STANFORD RESEARCH INSTITUTE
MENLO PARK, CALIFORNIA

A BRIEF SURVEY OF COMPUTER LANGUAGES FOR SYMBOLIC AND ALGEBRAIC MANIPULATION*

By:

B. Raphael
Stanford Research Institute

D. G. Bobrow
Bolt, Beranek and Newman

L. Fein
Synnoetic Systems

J. W. Young
National Cash Register Company

(A report of the Comparison of Languages Subcommittee of the ACM Special Interest Committee on Symbolic and Algebraic Manipulation. This report was presented at the IFIPS Working Conference on Symbol Manipulation Languages, Pisa, Italy, September 1966.)

*The work reported herein was supported in part at Stanford Research Institute by the Air Force Office of Scientific Research, under Contract No. AF 49(638)-1752, and in part at Bolt, Beranek and Newman by the Advanced Research Projects Agency.

ABSTRACT

This paper is the result of a study conducted by the Comparison of Languages Subcommittee of the ACM Special Interest Committee on Symbolic and Algebraic Manipulation (SICSAM). It reports on the following programming languages: ALTRAN, AMBIT, COGENT, COMIT, CONVERT, CORAL, DYSTAL, FLIP, FORMAC, FORMULA ALGOL, IPL-V, LISP1.5, LISP2, L^6, PANON, SLIP, SNOBOL, AND TRAC. Several other languages are also briefly discussed.

The paper classifies each language as primarily a list processor, general-purpose language, linked block language, algebraic formula manipulator, pattern-directed string processor, or pattern-directed structure processor. For each category the paper:

(1) Describes properties that members of the group have in common;

(2) Gives a brief description of each language in the group including an excerpt from a program in the language that demonstrates the kind of problem for which the language is well suited, and

(3) Compares the features of the languages in the group.

The paper also contains as appendices:

(1) A reference chart that summarizes the features of all the languages covered;

(2) A comparison chart that emphasizes the salient distinctions between selected pairs of similar languages; and

(3) A set of annotated examples of programs in various languages that solve similar problems, thus illustrating differences in data representations, program forms, and notations.

A board of consultants, including experts in each of the languages, contributed data and reviewed a draft of this paper for the authors.

I INTRODUCTION

For conventional applications, stored-program digital computers are viewed primarily as number processors; the users require units of data that are numbers to be mapped into other numbers. On the other hand, for algebraic formula manipulation, information retrieval, computational linguistics, automatic decision-making, and other increasingly important applications, computers are more conveniently viewed as primarily symbol processors; the users require units of symbolic data to be transformed into other symbols or symbolic structures. Several papers in the literature describe the advantages and techniques of symbol processors.[1,2,3,4]* This paper surveys currently available programming languages for symbolic, rather than arithmetic, computation.

II NATURE OF THE SURVEY

The purpose of this report is to provide an overview, rather than specific detailed descriptions and analyses of the many currently available symbol-manipulation languages. No attempt is made to list all experimental or proposed symbol-manipulation languages. The following criteria for inclusion in this survey were used:

(a) The language should contain symbolic or algebraic manipulation facilities as integral features.

(b) It should be fully implemented by the time of this writing (July 1966).

(c) It should contain features that make it uniquely preferable over any other language for some class of users, conditions, or problems.

An expert in each of the included languages submitted data about his language; however, the authors alone are responsible for many of the judgments concerning the merits of various languages.

The following languages are covered: ALTRAN, AMBIT, COGENT, COMIT, CONVERT, CORAL, DYSTAL, FLIP, FORMAC, FORMULA ALGOL, IPL-V, LISP1.5, LISP2, L[6], PANON, SLIP, SNOBOL, and TRAC. For each of six groups of languages having somewhat similar characteristics, we shall (1) describe properties that members of the group have in common; (2) give a brief description of each language in the group, including an excerpt from a program in the language that demonstrates the kink of problem for which the language is well suited; and (3) briefly compare the features of the languages in the group.

---

* Superscript numbers refer to references given at the end of this paper.

Three appendices at the end of this paper contain the following: (1) a reference chart that summarizes the features of all the languages covered; (2) a comparison chart that emphasizes the salient distinctions between selected pairs of similar languages; and (3) a set of annotated examples of programs in various languages that solve similar problems, thus illustrating differences in data representations, program forms, and notations.

III LIST PROCESSORS

IPL-V[5,6] and LISP1.5[7,8] are the oldest and most widely-used list-processing languages. Each permits the construction and analysis of certain well-defined forms of tree and list structures. These languages are well suited only for those problems whose data can conveniently be encoded into the particular forms. In most implementations of these two languages, the representation of numbers and calculation of arithmetic results are particularly awkward and inefficient.

IPL-V[5,6]

IPL-V is an autonomous programming system having more than 100 list-processing, housekeeping, input-output, and arithmetic instructions. It is well documented and has been widely implemented. However, such newer developments as the notational and arithmetic convenience of SLIP;[9] the power of recursive definitions in LISP;[7,8] and the flexibility of low-level operations in L[6];[10,11] suggest that IPL-V is obsolescent.

IPL-V programs resemble programs written in machine language. Storage allocation, including retrieval of abandoned list cells for reuse ("garbage collection"), is the programmer's responsibility.

The following routine named R1, which reverses the order of the elements on a list, shows the general appearance of IPL-V code:

```
R1      J60      Skip first element
   70   J8       Terminate if done
   40   H0       Save current place
        R1       Reverse rest of list (recursively)
   12   H0       Get current element
        J65      Insert at end
        J68   0  Delete from top and stop
```

LISP1.5[7,8]

LISP1.5 is a language for defining and applying manipulative functions to binary trees of symbols. Standard encoding rules permit the programmer to manipulate data-list structures represented by the comma-and-parenthesis notation, e.g.

```
(THIS,IS,(A,LIST,STRUCTURE),OF,(((VARYING),DEPTH))).
```

LISP programs (i.e., function definitions) are usually represented internally as list structures themselves; however, a more understandable and less heavily parenthesis-laden "metalanguage" notation is used in some new LISP1.5 implementations. The most common program structure is a high-level conditional expression that permits recursion.

Push-down store maintenance and garbage collection are handled automatically. A function defined as follows creates a list which is the reverse of its argument list x ('car[x]' is the first element of the list x, and 'cdr[x]' is the rest of the list):

```
reverse[x] := [if null[x] then NIL
                  else append[reverse[cdr[x]];list[car[x]]]].
```

Comparison

IPL-V is a straightforward low-level list-processing language. LISP1.5 is a more sophisticated, more automatic language with a powerful but somewhat unconventional way of programming. More detailed descriptions and comparisons of these languages (along with SLIP[9] and COMIT[12]) may be found in Reference 13.

IV GENERAL-PURPOSE LANGUAGES

Some problems require both some symbolic and some numerical computation. Systems have been developed to provide the capabilities of both a symbolic-manipulation system and a numerical algebraic compiler. SLIP[9] and DYSTAL[14] are systems that give existing algebraic languages list-processing capabilities. LISP2[15] and FORMULA ALGOL[16] are new systems designed for both symbolic and numeric processing. The following discussions will focus upon the symbol-manipulation features of these four languages.

SLIP[9]

SLIP is an extension of FORTRAN, MAD, ALGOL, or any similar algebraic language. It consists of a repertoire of symbol-manipulating subroutines that may, except for a few machine-coded basic processes, all be written in the host language.

SLIP has been operational for several years. It has been embedded into many host languages on a variety of machines. Because SLIP is by definition always an extension of some well-known language, it is easy to implement and learn.

Its basic symbolic data form is a symmetric list structure with an identifying "head" cell. This permits pointers to scan forwards or backwards through list structures. A "reference" counter in each head cell permits continual, largely automatic garbage collection.

With the recent addition of a string pattern-matching feature (see Section VII), SLIP's versatility has been increased. The processing of arrays, list structures, and strings may now be mixed.

The following example creates a complex SLIP list structure:

```
      DIMENSION LST(17)
C     **'LIST(X)' STORES THE NAME OF AN EMPTY LIST INTO X
      CALL LIST (LST(1))
      DO 1 I = 2,17
C     **'NEWTOP(X,Y)' MAKES X A NEW FIRST ELEMENT OF
C     **    LIST NAMED IN Y
1     CALL NEWTOP(LIST(LST(I)),LIST(I-1))
```

DYSTAL[14]

DYSTAL is essentially a set of FORTRAN subroutines. The basic data element is a list, whose elements may be numbers, alphanumeric strings, or names of other lists. However, lists of names of other lists should generally be kept separate from lists of data. Each list is stored in a block, including a five-cell header, of consecutively addressed memory locations (rather than the usual linked-pointer structure). This system greatly speeds up the retrieval of the n-th item on a list (because it is now accessible by normal FORTRAN array addressing). However, the major advantage of variable-length list storage is lost. Each list-storage area is assigned, from a single available space block, at the time it is needed in the program. A list may be erased only under program control. Erasure consists of resetting the available space pointer, thereby erasing that list and all subsequently created ones. (Some additional flexibility is achieved by allowing the available space block to grow or shrink from either end.)

The following example is a DYSTAL program segment that reads in lists of attributes, sorts them according to a specified list of keys, and prints the records out in the new sorted order:

```
C     **READ IN ARRAY OF KEY ITEM NUMBERS,
C     **    AND READ DATA RECORDS
      LKEY = LSREAD(NKEY)
      CALL LSREAD(NAME)
C     **CALL SORTING ROUTINE, THEN PRINT.
C     **    "LOT" GETS CELL CONTENTS.
      CALL MSORT(NAME,LKEY)
      N = LOT(NAME)
      DO 10 I = 1,N
      LIST = ITEM(I,NAME)
      IMAT = LMAT(LIST)
10    CALL IPRINT(LOT(IMAT),1,LOT(LIST),LIST)
```

LISP2[15]

LISP2 offers the features of a list processor, a numerical algebraic compiler, and a pattern matcher, in a single uniform programming

system. In addition, low-level bit and logical operations give LISP2 capabilities comparable to machine languages.

LISP2 combines the semantics of ALGOL and LISP1.5 with a syntax that resembles that of ALGOL, but is augmented to include more data representations. Although at the top level every program consists of the evaluation of functions applied to arguments (as in LISP1.5), a function definition usually consists of block-structured declarations and program statements (as in ALGOL).

As long as only arithmetic functions, variables, and data are employed, LISP2 produces code comparable to that produced by a good ALGOL compiler. When list processing, array references, and arithmetic are mixed, the efficiency of the resulting code depends upon the programmer's insertion of appropriate declarations.

The following program computes the length of the longest initial segment common to two lists X and Y:

```
INTEGER FUNCTION LCOM(X,Y);
     BEGIN INTEGER N;
          FOR N←0 STEP 1 WHILE X AND Y AND CAR X = CAR Y DO
          BEGIN X←CDR X; Y←CDR Y END;
          RETURN N
     END
```

FORMULA ALGOL[16]

This language uses the control structure and most other attributes of ALGOL. However, it provides capabilities to do arithmetic and logical computation, algebraic formula manipulation, list processing, and some string processing.

The data space and operation set of ALGOL 60 were extended by adding two simulated machines in the form of packages of run-time routines to create and manipulate formulae and list structures. New kinds of declarations, expressions, and statements were added to the ALGOL syntax.

Transfer functions in FORMULA ALGOL map one type of data object onto another; e.g., an algebraic formula created by means of list-processing and symbolic-data manipulation may be "transferred" into a real number by complete substitution of numbers for variable names and by evaluation of the result. Transformations may be defined which map a given class of data objects onto another.

One area to which FORMULA ALGOL is particularly well suited is formula manipulation in unusual mathematical systems. For example, if the distributive and commutative laws do not hold, several features of the language dealing with formula manipulation can still conveniently

be used--e.g., Markov algorithms, substitution processes, and formula evaluations.

The following sample FORMULA ALGOL statements are from a program that solves algebraic equations containing a single occurrence of the variable X on the left hand side: (See Appendix III for further explanation.)

```
BEGIN
     FORM A,B,C,X; SYMBOL PLUS,TIMES,S;
     .
     .
     .
     B←B: ANY;
     C←C: ANY;
     PLUS←/[OPERATOR:+][COMM:TRUE];
     TIMES←/[OPERATOR:*][COMM:TRUE];
     S←[[
       (A|TIMES|B) = C → .A = .C / .B,
       A / B = C → .A = .C * .B,
       .
       .
       .
       X = C .→ .X = .C ]];
     E ← K*H + (M/(H-K)+X)*N = P;
     PRINT (E,E.↓S);
END
```

Comparison

All four languages discussed in this section have both symbolic and arithmetic computation facilities. SLIP and DYSTAL extend host languages to fit them for certain classes of list processing. SLIP has been used successfully for several list-processing applications. DYSTAL is a new and relatively untested language. Its limitations of (almost) fixed-length lists, and of sequential erasure of symbolic-data storage, make it best suited for numeric applications requiring only limited amounts of list processing. However, sequential list storage will simplify time-sharing implementations.

The two eclectic systems, LISP2 and FORMULA ALGOL, are both still in an early experimental stage; each exists as a single implementation on a little-known computer. LISP2 seems to have advantages for general list manipulation and string processing; however, it is designed to operate only on a very large computer in a time-shared environment. FORMULA ALGOL offers built-in packages for algebraic formula manipulation; implementation is practical for medium-size, batch-processing computer systems. However, its general list-processing facilities are somewhat limited.

V LINKED-BLOCK LANGUAGES

In the aforementioned symbol processors, computer memory cells are assigned to data automatically. L6[10,11] and CORAL,[17,18] on the other hand, give the user low-level control--and responsibility--for details of memory allocation such as the size of blocks of sequential storage to be reserved and the locations of link pointers within those blocks. At the expense of considerable attention to memory organization and housekeeping details, an L6 or CORAL programmer can construct extremely efficient list-processing programs tailored to particular problems.

L6[10,11]

The elementary data unit in L6 is the memory block, which may be defined to contain 1,2,4,..., up to 128 words. The programmer defines bit fields within the blocks. Each field has a one-character name; this name refers to a specified set of bits (say 4-25) of a particular word (say the 3rd) in every block in the system.

Twenty-six base registers are each identified by a single character. A base-register identifier followed by a sequence of field identifiers specifies a field. Thus ABC means "Take base register A; take the B field of the block addressed by A; the C field of the block it points to contains the data."

Operations in L6 can move, compare, and test fields; perform arithmetic and logical functions; save and restore field contents; and redefine fields.

An example of coding from a routine for sorting data follows:

```
ORDER     THEN(S,FC,X)(X,P,WA)
NDTEST    IF(XA,E,0)THEN(R,FC,X)DONE
BACK      IF(XB,E,XDB)THEN(XDA,P,XA)(XAD,P,XD)(X,FR,XA)NDTEST
          IF(XB,L,XDB)THEN(XB,IC,XDB)(X,D)BACK
          THEN(X,A)NDTEST
```

CORAL[17,18]

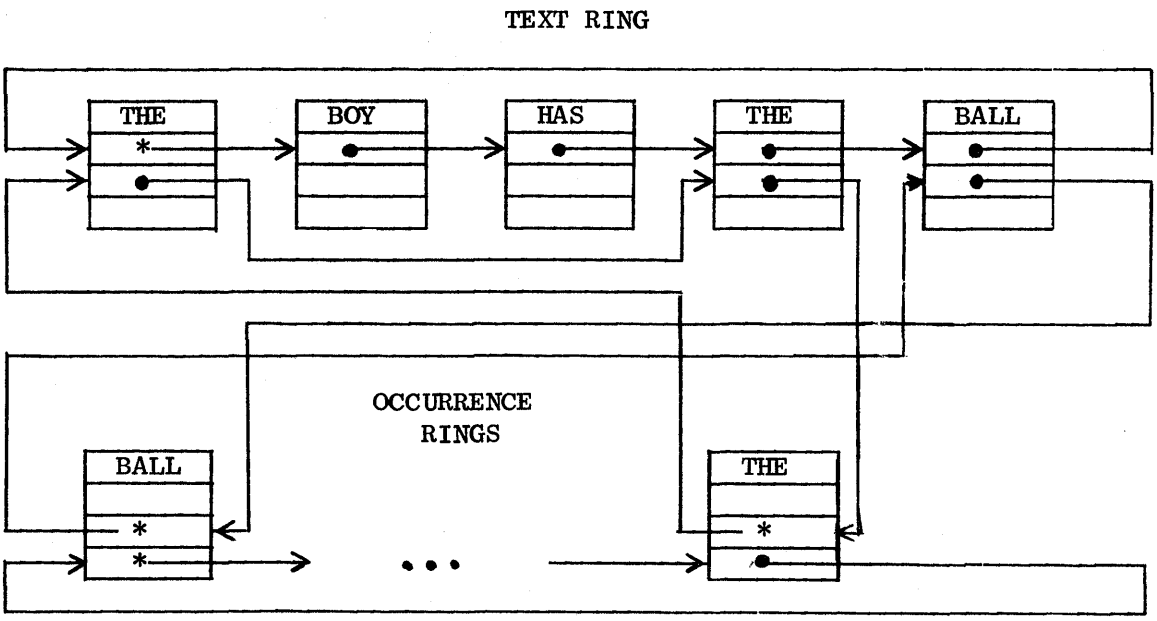
CORAL includes operations for creating and modifying linked-block structures, doing arithmetic, and performing higher-level list-processing functions such as "recursive-delete" which removes from a structure all blocks which are subordinate to a given block.

Perhaps the major restriction in the system is that the number of pointers in a block is fixed at program-writing time, so that the general nature of the information in a structure is not variable (although of course the individual connections are).

The principal implementation of CORAL is on the TX-2 computer at MIT Lincoln Laboratories. The syntax of this language, which uses the TX-2 keyboard character set, is too obscure to permit an understandable example. The following program statement is typical:

DICTYES→ ((TEXTP(β)|TIE) ⇓ 1)(≤)(DICTW ⇓ 1) ← CONTROL

However, the following diagram illustrates the sort of data structure that may be used. Here some text is organized into "rings" by word sequence and occurrence:



* Ring Start

Comparison

CORAL and L^6 both have linked blocks. However, the L^6 programmer specifies the link elements directly and operates on them individually. L^6 is a relatively small system, easily implemented on new machines. Higher, more sophisticated languages can be written in L^6. CORAL is a higher-level linked-block processor with a set of more complex operations. New notation would have to be invented for new implementations of the CORAL language.

CORAL has been used extensively for on-line modeling of graphic display data. L^6 has a much simpler control language and is more suitable for batch-processing.

VI ALGEBRAIC-FORMULA MANIPULATORS

For many problems in applied mathematics it is necessary to carry out extensive and complex algebraic or analytic derivations on symbolic mathematical expressions. A recent bibliography[36] surveys the use of computers for various non-numerical mathematical tasks. FORMAC[4,19] and ALTRAN[20,21] are systems designed specifically for the manipulation of symbolic-algebraic expressions. AMBIT[22,23] is a more general symbol-manipulation system that somewhat resembles the string pattern-matching systems of Section VII. However, its facilities are so strongly oriented towards operations on formal mathematical expressions that we include it in this group.

FORMAC[4,19]

FORMAC is an extension of FORTRAN IV. It can manipulate algebraic expressions what may be of any form, within roughly the family of explicitly defined real elementary functions. An expression can be named, and the name used as an argument of an arithmetic operation, of a function (EXP, SIN, differentiate, binomial coefficient, etc.), or of a system command.

The most important system commands are:

LET (Assignment statement for symbolic data)

SUBST (Substitution)

EXPAND (Multinomial expansion and distributive law)

COEFF (Determines the coefficient of a power of a variable)

PART (Produces the first well-formed part of an expression, i.e., yields the first term or factor of a sum or product)

ORDER (Specifies the sequence of variables, etc., in an expression)

FIND (Determines if one or more variables appear, either explicitly or by implicit dependence, in an expression)

EVAL (Produces a FORTRAN numeric value from a FORMAC expression)

MATCH (Compares two expressions for exact identity or mathematical equivalence)

The limited amount of working space available for expressions is the severest limitation of FORMAC. Automatic simplification of resulting expressions occurs after execution of every FORMAC instruction; however, since explosive expression growth is inherent in straightforward formula manipulation, careful programming is still necessary.

The following program segment finds the first M coefficients $a_i(x)$ of the power series expansion of the function G(x,t) about t = 0:

```
      DO 100 N = 0,M
C     ** FMCFAC MEANS FACTORIAL, FMCDIF MEANS DIFFERENTIATE
      LET A(N+1) = 1/FMCFAC(N)*FMCDIF(G,T,N)
C     ** REPLACE T BY ZERO
      LET A(N+1) = SUBST A(N+1),(T,0)
C     ** MULTIPLY OUT PRODUCTS AND POWERS OF SUMS
100   LET A(N+1) = EXPAND A(N+1)
```

ALTRAN[20,21]

ALTRAN, like FORMAC, is an extension of FORTRAN. It is specifically directed toward large-scale computations with rational functions of several (perhaps many) variables. An earlier, stand-along formula-manipulation system called ALPAK[21] also had facilities for truncated power series with rational-function coefficients and for systems of linear equations with rational-function coefficients. However, these have not yet been implemented in ALTRAN.

ALTRAN permits addition, subtraction, multiplication, division, integral exponentiation, substitution, differentiation, and GCD computation on rational algebraic functions. The system also allows the user to specify "side relations" which handle, in a limited way, certain irrational quantities.

Typical fragments of an ALTRAN program follow:

```
      POLYNOMIAL A1,A2,A3,B1,B2,B3,F
C     ** '=' '*' AND OTHER OPERATORS ARE NOW SYMBOLIC, NOT
C     ** FORTRAN ARITHMETIC, FOR A1,A2,...,F.
      A1 = (R0+R1)**2
      A2 = (R0+R2)**2
      B1 = R0**2+R0*R2+R0*R3-R2*R3
      .
      .
      .
      F = A1*A2*A3-A1*B1**2-A2*B2**2-A3*B3**2+2B1*B2*B3
      PRINT F, F(R0=1/C0,R1=1/C1,R2=1/C2,R3=1/C3)
```

AMBIT[22,23]

An AMBIT program processes a single data string, consisting of a sequence of characters. The sequence is divided, by blanks, into segments. Typical segments are an identifier, a number, a mathematical operator, a parenthesis, or a special place-marker called a pointer, which acts as a reference point for manipulation of the data string.

The basic operation in AMBIT is the replacement rule S1→S2, where S1 and S2 are string descriptions and "→" is read "shall be replaced by". A string description consists of literals and dummy

variables; the variables may be declared to represent particular classes of subsequences. The pattern string description S1 must contain a pointer; the AMBIT rule is similar to a statement in a pattern-matching language (Section VII); however, the pattern string description must contain a pointer and balanced parentheses. These restrictions permit the efficient manipulation of parenthesized strings.

The following program moves a pointer pΔ through an expression to "multiply out" all products of sums or differences.

```
SCAN:   EQ2Δ Q → EQ2Δ  pΔ  Q;
MULT:   if pΔ  AX(B sign C) → (AXB) sign (AXC)
        or pΔ  (A sign B)XC → (AXC) sign (BXC)
        then go to SCAN;
        pΔ  seg → seg pΔ  ;
        if EQ2Δ  Q pΔ  → EQ2Δ  Q
        then go to EXIT
        else go to MULT;
```

Comparison

AMBIT is an excellent tool to aid in the writing of a formula manipulator, but is clearly not one itself. Its operations are oriented toward the programmer who would like to design a formula-manipulation system, not the engineer who would use it. AMBIT could also be used for applications other than formula manipulation where the manipulation of parenthesized strings of symbols is useful.

ALTRAN runs with high efficiency in time and storage for the limited class of problems for which it was designed, i.e., manipulation of rational functions. FORMAC, however, has much greater flexibility. It offers the user an assortment of both low-level operations (e.g., COEFF) which allow him to program his own detailed manipulation, and high-level ones (like differentiation) which carry out very complex operations.

VII STRING PROCESSORS

A. Pattern-Directed

The pattern-directed string-processor languages have a common basic structure, related to the mathematical model of processing known as a Markov algorithm. A specified data string is compared with a pattern. If it matches (according to the criteria of the language) then the input string is transformed according to a format associated with the pattern, usually utilizing the parsing of the original string determined in the matching process. If the match is unsuccessful then no transformation occurs. In some languages the results of the parsing can be saved and the segments found can be utilized later in the program, outside the individual matching statement.

The basic units of the data string, called items, may be either individual alphanumeric characters, special characters, or pre-specified strings of characters. The entire data string to be scanned for a match, called the workspace, may be unique in the system or it may be specified independently for each rule.

A pattern consists of a sequence of elementary patterns. A pattern matches the workspace if each of the successive elementary patterns matches successive contiguous segments of the workspace. Typical elementary patterns include literals; variables which match any segment of the workspace; variables which match any segment of specified length, i.e., containing a specified number of items; segments identified by previously-assigned names; and segments which have special properties or belong to certain classes.

The construction of a new string is specified by a format which is a sequence of elementary formats. Typical elementary formats include literals, segment names, and string-segment-valued function calls. The constructed string usually replaces the matched segment of the workspace.

COMIT II[12]

COMIT is the oldest and best documented pattern-directed string processor. COMIT II is a streamlined system that contains several new features but can run all old COMIT programs.

String items are strings of characters optionally tagged with subscripts. The characters and subscripts in an item may easily be changed under program control. Numerically-addressed "shelves" provide temporary storage for strings when they are not in the workspace.

The following program will read a deck of cards and punch out just those cards containing the word "THE".

```
          COM
    K  BLANKS
A $+THE+$//*WAI1 2 3 *
* $      //*RTK1 A
          END
```

SNOBOL3[24,25]

The SNOBOL string items are always individual characters. However, any segment can easily be given a mnemonic name. Such names are assigned automatically to variable symbols during matches, and may be used to retrieve the segments for future processing. Functions may be defined, in SNOBOL, to test or transform strings.

The following program segment removes all occurrences of the letters A, E, I, O, U from a string named TEXT:

```
START VOWEL = "A,E,I,O,U,"
V1    VOWEL *V* "," =        /F(END)
V2    TEXT V =           /S(V2)F(V1)
END          •••
```

PANON-1B[26]

This new language allows the programmer to define variables that will match any string accepted by a specified context-free grammar. The production rules of the grammar appear as part of the PANON program.

The following PANON statements are from a program that converts fully-parenthesized arithmetic expressions into Polish prefix notation:

```
^-CP*   ^E    ^-/*        ^-LET
              ^-/*        (^E ^OP ^E)
^-CD*   ^OP   ^-/*   +    ^-/*  -  ^-/*  *  ^-/*  /
^-TR*/CONV*
   (^E*/1  ^OP  ^E*/2*)  ^==  ^OP  ^E*/1  ^E*/2
                                  ^-GOTO*/CONV
```

Comparison

COMIT has flexible input-output and subroutine-linkage facilities, a "list rule" for fast dictionary searches, and logical subscripts allowing convenient multidimensional tagging of items. SNOBOL permits mnemonic names for strings, explicit function calls, and automatic checking for parenthesis-balanced strings. The languages are so close in capabilities that considerations such as availability of information and implementation would probably be dominant selection criteria.

PANON is a relatively untested language that offers new power for recursive scanning of highly structured string data.

B. Macro-Expander

A somewhat unique string processor called TRAC has recently been introduced. The TRAC (Text Reckoning And Compiling) language[32] is an interpretive, string-manipulating language designed for on-line interactive use. In the TRAC language, one can write procedures for accepting, naming, and storing any character string emitted from a teletypewriter or other device; for modifying any string; for treating

any string as an executable procedure, as a name, or as a text; and for printing out any string.

The published version of TRAC contains primitive functions for: typewriter input and output, naming and calling strings, text and procedure macro generation, management of back-up storage, test and branching, integer arithmetic, logic vector (strings of 0's and 1's) manipulation, and diagnostics. The logical foundations of TRAC derive from the notion of a macrogenerator (specifically Eastwood and McIlroy's "Macro-SAP" of 1959) as extended to test strings. TRAC is particularly convenient for synthesis of strings and executions of procedures, but in its present state it is relatively clumsy in certain analytical situations, such as parsing fully parenthesized arithmetic expressions.

At the installations where the TRAC system is now implemented, the more interesting directions of experimental application seem to be: (1) for use in various kinds of text management and storage; e.g., editing and machine aided instruction; (2) for a command system for the control of hardware devices, e.g., driving an automatic telephone dialling set, or a graphic display; (4) as a modular logical base for the insertion of additional machine-coded primitives, e.g., floating point arithmetic capabilities, pattern-matching; and (5) as the complete operating system for multiple-user management of a computer, i.e., by dropping a time-sharing package into the TRAC translator which is shared by all the users.

The following illustration of TRAC programming illustrates naming and storage of text, defining and calling of an iterative (recursive) command procedure, and "plain language" user interaction under control of the command procedure. The apostrophe is the terminator of the input string, and what is typed out by the computer is underlined.

#(DS,TEXT,THIS IS TRAC)'	names and stores string
#(DS,PROGRAM,(#(PS,(**))#(PS,( )#(CL,#(RS)))#(CL,PROGRAM)))'	records command procedure
#(CL,PROGRAM)'	initiates command procedure
**TEXT'	computer asks for input, name provided
THIS IS TRAC	named string is printed out
**	waiting for next input command

VIII PATTERN-DIRECTED STRUCTURE PROCESSORS

Pattern-directed data-processing features, similar to features of the aforementioned string processors, have been embedded into more-general symbol-manipulation languages. [For a discussion of some of the advantages and disadvantages of embedding, see Reference 27.] The principal advantages of embedding are that the pattern-directed

processor may defer to the host language for such facilities as input, output, storage management, and flow of control; and the capabilities of both the pattern matcher and the symbol manipulator are available, producing a more powerful overall system.

Pattern-directed features have been implemented in SLIP, and will be important components of both LISP2 and FORMULA ALGOL. CONVERT[28] and FLIP[29] are essentially pattern-directed structure processors embedded into LISP1.5. COGENT[30,31] represents a somewhat different approach to manipulating highly structured data.

CONVERT[28] and FLIP[29]

These languages are like the pattern-directed string processors except that they operate on LISP list structures, rather than character strings. The following additional types of elementary patterns are available: subpatterns, i.e., specifications on the substructure of a matched list-structure item; special matches, such as elementary patterns for repeated segments, a segment which matches one of several alternatives, or an item not equal to a specified item; and function patterns--i.e., within a match an arbitrary LISP function may be called to test the acceptability of a segment or influence the match in other ways.

Suppose we wish to merge a list of two lists into a single list by taking alternate elements, e.g., merge[((0 1 2)(A B C))]= (0 A 1 B 2 C). After declaring that P and Q are element variables and PPP and QQQ are segment variables, the following two CONVERT rules will do the job. The first rule recursively merges the two sublists, and the second is the terminating condition: If the two sublists of the input are empty, then the output is the empty list.

```
(((P PPP)(Q QQQ))(P Q (*BEGN* ((PPP)(QQQ)))))
((NIL    NIL)    NIL)
```

An example of the pattern part of a FLIP rule is:

```
($3 $3/(EQUAL(=REVERSE 1)) )
```

The list following a "/" modifies the immediately preceding elementary pattern. The "1" refers back to the first elementary pattern, i.e., the first $3. This pattern will match a list of six elements, the second three of which are the same as the first three but in reverse order.

COGENT[30,31]

This is a pattern-directed system with aspects of both string and list processing. Externally, the data consists of phrases of some context-free phrase-structure language whose syntax is specified by "production rules" in the COGENT program. The basic operations of COGENT synthesize and analyze these phrases according to patterns which are phrases of the data language containing variable subphrases. Internally, however, both the data and patterns are represented by list structures which are obtained by parsing the data language, so that the basic operations are a type of list structure pattern operation. More general types of list structures may also be handled, and arithmetic and symbol-table facilities are included.

The following COGENT routine will accept two list structures representing algebraic expressions and produce a structure representing their product without introducing redundant parentheses:

```
$GENERATOR PRODUCT((X,Y)
  +1 IF X =/ (EXP/(TERM)),X. X/=(TERM/(()(EXP)())),X.
1/+2 IF Y =/ (EXP/(FACTOR)),Y. Y /=(FACTOR/(()(EXP)())),Y.
2/      X /= (EXP/(TERM)*(FACTOR)),X,Y. $RETURN(X). ).
```

(The symbol "=/" indicates analysis, "/=" indicates synthesis, and "+" denotes "go to".) This routine can be used only after the syntax of EXP, TERM, and FACTOR has been described by production rules including, for example,

```
(TERM) = (FACTOR), (TERM)*(FACTOR), (TERM)/(FACTOR).
```

Comparison

CONVERT and FLIP are both powerful systems which have concise notations for mixing pattern matching with general list processing. The differences between them, aside from gross notational differences, are subtle and beyond the scope of this report.

COGENT is an experimental system specifically designed for processing sentences of some context-free phrase-structure language.

IX OTHER LANGUAGES

To the best of the authors' knowledge, this paper is a report of all languages that come close to meeting the criteria of Section II. The SICSAM Comparison of Languages Subcommittee would appreciate hearing about omissions or new developments as they occur for inclusion in future reports.

The following languages, although not surveyed in detail here, are of sufficient interest to mentioned briefly:

GPM[33]

The General Purpose Macrogenerator developed on the Atlas 2 computer at Cambridge, England, is in many respects similar to TRAC. However, it is intended for use by experienced system programmers rather than by scientists.

AXLE[34]

The basic idea of this pattern-directed string processor is similar to that of PANON. Side conditions, in the form of production rules, define the syntax of acceptable strings. AXLE has not been implemented.

TREET[35]

This experimental variant of LISP1.5 is implemented on the STRETCH computer at Mitre Corp. The basic data structure is a trinary tree. A subsystem called OAKTREET operates on-line display and light-pen facilities.

AED

The AED languages, being developed by Dr. Douglas Ross at MIT, are designed to combine the power of Algol with special symbol-manipulation facilities for computer-aided design and on-line display control research.

PL/I

This "universal" language will have various symbol-manipulation facilities. However, they have not been emphasized sufficiently to be considered in this report.

X ACKNOWLEDGEMENTS

The authors wish to acknowledge the invaluable contributions made to this report by the language consultants listed in the reference chart (Appendix I). These consultants may be contacted directly for further information about their respective languages.

APPENDIX I

The attached reference chart summarizes the information in this report and tabulates details, such as implementations, that were not included in the text.

NAME	DATA FORM: PROGRAMMER'S	DATA FORM: INTERNAL	PROGRAM DESCRIPTION	IMPLEMENTATIONS	BASIC CAPABILITIES	THUMBNAIL EVALUATION	CONSULTANT
IPL-V [5,6]	Lists of elements which name data terms or other lists.	Binary trees (i.e., 2 address fields per word) representing list structures.	Lists of elementary operations and subroutine calls.	IBM709/90,650 CDC 1604 Bendix G20 Philco 2000 Univac 1105 AN/FSQ-32 Prob. others	Low-level, pure list processing	Obsolescent	Prof. A. Newell, Carnegie Inst. of Tech., Pittsburgh, Pa.
LISP1.5 [7,8]	Parenthesized list structures and dotted pairs.	Binary trees.	Functions, defined by conditional expressions and recursion, applied to arguments.	IBM709/90,1620 AN/FSQ-32 PDP-1,6 AFCRL M460 SDS 930/40 B5500 Prob. others	High-level, pure list and tree processing.	Elegant, but lacks some practical needs.	Dr. B. Raphael, Stanford Research Inst., Menlo Park, Calif.
SLIP [9]	Parenthesized list structures and standard host language input data.	Headed lists of 2-word blocks linked both ways and all permissible host language symbols.	Host language extended with list-process. functions.	IBM7090/94 CDC 1604,3200 IBM7044,1620 Philco 2000 Atlas; IBM360 AN/FSQ-32 Prob. others	Host language arithmetic mixed with symbol and text manipulation.	A successful intrusion of list processing into numeric algebraic languages.	Prof. J. Weizenbaum, Project MAC, MIT, 545 Technology Sq., Cambridge, Mass.
DYSTAL [14]	Character strings, numbers or arrays; chains of symbolic arrays.	Arrays with special 5-word heading, and all FORTRAN symbols.	FORTRAN extended with symbolic-array manipulation functions.	IBM7070; IBM360-50, basic subroutines only.	Arithmetic mixed with limited kinds of list and string processing.	Minimal list-processing capabilities.	Prof. J. Sakoda, Brown University, Providence, R.I.
LISP2 [15]	Numbers, character strings, truth values, parenthesized list structures and dotted pairs, arrays.	Binary trees, numbers and arrays.	Functions, defined by ALGOL-like block structures, applied to arguments.	AN/FSQ-32 IBM360-65* PDP-6* *Scheduled by 1/67.	Arbitrary mix of arithmetic and symbolic computation.	Highly promising, but presently experimental.	Dr. S. Kameny, SDC, 2500 Colorado Ave., Santa Monica, Calif.
FORMULA ALGOL [16]	Numbers, truth values, arrays, algebraic formulae, list structures.	Numbers, arrays, linked lists, binary trees.	ALGOL extended with formula and list-processing constructs.	CD G-21; Planned for IBM360-67	Mix of arithmetic and symbolic computation, emphasizing formula manipulation.	Highly promising but presently experimental.	Mr. T. Standish, Computation Center, CIT, Pittsburgh, Pa.

NAME	DATA FORM: PROGRAMMER'S	DATA FORM: INTERNAL	PROGRAM DESCRIPTION	IMPLEMENTATIONS	BASIC CAPABILITIES	THUMBNAIL EVALUATION	CONSULTANT
L^6 [10,11]	Character string, formatted under program control.	Linked blocks of various sizes; linkages and fields specified by programmer.	Conditional statements involving concatenated field identifiers.	MOBIDIC (Nat'l Bureau of Stds, Washington, DC) IBM7094,7040 Planned for: GE 635/45 IBM 360; PDP-7 SDS 940	Low-level program control of linked-block memory structures.	A chance for programmers to do efficient low-level list processing.	Dr. K. Knowlton Bell Telephone Labs., Murray Hill, N.J.
CORAL [17,18]	Character string, light-pen or button action.	Linked blocks of format specified at start of run.	Obscure notation, based on TX-2 character set, for manipulating ring structures.	MIT TX-2 PDP-7	Linked-block data with independent threaded rings.	Similar to L^6; more sophisticated but less easily obtainable and less flexible.	Mr. W. Kantrowi MIT Lincoln Lab Lexington, Mass
FORMAC [4,19]	Mathematical expressions representing explicitly defined real elementary functions.	Prefix-delimiter Polish strings.	FORTRAN IV extended for formal algebraic operations.	IBM7090/94	Formula manipulation and arithmetic.	Practical formula-manipulation system.	Mr. Peter Marks IBM, 545 Techno Sq., Cambridge, Mass.
ALTRAN [20,21]	Mathematical expressions representing rational functions.	Blocks linked into rooted directed graphs with no loops	FORTRAN II extended for formal rational function operations.	IBM7040/44 IBM7090/94	Formula manipulation and arithmetic.	More efficient than FORMAC for a restricted class of users.	Dr. W. Brown, Bell Telephone Labs., Murray Hill, N.J.
AMBIT [22,23]	Parenthesis-balanced character string.	Symmetric linked lists.	Replacement rules structured by pointers or parentheses.	CD 1604 Written entirely in ALGOL, mostly machine-independent.	Detailed processing of formulae and other highly structured strings.	For the formula-manipulation systems designers.	Mr. C. Christen Computer Assocs Lakeside Office Wakefield, Mass
COMIT [12]	Strings of character strings which may have subscripts.	Linked 2-word blocks with short-cut links.	Pattern-directed string processing.	IBM7040/44 IBM7090/94	Arbitrary string transformations.	An established string-processing system.	Prof. V. Yngve, Graduate Librar School, U. of Chicago, Chicag Ill.

NAME	DATA FORM: PROGRAMMER'S	DATA FORM: INTERNAL	PROGRAM DESCRIPTION	IMPLEMENTATIONS	BASIC CAPABILITIES	THUMBNAIL EVALUATION	CONSULTANT
SNOBOL [24,25]	Strings of characters.	Indexed blocks of string symbols.	Pattern-directed string processing.	IBM360/40 IBM7040/44 IBM7090/94 RCA601/604 SDS930/940 CDC3100	Arbitrary string transformations.	An established string-processing system.	Dr. R. Griswold, Bell Telephone Labs., Holmdel, N.J.
PANON-1B [26]	Strings of characters.	Sequence of string symbols.	Context-free string descriptions and pattern-directed string processing.	CSCE/CEP (U. of Pisa) Planned for: IBM 7040/90	Recursive pattern-directed transformations of phrase-structured data strings.	Experimental new language.	Dr. A. Caracciol Centro Studi Calcolatrici Elettronichi, Via Santa Maria, Pisa, Italy
TRAC [32]	Strings of characters.	Linear strings or binary trees.	Nested functions applied to arguments.	PDP-1,5,8 SDS 930 GE Datanet 30 IBM 7094 (MIT) SAAB D-21 (Swedish) ICT 1202 (British) Others	High level interactive string manipulation.	Experimental versatile on-line language.	Mr. C. N. Mooers Rockford Researc Inst., Inc., 140 Mt. Auburn St., Cambridge, Mass.
CONVERT [28]	Numbers, list structures, arrays, character strings.	Binary trees, numbers and arrays.	Pattern-directed structure processing.	IBM 7090/94 AN-FSQ-32 PDP-6	Pattern-directed transformations mixed with arbitrary symbolic computation.	An experimental approach to concise high-level programs via mixed pattern matching and list processing.	Mr. A. Guzman, Project MAC, 545 Technology S Cambridge, Mass.
FLIP [29]	Parenthesized list structures and dotted pairs.	Binary trees.	Pattern-directed structure processing.	IBM 7094 PDP-1	Pattern-directed transformations mixed with arbitrary symbolic computation.	Similar to CONVERT.	Dr. D. G. Bobrow Bolt, Beranek & Newman, 50 Moult St., Cambridge, Mass.

NAME	DATA FORM: PROGRAMMER'S	DATA FORM: INTERNAL	PROGRAM DESCRIPTION	IMPLEMENTATIONS	BASIC CAPABILITIES	THUMBNAIL EVALUATION	CONSULTANT
COGENT [30]	Sentences of some context-free phrase-structured language.	Linked blocks.	Production rules defining data language; analysis and synthesis function generators.	CDC3600/3800	Analysis and synthesis of sentences of a suitable formal language.	Powerful for limited application, somewhat obscure.	Prof. J. C. Reynol Applied Math. Div Argonne Nat'l. Lal Argonne, Ill.

APPENDIX II

Comparison Chart

Some of the languages discussed in the paper are, at least superficially, quite similar to each other. The attached comparison chart emphasizes the salient distinctions between selected languages. The language characteristics listed on this chart were chosen primarily for their ability to discriminate between the selected languages, rather than their importance in the use of the languages.

Comparison Chart

(a) List Processors

	LISP1.5	IPL-V
Program form	Function definitions and evaluations	Lists of executable statements
Symbols	Arbitrary mnemonics	Highly restricted
Data	Arbitrary binary trees	List structures only
Storage maintenance	Automatic "garbage collection"	Under program control
Recursion and Subprogram linkage	Fully automatic	Push-down stores and other "housekeeping" under program control

(b) Linked Block Languages

	L^6	CORAL
Implementations	Many (see Ref. Chart, Appendix I)	Relatively inaccessible
Program formalism	Primitive statement forms, restricted character set	Concise but difficult to learn notations, large and unconventional character set
Data block sizes	2^n cells per block, n=0,1,...,7. Mixture of block sizes and pointer structure under program control.	Arbitrary size, but must be fixed (with pointer conventions established) at start of run.
Programming	Low level; arbitrary definitions of, and operations on, data fields.	Built-in processes for manipulating "rings," particular higher-level structures of data blocks.

(c) Pattern-Directed String Processors

	COMIT	SNOBOL
Data	Strings of "constituents" which are single or groups of characters with optional subscripts.	Strings of individual characters.
References for temporary storage	Shelf numbers	String names
Working string segments referenced by:	Position numbers	Automatically assigned names
Special pattern match conditions	Conditions on subscripts	Parenthesis balance

(d) Algebra Manipulators

	FORMAC	ALPAK-ALTRAN
Data	Any mathematical formula	Rational algebraic expressions
Simplification	"Zero and one" simplification automatic; factoring and expansion under program control.	Automatic reduction by GCD
Stress	Generality	Efficiency, for restricted class of data.
Program form	New syntactic forms; e.g., "LET" statements	Extended semantics, with TYPE ALGEBRAIC declarations

(e) General-Purpose Languages

	Formula Algol	LISP2
Special data type	Algebraic expression	Functional expression
Derivation	Algol, extended for list processing	Algol syntax, LISP program and data structures
Implementations	G-21; 360-67 planned	Q-32; 360-65 and PDP-6 in progress
Declarations	Required for all variable types	Optional; necessary only to ensure efficiency
System size	Moderate	Presently impractically large

Appendix III: Annotated Sample Programs

In order to provide a basis for comparing symbol-manipulation languages with respect to data representations, program forms, and notations, a choice of two problems was offered to the language consultants listed in the Reference Chart (Appendix I):

Problem 1: Convert a fully-parenthesized algebraic expression into prefix form.

Problem 2: Solve an algebraic equation for the single occurrence of some variable x.

Each consultant was asked to select the problem most easily programmed in his language, and to submit an annotated program for its solution.

For certain languages, e.g., TRAC, neither problem was considered suitable. Also, we must realize that these brief examples cannot begin to show the full power of most of these languages. Nevertheless, the programs printed below should be illuminating.

Only two programs were submitted for problem 2 ("Solve an equation for x"): one in ALTRAN and one in FORMULA ALGOL. Of the twelve solutions for problem 1, ("convert to prefix form"), several different interpretations of the problem lead to different solution algorithms. In particular, some of the programs output a Polish parenthesis-free string; others retain the parentheses, thereby allowing operators with an indefinite number of arguments, but rearrange terms into the operator-first "Cambridge Polish" form. The particular problem being solved should be clear from the discussion associated with each program.

A. IPL-V

This program translates, from infix to prefix form, a list structure defined by the BNF

```
e:: = symbol | (e symbol e)
```

In this case it is unnecessary to distinguish the primitive terms from the operators.

The method relies on the fact that the prefix form can be obtained by replacing the open parenthesis by the operator and deleting the operator and the closing parenthesis. Note that the executive is independent of the routine.

Program:

```
EXEC                                E0      11W20
(SET TO PRINT ON TELETYPE)                    J125
CONVERT L0                                  50L0
                                              910
CONVERT M0                                  10M0
                                              910
CONVERT N0                                  10N0
(PRINT TELETYPE POST MORTEM)                  910     $18
LOCAL SUBROUTINE:                   910     40H0
EXECUTE P0 ON INPUT AND PRINT                 P0      J150
CONVERT INPUT TO PREFIX FORM        P0        J60
DO LOCAL SUBROUTINE 910                       910     J8
GET FIRST SYMBOL                    910     12H0
                                            10(
TEST IF FORM:  (EXP OPR EXP)                  J2
IF NOT, OR FORM:  EXP. GO PAST              70J60
PUSH (-LOCATION STACK                       40W0
SAVE (-LOCATION                             60W0
GO TO NEXT                                    J60
RECURSE ON LEFT SUBEXPRESSION                 910
GET OPR SYMBOL                              12H0
REPLACE ( BY OPR (MAKE PREFIX)              21W0
POP (-LOCATION STACK                        30W0
DELETE MIDDLE OPR                           31H0
RECURSE ON RIGHT SUBEXPRESSION                910
DELETE FINAL )                              31H0    0
```

Data:

```
                                    L0      0
                                            (
                                            A
                                            +
                                            (
                                            B
                                            -
                                            C
                                            )
                                            )       0
                                    M0      0
                                            (
                                            (
                                            A
                                            +
                                            B
                                            )
                                            -
                                            C
                                            )       0
```

```
NO      0
        (
        (
        A
        +
        B
        )
        *
        (
        C
        -
        D
        )
        )       0
```

Resulting output:

```
LO      04 0
        00 +0
        00 A0
        00 -0
        00 B0
        00 C0

MO      04 0
        00 -0
        00 +0
        00 A0
        00 B0
        00 C0

NO      04 0
        00 *0
        00 +0
        00 A0
        00 B0
        00 -0
        00 C0
        00 D0
```

B. LISP1.5

The function in2pre translates an expression of the form defined by the BNF

e:: = symbol | (e symbol e) | (- e)

The prefix form is obtained by moving a symbol in the middle (operator) position to the front of its list. Parentheses remain in the output. Note that negations are already in prefix form.

Program:

```
(IN2PRE (LAMBDA (X) (COND
   ((ATOM X) X)
   ((EQ (CAR X) (QUOTE -)) (LIST (CAR X)(IN2PRE (CADR X))))
   (T (LIST (CADR X)(IN2PRE (CAR X))(IN2PRE (CADDR X)))))))
```

CAR, CADR, and CADDR are functions which extract the first, second, and third elements, respectively, of a list.

The first line defines IN2PRE to be a LISP function, in conditional expression form, of one argument.

The second line asserts that an atomic symbol is unchanged by IN2PRE.

The third line does the appropriate thing in the special case of the unary "-" operator.

The fourth line forms the output as a list of the main operator (CADR X) followed by the results of applying IN2PRE, recursively, to its two arguments.

Data and resulting output:

```
IN2PRE (((A * B) + (C / (E - (F ↑ (-G)))))
(+ (* A B)(/ C (- E (↑ F (-G)))))
```

C. DYSTAL

Program:

```
C     ** THIS IS A PROGRAM TO CONVERT A FULLY PARENTHESIZED ALGEBRAIC
C     ** EXPRESSION INTO PREFIX FORM.
C     ** THE FOLLOWING SPECIFICATION STATEMENTS SET UP THE DYNAMIC
C     ** STORAGE AREA.
      DIMENSION LOT (8191), FLOT(8191)
      COMMON LOT
      EQUIVALENCE(LOT(1), FLOT(1))
      EQUIVALENCE(LOT(21),LOP), (LOT(22),LSET), (LOT(23),LFORM),
     1(LOT(24),LPREFX)
      CALL INLOT(20,8191)
C     ** INPUT USING LSREAD, WHICH IS CAPABLE OF READING MULTIPLE
C     ** ARRAYS, MATRICES OR TREE STRUCTURES.  LOP, THE FIRST LIST,
C     ** CONTAINS THE SYMBOLS $, (,). LSET, THE SECOND LIST,
C     ** CONTAINS THE ORIGINAL ALGEBRAIC EXPRESSION SUCH AS
C     ** (( ALPHA + (BETA * E1)) - (C/D)). LSET WILL BE READ IN WITH A
C     ** STANDARD DYSTAL FORMAT OF 5 FIELDS OF 4 CHARACTER WORDS PER
C     ** CARD UNLESS A VARIABLE FORMAT IS SPECIFIED AT INPUT TIME.
      CALL LSREAD(NAME)
      LOP = LOT (NAME + 1)
      LSET = LOT(NAME + 2)
```

```
      NSET = LOT (LSET)
      NSYMB = LOT(LOP + 1)
      NDPAR = LOT(LOP + 3)
C     ** CREATION OF ARRAYS.  LFORM WILL BE
C     ** USED AS A PUSHDOWN STACK TO HOLD PORTIONS OF EXPRESSIONS
C     ** UNTIL THEY ARE READY TO BE USED.  LPREFX HOLDS THE PREFIXED
C     ** FORM OF THE EXPRESSION. BOTH LFORM AND LPREFX ARE SET AT A
C     ** MAXIMUM OF NSET, THE LENGTH OF LSET.
      CALL LSTALL(4, NSET, LFORM)
      CALL LSTALL(4, NSET, LPREFX)
C     ** STRATEGY. WHENEVER A RIGHT PAREN IS ENCOUNT-
C     ** ERED A SUBEXPRESSION IS FORMED AND PLACED ON LPREFX AND A $
C     ** IS LEFT IN ITS PLACE ON LFORM.  THE PREFIXED EXPRESSION IS
C     ** FORMED ON LPREFX IN REVERSE ORDER FOR THE SAKE OF CONVENIENCE
C     ** AND LATER REVERSED.
      DO 100 I = 1, NSET
C     ** IPT IS THE LOCATION OF EACH ITEM ON LSET
      IPT = LSET + I
C     ** SEARCH FOR RIGHT PARENTHESIS.
      IF(LOT(IPT) - NDPAR) 40, 50, 40
   40 CALL LOAD(LOT(IPT), LFORM)
      GO TO 100
C     ** STORE LAST THREE WORDS IN 3, 1, 2 ORDER.  SKIP STORAGE IF
C     ** THE FIRST OR THIRD IS A $.
   50 IWD = ITEM(LOT(LFORM), LFORM)
      IF(IWD - NSYMB) 60, 70, 60
   60    CALL LOAD(IWD, LPREFX)
   70    IWD = ITEM(LOT(LFORM) - 2, LFORM)
         IF(IWD - NSYMB) 80, 90, 80
   80    CALL LOAD(IWD, LPREFX)
   90 CALL LOAD (ITEM (LOT(LFORM) - 1, LFORM), LPREFX)
C     ** PLACE $ IN PLACE OF (.
      CALL IPLACE(NSYMB, LFORM + LOT(LFORM) - 3)
C     ** REDUCE LFORM COUNTER BY 3
      LOT(LFORM) = LOT(LFORM) - 3
  100 CONTINUE
C     ** REVERSE LPREFX
      CALL LREVER(LPREFX)
C     ** PRINT OUT DYNAMIC STORAGE AREA
      CALL KDUMP
      STOP
      END
```

Data:

```
     NAME                    1              10
  1  LOP                     4               3              3
     $          (          )
  2  LSET                    4              17             17
     (          (          ALPH          +               (
     BETA        *         EL            )               )
     -          (          C             /               D
     )           )
     STOP
/*
/*
```

Resulting output:

```
DUMP OF LIST   1  NAME   5
 IDEN =       NODE =    0  MODE =     2  NMAX =   35  NCTR =   35

        134          8191          8191            0            0
          0             0             0            0            0
          0             0             0            0            0
         60            68            90          112            0
          0             0             0            0            0
          0             0             0            0            0
          0             0             0            0            0

DUMP OF LIST   2  NAME   45
 IDEN =  NAM  NODE =    0  MODE       1  NMAX =   10  NCTR =    2

         60            68

DUMP OF LIST   3  NAME  60
 IDEN =  LOP  NODE =    0   MODE =    4  NMAX =    3  NCTR =    3

        $             (             )

DUMP OF LIST   4  NAME  68
 IDEN =  LSE  NODE =    0  MODE =     4  NMAX =   17  NCTR =   17

        (             (             ALPH          +             .(
        BETA           *            El            )              )
        -             (             C             /              D
        )             )

DUMP OF LIST   5  NAME  90
 IDEN =       NODE =    0  MODE =     4  NMAX =   17  NCTR =    1

        $
```

```
DUMP OF LIST    6  NAME   112
 IDEN =        NODE =       0  MODE      4  NMAX =   17  NCTR = 9

          -             /             C             D          +
          ALPH          *             BETA          E1
```

D. LISP2

It is assumed here that the input to the function PREFIX is a list containing variables (i.e., literal atoms), binary operators (i.e., +, -, *, ↑, /) and subexpressions of the same form, appearing in parentheses. Syntax equations for the input arithmetic expression are:

expression ::= primary | primary operator expression

primary ::= basic | (expression)

basic ::= number | variable

and number, variable, and operator are not defined further, but represent tests that can be made. It is also assumed that the function PREFIX should detect errors if the expression given it is ill-formed.

Program:

```
SYMBOL SECTION TEST:
FUNCTION PREFIX (L):
     IF BASIC (L) THEN L ELSE
     DO IF ATOM L THEN E : EXIT (L . '(IS NOT A LEGAL EXPRESSION));
          BLOCK (X ← PREFIX (CAR L), Y ← CDR L):
          IF NULL Y THEN RETURN X;
          IF ATOM Y THEN GO E;
          BLOCK (Z ← CAR Y):
               Y ← PREFIX (CDR Y);
               IF OPERATOR (Z) THEN RETURN LIST (Z,X,Y)
                    ELSE EXIT (Z . '(IS NOT AN OPERATOR))END END END,
FUNCTION BASIC (X) : NUMB (X) OR VARIABLE (X),
FUNCTION PREFIXSUPV ():
     BLOCK (X):
     A: X ← READ ();
          IF X = 'FINISHED THEN RETURN X;
          TRY (X, C, X ← PREFIX (X));
```

```
B: PRINT (X);
     GO A;
C: PRINT ('ERROR..); GO B; END;
```

The above function definitions use the general LISP list-processing functions CAR, CDR and testing predicates ATOM, NULL, and the list-building functions LIST and CONS (represented in source language by an infix dot). NUMBP is true for a number, and it is assumed that VARIABLE and OPERATOR are true for variables and operators, respectively. A supervisor function is included here to show the ease with which LISP2 enables the construction of supervisors to be done, and makes the user see a different system. (The TRY statement used in the supervisor allows control of error exit. The third argument of TRY is a statement to be operated; if EXIT is encountered in its execution, the value of the exit expression is placed into the first argument of TRY, and control reverts to the label given as the second argument of TRY.)

PREFIX can be called directly, as in the following example:

```
input:  PREFIX ('(A + (B * Z) - (C ↑ D) / (F + G)));
output:  (+ A (- (* B 2) (/ (↑ C D) (+ F G))))
```

Instead, PREFIXSUPV () can be called, and the following conversation could ensue:

```
input:   3
output:   3
input:   (((((A))) + ((((B))))))
output:   (+ A B)
input:   (A B C )
output:   ERROR..
               (B IS NOT AN OPERATOR)
input:   (A + ((B * 2) / (C + D - E))))
output:   (+ A(/ (* B 2) (+ C (- D E))))
input:   FINISHED
output:   FINISHED
```

E. L^6

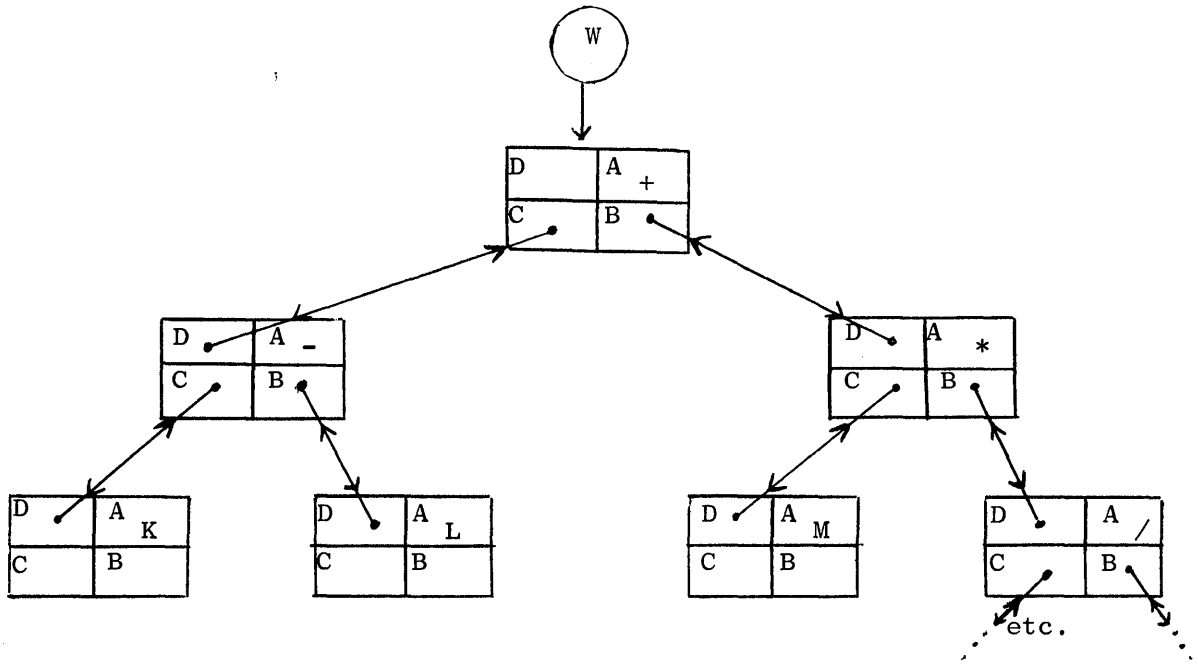
Assume as input a well-formed fully parenthesized algebraic expression with single-letter variables and binary operators +, -, * and /, such as

```
((K-L)+(M*((N-O)/(P+(Q+((((R*S)/(T*U))+V)*((W+X)-(Y+Z))))))))
```

This can be converted into the corresponding prefix form

```
+-KL*M/-NO+P+Q*+/*RS*TUV-+WX+YZ
```

by a program which builds the appropriate binary tree, using 2-word blocks divided into fields A, B, C and D as shown:



The one-line subroutines and two subordinate subroutines which do this are coded as follows:

```
PREFIX THEN    (W,GT,2)(DO,TREE)(DO,OUTPUT)(W,IN,73)(1,PU,77)DONE

  TREE THEN    (WA,IN,1)
       IF      (WA,NO,74)DONE
       THEN    (WC,GT,2)(WCD,P,W)(W,C)(DO,TREE)(W,D)(WA,E,O)(WA,IN,1)
       THEN    (WB,GT,2)(WBD,P,W)(W,B)(DO,TREE)(W,D)(W,IN,1)(W,R,6)DONE


    OUTPUT THEN     (1,PU,WA)
           IF       (WC,N,0)THEN(W,C)(DO,OUTPUT)(W,B)(DO,OUTPUT)
           THEN     (W,FR,WD)DONE
```

The subroutine PREFIX begins with bug (i.e. base register) W GeTting a 2-block from free storage (W,GT,2). Then the subroutine TREE is performed, which builds the tree, followed by subroutine OUTPUT which prints out the results and throws the tree back to free storage. Next an attempt is made to shift 73 characters of input into bug W by (W,IN,73), a process which automatically stops at the end of the current card, and thus positions the input mechanism for reading the next card. Finally the special end-of-line character, 77_8, is "PUnched" to terminate this line and cause actual punching from the output buffer.

The subroutine TREE begins by reading one character of INput into field WA (i.e. the A field of the block that W points to) by the operation (WA,IN,1). Then if the contents of WA are Not Octal (a left parenthesis), i.e., if this expression or subexpression is not compound, then the subroutine is exited by the special go-to DONE. Otherwise field WC "gets" a 2 block by (WC,GT,2), the D field of that block is made to Point back to W's block by (WCD,P,W), W jumps to point to this new block by (W,C) and this same subroutine is entered recursively, to build a subtree for the left-hand subexpression. Then W jumps back up by (W,D). It sets the A field of its block equal to zero by (WA,E,0) and shifts the next character, which should be the connective, into this field by (WA,IN,1). The subroutine proceeds similarly for the right-hand subexpression and finally disposes of the right parenthesis by shifting it into W by (W,IN,1) and the expelling it by shifting contents of W right 6 bits by (W,R,6).

The subroutine OUTPUT PUnches one character of output from the right-end of WA by (1,PU,WA). If there are subtrees (i.e. if WC is Not 0), W then jumps down to where the C field of its block pointed (W,C), performs the OUTPUT subroutine, and then does the same for the right-hand subtree. The final operation (W,FR,WD) FRees the block that W was pointing to but refills W with what was the contents of WD. Thus on exit, W is one level higher in the tree than when the subroutine was entered, and an entire subtree has been returned to free storage.

Running time for this subroutine on the 7094, for the above sample problem not including buffered input/output, is 20 msec.

F. CORAL

The CORAL program described here considers the algebraic expression as a string of elements of the following four types: left parenthesis, right parenthesis, operator, and term. Each element is represented by a CORAL bloc which is identifiable as to type of element, and these blocks are strung sequentially in a CORAL ring. The program accepts this ring as "input"; it rearranges and modifies the ring so that the "output" ring represents the prefix form of the expression.

Program:

```
CONVERT→  START[>]SUBR→CUR►DONE
SUBR→     ([T])[=]RIGHT⊃RSUBR
          [=]OPR⊃OPRSUBR|MORE
RSUBR→    CUR[<](([T])[=]LEFT⊃([*](PREV⊗)|(CUR⊗)►END)|MORE)→PREV►MORE
OPRSUBR→  CUR[<](([T])[=]LEFT (CUR⊜ PREV►END)|MORE)→PREV► MORE
```

The algorithm used is the following: Proceed through the ring from "left to right" performing an action at each element. If the element is a right parenthesis, delete it and the most recently encountered left parenthesis; otherwise, if the element is an operator, move it to the right of the most recently encountered left parenthesis; otherwise nothing. After all of the elements have been scanned and the appropriate actions performed, the expression has been converted to prefix form.

The line labelled CONVERT is the "go around the ring" [⊵] causing the subroutine SUBR to be done to each element. SUBR and the next line test in turn whether the current element CUR is a right parenthesis, in which case control goes to RSUBR [=RIGHT⊃RSUBR]; or whether it is an operator, in which case control goes to OPRSUBR [=OPR⊃OPRSUBR]; if neither, then the go-around is returned to [|MORE].

RSUBR backs up through the ring searching for a left parenthesis [CUR⊴((T)=LEFT⊃ . . . |MORE)→PREV]; when found, it and the current element are deleted from the ring [(*(PREV⊗) | (CUR⊗)►END)] and control returns to the go-around.

OPRSUBR similarly backs up through the ring. When it finds a left parenthesis, the current element CUR is put to the right of it [CUR⊜PREV►END] and control returns to the go-around.

When CONVERT is done, it exits at DONE [►DONE].

CORAL is reasonably well suited for the prefix conversion problem when performed on a pre-existing data base. However, due to the unavailability of explicit input-output facilities in TX-2 CORAL, no meaningful sample data and outputs can be included here. It should be noted that CORAL is not intended for such stand-alone problems. Rather, CORAL was designed for problems which involve manipulations of complex inter-related data bases, such as those which occur in graphics applications.

G. AMBIT

This example is in the form established in [23], where four other examples of AMBIT are given.

Given an input of the form 'QΔ expl' where expl is a fully parenthesized algebraic expression, this program produces an output string of the form 'QΔ exp2' where exp2 is the Polish prefix equivalent of expl.

Program:

```
1.   begin     phrase dummy A, B, expl;
2.             mark dummy op;  word dummy EO;
3.             QΔ expl → QΔ pΔ expl
4.   LOOP:     pΔ ( A op B ) → op pΔ A pΔ B   or
5.             pΔ ( op B )   → op pΔ B        or
6.             pΔ EO         → EO             ;
7.             if ∃ pΔ then go to LOOP;
8.   end
```

Lines 1 and 2 of the program declare A, B, and expl to each represent arbitrary operands; declare op to represent an arbitrary operator; and declare EO to represent an arbitrary elementary operand (variable or constant).

Line 3 is the beginning of the executable part of the program and inserts the initial instance of the pointer 'pΔ' to the left of the given expression.

Lines 4-6 perform the desired transformation for a binary expression, a unary expression, or an elementary operand. These lines detect ill-formed input by failing to apply.

Line 7 is the end test, and succeeds only if an instance of the pointer 'pΔ' remains.

Note: Line 4, if it succeeds, replaces one instance of 'pΔ' with two instances, one for each operand. This is not recursive program execution, although it might be called "data directed recursion".

Data:

```
QΔ ((A+B)+(PHI (-3)))
```

Resulting output:

```
QΔ = + A B  PHI  -  3
```

Trace of Example Execution. The following trace shows each modification of the data string in the course of execution. Each line shows an execution history (not part of the data string) and a copy of the current data string.

```
(input)            QΔ  ((A + B) = (PHI↑(-3)))
3 succeeds:        QΔ  pΔ1   ((A + B) = (PHI↑ (-3)))
4s:                QΔ  =  pΔ1   (A + B)  pΔ2  (PHI↑ (-3))
7s,4s:             QΔ  =  pΔ1   (A + B)  ↑  pΔ2  PHI   pΔ3  (-3)
7s,4f,5s:          QΔ  =  pΔ1   (A + B)  ↑  pΔ2  PHI   -  pΔ3  3
7s,4f,5f,6s:       QΔ  =  pΔ1   (A + B)  ↑  pΔ2  PHI   -  3
7s,4f,5f,6s:       QΔ  =  pΔ1   (A + B)  ↑  PHI  -  3
```

```
7s,4f:                Q∆  =  +  p∆1  A  p∆2  B  ↑  PHI  -  3
7s,4f,5f,6f:          Q∆  =  +  p∆1  A  B  ↑  PHI  -  3
7s,4f,5f,6s:          Q∆  =  +  A  B  ↑  PHI  -  3
7s,4f,5f,6f (exit)    (output)
```

H. COMIT

This program translates, from infix to prefix form, a string of symbols defined by the BNF

e::= symbol | (e symbol e)

The output string consists of operator and operand symbols separated by spaces and without parentheses.

Program:

```
*    COMIT
CONVERT   *( + $1 + $1 + $1 + *) = 3 + - + 2 + - + 4 // *K1 2 3 4 5 /
*  $ // *WAI1, *RTK1  CONVERT
END
```

This COMIT program repeatedly locates one of the infix expressions which has no parenthesized subexpressions and replaces it by the equivalent prefix form. The data is read into the workspace with each constituent being a string of letters or a plus, minus, slash, asterisk, or parenthesis. Blank characters have been deleted. As prefix-form subexpressions are formed, they are compressed to a single constituent. The convert rule says: Search the workspace for a left parenthesis followed by any three constituents and a right parenthesis. If the pattern is not found, go to the next rule; if it is found, replace the pattern by the operator, a blank, the left operand, a blank, and the right operand, all compressed into a single constituent and then apply the same rule again. In order to demonstrate the convert routine, a second rule has been added which reads infix expressions from data cards and punches the prefix expressions. This rule takes whatever is in the workspace, punches it, and replaces it by the contents of the next data card. If there are no more data cards, control passes to the next rule, not shown, and the program is finished. The program starts at the first rule; the first time through, the first rule does not find a match, and there is nothing for the second rule to punch.

Data:

```
( A + B )
((9-1)/(2*2))
((PRESSURE*VOLUME)/TEMPERATURE)
(((A+B)*(C+D))/((E-F)*(G-H)))
```

Resulting output:

```
+ A B
/ - 9 1 * 2 2
/ * PRESSURE VOLUME TEMPERATURE
/ * + A B + C D * - E F - G H
```

I. SNOBOL

The following program works on the same class of data strings as the preceding COMIT program. Where COMIT used repeated scans of the data and identified processed substrings by compressing characters into single "constituents", SNOBOL recursively identifies and processes parenthetically-balanced subexpressions.

Program:

```
          DEFINE("P(P)","P","U,V,OP")                                    1
READ      TEST      =    TRIM(SYSPIT)        /F(END)                     2
          SYSPOT    =                                                    3
          SYSPOT    =    TEST                                           4
          SYSPOT    =    P(TEST)             /(READ)                    5
P         P  "(" *(U)* *OP/"1"* *(V)* ")" = OP P(U) P(V)  /(RETURN)     6
END                                                                      7
```

Statement 1 defines a function P which generates the prefix form. For convenience the formal argument is chosen to be the same as the name of the function. U, V and OP are local variables of the function, and execution of the function begins at the statement with label P (Statement 6).

Statement 2 reads in a card, removing trailing blanks. On a read failure, the program is terminated by a transfer to end. SYSPIT stands for "system peripheral output tape."

Statement 3 prints a blank line for listing format. SYSPOT stands for "system peripheral output tape."

Statement 4 prints the algebraic expression.

Statement 5 calls the function P to convert the expression to prefix form and prints the results.

Statement 6 is a one-line function which recursively generates the prefix form. The string variables *(U)* and *(V)* match parenthesis balanced strings. The pattern matched is replaced by its prefix form. By SNOBOL convention, the value returned is the value of the name of the function, P.

This program assumes fully parenthesized expressions (without redundant parentheses), single character variables and single character operators.

Data and resulting output:

```
(((A+B)+C)+D)
+++ABCD

((A-B)-(C-D))
--AB-CD

((A+(B*D))/(U-V))
/+A*BD-UV

((((A*B)*C)+D)-((E*F)+G))
-+**ABCD+*EFG

(((A+B)+(C+D))*((E/F)/G))
*++AB+CD//EFG
```

J. PANON

The following PANON program reads a fully parenthesized arithmetic expression followed by the symbol # (rules/READ,/1,/2,/3) and converts it to Polish prefix notation (rule CONV). Such a class of expressions is recursively defined within the program and noted as E*. If the string being read does not belong to the class E* an error message is printed. Otherwise the converted string is printed by the rule/PRINT.

Program:

```
^=CM*  THE SYMBOL      ^=LET*  DENOTES THE CLASS OF LETTERS
       THE SYMBOL      ^=DMC*  DENOTES THE CLASS OF DUMMY CHARACTERS
^=CD*          ^E      ^=/*          ^=LET
                       ^=/*          (^E  ^OP  ^E*)
^=CD*          ^OP      =/*    +     =/*  -  =/*   *   =/*   /
^=TR*/O         -      ^==*          #          ^=GOTO/*READ
^=TR*/READ     ^=DMC   ^==*                     ^=GOTO*/1
^=TR*/1        ^E*  ##  ^==*          ^E         ^=GOTO*/CONV
^=TR*/2            # #  ^==*          ?
               ^=PRINT * ERROR:       ^=AS*      ^=GOTO*/STOP
^=TR*/3*           #    ^==*          ^=NC*  #   ^=GOTO*/READ
^=TR*/CONV*
     (^E*/1^OP^E*/2*) ^==*^OP  ^E*/1^E*/2
                                                 ^=GOTO*/CONV
```

^=TR*/PRINT ^==*

^=PRINT* ^=AS* ^=GOTO*/STOP

^=END*

Trace of Typical Run:

The notation < NC' ← φ > indicates that:

(a) The symbol -NC* is created.
(b) It is replaced by the symbol φ which is the new character read in.

i) Input string: ((A+B) * C) #

Initially the string is empty, then by applying the rule named in the first column it is successively changed into the string to the right:

```
0                                          #
3                                    < NC ← (> #
3                                  ( < NC ← (> #
3                                (( < NC ← A> #
3                              ((A < NC ← +> #
3                            ((A + < NC ← B> #
3                          ((A + B < NC ← )> #
3                        ((A + B) < NC ← *> #
3                      ((A + B)* < NC ← C> #
3                    ((A + B)*C < NC ← )> #
3                  ((A + B)*C) < NC ← #> #
1                            ((A + B)* C)
CONV                         *(A + B) C
CONV                         *+ABC
PRINT                        *+ABC              [and prints: * +ABC]
ii) Input string:    (A + #
0                                          #
3                                    < NC ← (> #
3                                  ( < NC ← A> #
3                                (A < NC ← +> #
3                              (A + < NC ← #> #
2                            (A+?               [and prints:  ERROR:(A+?]
```

K. CONVERT

The CONVERT program, like all varients and descendents of LISP, prefers to leave the parentheses in its output. This particular program assumes that unary operators are really binary with an implicitly zero first argument, so that p and m (+ and -) are handled appropriately for both one and two arguments. The input is not assumed to be fully parenthesized; instead, standard operator procedence rules are employed.

Program:

```
                                DEFINE((
 1    (FORMTRAN ( LAMBDA  (U) (CONVERT
 2    (LIST)
 3    (QUOTE ( X ( LLL) (RRR)))
 4    U
 5    (QUOTE ( * (
 6         ( (LLL P RRR) (P (=BEGN= (LLL)) (=BEGN= (RRR))) )
 7         ( (LLL M RRR) (M (=BEGN= (LLL)) (=BEGN= (RRR))) )
 8         ( (LLL * RRR) (* (=BEGN= (LLL)) (=BEGN= (RRR))) )
 9         ( (LLL / RRR) (/ (=BEGN= (LLL)) (=BEGN= (RRR))) )
10         ( (LLL ** RRR) (** (=BEGN= (LLL)) (=BEGN= (RRR))) )
11         ( (X) (=BEGN= X))
12         (() 0 )
13         )))
14    )))
                                      ))
```

(1) Defines "FORMTRAN" as a function of one argument, U. Then it calls to CONVERT.

(2) The 1st argument of CONVERT, the dictionary, is empty.

(3) We define X as an undefined variable [UAR] and LLL and RRR as undefined fragments.

(4) The 3rd argument of CONVERT is U, namely, the expression we want to transform.

(5) In this line begins the CONVERT program; it consists of one set of rules, named *.

(6) This is the 1st rule. Its left half is the pattern (LLL P RRR)
Its right half is the skeleton (P (=BEGN= (LLL))
(=BEGN= (RRR)))

The pattern match scans the expression for a plus sign (we are using here P for plus sign and M for minus), and assigns the name LLL to the fragment or string to the left of such sign, and RRR to the fragment to its right.

If the pattern matches (the search is successful), we replace the scanned structure by the result of substituting into the skeleton, (P (=BEGN=(LLL)) (=BEGN=(RRR))), and this will be the result of our program. This skeleton says that we should form a list of 3 elements:

```
P
(=BEGN= (LLL))
(=BEGN= (RRR))
```

Result = skeleton properly replaced:

First element: The atom P gets replaced and, since it does not appear in the dictionary, it stands for itself.

Second element: The skeleton (=BEGN=(LLL)) will get replaced also, and its value is the result of applying the same CONVERT program to (LLL), that is, to the left fragment.

Third element: The skeleton (=BEGN= (RRR)) gets replaced similarly. =BEGN= is the recursive call or entry. In this case this entire CONVERT program will be applied to (RRR); that is to say, we will convert to prefix form the string which was found to the right of P.

If the pattern of rule (6) does not match our expression (4), we apply the next rule (7).

(7) Does the same as (6), but it looks for minus sign.
(8)
(9) } Test and conversion for times, quotient, and exponentiation.
(10)
(11) This line handles parenthesized subexpressions
(12) The rule (()0) is used for unary - and +, for instance, -Q goes into (0-Q); +7 goes into (0+7).

If none of the above rules apply, the expression (4) is returned unchanged as value.

The order of the rules (6) - (12) is important, and depends on the hierarchy of the different operators; here we assume the standard FORTRAN conventions.

Let us follow an example. Suppose we want to convert a^2+9.

```
(4)          (a ** 2 P 9  )
(6)          (LLL    P RRR)
```

The 1st rule is tested and its pattern matches (4); now LLL = a ** and RRR = 9

We then substitute into the skeleton (P (=BEGN=(LLL)) (=BEGN=(RRR))), obtaining (P (=BEGN=(a ** 2)) (=BEGN= (9))).

(1) (=BEGN= (a ** 2)) - The expression to transform is (a ** 2); we apply to it the entire CONVERT program.
Rules (6) to (9) are tested and fail.
Rule (10) succeeds: (a ** 2)
(LLL ** RRR) now LLL = a
RRR = 2

So the answer will be (** (=BEGN=(A)) (=BEGN=(2))) In order to compute (=BEGN=(A)), we apply this entire program to (A); when doing this, rules (6) to (10) fail; rule (11) succeeds and tells us to compute (=BEGN=X), that is, (=BEGN=A); this value is computed by recursively entering the program again, resulting in A. (All rules fail when their

patterns are compared with A, so A is returned unchanged). Similarly, the value of (=BEGN= (2)) is 2. Therefore, (** (=BEGN=(A)) (=BEGN= (2))) gets replaced by (** A 2).

(2) (=BEGN= (9)) gets evaluated and results in 9.

Therefore, (P (=BEGN= (a ** 2)) (=BEGN= (9))) becomes (P (** A 2) 9) and this is the final result.

Simple rules govern the use of patterns and skeletons, when transforming a given expression:

When we arrive to a certain rule:
We compare its pattern (left half of the rule) against the expression.
If the comparison succeeds, we replace into the 2nd half or skeleton of the successful rule, and return this as value.
If the comparison fails, we try the same with the next rule.
If no more rules, return as value the expression unchanged.

Data and resulting outputs:

```
formtran((  a ** 2 ))
  (** A 2)

formtran((  a ** 2 p 9 ))
  (P ( ** A 2) 9)

formtran((  a ** 2 m 9 / 6 ))
  (M (** A 2) (/ 9 6))

formtran((  (a** 2) m (9 / 6 ) ))
  (M (** A 2) (/ 9 6 ))

formtran(( a p b * x / y p c * x ** 2 / y ** 2 p d * x ** 3 /
 y ** 3  ))
 (P A (P (* B (/X Y )) (P (* C (/ (** X 2) (** Y 2))) (* D (
 / (** X 3) (** Y 3 )))) ))

formtran ((  (a m a0) p (b m b0) * (x m x0) / (ym y0) ** 2  ))
 (P (M A A0) (* (M B B0) (/ (M X X0) (**(M Y Y0) 2))))
```

L. FLIP

The following FLIP program operates on the same class of expressions as the LISP program (part (b), above). This example illustrates the complementary natures of LISP and FLIP. While LISP is well suited for recursive problems, FLIP is designed for specifying a pattern-directed transformation of a list.

Program:

```
DEFINE (((IN2PRE
(LAMBDA (X) (COND ((ATOM X) X) (T (FLIP X (QUOTE
(EITHER['-$1; $1 EITHER['↑; '*; '/; '+] $1]))
(QUOTE (EITHER['MINUS =(IN2PRE #2)
  EITHER['EXPT;'TIMES;'QUOTIENT; 'PLUS] =(IN2PRE #1) =(IN2PRE #3)])
  ))))))))
```

FLIP is a LISP function which takes three arguments: a list, a pattern which is used to match the list, and a format for constructing a new list using the result of the match. For this particular problem, there are two cases to be distinguished. In the first case, the list is of the form (-X); in the second case, it is of the form (X operator Y) where operator is either "+", "/", "*", or "↑". These two cases are resolved by use of the EITHER pattern in FLIP. By using the EITHER pattern inside of another EITHER pattern, in the second case, we can distinguish among the various operators.

The format given to FLIP utilizes the EITHER format. This format selects a format corresponding to the pattern matched by the associated EITHER pattern. Thus, if the input list is of the form (-X), the first format, or

```
'MINUS =(IN2PRE #2)
```

will be used for constructing a new list. In this format, 'MINUS causes the atom "MINUS" to be inserted in the list being constructed, while =(IN2PRE #2) specifies a call to IN2PRE giving it as its argument the second element in the match, i.e., that element matched by $1. The result of this call is inserted in the new list following MINUS.

If the list is of the form (X operator Y), then the second format is used. This latter format in turn utilizes another EITHER format for selecting among the different operators. Thus if the operator is "+", then the format 'PLUS is used, if "*", then 'TIMES is used.

If the input list is of the form (=X), a recursive call to IN2PRE giving it X as input is used in the construction of the new list. If the list is of the form (X operator Y), two calls to IN2PRE are necessary, one with input X and one with input Y. The results of these calls are inserted in the list which is being constucted.

Thus for the input list (A + (B * C)), a new list if formed with PLUS as its first element, IN2PRE applied to A as its second element, and IN2PRE applied to (B * C) as its third element. Since IN2PRE applied to A is A, and IN2PRE applied to (B * C) is (TIMES B C), the final result if (PLUS A (TIMES B C)).

Data and resulting outputs:

```
IN2PRE ((A + B))
  (PLUS A B)
IN2PRE ((A + (( - C) * (D / ((B * C) + (- X))))))
  (PLUS A (TIMES (MINUS C) (QUOTIENT D(PLUS (TIMES B C) (MINUS X)))))
```

M. COGENT

The following program converts arithmetic expressions form infix to prefix notation. Its input is a sequence of expressions followed by periods, where each expression is built up from alphanumeric variables and integer constants, using the binary operators "+", "-", "*", "/", and "**", and the unary operators "+", and "-". Precedence rules similar to ALGOL are used so that full parenthesation is unnecessary. The output is a sequence of prefix expressions in which operators and operands are separated by blanks, and unary operators are preceded by "$" (since prefix notation is ambiguous unless unary and binary operators are distinguishable).

The program consists of four sections:

1. Character definitions which establish ($EOF) and ($B) as special symbols denoting an end-of-file and a blank.

2. A sequence of production rules describing the input language syntax. Labels on the production rules refer to generators (subroutines) and indicate how the corresponding input phrases are to be translated: < var >'s are converted to packed BCD symbol table entries, < const >'s are converted to numbers, < primary >'s < factor >'s < term >'s and < exp >'s are converted at each syntactic level into equivalent prefix expressions, and translated < sentence >'s are outputted by the generator PRINT.

3. Additional productions describing the output language syntax (beginning with $SECSYN).

4. Generator definitions. The generators UNARY and BINARY synthesize prefix expressions from operators and operands (which may themselves be previously-converted prefix expressions) by substitution into a pattern phrase of output language. The generator PRINT outputs a prefix expression and closes the line buffer.

Overall, the program is a straightforward example of syntax-directed translation and is little more than a description of the input and output language syntax plus a direct mapping from input to output phrases. The problem is too simple to illustrate any of the features of COGENT which go beyond syntax-directed methods, e.g., analysis statements, conditional transfers, or recursive generators.

Program:

```
     $TITLE INFIX-TO-PREFIX-TRANSLATION.
     $CHARDEF ($EOF) = (101)101.  ($B) = (60)60.
     $PRIMSYN ((INPUT)($EOF))
          (LETR) = A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z.
          (DIGIT) = 0,1,2,3,4,5,6,7,8,9.
          (EXPN OP) = **.
          (MULT OP) = *,/.
          (ADD OP) = +,-.
          (NAME STR) = (LETR),(NAME STR)(LETR),(NAME STR)(DIGIT).
          (DIGIT STR) = (DIGIT),(DIGIT STR)(DIGIT).
$IDENT,1/ (VAR) = (NAME STR).
    $DEC/ (CONST) = (DIGIT STR).
     NOP/ (PRIMARY) = (VAR),(CONST),(()(EXP)()).
     NOP/ (FACTOR) = (PRIMARY).
  BINARY/ (FACTOR) = (FACTOR)(EXPN OP)(PRIMARY).
     NOP/ (TERM) = (FACTOR).
  BINARY/ (TERM) = (TERM)(MULT OP)(FACTOR).
     NOP/ (EXP) = (TERM).
   UNARY/ (EXP) = (ADD OP)(TERM).
  BINARY/ (EXP) = (EXP)(ADD OP)(TERM).
   PRINT/ (SENTENCE) = (EXP)(.).
          (INPUT) = (SENTENCE),(INPUT)(SENTENCE).
$SECSYN   (OP) = (EXPN OP),(MULT OP),(ADD OP).
           (PFX EXP) = (VAR),(CONST),$(ADD OP)($B)(PFX EXP),
                  (OP)($B)(PFX EXP)($B)(PFX EXP).
$PROGRAM
$GENERATOR UNARY((OP,X)
     X /= (PFX EXP/$(ADD OP)($B)(PFX EXP)),OP,X.  $RETURN(X). ).
$GENERATOR BINARY((X,OP,Y)
     X /= (PFX EXP/(OP)($B)(PFX EXP)($B)(PFX EXP)),OP,X,Y.  $RETURN(X). ).
$GENERATOR PRINT((X)
     STANDSCN(X,PUTP).  OUTP().  ).
```

Data:

```
(-X/Y+3)*Z.
ALPHA/(-BETA*(GAMMA-16)).
-I/2+J*K**(-3).
```

Output:

```
* + $- / X Y 3 Z
/ ALPHA $- * BETA - GAMMA 16
+ $- / 1 2 * J ** K $- 3
```

The ALTRAN and Formula Algol programs following, solve variants of problem 2: "Solve an algebraic equation for the single occurrence of some variable x". The problem is trivial for ALTRAN provided the equation is composed of polynomials. Formula Algol offers several equally complicated-appearing solutions to a much more general version of the problem.

N. ALTRAN

Problem:

Read in a polynomial F(X,Y,Z) linear in X. Verify this linearity. Then solve the equation F(X,Y,Z) = 0 for X, and print the result.

Solution:

Clearly $F(X,Y,Z) = A(Y,Z)X + B(Y,Z)$

where A and B are polynomials, and the desired result is

$$X = G(Y,Z) \equiv \frac{B(Y,Z)}{A(Y,Z)}$$

The first step is to declare all needed indentifiers and to establish a layout for the polynomials and algebraics.

```
POLYNOMIAL A, B, F
ALGEBRAIC G
INTEGER FUNCTION DEG
POLYNOMIAL FUNCTION  COEFF
LAYOUT (L) X 12, Y 12, Z 12
```

The first of these declarations specifies that A, B, and F are language variables whose values will be polynomials. Similarly, the second says that G is a language variable whose values will be algebraics, or in other words, rational functions. The third declares that DEG is an integer-valued function, and similarly the fourth declares that COEFF is a polynomial-valued function. Finally the LAYOUT statement establishes a layout (L) which specifies that X, Y, and Z are a set of data variables. (i.e., variables from which polynomials and algebraics in the data may be composed), and that 12 bits per term are to be allocated for the exponents of each. (Since a data variable represents only itself, no value can be assigned to it. Therefore data variables are sometimes called symbolic constants.)

We now proceed to the program's executable part, which is almost self explanatory to anyone familiar with FORTRAN II.

```
      READ (L) F
      IF (DEG(F)-1) 90, 10
10    A = COEFF(F,X,1)
      B = COEFF(F,X,0)
      G = B/A
      PRINT "RESULT", G
      STOP
90    PRINT "ERROR IN DATA"
      STOP
      END
```

Possible input data and corresponding results:

Suppose $F = XY^2 + 2XYZ + XZ^2 - Y^2 + Z^2$. Then the input data would have the form

```
 1  1 2 0
 2  1 1 1
 1  1 0 2
-1  0 2 0
 1  0 0 2
 0
```

The first five lines represent the five terms of F, and can occur in any order. In each line the first integer is the coefficient of the corresponding term, and the other three are the exponents. The last line, consisting of a single zero, terminates F.

Clearly the desired result is G = (Y-Z)/(Y+Z). The statement

```
PRINT "RESULT", G
```

in our program will produce the output

```
RESULT

G =

     NUMERATOR          X Y Z
                    1   0 1 0
                   -1   0 0 1
   DENOMINATOR          X Y Z
                    1   0 1 0
                    1   0 0 1
```

Although the representation of a polynomial as an array of coefficients and exponents is admittedly inconvenient for small polynomials ($\leq$ 10 terms), it is both easier to read and easier to write for large ones ($\geq$ 100 terms). In the Princeton version of ALTRAN there is an optional variant of the READ statement which permits input data in FORTRAN form.

0. Formula Algol

A programming language is made rich by the availability within it of a variety of programming techniques. The attached computer output present three ways that Formula Algol can be used to solve an algebraic equation for the single occurrence of the variable X. These three solutions are by Markov Algorithms, by recursion, and by iteration. Formula Algol is well suited to programming this problem because its data structures and source language instructions were chosen to be well adapted to problems in formal algebraic manipulation. It can be seen from the attached programs that the Formula Algol programmer has detailed control over the specification of formula manipulation algorithms and that, at the same time, abbreviation devices, such as the Markov Algorithm, make it convenient to write them. Brief explanations of the three solutions are as follows:

I Markov Alogirthm Solution

Lines 12 to 29 define a Markov Algorithm which gives the rules of transformation by which equations are to be solved for X. The equation to be solved for X is stored as the value of the variable E in line 30, and line 31 prints both E and E.↓S, the result of apply the Markov Algorithm S to E, which result is the solved equation. In lines 10 and 11, plus and times are defined to be operators with commutative properties so that in line 14 and 15 commutative instances of A X B and A+B will be considered. Lines 7, 8, and 9 define A to be a formula pattern which will match any subexpression of a formula containing an occurrence of X, and B and C to be formula patterns which will match any arbitrary subexpression of a formula. The A's, B's, and C's are used in the construction of the left hand sides of the transformations in the Markov Algorithm and stand for patterns with these properties. On the right hand sides of the transformations the .A's, .B's, and .C's are objects which are replaced by the subexpressions which match the A's, B's, and C's when a given transformation applies to an input equation.

II Recursive Solution

Lines 4, 5, and 9 define patterns A, B, and C with the same properties as in the Markov Algorithm solution. The recursive procedure Solve (LHS, RHS) given in lines 8 to 28 analyzes the form of the left hand side of the equation LHS, which is assumed to contain X, and recursively calls Solve with that subexpression of LHS containing X as its new first parameter, and an appropriate inverse expression composed of an appropriate inverse operator applied to RHS and a subexpression of LHS not containing X as its new second parameter. The procedure Answer(E) given in

lines 30 to 34 analyzes the input equation E to see which side contains X and passes the side containing X as the left hand side and the side not containing X as the right hand side to Solve which delivers the answer to the problem. An equation is assigned to E in line 36 and both E and Answer (E) are printed in line 37. The printed solution is the same as that given in the first and third solutions.

III Iterative Solution

Lines 6 and 7 define two operator classes OP1 and OP2 consisting respectively of the binary operators to be used in input equations and the unary operators to be used in input equations. An integer variable I is attached to the definition of each operator class as an "Index". In lines 12 and 13 the input equation G is compared with two patterns. The first pattern matches if the left hand side of G contains a binary operator in the class OP1 and the index variable I is set to contain an integer denoting the ordinal position of this operator in the list of operators given on line 6. Similarly, the second pattern matches if G's left hand side is of the form < unary operator > (< expression >) and the index I is set to the ordinal position of the unary operator in the list of unary operators in line 7. The integer value of this index I is used in a designational expression containing a switch to transfer control to an appropriate statement to perform the required transformation of the equation. These transformations are given in lines 15 to 27. The iteration is under the control of a for-while statement and halts when the equation G has X as its left hand side. The printed solution is the same as that for solutions I and II.

IV Comparison of the Three Solutions

	Markov Algorithm	Recursion	Iteration
seconds required	5 ± 1	4 ± 1	3 ± 1
cells required	232	471	183
code required	771	826	595

The times given here are not measured as precisely as they should be for a truly useful comparison.

Markov Algorithm:

```
002:    AL BEGIN
003:      FORM E,K,M,H,N,P;
004:      FORM A,B,C,X; SYMBOL PLUS, TIMES, S;
005:      BOOLEAN PROCEDURE HASX(F); VALUE F; FORM F;
006:      HASX ← F >> X;
007:      A←A:OF(HASX);
008:      B←B:ANY;
009:      C←C:ANY;
010:      PLUS←/[OPERATOR:+][COMM:TRUE];
```

```
011:      TIMES←/[OPERATOR:*][COMM:TRUE];
012:      S ← [
013:      [
014:      (A|TIMES|B) = C → .A = .C / .B,
015:      (A|PLUS |B) = C → .A = .C - .B,
016:      A    -   B  = C → .A = .C + .B,
017:      B    -   A  = C → .A = .B - .C,
018:      A    /   B  = C → .A = .C * .B,
019:      B    /   A  = C → .A = .B / .C,
020:      A    ↑   B  = C → .A = .C ↑ (1/.B),
021:      B    ↑   A  = C → .A = LN(.C)/LN(.B),
022:           -   A  = C → .A = -.C,
023:          EXP(A) = C → .A = LN(.C),
024:           LN(A) = C → .A = EXP(.C),
025:         SQRT(A) = C → .A = .C ↑ 2,
026:       ARCTAN(A) = C → .A = SIN(.C)/COS(.C),
027:          SIN(A) = C → .A = ARCTAN(.C/SQRT(1-.C↑2)),
028:          COS(A) = C → .A = ARCTAN(SQRT(1-.C↑2)/.C),
029:              X  = C.→ .X = .C ] ];
030:      E ← K ↑2 + LN(M + SIN( (X↑3-K)/(H+4)*M↑5 )↑N - K)*M = P;
031:      PRINT( E, E.↓S );
032:       PRINT(CELLS);
033:       END;

K↑2 + LN(M + SIN(X↑3 - K)/(H + 4)*M↑5)↑N - K)*M=P
X=(ARCT((EXP((P - K↑2)/M) + K - M)↑(1/N)/SQRT(1 - EXP((P
-K↑2)/M) + K - M)↑(1/N)↑2))/M↑5*(H + 4) = K)↑(
.33333333333 +00)
```

Recursive Solution:

```
2.   BEGIN FORM E,K,M,N,H,P,F,G,X;
3.   SYMBOL PLUS,TIMES;
4.   BOOLEAN PROCEDURE HASX(F): VALUE F; FORM F; HASX←F>>X;
5.   PLUS←/[OPERATOR:+][COMM: TRUE]; TIMES←/[OPERATOR:*][COMM: TRUE];
6.
7.      BEGIN
8.      FORM PROCEDURE SOLVE(LH,RHS); FORM LHS,RHS;
9.        BEGIN FORM A,B,C;A←A:OF(HASX);B←B:ANY;C←C:ANY;
10.       IF LHS == (A|PLUS|B) THEN SOLVE←SOLVE(A,RHS-B);
11.       IF LHS == (A|TIMES|B) THEN SOLVE←SOLVE(A,RHS/B);
12.       IF LHS == A-B THEN SOLVE ← SOLVE(A,RHS+B);
13.       IF LHS == B-A THEN SOLVE ← SOLVE(A,B-RHS);
14.       IF LHS == A/B THEN SOLVE ← SOLVE(A,RHS*B);
15.       IF LHS == B/A THEN SOLVE ← SOLVE(A,B/RHS);
16.       IF LHS == A↑B THEN SOLVE ← SOLVE(A,RHS (1/B));
17.       IF LHS == B↑A THEN SOLVE ← SOLVE(A,LN(RHS)/LN(B));
18.       IF LHS == -A THEN SOLVE  ← SOLVE(A,-RHS);
19.       IF LHS == EXP(A) THEN SOLVE ← SOLVE(A,LN(RHS));
20.       IF LHS == LN(A) THEN SOLVE  ← SOLVE(A,EXP(RHS));
21.       IF LHS == SQRT(A) THEN SOLVE ← SOLVE(A,RHS↑2);
```

```
22.         IF LHS == ARCTAN(A) THEN SOLVE ← SOLVE(A,SIN(RHS)/COS(RHS));
23.         IF LHS == SIN(A) THEN
24.                     SOLVE ← SOLVE(A,ARCTAN(RHS/SQRT(1-RHS↑2)));
25.         IF LHS == COS(A) THEN
26.                     SOLVE ← SOLVE(A,ARCTAN(SQRT(1-RHS↑2)/RHS));
27.         IF LHS == X THEN SOLVE ← X = RHS;
28          END;
29.
30.       FORM PROCEDURE ANSWER(E); FORM E;
31.         BEGIN FORM F,G;
32.         IF E == G:ANY=F: ANY THEN BEGIN IF F>>X THEN
33.         ANSWER←SOLVE(F,G) ELSE ANSWER←SOLVE(G,F) END ELSE
34.         ANSWER←.NOEQUATION; END;
35.
36.       E ← K↑2 + LN(M + SIN((X↑3-K)/(H+4)*M↑5)↑N-K)*M =P;
37.       PRINT(E,ANSWER(E)); PRINT(CELLS);
38.       END;END;
```

```
K↑2 + LN(M + SIN((X↑3 - K)/(H + 4)*M↑5)↑N - K)*M=P
X=(ARCT((EXP((P - K↑2)/M) + K - M)↑(1/N)/SQRT(1 - EXP((P
- K↑2)/M) + K - M)↑(1/N)↑2))/M↑5*(H + 4) + K)↑(1/3)
```

Iterative Solution

```
002:        BEGIN
003:        FORM G,K,M,H,N,P,A,B,C,X;SYMBOL OP1,OP2;
004:        INTEGER I; SWITCH L← L1,L2,L3,L4,L5;
005:        SWITCH Q ← Q1,Q2,Q3,Q4,Q5,Q6,Q7;
006:        OP1←/[OPERATOR:*,+,-,/,↑][INDEX:I];
007:        OP2←/[OPERATOR:-,EXP,LN,SQRT,ARCTAN,SIN,COS][INDEX:I]
008:        G←K↑2 +LN(M+SIN( (X↑3-K)/(H+4)*M↑5) N-K)*M=P;
009:
010:        FOR G ← G WHILE -(G == X=ANY ) DO
011:           BEGIN
012            IF G == (A:ANY|OP1|B:ANY)=C:ANY THEN GO TO L[I];
013:           IF G == (+|OP2| A:ANY)=C:ANY THEN GO TO Q[I];
014:           PRINT(.NOEQUATION): GO TO EXIT;
015:           L1:G←IF A>>X THEN A=C/B ELSE B=C/A; GO TO CONTINUE;
016:           L2:G←IF A>>X THEN A=C-B ELSE B=C-A; GO TO CONTINUE;
017:           L3:G←IF A>>X THEN A=C+B ELSE B=A-C; GO TO CONTINUE;
018:           L4:G←IF A>>X THEN A=C*B ELSE B=A/C; GO TO CONTINUE;
019:           L5:G←IF A>>X THEN A=C↑(1/B) ELSE B=LN(C)/LN(A);
020:             GO TO CONTINUE;
021            Q1:G←A=-C; GO TO CONTINUE;
022:           Q2:G←A=LN(C); GO TO CONTINUE;
023:           Q3:G←A=EXP(C); GO TO CONTINUE;
024:           Q4:G←A=C↑2; GO TO CONTINUE
025:           Q5:G←A=SIN(C)/COS(C); GO TO CONTINUE;
026:           Q6:G←A=ARCTAN(C/SQRT(1-C↑2)); GO TO CONTINUE;
027:           Q7:G←A=ARCTAN(SQRT(1-C↑2)/C); GO TO CONTINUE;
```

```
028:        CONTINUE:  ;
029:        END;
030:        EXIT: ;
031:        PRINT(G);PRINT(CELLS);
032:        END;

X=(ARCT((EXP((P - K↑2)/M + K - M)↑(1/N)/SQRT(1 - (EXP((P
- K↑2)/M) + K - M)↑(1/N)↑2))/M↑5*(H + 4) + K)↑(1/3)
```

REFERENCES

1. B. F. Green, "Computer Languages for Symbol Manipulation," IRE Trans. HFE **2** (March 1961).

2. B. Raphael, "Aspects and Applications of Symbol Manipulation," Proc. 21st Nat. Conf. ACM (August 1966).

3. M. V. Wilkes, "Lists and Why They Are Useful," Proc. 19th Nat. Conf. ACM (August 1964).

4. J. E. Sammet, "Formula Manipulation by Computer," TR 00.1363, IBM Systems Development Division, Poughkeepsie, New York (November 1965).

5. A. Newell, ed., Information Processing Language-V Manual, Prentice Hall, Englewood Cliffs, N.J., 2nd edition (1963).

6. A. Newell, "Documentation of IPL-V," Comm. ACM **6**, No. 3 (March 1963).

7. J. McCarthy, et al., LISP1.5 Programmer's Manual, MIT Press, Cambridge, Mass. (1962).

8. E. C. Berkeley and D. G. Bobrow, eds., The Programming Language LISP: Its Operation and Applications, MIT Press, Cambridge, Mass. (1966).

9. J. Weizenbaum, "Symmetric List Processor," Comm. ACM **6**, No. 9 (September 1963).

10. K. C. Knowlton, "A Programmer's Description of L^6," Comm. ACM **9**, No. 8 (August 1966).

11. Two excellent 16mm sound films describing L^6 are available on loan from Technical Information Libraries, Bell Telephone Laboratories, Inc., Murray Hill, N.J.

12. V. H. Yngve, COMIT Programming, MIT Press, Cambridge, Mass. (in preparation 1966); see also SHARE distribution on COMIT.

13. D. G. Bobrow and B. Raphael, "A Comparison of List-Processing Computer Languages," Comm. ACM **7**, No. 4 (April 1964).

14. J. M. Sakoda, DYSTAL Manual, Sociology Computer Laboratory, Brown University, Providence, R.I. (1965).

15. P. W. Abrahams, et al., "The LISP2 Programming Language and System," AFIPS Proc. FJCC **29** (November 1966)

16. Formula Algol User's Manual, Carnegie Institute of Technology, Pittsburgh, Pa. (in preparation).

17. L. G. Roberts, "Graphical Communication and Control Languages," Second Conf. on Information System Science, Hot Springs, Va. (1964).

18. W. R. Sutherland, "The CORAL Language and Data Structures," contained in Tech. Report 405, MIT Lincoln Laboratory, Lexington, Mass. (1966).

19. FORMAC Manual, IBM Corp., Program Information Dept., 40 Saw Mill River Road, Hawthorne, N.Y.

20. W. S. Brown, "A Language and System for Symbolic Algebra on a Digital Computer," Proc. IBM Scientific Computing Symposium on Computer-Aided Experimentation (October 1965).

21. W. S. Brown, et al., "The ALPAK System for Nonnumerical Algebra on a Digital Computer," Bell System Tech. J. 42, pp. 2081-2119 (1963); 43, pp. 785-804 (1964); 44, pp. 1547-1562 (1964).

22. C. Christensen, "On the Implementation of AMBIT, a Language for Symbol Manipulation," Comm. ACM 9, No. 8 (August 1966).

23. C. Christensen, "Examples of Symbol Manipulation in the AMBIT Programming Language," Proc. 20th Nat'l. Conf. ACM (August 1965).

24. D. J. Farber, et al., "SNOBOL, a String-Manipulation Language," J. ACM 11, No. 1 (January 1964).

25. D. J. Farber, et al., "The SNOBOL3 Programming Language," Bell System Tech. J. (July-August 1966).

26. A. Caracciolo, et al., "PANON-1B, a Programming Language for Symbol Manipulation," abstract in Comm. ACM 9, No. 8 (August 1966).

27. D. G. Bobrow and J. Weizenbaum, "List Processing and Extension of Language Facility by Embedding," IEEE Trans. EC 13, No. 4 (August 1964).

28. A. Guzman and H. McIntosh, "CONVERT," Comm. ACM 9, No. 8 (August 1966).

29. W. Teitelman, "FLIP--a Format List Processor," Memo MAC-M-263, MIT Project MAC, Cambridge, Mass. (1966).

30. J. Reynolds, "An Introduction to the COGENT Programming System," Proc. 20th Nat. Conf. ACM (August 1965).

31. J. Reynolds, "COGENT Programming Manual," ANL-7022, Argonne National Laboratory (March 1965).

32. C. N. Mooers and L. P. Deutsch, "TRAC, a Text Handling Language," Proc. 20th Nat'l. Conf. ACM, pp. 229-246 (August 1965).

33. C. Strachey, "General-Purpose Macrogenerator," Computer J. (October 1965).

34. K. Cohen and J. H. Wegstein, "AXLE: an Axiomatic Language for String Transformations," Comm. ACM 8, No. 11 (November 1965).

35. E. C. Haines, "The TREET List-Processing Language," SR-133, Mitre Corp., Bedford, Mass. (April 1965).

36. J. E. Sammet, "An Annotated Descriptor Based Bibliography on the Use of Computers for Doing Non-Numerical Mathematics," Computing Reviews 7, No. 4 (July-August 1966).