A FORmula TRANslator for Forth

J.V. Noble
Institute for Nuclear and Particle Physics
University of Virginia
Charlottesville, Virginia 22901

Abstract

This paper discusses problems in automating the translation of FORTRAN expressions into Forth code. To avoid multiple passes, the translator recursively decomposes a complex expression into simpler sub-expressions residing on an expression stack, with corresponding arithmetic operators residing on a parallel operator stack. When the top of the stack can no longer be decomposed, its code is emitted. The program has finished when the stack is empty. The stack-based parser seems simpler both in concept and execution than the trees usually advocated in compiler design.

1. Introduction

In a recent article **[NOBL88]** I celebrated the imminent demise of FORTRAN, claiming Forth is the only modern programming language able to supplant FORTRAN in scientific computing. Even Forth apologists may find this claim extravagant, since Forth is reputedly more suited to machine control and interfacing than to number crunching **[DUNC88]**. However, with numerical calculations delegated to specialized dedicated coprocessors, computation in *any* language becomes an exercise in machine control and interfacing, Forth's acknowledged metier.

One of Forth's virtues — shared with some C's and Pascals — is its ability to intersperse high-level and assembler language. This feature permits simple, efficient, machine-specific optimizations while maintaining portability.

Extensibility is a second virtue. While C, Pascal Modula-2 (and now some BASICs) enable programmers to define new data structures, only Forth, to my knowledge, defines new *operations* (and operators) on the same footing as the old ones. Thus, *e.g.*, it proved easy to supplement integer operators (+ - * /) with floating-point (`F+ F- F* F/`) or complex (`X+ X- X* X/`) ones. In fact, so far as I know, only Forth among compiled languages rivals FORTRAN in the graceful extension of arithmetic to the complex number field.

Forth seems to me more pleasant than FORTRAN in several respects: first, compilation is both immediate and incremental hence testing and debugging proceed in tandem with program design (an example will be given below). Second, Forth enforces good structure because it lacks statement labels and GOTO's. Its control and looping arrangements are simpler and more logical than FORTRAN's. Finally, Forth's simplified subroutine linkage mechanism imposes little calling overhead and thereby encourages fine-grained decomposition into short definitions.

Nevertheless, FORTRAN — despite its manifold deficiencies relative to Forth — contains a useful and widely imitated invention that helps maintain its popularity despite competition from more modern languages. This is the *FOR*mula *TRAN*slator from which the name FORTRAN derives.

© 1990 Institute for Applied Forth Research, Inc.

Forth's lack of a formula translator is keenly felt. Years of scientific Forth programming have not entirely eliminated my habit of first writing a pseudo-FORTRAN version of a new algorithm before reexpressing it in Forth. Realizing this, I have written a Forth formula translator to supplement the scientific Forth lexicon I have described elsewhere [NOBL??]. My aim was to maintain portability by employing only the standard Forth kernel. This article describes a Forth program, FORTRAN.3, that converts a FORTRAN formula to Forth code.

Clearly, the Forth formula translator could become the kernel of a more complete FORTRAN → Forth filter. Although my original aim was to write a full cross-compiler, I have (subsequent to writing the present article) only so far augmented the formula translator with routines to translate FORTRAN DO loops and logical IF...THEN... ELSE...ENDIF statements into their Forth equivalents. The substantial effort and (in my opinion) limited utility of extensions to translate obsolescent and unstructured FORTRAN constructs such as assigned and computed GOTO's, EQUIVALENCE, COMMON and block COMMON, and arithmetic IF's, have discouraged me from tackling an ANSI standard FORTRAN cross-compiler. The formula translator of this article is therefore intended for quick-and-dirty translation of single formulae; with the understanding that a user will ultimately edit the resulting Forth code for style and efficiency. Although some hand work is therefore needed, the incompleteness of the Forth formula translator has not materially hindered my porting several standard FORTRAN library routines.

The radical differences in the ideal forms of Forth and FORTRAN programs also discourage further extensions: stylistically acceptable FORTRAN usually translates into stylistically mediocre Forth or worse (even when no loss of execution speed is incurred). FORTRAN subroutines tend to be long and poorly factored, as noted earlier [NOBL88]. Forth, conversely, can be made almost self-documenting and quite readable, with only moderate effort in choosing names. The program fragments in the text, and the full listing in Appendix C illustrate this aspect of Forth.

The general principles of compiler writing are of course well understood and have been described extensively elsewhere. Several computer science texts expound programs for formula evaluators, *e.g.*, [KRUS87]. The interest of the translator described here lies more in the surprisingly large amount of detailed information about FORTRAN that must be incorporated in the program than in the principles embodied in the compiler itself.

To see how to proceed, let us translate a FORTRAN formula into Forth code by hand. For simplicity, we eschew integer arithmetic and assume all literals will be placed on the floating point stack (*fstack*). Similarly we assume all variable names in the program refer to FVARs (see Appendix A). A word that has become fairly standard is %, which interprets a following number as floating and places it on the *fstack*. With these conventions, we see that we shall want to translate an expression like

$$A = -15.3E7*EXP(7/X) + Z/(W-SIN(THETA*PI/180)/4) \qquad (1)$$

into Forth code something like this (for convenience we give a glossary of unfamiliar floating-point words in Appendix B):

```
% 4  % 18Ø F=PI F\ THETA F*  FSIN  F\  W FR-
Z F\  X % 7 F\  FEXP  % -15.3E7 F*  F+  IS A
```

FORTRAN.3 is invoked with >FTH, as in

```
>FTH ?
A=-15.3E7*EXP(7/X)+Z/(W-SIN(THETA*PI/18Ø)/4); ok
```

and emits the code

```
% 4
% 18Ø
PI F
THETA F*
  FSIN
FNEGATE F
W F+
Z F
X
% 7 F
  FEXP
% -15.3E7 F*
F+
IS A
```

which is similar to, and functionally identical with, the hand-coded version. (The word **PI** is synonymous with **F=PI**: it seemed simpler to make it so than to include code to recognize **PI** as an operator rather than a variable name.)

Section 2 below lists the FORTRAN rules that the translator must preserve. Following these rules, we work out in §3 the algorithm used to achieve the above translation. Section 4 describes the data structures and their associated operators used in the implementation of the algorithm, and §5 details the actual code.

2. *The rules of FORTRAN*

A FORTRAN expression obeys the rules of algebra in a generally obvious fashion. Parentheses can be used to eliminate all ambiguity and force a definite order on the evaluation of terms and factors. However, to reduce the number of parentheses, FORTRAN adopted a hierarchy of operators that has been followed by all other languages that incorporate semi-algebraic replacement statements like Eq. 1 above. The hierarchy, in decreasing order of priority, is

0. FUNCTION
1. EXPONENTIATION (^ or **)
2. * or /
3. + or –
4. , (argument separator in lists)

Our eventual algorithm must enforce these rules. It must also have rules for resolving ambiguities involving operators at the same level. Thus, *e.g.*, does the fragment

A/B*C

mean A/(B*C) or (A/B)*C ? Many FORTRAN compilers follow the latter convention, so we should maintain this tradition.

A second issue is the FORTRAN function library. The formula translator must recognize functions, and be able to determine whether a given function is in the standard library. Thus FORTRAN.3 recognized EXP and SIN as standard library functions and emitted the requisite Forth code that invokes them. A beauty of Forth is that there are several easy ways to accomplish this, using components of the Forth kernel.

A third issue is the ability of a true FORTRAN compiler to perform mixed-mode arithmetic, combining INTEGER*2, INTEGER*4, INTEGER*8, REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 types *ad libitem*. FORTRAN does this using the information contained in the type declarations at the beginning of a routine. A pure formula translator has no such noncontextual information available to it, hence has no way to decide how to insert the proper Forth words during compilation. One way around this is the use of generic operators such as described below, that make the decision at run-time. As we shall see, this approach has advantages and disadvantages relative to classic FORTRAN compilers.

3. Parsing

Let us hand-parse the expression Eq. 1, reproduced below:

A=−15.3E7*EXP(7/X)+Z/(W−SIN(THETA*PI/180)/4) (1)

The first and most obvious thing to do is split at the "=" sign, and interpret the text to its left as a variable name. Since we want to emit the code **IS A** last, yet have parsed it first, we have to hold it somewhere. Clearly the buffer where we store it will be a first-in last-out type; and by induction, last-in, first-out also. But a LIFO buffer is a stack. Hence the fundamental data structure needed in a parsing algorithm is a stack whose elements are strings. So we might imagine that after the first parsing step the string stack contains two strings, as in Fig. 1 below.

$STACK	Notes
IS A	\ last line of code
−15.3E7*EXP(7/X)+Z/(W−SIN(THETA*PI/180)/4)	\ everything else

Fig. 1 The stack used to parse a FORTRAN expression

Following the rule that the lowest priority operator is "+" or "−" (this example contains no commas), we see that the top expression should be broken at the + sign between ")" and Z. We should think of the two sub-expressions

−15.3E7*EXP(7/X)

and

Z/(W−SIN(THETA*PI/180)/4)

as numbers on the *fstack*; hence their code should be emitted before the addition operator (that is, these expressions are higher on the string stack than the addition operator F+). The ***$stack*** now looks like Fig. 2. We may anticipate a new problem: suppose we have somehow — no need to worry about details yet — emitted the code that represents the expression labelled "new TOS" in the Figure. Then we would have to parse the line −15.3E7*EXP(7/X) F+ . Assuming the program knows how to handle the first part, −15.3E7*EXP(7/X), how will it deal with the **F+**? We do not want to use the space as a delimiter (an obvious out) because this will cause trouble with **IS A**.

$STACK	Notes
IS A	\ last line of code
−15.3E7*EXP(7/X) F+	\ this has the F+
Z/(W−SIN(THETA*PI/180)/4)	\ new TOS

Fig. 2 The stack after splitting at +

The difficulty came from placing **F+** on the same line as

−15.3E7*EXP(7/X).

What if we had placed it on the line above, as in Fig. 3? Eventually we realize this solution merely exchanges the problem for another of equal difficulty. How do we distinguish a factor or term ("atom") that contains no more mathematical operators or functions — and is therefore ready to be emitted as code — from the operator F+ , which contains a "+" sign? Now we need complex expression recognition, which will lead to a slow, complicated program.

$STACK	Notes
IS A	\last line of code
F+	\put F+ here
−15.3E7*EXP(7/X)	\this has no F+
Z/(W−SIN(THETA*PI/180)/4)	\new TOS

Fig. 3 The stack after splitting at + (second try)

When this sort of impasse arises (and I am pretending it had been realized early in the design process, although the difficulty did not register until somewhat later) it signals that a key aspect of the problem has been overlooked. Here, we have not properly distinguished Forth words from FORTRAN expressions. By putting them in a common string we have, in effect, mixed disparate data types (like trying to add scalars and vectors). Worse, we discarded too soon information that might have been useful at a later stage. This leads to a programming tip, *a la* Brodie [BROD84]:

TIP: Never discard information. You might need it later.

Phrased this way, the solution becomes obvious: keep the operators on a separate stack, whose level parallels the expressions. So we now envision an expression stack and an operation stack, which we call `E/S` and `O/S` for short. Let us recapitulate the parsing steps taken so far, but on the two stacks, shown in Fig. 4.

E/S	O/S	Notes
IS A	NOP	
−15.3E7*EXP(7/X)+Z/(W−SIN(THETA*PI/180)/4)	NOP	\step 1
IS A	NOP	
−15.3E7*EXP(7/X)	F+	
Z/(W−SIN(THETA*PI/180)/4)	NOP	\step 2

Fig. 4 Parsing with two stacks.

Let us pause briefly to explain the NOP that appears on the `O/S`. Since we want to keep the stack levels the same (so we do not have to check both when POPping off code strings) it is easier to put a nooperation code on the `O/S` to balance a string on the `E/S`. The top of the `O/S` will usually have a NOP. But NOP's can appear elsewhere, if needed.

We are now ready to proceed to Step 3: notice that there are no more exposed (in this context "exposed" means "not contained between parentheses") "+" or "−" operators. The only "−" signs are either a leading "−" in the second line from TOS (unary operator — it needs special treatment) and a "−" buried within parentheses (TOS). But there are exposed operators at a higher priority-level: the "/" in the TOS. So we split the top string at this point, issuing a reverse-divide instruction `F\` (the reason for a reverse-divide is explained below) and putting the left and right sub-expressions on the `E/S` as shown in Fig. 5. A little thought now explains why the `F\` instead of `F/`. The number that the expression (W−SIN(THETA*PI/180)/4) evaluates to will be sitting on the *fstack*. Then Z will be pushed, but we want Z divided by Next-On-Stack, rather than the opposite. (If the math co-processor has no direct machine-instruction for `F\`, we can define it it as `FSWAP F/`.)

E/S	O/S
IS A	NOP
−15.3E7*EXP(7/X)	F+
Z	F\
(W−SIN(THETA*PI/180)/4)	NOP

Fig. 5 Parsing the divide operator.

The parentheses around the expression on TOS serve no purpose, so drop them:

E/S	O/S
IS A	NOP
−15.3E7*EXP(7/X)	F+
Z	F\
W−SIN(THETA*PI/180)/4	NOP

The top expression now has an exposed "−" sign, so the hierarchy of operators tells us to split at that point. This gives Fig. 6.

E/S	O/S
IS A	NOP
−15.3E7*EXP(7/X)	F+
Z	F\
W	F+
−SIN(THETA*PI/180)/4	NOP

Fig. 6 Parsing an embedded "−" as **F+** and **LEADING-**

Here is another decision needing clarification. Why issue F+ and keep the "−" sign with SIN in Fig. 6? The reason is simple: Any 9th grader can tell the difference between a "−" binary operator (binop) and a "−" unary operator (unop) in an expression. But, while not impossible, it is unnecessarily difficult to program this distinction. The Forth philosophy is "Keep it simple!" Simplicity dictates that we embrace every opportunity to avoid a decision, such as that between "−" binop and "−" unop. The algebraic identity

$$X - Y = X + (-Y) \qquad (2)$$

lets us issue only **F+**, as long as we agree always to attach "−" signs as unops to the expressions that follow them. Eventually, of course, we shall have to deal with the distinction between negative literals (−15.3E7, *e.g.*) and negation of variables. The first we can leave alone, since the literal-handling word **FLITERAL** (or **%**) surely knows how to handle a unary "−" sign; whereas the second case will require us to issue a strategic **FNEGATE**.

A consequence of this method for handling "−" signs is that the compiler will resolve the ambiguous expression

$$-X\wedge Y = -(X\wedge Y) \text{ or } (-X)\wedge Y$$

in favor of the latter alternative. If the former is intended, it must be specified with explicit parentheses. By the way, failure to resolve such ambiguities is one of the little ways FORTRAN code that works with one compiler can develop bugs with another.

The next exposed operator is "/", giving

E/S	O/S
IS A	NOP
−15.3E7*EXP(7/X)	F+
Z	F\
W	F+
−SIN(THETA*PI/180)	F\
4	NOP

The parsing has now reached a turning point: the top expression on the **E/S** can be reduced no further. The program must recognize this and emit the corresponding line of code:

```
%4                  \send FORTH
```

The stacks now look like this:

E/S	O/S
IS A	NOP
−15.3E7*EXP(7/X)	F+
Z	F\
W	F+
−SIN(THETA*PI/180)	F\

As predicted, the translation has reached the point where the leading "−" preceding SIN() must be dealt with. We want the decomposition of

−SIN(THETA*PI/180)

to lead to the stack

E/S	O/S
IS A	NOP
−15.3E7*EXP(7/X)	F+
Z	F\
W	F+
NOP	F\
NOP	FNEGATE
SIN(THETA*PI/180)	NOP

To preserve the proper ordering on emission we will want a word **LEADING-** that puts the token for "FNEGATE" on the **O/S** and moves the string

SIN(THETA*PI/180)

to the TOS, issuing a NOP on the **E/S**. (We show explicit NOP's on the **E/S** for clarity, but in operation the program will emit only blanks in the stream of Forth code.)

The function recognition code must now recognize SIN(THETA*PI/180) as a function, and decompose it as

E/S	O/S
IS A	NOP
−15.3E7*EXP(7/X)	F+
Z	F\
W	F+
NOP	F\
NOP	FNEGATE
NOP	FSIN
THETA	F*
PI/180	NOP

Continuing, we find the successive stacks and Forth code emissions

```
E/S                     O/S
-------------------------------------------
IS A                    NOP
-15.3E7*EXP(7/X)        F+
Z                       F\
W                       F+
FNEGATE                 F\
NOP$                    FSIN
THETA                   F*
PI                      F\
180                     NOP
- - - - - - - - - - - - - - - - - - - - - -
% 180                   \send FORTH
PI F\
THETA F*
    FSIN
FNEGATE F\
W F+
Z F\
- - - - - - - - - - - - - - - - - - - - - -
E/S                     O/S
-------------------------------------------
IS A                    NOP
NOP$                    F+
-15.3E7*EXP(7/X)        NOP
E/S                     O/S
-------------------------------------------
IS A                    NOP
NOP$                    F+
-15.3E7                 F*
EXP(7/X)                NOP
E/S                     O/S
-------------------------------------------
IS A                    NOP
NOP$                    F+
-15.3E7                 F*
NOP$                    FEXP
7/X                     NOP
E/S                     O/S
-------------------------------------------
IS A                    NOP
NOP$                    F+
-15.3E7                 F*
NOP$                    FEXP
7                       F\
X                       NOP
- - - - - - - - - - - - - - - - - - - - - -
X                       \send FORTH
% 7 F\
    FEXP
% -15.3E7 F*
    F+
IS A
```

We can derive the parsing rules from having observed the steps in the translation.

1. An expression can be either simple or complex. A simple expression contains no arithmetic operators, except in floating-point exponents and leading "−" signs. Simple expressions contain no parentheses (or commas, which we use to delineate arguments).
2. If the TOS expression is simple, POP the `E/S` and `O/S`, emitting the corresponding Forth code.
3. If the TOS is complex, is it a function? A function is defined as a string containing no exposed operators and ending in ")" , in which the character immediately preceding the first "(" is not an arithmetic operator.
4. If the expression is a function, decompose into a function name and an argument string. Look up the function name in the library and issue the requisite code if it is found.
5. If the name is not found, PUSH the name on the `E/S` as a text string. See below for possible conventions in user-defined functions. PUSH the argument string on the `E/S` and PUSH NOP on the `O/S`.
6. If the expression is not a function, dissect it into subexpressions by splitting appropriately at the arithmetic operators, in the following order: + −, */, ^ .
7. Dissecting at "−" is handled as an exception, as discussed above.
8. To the preceding rule, possibly add the even lower-priority "arithmetic operator" "," (comma) separating arguments within parentheses in a multi-argument function. Thus, dissecting at "," precedes dissecting at the other arithmetic operators.
9. When dissecting, do not recognize operators contained within parentheses ("hidden" operators).
10. Leading "−" signs are handled by FNEGATE if they precede text, including a "(", and by retaining them if they precede a number.
11. A sub-expression is dissected out including all parentheses. A function that removes the outermost parentheses of the TOS expression is therefore required.

In pseudocode we might express the algorithm as

```
: PARSE
    FP#?          IF  CR  ." % "   SEND.FORTH   ELSE
    SIMPLE?       IF  LEADING-  CR  SEND.FORTH  ELSE
    HIDDEN?       IF  LEADING-  EXPOSE          ELSE
    FUNCTION?     IF  LEADING-  FUNCTION!       ELSE
                          DISSECT  THEN  THEN  THEN  THEN
    TOS>  Ø>  IF  RECURSE  THEN  ;
```

4. Data structures

The soul of any computer program is its data structures [KRUS87]. We have seen that parsing a FORTRAN expression uses two stacks. Since stacks are familiar to Forth programmers, the code defining the `E/S` and `O/S` should be fairly self-explanatory. The `E/S` is a stack of integer addresses into a string buffer `<E/S>` of reasonable length. For safety I took `<E/S>` to be a kilobyte, and the depths of `E/S` and `O/S` to be 16; but I am certain that reasonable expressions of < 256 characters will not require more than a stack of depth lg(256) + 1 = 9, and a 512-byte buffer. The code is shown in the full listing, in the section "USER-STACK DEFINITIONS."

Note that the words **`PUSH`** and **`POP`** include run-time error checking. This is useful for debugging. The stacks are addressed by giving their names, then saying **`PUSH`**, **`POP`**, **`S.@`**, **`S.INIT`** or **`TOS>`**. The word **`S.LEN`** was included for clarity, but for optimization it would be replaced by a simple `@`. In HS/Forth, **`S.LEN`** can be made synonymous with `@` *via*

```
SYNONYM S.LEN @
```

thereby avoiding a run-time speed penalty.

5. Coding the FORmula TRANSlator

As usual in Forth program development, we proceed by developing various components. For example, we need to be able to recognize whether a piece of text is a floating point number. The word that does this is **FP#?**. **FP#?** steps through a string and checks each character to see whether it conforms to certain rules:

a. The first character can be a numeral, a "−" sign or a decimal point.

b. After a decimal point is encountered, further decimal points are forbidden.

c. After the first numeral, an exponent (E or e) is permitted.

d. A decimal point is forbidden after an E or e.

e. After the first character, a + or − sign is forbidden, *except* immediately following the E or e of an exponent.

f. Only numerals are accepted after the + or − sign in an exponent.

g. If the first character is a letter, or if the string contains an arithmetic operator (including parentheses or comma) then it is *not* a floating point number. However, if it contains 2 E's or 2 decimal points, or if other characters (not operators) follow numerals, then it is wrong and should lead to an **ABORT**.

Coding **FP#?** is harder than it looks. Parsing rules like the above are most naturally implemented *via* finite state machines (FSMs) [AHO86][SEDG83]. Although Forth offers extremely natural ways to implement FSMs, here we use **BEGIN**...**WHILE**...**REPEAT** loops to simulate an FSM.

The only tricky part is finding a fast way to determine whether a character is one of the permissible ones. The word **WITHIN** lets us find out whether a character is between ASCII 0 and ASCII 9, since these are contiguous in the ASCII code. However, ".", "+", "−", "E" and "e" have codes 46d, 43d, 45d, 69d and 101d, respectively, so the brute-force method requires 5 tests and 6 **OR**'s in addition to **WITHIN**. It is faster simply to remap the desired ASCII characters into contiguous 8-bit numbers in a table. The code for this appears in the section CONTIGUOUS ENCODING TABLE (Appendix C).

By the way, the table includes "*", "/" and "^", "(", ")" and ",". This anticipates having to determine whether a string is "simple" or a function name, according to parsing rule #3 above. Clearly, it will help then to have these characters mapped into contiguous integers. Again, this is retrospective prescience since I realized the utility of the extended table only after having first defined a smaller table, arranged in a less useful order.

With the help of the table lookup word **ASC>FP$** and some auxiliary definitions for good factoring,

```
: WITHIN  DDUP MAX  -ROT MIN  ROT UNDER MAX  -ROT MIN = ;
: DO.ADR  ( $adr - $adr+n+1 $adr+1 ) COUNT OVER + SWAP ;
: Øto9?   ( n - f)   ASCII Ø ASCII 9  WITHIN  ;
: an.op?  ( n - f)   ASC>FP$   13 2Ø  WITHIN  ;
: Ee?     ( n - f)   ASC>FP$   11 12  WITHIN  ;
: +-?     ( n - f)   ASC>FP$   13 14  WITHIN  ;
: SKIP-   ( $adr - $adr or $adr+1)  DUP  C@   ASCII -  =  -  ;
: skip.nums     ( $beg $end - $beg' $end)  >R
        BEGIN   R@  OVER  >                  \ not done?
        OVER   C@   Øto9?  AND               \ and a numeral?
        WHILE   1+   REPEAT   R>  ;
: do.exponent   ( -1 $beg $end - flag or abort)
        OVER   C@    Ee?   ABS  ROT +  SWAP    \ incr. pointer if E or e
        OVER C@ +-?        ABS  ROT +  SWAP    \ incr. pointer if + or -
  skip.nums
        OVER C@   DUP
        Øto9?  SWAP   an.op?  OR               \ acceptable char.?
        IF   =   AND                           \ "true" if end of string,
                                               \ "false" if not eos
        ELSE   ." Improper fp#." ABORT  THEN  ; \ error
```

we may define the key word

```
: FP#?   ( $adr - f)  DO.ADR   SKIP-  >R 1- R>   ( - $end $beg')
        -1  SWAP ROT      ( - -1 $beg $end)
           skip.nums
        OVER C@   ASCII . =                    \ stopped at "." ?
        IF   >R 1+ R>                          \ incr. pointer
           skip.nums
        ELSE   OVER C@  ASC>FP$   32 =         \ an incor. char.?
           IF   DDROP  NOT  EXIT   THEN        \ leave "false"
        THEN   do.exponent   ;
```

(The word SKIP- above used a "hack" — it assumed logical "true" was −1. On some older Forth systems, "true" is +1. SKIP- can be made universal by replacing the final - operator with the phrase ABS +).

We now need to determine whether a string represents a "simple" or compound expression. This is straightforward: a "simple" expression contains no parentheses, commas or operators. Thus

```
: SIMPLE?   ( $adr - f)   -1 SWAP   DO.ADR
     SKIP-   DO   I C@   ASC>FP$   13 2Ø  WITHIN
             IF   NOT   LEAVE   THEN   LOOP  ;
```

To simplify an expression we slice it at exposed arithmetic operators. The "−" operator is an exception, as noted previously. Moreover, the slicing word must know not to slice if the operator is not found. (We can handle a "+" or "−" within a floating point exponent most simply by not FINDing it.) The procedure is

```
: SLICE ( $adr char - flag or $end1 $beg1 $end2 $beg2 )
    OVER  SWAP   FIND)?(  ( - $adr, >op OR Ø)     \ find exposed char
    DUP  Ø=   IF  PLUCK  EXIT  THEN               \ not found. leave Ø
    \ now handle leading "-" exception -----------------------------------
    DUP C@  ASCII -  <>  ABS   >R              \ Ø if -, 1 else
    \ ------------------------------------------ finished handling exception!
    >R  $.ENDS                    ( $adr >op - $end2 $beg1 )
    R@ 1-  SWAP                   ( - $end2 $end1 $beg1 )
    ROT  R>  R> +  ;              ( - $end1 $beg1 $end2 $beg2 )
```

Note how the exception for a "−" is handled: we use the phrase

```
C@ ASCII - <>
```

to ask whether the found operator is a "−". If it is not, then −1 ("true") will be on the stack (this is the case in Forth-83, and also in HS/Forth. In older Forths "true" is +1. By computing with logic (and **ABS**) rather than making decisions with **IF...ELSE...THEN** we not only speed execution, we vastly simplify the program). By taking the absolute value this is converted to 0 for a "−" and 1 for any other arithmetic operator (the result is stored on the *rstack*). This number will eventually be added to the pointer to the operator, >op, to convert it to the beginning of the right-hand expression: $beg2 = >op + (0 or 1). Conversely, the end of the left-hand expression is always $end1 = >op − 1.

The definition of **SLICE** requires two other words: **FIND)?(**, that finds an exposed character in a counted string, and **$.ENDS**, that computes the actual beginning and end of the text in a counted string. The latter is easy:

```
: $.ENDS  ( $adr - $end $beg )  COUNT  OVER  +  1-  SWAP  ;
```

As noted above, we must not find a "+" or a "−" if it is part of a floating point exponent. The characters preceding such a sign are numerals followed by "E" or "e"; while the character following must be a numeral from 0 to 9. Unfortunately, FORTRAN permits expressions that fool this simple rule, e.g., A30E−273.5 (A30E is a permissable variable name, and 273.5 a number). The correct solution is simply to skip over a floating point number:

```
: skip   ( $end $beg f - $end $beg')    >R  OVER   <
      IF   R>  -   ELSE  RDROP   THEN  ;
: skip.fp#   ( $end $beg - $end $beg')
      SWAP  skip.num   SWAP
      DUPC@   ASCII .   =   skip
      SWAP  skip.num   SWAP
      DUPC@   Ee?
      IF   1+  DUPC@   +-?  skip  SWAP  skip.num  SWAP  THEN  ;
```

The word **FIND)?(** finds only exposed operators, not ones enclosed between parentheses. The code looks like this:

```
: +level  ( level - level')  ASCII ( =   ABS  +  ;
: -level  ( level - level')  ASCII ) =   ABS  -  ;
```

```
    : FIND)?(    ( $adr char - >op OR Ø="not found")
         Ø  ROT                             \ set parens.level = Ø
         DO.ADR   SKIP-                     \ ignore leading -
         DO  ( - char parens.level)
              R>  R>  SWAP  skip.fp#  SWAP  >R  >R
              I C@  >R
               OVER  R@ =      ( char=$[i] ?)
               OVER  Ø=  AND  ( AND level=Ø ?)
               IF  RDROP  DROP  I  LEAVE                \ found
               ELSE      R@  +level                     \ incr. ()level
                         R>  -level                     \ decr. ()level
                         DUP  Ø<  ABORT" Too many ) "
               THEN
         LOOP    ( - char >op OR Ø )    PLUCK   ( - >op OR Ø )
         DUP  1 2Ø  WITHIN   ABORT" Too many ( "  ;
```

Note the parenthesis-level is maintained on the stack, hence is a local variable, rather than a `VARIABLE` or `VAR`.

We can now write the words at the heart of the algorithm:

```
: PREDICATE  ( $end $beg - ) MAKE$   PAD  $PUSH  ;
: BREAK.AT     ( .op char - f )
    TOP.LINE  DUP  $BUF.Ø  $!   NOP$  SWAP  $!   \ top line in buffer
    $BUF.Ø   SWAP   SLICE   DUP  Ø=
    IF      $BUF.Ø  TOP.LINE  $!   PLUCK         \ Uncompleted. Leave false.
    ELSE    ?POP.NOPS  DSWAP  PREDICATE  PREDICATE
            O/S  PUSH   .NOP  O/S  PUSH
    -1     THEN   ;                              \ Completed. Leave true.
: DISSECT
        .NOP ASCII ,  BREAK.AT  DEBUG  NOT       \ parse function arguments
     IF  .F+  ASCII +  BREAK.AT  DEBUG  NOT
     IF  .F+  ASCII -  BREAK.AT  DEBUG  NOT
     IF  .F*  ASCII *  BREAK.AT  DEBUG  NOT
     IF  .F   ASCII /  BREAK.AT  DEBUG  NOT
     IF  .F** ASCII ^  BREAK.AT  DEBUG  DROP
     THEN  THEN  THEN  THEN  THEN  ;
```

The word `DEBUG` is debugging code that dumps both `E/S` and `O/S` when enabled by `DEBUG-ON`; it is disabled by `DEBUG-OFF`. See the section of the listing marked "DEBUGGING CODE."

The nested `IF`...`THEN`'s appear ugly to me, but Forth seems to offer no simpler technique of successive decisions. The mnemonics for floating point operation codes, `.F+`, *etc.*, are `CONSTANTS`, as in

```
     Ø   CONSTANT .NOP
    1Ø   CONSTANT .F+
    ..................
```

We have also supposed that `FUNCTION?`, `FUNCTION!`, and `$PUSH` have been defined previously. The definition of `$PUSH` is

```
    : TOP.LINE  E/S  S.@  ;
    : $PUSH   ( $adr - )                      \ text at $adr
         TOP.LINE  DUPC@  + 1+                \ new$adr in <E/S>
         DUP  E/S PUSH   ( - $adr new$adr )  $!  ;
```

The word `FUNCTION?` determines whether a string is a FORTRAN function. Clearly there is some ambiguity here, since in FORTRAN the notation A(I,J) could mean, *ab initio*, either the I,J'th element of an array or a function whose arguments are I and J. A FORTRAN compiler resolves this by requiring arrays to be declared in DIMENSION statements. Any undeclared construct of this sort would then be interpreted as a function. Since the formula translator has no declaration statements to guide it, it *always* assumes A(I,J) is a user-defined function.

A function is defined to be a string that ends with a right parenthesis, that has no exposed arithmetic operators (+−*/^), and whose name (*i.e.* all characters preceding the first left parenthesis) contains no arithmetic operators. Because we test whether a string is a function *before* we DISSECT (that is, we have not yet tested for exposed operators) we must exclude cases such as

$$(A+B)/(C-D)$$
$$SIN(A+B)/(C-D)$$
$$SIN(A+B)/EXP(C-D)$$

that is, compound expressions that might or might not contain functions. The simplest method to perform this test, it seems to me, is to work from right to left. This leads to

```
: FUNCTION? ( $adr - f ) -1 -1 ROT ( - -1 -1 $adr)
    $.ENDS    ( - $end $beg )  DUPC@   SKIP-
    SWAP      ( - $beg' $end ) DUPC@  ASCII ) <>       \ test for final ")"
    IF  DDROP  DROP  NOT  EXIT  THEN     1-            \ decr. $end
    DO  I C@  DUP>R   -level   R@   +level
         R>  ASC>FP$  13 17 WITHIN   OVER Ø=  AND
         IF  SWAP  LEAVE  THEN
    -1  +LOOP  DROP  ;                                 \ drop ()level
```

Forth lets us test this definition immediately:

```
$" -(A+B)*(C-D)"       FUNCTION? . Ø  ok
$" -SIN(A+B)*(C-D)"    FUNCTION? . Ø  ok
$" -SIN(A+B)/COS(C-D)" FUNCTION? . Ø  ok
$" -SIN(A+B)"          FUNCTION? . -1  ok
$" -SIN(COS(A+B))"     FUNCTION? . -1  ok
```

To save an extra comparison (having tested for it in the third line, we *know* the last character is ")") the parenthesis level is initialized to −1 and the DO...LOOP begins with the next-to-last character (this is ensured by the 1- following the first IF...THEN). The backward counting is taken care of by SWAPing the pointers to the beginning and end of the text, and by the phrase -1 +LOOP.

Now that we know how to tell whether a text fragment is a function, how do we apply this knowledge? The word FUNCTION! must perform these operations, presented as pseudocode:

```
: FUNCTION!
    split the text at the first "(" to get NAME and ARG.STRING

                        ⎧Y:  push its Forth equivalent on the O/S
    NAME in library?    ⎨    and NOP$ on the E/S
                        ⎩N:  push NAME on E/S, .NOP on O/S

    push ARG.STRING on the E/S, .NOP on the O/S  ;
```

There is only one way to determine whether NAME is in the function library. We require a table of library function names (tokens), a set of words that will look up a candidate NAME in the table, and a method for translating NAMEs to their Forth equivalents, *e.g.* SIN → .FSIN (on the O/S).

Although it is not very hard to program these operations, the accessibility of Forth's compiler offers a better way. Looking up names in a dictionary is of course the key to both compilation and interpretation in Forth. Let us suppose we have a method to find the CFA's of words bearing the names of FORTRAN functions. Then we can simply define the action of such a library function appropriately and EXECUTE it after finding its CFA. We can use the mnemonics for the

Forth floating point library, *e.g.* `.FSIN`, `.FEXP` ..., as shown in the code marked "FUNCTION LIBRARY" in the full listing.

When we have located the CFA of `EXP`, we `EXECUTE` it and the result would be to add the new top line

E/S	O/S
....	....
NOP	EXP

The various dialects of Forth differ slightly in the methods they require for finding the relevant CFA. In Forth-83 the word `FIND` expects the address of a counted string. If the string matches a dictionary entry, `FIND` leaves the CFA and −1 for an ordinary word and +1 for an IMMEDIATE word; if no match is found, the address of the string and 0 are left. The Forth-79 version of `FIND` operates differently: it takes the text as the next blank-delimited token in the input stream, seeks a dictionary match and leaves either the CFA or 0, depending on whether or not the token is found. My initial approach — as HS/Forth is mainly Forth-79 except for following Forth-83 conventions in "true" and "false" — was to use the old `FIND` in conjunction with the word `EVAL` [TRAC87], as follows:

```
$" FIND " $CONSTANT FIND$
: IN.LIBRARY?  ( $adr - cfa OR Ø )  FIND$  SWAP  $+   PAD   EVAL ;
```

Since HS/Forth already contains the equivalent of `PAD EVAL` (called `PLOAD`) this method was effortless.

The equivalent Forth-83 definition would be

```
: IN.LIBRARY?  ( $adr - cfa OR Ø )  FIND  DUP  Ø=  IF PLUCK THEN ;
```

which is obviously simpler and faster. A word that would do the same thing as the Forth-83 `FIND`, but in Forth-79, could be defined given an intimate knowledge of the interpreter. The HS/Forth equivalent word would be

```
: IN.LIBRARY?  ( $adr - cfa OR Ø )  CONTEXT @  <FIND>
     PLUCK  SWAP  @E  SWAP ;
```

(Since we envision putting the function library in the same vocabulary as the parser, there is no need to make this word search all vocabularies. The HS/Forth word `@E` is specific to this dialect and the Intel 80x86 family: it fetches using the given offset and the ES segment register, which has been properly set to the segment containing the current vocabulary by the phrase `CONTEXT @ <FIND>`.)

Finally, how do we handle something that is obviously a function but not in the library? Since I have been using a function notation designed to clarify Forth programs ([3] and Appendix A),

```
FUNC{ name }TION ,
```

I made the translator emit a user-defined function in the above form. This is convenient for me, and also alerts the user that the translator thinks his code is a function, not an array. The words that implement this approach are

```
$" FUNC{ "  $CONSTANT  FUNC{
$" }TION "  $CONSTANT }TION
: USER.FN!  ( $buf - )  FUNC{  OVER  $+   PAD OVER  $!
     }TION  $+   PAD  $PUSH   .NOP  O/S PUSH  ;
```

With the definitions of `IN.LIBRARY?` and `USER.FN!` we define `FUNCTION!` as

```
256 $VARIABLE  $BUF
32  $VARIABLE  fn.name
```

```
: FUNCTION!
    TOP.LINE  DUP  $BUF  $!   NOP$  SWAP  $!      \ TOS in $BUF, NOP in TOS
    ?POP.NOPS                                     \ drop line of NOP's
    $BUF  ASCII (  SLICE   1-                     \ slice into name, arg.list
    DSWAP  MAKE$  PAD  fn.name  $!                \ put name in a buffer
    fn.name  IN.LIBRARY?   ?DUP  Ø=               \ not in library if Ø
    IF  fn.name  USER.FN!  ELSE   EXECUTE   THEN
    PREDICATE   .NOP  O/S PUSH  ;                 \ arg.list on TOS
```

Using the tools developed so far, the words LEADING-, and EXPOSE are easily defined:

```
: LEADING-   ( - )   TOP.LINE    \ remove leading "-" from expression
     DUP  $.ENDS  DUPC@   ASCII -  =              \ leading "-" ?
     IF  1+  MAKE$
         NOP$  SWAP  $!   O/S S.@  .NOP =  IF  O/S  POP DROP  THEN
         .FNEGATE  O/S PUSH                       \ issue FNEGATE
         PAD   $PUSH    O/S TOS>  E/S TOS>  <     \ new top.line
         IF  .NOP    O/S   PUSH  THEN             \ balance stacks
     ELSE  DDROP  DROP  THEN  ;                   \ not a leading "-"
: EXPOSE    ( - )   TOP.LINE     \ remove outer parens from top expression
     DUP  $.ENDS    DUPC@  ASCII ( =  -    SWAP
                    DUPC@  ASCII ) =  +    SWAP   ( - $end-1 $beg+1 )
     MAKE$    PAD  SWAP  $!  ;
```

The final word we need is PARSE, which we pseudocoded above in §3. Its definition is essentially the same (note we explicitly coded the tail recursion as a BEGIN...WHILE...REPEAT loop in this version).

```
: SEND.FORTH    E/S POP    $.   O/S POP .FCODES ;    \ emit Forth code
: PARSE  ( - )                                       \ FORTRAN -> Forth
    BEGIN    E/S TOS>  Ø>   WHILE
    TOP.LINE  FP#?      IF   CR  ." % "     SEND.FORTH    ELSE
    TOP.LINE  SIMPLE?   IF   LEADING-  CR  SEND.FORTH    ELSE
    TOP.LINE  HIDDEN?   IF   LEADING-  EXPOSE            ELSE
    TOP.LINE  FUNCTION? IF   LEADING-  FUNCTION!         ELSE
                             DISSECT   THEN  THEN  THEN  THEN
    REPEAT  ;
```

The process of translation is virtually self-documenting. With the addition of words to read the keyboard, or to read lines from a file, we have the germ of a FORTRAN compiler.

Notice that I have followed the BASIC convention of using ^ for exponentiation, rather than FORTRAN's **. The token ** can be handled easily by substituting ^ for it in the input string, prior to parsing. But this would be needed only when translating existing FORTRAN programs, as ^ is clearly better for expressions entered from the keyboard.

An obvious improvement that has not escaped my attention (bypassed in the initial design in the interest of expeditiously getting the program working) would be to replace the expression address stack and the 1 kbyte expression stack by a stack holding addresses (relative to EXPRESSION+1) and lengths of sub-strings of the original expression (note standard 2-byte cells will hold both data). Then moving strings around would become unnecessary, and the workspace could be reduced.

6. Typed variables

FORTRAN permits mixed number-types in arithmetic expressions. It achieves this at the cost of demanding compile-time typing of variables, so the compiler will have the information needed to choose which of the 36 possible subroutines to use, *e.g.*, in the expression A**B.

Elsewhere [NOBL90] I have proposed a system of low-overhead run-time data typing. The key to run-time typing is that **SCALAR** must be a defining word that makes a data structure containing a type label as well as storage for its data. The generic operations must then be able to use this information to vector to appropriate routines.

Among other things, run-time data typing makes practical a library of generic functions and subroutines, that work with any of the four most-used data types (4-byte REAL, 8-byte DREAL, 8-byte COMPLEX and 16-byte DCOMPLEX) with equal facility. The modifications to the formula translator are trivial: the mnemonics would be replaced with generic ones, the variable names are emitted with the code **G@** (generic fetch), the terminating phrase **IS A** would be replaced with **A G!**, the function library would be extended, *etc.* One could even retain the **VAR** notation by using multiple code-field words, but I am not sure this is worth the trouble.

References

[NOBL88] Noble, J.V., "Fortran is Dead! Long Live Forth!," *JFAR*, **5**, 2 (1988) pp.261.

[DUNC88] Duncan, R., *Programmer's Journal*, **6**, 6 (1988) pp.56.

[NOBL??] Noble, J.V., "Scientific Forth: A Modern Language for Scientific Computing," in preparation.

[KRUS87] Kruse, R.L., *Data Structures and Program Design*, 2nd Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[BROD84] Brodie, L., *Thinking FORTH*, Prentice-Hall, Inc., NJ, 1984.

[AHO86] Aho, A.V., Sethi, R. and Ullman, J.D., *Compilers: Principles, Tools and Techniques*, Addison Wesley Publishing Company, Reading, MA, 1986.

[SEDG83] Sedgewick, R., *Algorithms*, Addison Wesley Publishing Company, Reading, MA, 1983.

[TRAC87] Tracy, M., *Dr. Dobb's Journal*, 12/87, pp.152. (**EVAL** expects the address of a counted string containing Forth words, and executes them as though they had been input from the keyboard.)

[NOBL90] Noble, J.V., "Data Structures for Scientific Programming in Forth," *JFAR*, **6**, 1 (1990) pp.47.

Dr. Noble received his B.S. from Caltech in 1962, his M.S. from Princeton in 1963 and his PhD from Princeton in 1966, all in Physics. Beginning with Fortran I in 1960, he has programmed in Basic and Assembler, but almost exclusively in Forth since 1985. His interests include theoretical physics (nuclear, particle and astrophysics), theoretical biology (epidemiology and chaos), and science and public policy. Dr. Noble's major use of Forth is in number crunching.

Appendix A

A.1 Multiple code field words

A **VAR** is a data structure with three code fields, invoked at compile time by the presence or absence of the word **IS**, or by the presence of the word **AT**. Thus, the code fragment below defines/initializes a **VAR** named **N**; then shows how **VAR** can behave like a constant (its name brings its contents to the stack); then illustrates how it can act like a variable, since **IS** allows its contents to be changed. And finally, the third code field, invoked with **AT**, places its address on the stack should direct access be necessary.

```
Ø VAR  N
: .N   N . ;  ( - )
: SET.N   IS N ;  ( n - )
```

An FVAR is exactly the same, except it uses the *fstack* and holds a floating point number. Some systems use the name QUAN rather than VAR.

An FVAR can be simulated on systems lacking VARs *via*

```
VARIABLE <is>
VARIABLE <at>
: <reset>   Ø <is> !  Ø  <at> !  ;   <set>
: IS   -1 <is> !  Ø  <at> !  ;
: AT   -1 <is> !  Ø  <is> !  ;
: F,   HERE  R32!  4 ALLOT  ; IMMEDIATE
CASE:  DOFVAR    R32@   R32!   NEXT  ;CASE
: FVAR   ( n - )  CREATE  F,  DOES>
     ( a - )  <at> @  2 AND  <is> @  1 AND +   DOFVAR   <reset> ;
```

A.2 A notation for functions in Forth

The HS/Forth definition of FUNC{ is

```
: FUNC{   [COMPILE] '  CFA  ;  IMMEDIATE
```

whereas }TION is a SYNONYM of EXECUTE.

Appendix B: Non-standard Definitions

%	interpret following text as fp#, place on *fstack*
FVARIABLE	create a named 4-byte fp variable (– adr)
FCONSTANT	create a named 4-byte fp constant (:: – x)
F=Ø	(:: – Ø)
F=1	(:: – 1)
F=PI	(:: – 3.14159...)
F=E	(:: – 2.71828...)
F=LN(1Ø)	(:: – 2.3Ø258...)
F*	(:: x y – x*y)
F**2	(:: x – x*x)
F2*	(:: x – x*2)
F2/	(:: x – x/2)
F+	(:: x y – x+y)
F/	(:: x y – x/y)
F\	(:: x y – y/x)
F-	(:: x y – x-y)
FR-	(:: x y – y-x)
FSQRT	(:: x – x^Ø.5)
FEXP	(:: x – e^x)
FLN	(:: x – ln[x])
F**	(:: x y – x**y)
	integer in name indicates no. of bits precision
R32@, R32!	(adr[x] – :: – x), (adr[x] – :: x –)
R64@, R64!	(adr[x] – :: – x), (adr[x] – :: x –)
R8Ø@, R8Ø!	(adr[x] – :: – x), (adr[x] – :: x –)

Appendix C: Listing of FORTRAN.3

```
CR CR
.( TRANSLATE A FORTRAN EXPRESSION TO FORTH CODE: ) CR
.(                 ARITHMETIC AND MATHEMATICAL FUNCTIONS )
CR CR  .(       Copyright NobleHouse Software 1989 ) CR
.(       Modification or sale of this software or removal of this ) CR
.(       copyright notice will constitute grounds for legal action ) CR

TASK FTN2FTH

\ STRING HANDLING ( HS/FORTH COPYRIGHT DEFINITIONS AS SHOWN ) --------------

: WITHIN   ( n a b - f) DDUP MIN  -ROT  MAX  ROT UNDER MIN  -ROT MAX  = ;

\ Reproduced with permission from HARVARD SOFTWORKS for non-commercial ------
\ personal use in this utility only ----------------------------------------
HEX
: $VARIABLE (         #bytes  -  )   CREATE 1+ ALLOT  ;' DOVAR
: $CONSTANT ( srcadr   - ) CREATE HERE OVER C@ 1+ DUP ALLOT CMOVE ;' DOVAR
: $!        ( srcadr  dstadr  -  )    OVER C@ 1+ CMOVE ;
: $+        ( adr$1 adr$2  - pad )
            DUPC@ >R 1+  OVER C@ PAD + 1+     R@  <CMOVE
            PAD $!       R> PAD C@ +   Ø MAX FF MIN    PAD C!  ;
DECIMAL
\ END HS/FORTH DEFINITIONS -------------------------------------------------

\ BEGIN FORMULA TRANSLATOR =================================================

\ MNEMONICS  FOR FLOATING-POINT OPERATIONS ---------------------------------
Ø CONSTANT .NOP                30 CONSTANT .FSQRT
1 CONSTANT .R32!               31 CONSTANT .FEXP
2 CONSTANT .R32@               32 CONSTANT .FLN
                               33 CONSTANT .F**
                               34 CONSTANT .FSIN
                               35 CONSTANT .FCOS
                               36 CONSTANT .FTAN
                               37 CONSTANT .FATAN
 1Ø CONSTANT .F+               38 CONSTANT .FASIN
 11 CONSTANT .F*               39 CONSTANT .FACOS
 12 CONSTANT .F/               4Ø CONSTANT .FSINH
 13 CONSTANT .F\               41 CONSTANT .FCOSH
 14 CONSTANT .FINV             42 CONSTANT .FTANH
 15 CONSTANT .F2*              43 CONSTANT .FASINH
 16 CONSTANT .F2/              44 CONSTANT .FACOSH
                               45 CONSTANT .FATANH
 2Ø CONSTANT .FNEGATE
 21 CONSTANT .FABS
 22 CONSTANT .FMAX
 23 CONSTANT .FMIN
```

```
: .FCODES   BEGIN-CASE     ( OUTPUT MNEMONICS AS TEXT )
            Ø CASE-OF  ."    "      ELSE
            1 CASE-OF  ."   R32! "   ELSE
            2 CASE-OF  ."   R32@ "   ELSE

            1Ø CASE-OF ."   F+ "     ELSE
            11 CASE-OF ."   F* "     ELSE
            12 CASE-OF ."   F/ "     ELSE
            13 CASE-OF ."   F\ "     ELSE
            14 CASE-OF ."   FINV "   ELSE
            15 CASE-OF ."   F2* "    ELSE
            16 CASE-OF ."   F2/ "    ELSE

            2Ø CASE-OF ."   FNEGATE "   ELSE
            21 CASE-OF ."   FABS "      ELSE
            22 CASE-OF ."   FMAX "      ELSE
            23 CASE-OF ."   FMIN "      ELSE

            3Ø CASE-OF ."   FSQRT "   ELSE
            31 CASE-OF ."   FEXP "    ELSE
            32 CASE-OF ."   FLN "     ELSE
            33 CASE-OF ."   FSWAP  F** "     ELSE
            34 CASE-OF ."   FSIN "    ELSE
            35 CASE-OF ."   FCOS "    ELSE
            36 CASE-OF ."   FTAN "    ELSE
            37 CASE-OF ."   FATAN "   ELSE
            38 CASE-OF ."   FASIN "   ELSE
            39 CASE-OF ."   FACOS "   ELSE

            4Ø CASE-OF ."   FSINH "   ELSE
            41 CASE-OF ."   FCOSH "   ELSE
            42 CASE-OF ."   FTANH "   ELSE
            43 CASE-OF ."   FASINH "  ELSE
            44 CASE-OF ."   FACOSH "  ELSE
            45 CASE-OF ."   FATANH "  ELSE
               DROP    ."    "      END-CASE ;

\ Ex: .NOP  .FCODES
\     .R32! .FCODES

\ -------------------------------------------------------------------------

\ READ AN EXPRESSION FROM THE KEYBOARD -------------------------------------

256 $VARIABLE  EXPRESSION
```

```
: GET.EXP     ." ? "  CR Ø                        \ emit "?", count on stack
       BEGIN  KEY DUP                             \ get letter from KB
              ASCII ;  <>                         \ is it a ";" ?
       WHILE                                      \ if not
          DUP   32 <>                             \ ignore BL
          IF   DUP
               8 =  IF  EMIT  1-                  \ is it a BS ?
                    ELSE  DUP EMIT                \ echo to CRT
                          SWAP  1+                \ increment count
                          UNDER   PAD +  C!       \ add to string at PAD
                    THEN
               DUP  73 MOD  72  =                 \ end of input line ?
               IF  CR  THEN                       \ newline
          ELSE  DROP
          THEN
       REPEAT  EMIT                               \ echo ";" to CRT
       PAD  UNDER  C!   EXPRESSION $! ;           \ store count, string

\ ---------------------------------------------------------------------------

\ ADDITIONAL $ WORDS -------------------------------------------------------

: $.ENDS  ( $adr - $end $beg=$adr+1 )  COUNT   OVER +  1-  SWAP ;
: DO.ADR  ( $adr - $end+1  $adr+1 )    COUNT   OVER +  SWAP ;

: MAKE$  ( $end $beg -)  \ make a string at PAD given endpoints
    UNDER  -  DUP  Ø<  ABORT" Improper endpoints"  ( - $beg n)
    1+  DUP>R  PAD  1+  SWAP   CMOVE   R>   PAD  C! ;
\ --------------------------------- END ADDITIONAL $ DEFINITIONS ----------

\ USER-STACK DEFINITIONS --------------------------------------------------

: STACK  CREATE  DUP ,  Ø ,  2*  ALLOT ;' DOVAR

: S.LEN   ( adr - len)  @  ;
: TOS>    ( adr - TOS)  2+  @  ;
: STACK.INIT  ( adr -)  2+  Ø!  ;

: PUSH  ( n adr -)  DUP>R   S.LEN    R@  TOS>  =        \ check if room
   IF   CR   R> CFA  .WORD  ." -STACK FULL"   ABORT THEN
   R@  TOS>  2*    R@  +  4+  !                         \ put # on u.stack
   R>  2+  1+!  ;                                       \ increment  TOS>

: S.@   ( adr - n)   DUP  TOS>  1-  2* +  4+  @ ;       \ copy TOS to p.stack

: POP   ( adr - n)   DUP  TOS>  Ø<                \ test for empty
   IF   CR      CFA  .WORD  ." -STACK EMPTY"  ABORT THEN
   DUP  S.@  SWAP                                 \ fetch TOS
   2+  1-!  ;                                     \ decrement TOS>
\ ------------------------------------------------ END U.STACK DEFINITIONS --

16 STACK E/S                                      \ expression address stack
16 STACK O/S                                      \ operations tokens stack
```

```
CREATE  <E/S>  1Ø24 ALLOT  OKLW                   \ the expression stack

\ DEBUGGING CODE ---------------------------------------------------------------

: .STACK   E/S  TOS>  Ø                           \ dump the stacks
      OVER Ø=  ABORT" EXPRESSION STACK EMPTY"
      DO    CR  I 2*  4+
           E/S OVER +  @ $.  0/S +  @ .FCODES
      LOOP ;

Ø VAR DEBUG?
: DEBUG-ON   -1 IS DEBUG?  ;  : DEBUG-OFF  Ø IS DEBUG?  ;
: DEBUG   DEBUG? IF .STACK CR  THEN ;
\ ------------------------------------------------- END DEBUGGING CODE ------

\ CONTIGUOUS ENCODING WORDS ----------------------------------------------------

CREATE FP$  128 ALLOT OKLW     \ MAKE A TRANSLATION TABLE
FP$ 128 BL FILL                \ FILL IT WITH ASCII 32
Ø ASCII Ø  FP$ +  C!       9  ASCII 9  FP$ +  C!      \ CONTENTS OF TABLE
1 ASCII 1  FP$ +  C!       1Ø ASCII .  FP$ +  C!
2 ASCII 2  FP$ +  C!       11 ASCII e  FP$ +  C!   18 ASCII ( FP$ + C!
3 ASCII 3  FP$ +  C!       12 ASCII E  FP$ +  C!   19 ASCII ) FP$ + C!
4 ASCII 4  FP$ +  C!       13 ASCII +  FP$ +  C!   2Ø ASCII , FP$ + C!
5 ASCII 5  FP$ +  C!       14 ASCII -  FP$ +  C!
6 ASCII 6  FP$ +  C!       15 ASCII *  FP$ +  C!
7 ASCII 7  FP$ +  C!       16 ASCII /  FP$ +  C!
8 ASCII 8  FP$ +  C!       17 ASCII ^  FP$ +  C!

: ASC>FP$   ( char -- n)   FP$ +  C@ ;
\ ------------------------------------------------- END ENCODING WORDS ------

\ EXPRESSION TESTING WORDS -----------------------------------------------------
: Øto9?    ( n -- f)   ASCII Ø ASCII 9   WITHIN  ;
: an.op?   ( n -- f)   ASC>FP$   13 2Ø  WITHIN  ;
: Ee?      ( n -- f)   ASC>FP$   11 12  WITHIN  ;
: +-?      ( n -- f)   ASC>FP$   13 14  WITHIN  ;
: SKIP-    ( $adr -- $adr or $adr+1)  DUP  C@   ASCII -  =  -  ;

: skip.nums     ( $beg $end -- $beg' $end)  >R
        BEGIN   R@  OVER  >                      \ not done?
        OVER    C@   Øto9?  AND                  \ and a numeral?
        WHILE   1+    REPEAT   R>  ;

: do.exponent   ( -1 $beg $end -- flag or abort)
        OVER    C@    Ee?   ABS  ROT +  SWAP     \ incr. pointer if E or e
        OVER C@  +-?        ABS  ROT +  SWAP     \ incr. pointer if + or -
             skip.nums
        OVER C@   DUP
        Øto9?  SWAP   an.op?  OR                 \ acceptable char.?
        IF   =   AND                             \ "true" if end of string,
                                                 \ "false" if not eos
        ELSE   ." Improper fp#." ABORT  THEN  ; \ error
```

```
: FP#?   ( $adr – f)  DO.ADR   SKIP-  >R 1- R>  ( – $end $beg')
         -1  SWAP ROT      ( – -1 $beg $end)
            skip.nums
         OVER C@   ASCII . =                    \ stopped at "." ?
         IF   >R 1+ R>                          \ incr. pointer
            skip.nums
         ELSE   OVER C@  ASC>FP$   32 =         \ an incor. char.?
            IF   DDROP  NOT  EXIT   THEN        \ leave "false"
         THEN   do.exponent  ;

: SIMPLE?   ( $adr – f )  -1  SWAP   DO.ADR     \ test for simple expression
     DUPC@  ASCII - =  -                        \ skip leading "-"
     DO  I C@   ASC>FP$  13 2Ø  WITHIN
           IF  NOT  LEAVE  THEN     LOOP  ;

: HIDDEN?   ( $adr – f )   DUP  $.ENDS          \ enclosed in parens ?
       DUPC@   ASCII - =  -                     \ incr. $beg if leading "-"
       C@  ASCII ( =   SWAP     C@  ASCII ) =   AND
       NOT  IF  DROP  Ø  EXIT  THEN             \ no leading and trailing ()
       -1  Ø  ROT   DO.ADR
       DO  I C@   DUP>R  ASC>FP$
           13 17 WITHIN   OVER Ø= AND           \ +-*/ OR ^ and ().level=Ø
           IF  RDROP  SWAP  LEAVE  ELSE         \ Exposed op. ( – Ø -1 )
              R@  ASCII ( =  -    R>  ASCII ) =  + THEN
       LOOP   ( – -1 Ø or Ø -1 )  DROP   ;
\ ----------------------------------------------------- END TEST WORDS ------

\ PARSING WORDS -------------------------------------------------------------
: skip   ( $end $beg f – $end $beg')    >R  OVER   <
     IF   R>  -   ELSE  RDROP   THEN  ;
: skip.fp#   (  $end $beg – $end $beg')
     SWAP  skip.num    SWAP
     DUPC@   ASCII .   =   skip
     SWAP  skip.num    SWAP
     DUPC@   Ee?
     IF   1+  DUPC@    +-?  skip  SWAP  skip.num  SWAP  THEN  ;

: +level  ( level – level')  ASCII ( =   ABS  +  ;
: -level  ( level – level')  ASCII ) =   ABS  -  ;
: FIND)?(   ( $adr char - >op OR Ø="not found")
     Ø  ROT                            \ set parens.level = Ø
     DO.ADR   SKIP-                    \ ignore leading -
     DO  ( - char parens.level)
         R>  R>  SWAP  skip.fp#  SWAP  >R  >R
         I C@  >R
         OVER  R@ =     ( char=$[i] ?)
         OVER  Ø=  AND  ( AND level=Ø ?)
     IF  RDROP  DROP  I  LEAVE                    \ found
     ELSE      R@  +level                         \ incr. ()level
               R>  -level                         \ decr. ()level
               DUP  Ø<  ABORT" Too many ) "
     THEN
     LOOP   ( - char >op OR Ø )   PLUCK   ( - >op OR Ø )
     DUP  1 2Ø  WITHIN   ABORT" Too many ( "  ;

: TOP.LINE  E/S  S.@  ;

: SLICE  ( $adr char - $end1 $beg1 $end2 $beg2 OR flag=Ø )
     OVER   SWAP   FIND)?(    ( - $adr >op OR Ø )
     DUP  Ø=   IF  PLUCK  EXIT  THEN   \ Not found. Leave Ø.
     DUPC@  ASCII -  =  1+             \ Slice. Include leading "-"
     ( - $adr >op n=Ø OR 1 )
            >R  >R  $.ENDS          ( - $end2 $beg1 )
            R@  1-  SWAP            ( - $end2 $end1 $beg1 )
            ROT  R> R> +  ;         ( - $end1 $beg1 $end2 $beg2 )

: $PUSH  ( $adr - )   TOP.LINE  DUPC@  +  1+    \ new $adr
      DUP  E/S   PUSH  $!  ;

$"   " $CONSTANT NOP$

: ?POP.NOPS    O/S  S.@   .NOP =    TOP.LINE  1+  C@  BL =   AND
               IF  E/S  POP    O/S  POP   DDROP   THEN  ;

: LEADING-   ( - )   TOP.LINE   \ remove leading "-" from expression
     DUP  $.ENDS  DUPC@   ASCII -  =                  \ leading "-" ?
     IF  1+  MAKE$
         NOP$  SWAP  $!   O/S S.@  .NOP =  IF  O/S  POP DROP  THEN
         .FNEGATE  O/S PUSH           \ issue FNEGATE
         PAD   $PUSH     O/S TOS>  E/S TOS>  <         \ new top.line
         IF  .NOP    O/S  PUSH  THEN
     ELSE  DDROP  DROP  THEN  ;

: EXPOSE    ( - )    TOP.LINE    \ remove outer parens from top expression
     DUP  $.ENDS    DUPC@  ASCII ( =  -   SWAP
                    DUPC@  ASCII ) =  +   SWAP    ( - $end-1 $beg+1 )
     MAKE$   PAD  SWAP  $!  ;

256 $VARIABLE  $BUF.Ø
64  $VARIABLE  $BUF.1

$" IS "  $CONSTANT  IS$
```

```
: SUBJECT   ( $end $beg - )  MAKE$    PAD  $BUF.Ø  $!
    IS$  $BUF.Ø  $+
    PAD  <E/S> $!   <E/S>  E/S PUSH   .NOP  O/S PUSH  ;

: PREDICATE  ( $end $beg - ) MAKE$   PAD  $PUSH  ;

: FIRST.LINE  ( $adr -)  \ used as EXPRESSION  FIRST.LINE
    E/S  STACK.INIT   O/S  STACK.INIT
    ASCII =  SLICE   DUP  Ø=  ABORT" No = in expression"
    DSWAP  SUBJECT                            \ make subject field
    PREDICATE    .NOP   O/S PUSH  ;           \ make predicate field

: BREAK.AT    ( .op char - f )
    TOP.LINE  DUP  $BUF.Ø  $!   NOP$  SWAP  $!   \ top line in buffer
    $BUF.Ø   SWAP   SLICE   DUP  Ø=
    IF      $BUF.Ø  TOP.LINE  $!   PLUCK         \ Uncompleted. Leave false.
    ELSE    ?POP.NOPS  DSWAP  PREDICATE  PREDICATE
            O/S  PUSH   .NOP  O/S  PUSH
    -1     THEN   ;                              \ Completed. Leave true.

: FUNCTION?  ( $adr - f ) -1 -1 ROT  ( - t -1 $adr)
    $.ENDS    ( - $end $beg )  DUPC@   SKIP-
    SWAP      ( - $beg' $end ) DUPC@  ASCII ) <>      \ test for final ")"
    IF   DDROP  DROP  NOT  EXIT   THEN                \ no ), f=Ø
    1-                                                \ decr. $end
    DO  I C@  DUP>R   -level   R@   +level            \ count ()
        R>  ASC>FP$  13 17 WITHIN   OVER Ø=  AND      \ exposed op?
        IF  SWAP  LEAVE  THEN                         \ ( - Ø t)
    -1  +LOOP  DROP  ;                                \ drop ()level

: COUNT()  DUP>R  ASCII ( = -   R>  ASCII ) = +  ;
: FUNCTION?   ( $adr - f )  Ø SWAP  ( - Ø $adr)
      $.ENDS  SKIP-                          \ skip leading -
      OVER C@
      ASCII ) <>  IF  DDROP   EXIT THEN      \ no )  - not a function
      1+  SWAP    ( - Ø $beg+1  $end)
      DO  I C@   COUNT()    I C@  ASC>FP$  13 17  WITHIN
                             OVER Ø=   AND
         IF  NOT  LEAVE  THEN      ( -  -1)  \ not a function
      -1  +LOOP   ( - n)   Ø=   ;            \ not a function if not Ø

$" FIND "  $CONSTANT FIND$

: IN.LIBRARY?  ( $adr - cfa OR Ø)  FIND$  SWAP  $+   PLOAD   ;
```

```
$" FUNC{ "  $CONSTANT  FUNC{
$"  }TION "  $CONSTANT  }TION
: FUNCTION!    TOP.LINE   DUP  $BUF.Ø  $!   NOP$  SWAP  $!
    ?POP.NOPS
    $BUF.Ø  ASCII (  SLICE   1-  DSWAP     ( - $end2 $beg2  $end1 $beg1 )
    MAKE$  PAD  $BUF.1  $!       ( - $end2 $beg2 )
    $BUF.1   IN.LIBRARY?   ?DUP  Ø=
    IF  FUNC{ $BUF.1  $+   PAD  $BUF.1  $!   $BUF.1  }TION $+
        PAD  $PUSH   .NOP  O/S PUSH          \ not found, assume function
    ELSE  EXECUTE   THEN                     \ found, look-up in library
    PREDICATE   .NOP  O/S  PUSH  ;           \ clean up arguments

: DISSECT
         .NOP ASCII ,  BREAK.AT  DEBUG  NOT    \ parse function arguments
     IF  .F+  ASCII +  BREAK.AT  DEBUG  NOT
     IF  .F+  ASCII -  BREAK.AT  DEBUG  NOT
     IF  .F*  ASCII *  BREAK.AT  DEBUG  NOT
     IF  .F\  ASCII /  BREAK.AT  DEBUG  NOT
     IF  .F** ASCII ^  BREAK.AT  DEBUG  DROP
     THEN  THEN  THEN  THEN  THEN  ;

: SEND.FORTH    E/S POP    $.   O/S POP .FCODES ;

: PARSE  ( - )
    BEGIN    E/S TOS>  Ø>   WHILE
    TOP.LINE  FP#?      IF   CR  ." % "    SEND.FORTH    ELSE
    TOP.LINE  SIMPLE?   IF   LEADING-  CR  SEND.FORTH    ELSE
    TOP.LINE  HIDDEN?   IF   LEADING-   EXPOSE           ELSE
    TOP.LINE  FUNCTION? IF   LEADING-  FUNCTION!         ELSE
                             DISSECT   THEN  THEN  THEN  THEN
    REPEAT  ;

: >FTH   GET.EXP  EXPRESSION  FIRST.LINE  PARSE  ;
\ ----------------------------------------------- END PARSING WORDS ---------

\ ------ FUNCTION LIBRARY -------------------------------------------------

: FUNC   ( .op -)  CREATE  ,
    DOES>  @  NOP$  $.ENDS  PREDICATE    O/S  PUSH  ;

.FEXP   FUNC  EXP       .FSQRT  FUNC  SQRT       .FLN    FUNC  LOG
.FSIN   FUNC  SIN       .FCOS   FUNC  COS        .FTAN   FUNC  TAN
.FATAN  FUNC  ATN       .FASIN  FUNC  ASIN       .FACOS  FUNC  ACOS
.FSINH  FUNC  SINH      .FCOSH  FUNC  COSH       .FTANH  FUNC  TANH
.FASINH FUNC  ASINH     .FACOSH FUNC  ACOSH      .FATANH FUNC  ATANH

\ ------------------------------------ END FUNCTION LIBRARY ----------------

\ ================================================== END PROGRAM =============
```