Corbato & Daggett, pp. 391-395

"Design of a Separable Transition Diagram Compiler" - Conway, pp. 396-408

Computers and Automation - "A History of Writing Compilers" - D. E. Knuth, December 1962

B. REFERENCES

ELBOURN, R. D., and WARE, W. H.

The Evolution of Concepts and Languages of Computing Procedures of the IRE /1962/pp. 1059-1066.

FARBER, D. J., GRISWOLD, R. E., and POLONSKY, I. P.

SNOBOL, A String Manipulation Language

Journal of the ACM Volume II/No. 1/Jan 1964/ pp. 21-30.

GAIFMAN, H.

Dependency Systems and Phrase Structure Systems

Rand Corporation, Santa Monica, California/ P-2315/ May 22, 1961/ 64 pp.

GARWICK, J. V.

GARGOYLE, A Language for Compiler Writing

Communications of the ACM Volume 7/ No. 1/ January 1964/ pp. 16-20.

HALSTEAD, M. H.

NELIAC

Communications of the ACM Volume 6/ No. 3/ March 1963/ pp. 91-92.

HARPER, K. E.

Dictionary Problems in Machine Translation

The Rand Corporation, Santa Monica, California/ May 29, 1961/P-2327/ 15 pp.

KEPLER, J. F.

XTRAN: A New Approach to Compiler Writing

TIE 6208-0128; August 7, 1962/ 10 pp.

KNUTH, D. E.

A History of Writing Compilers

Computers and Automation Dec/1962

LIND, J. H.

Implementation of List Structures

IBM SRI Term Project No 6-35/ 33 pp.

MARON, M. E.

A Logician's View of Language Data Processing

The Rand Corporation, Santa Monica, California/ April 24, 1961/ P-2279/44 pp.

NEWELL, A.

Documentation of IPL-V

Communications of the ACM Volume 6/No. 3/March 1963/ pp. 86-89

NEWELL, A.; SHAW, J. C.; and SIMON, H. A.

Report on a General Problem Solving Program for a Computer

Information Processing: Proc. Internat. Conf. Information Processing

pp. 256-264, Paris: UNESCO, 1960

ROSIN, R.

Translation of Artificial Languages by Compiler Programs

The Rand Corporation, Santa Monica, California/P-1771, September 3, 1959/ 13 pp.

SCHWARZ. A.

An Introduction to ALGOL

Communications of the ACM, February 1962/Volume 5/No. 2

SHAW, C.

A Specification of JOVIAL, Communications of the ACM

Volume 6/No. 12/ December 1963, pp. 721-735.

SHAW, C.

JOVIAL and Its Documentation, Communications of the ACM

Volume 6/ No. 3/ March 1963/ pp. 89-91

SIMON, H. A.

Experiments with a Hueristic Compiler

Rand Corporation, Santa Monica, California/P-2349/
June 30, 1961/85 pp.

SIMON, H. A.

Experiments with a Hueristic Compiler

Journal of the ACM October 1963/Volume 10/No. 4/pp. 493-506

SIMON, H. A.

The Hueristic Compiler

The Rand Corporation, Santa Monica, California/
Memorandum RM-3588-PR-May 1963/125 pp.

SLAGLE, J. R.

A Hueristic Program that Solves Symbolic Integration

Problems in Freshman Calculus

Journal of the ACM/October 1963/Volume 10/No. 4/pp. 507-520

TONGE, F. M.

Summary of a Hueristic Line Balancing Procedure

Rand Corporation, Santa Monica, California/P-1799/
September 18, 1959/ 44 pp.

TONGE, F. M.

The Use of Hueristic Programming in Management Science

A Survey of the Literature

Management Science/Volume 7/No. 3/April 1961/pp. 231-237'

UNGER, S. H.

GIT, A Hueristic Program for Testing Pairs of Directed

Line Graphs for Isomorphism

Communications of the ACM Volume 7/No. 1/January 1964/
pp. 26-34

YNGVE, P. H.

COMIT

Communications of the ACM Volume 6/No. 3/March 1963/
pp. 83-84

ZIEHE, T. W. and MARKS, S. L.

The Nature of Data in Language Analysis

The Rand Corporation, Santa Monica, California/January 17, 1961/
P-2197/16 pp.

Programming Systems Briefing - Assemblers and Compilers

IBM Corporation/April 1962/R25-1672-0

Report on the Algorithmic Language ALGOL 60, Communications of the

ACM/May 1960/Volume 3/No. 5

Structure and Use of ALGOL 60, Journal of the ACM, April 1962/

Volume 9/ No. 2.

"Compiling Techniques"

Teaching Notes, Howard Edelson, SRI

"Compiling Techniques"

Class Notes of Edith Taggert, Robert Long, and Marilyn Jensen, SRI