and factors. However, to reduce the number of parentheses, FORTRAN adopted a heirarchy of operators that has been followed by all other languages that incorporate semi-algebraic replacement statements like the above. The heirarchy is

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

The translator must both enforce these rules and resolve ambiguities involving operators at the same heirarchical 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 will maintain this tradition.

A second issue is the function library. The FORmula TRANslator must recognize functions, and be able to determine whether a given function is in the standard library. In the example above, F" recognized EXP and SIN as standard library functions and emitted the FORTH code to invoke 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. To get around this we employ the generic data and operator conventions developed in Chapter 5 §1.

§§3 Parsing

Let us hand-parse the example, reproduced below:

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

Clearly, we must apply the first rule

```
\ <assignment>   -> <subject> = <expression>
```

embodied in the word **<assignment>**. We split at the " = " sign, and interpret the text to its left as a **SCALAR**. Since we want to emit the phrase A FS> 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 our parsing algorithm is a string stack. So we might imagine that after the first parsing step the string stack contains two strings

$STACK	Notes
A FS>	\ <subject>
-15.3E7*EXP(7/X) + Z/(W-SIN(THETA*PI/180)/4)	\ <expressio

Next we apply the rule

```
\ <expr'n>          -> <term> | < term> & <expr'n>
```

This breaks the top expression at the + sign between ") " and Z. We should think of the two terms

```
-15.3E7*EXP(7/X)
```

and

```
Z/(W-SIN(THETA*PI/180)/4)
```

as numbers on the ifstack; hence the code to evaluate each should be emitted before the addition operator (that is, these expressions are higher on the string stack than the addition operator **G+**). We adopt a rule that the right term is pushed before the left, so the $stack now looks like

$STACK	Notes
A FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4) G+	\ <term>
-15.3E7*EXP(7/X)	\ <term>

We now anticipate a new problem: suppose we have somehow – no need to worry about details yet – emitted the code for the <term> **-15.3E7*EXP(7/X)** on top of the $stack. Then we would have to parse the line **Z/(W-SIN(THETA*PI/180)/4) G+** . Assuming the program knows how to handle the first part, **Z/(W-SIN(THETA*PI/180)/4)**, how will it deal with the **G+** ? We do not want to use the space as a delimiter (an obvious out) because this will cause trouble with **A FS>** .

The difficulty came from placing **G+** on the same line as **-15.3E7*EXP(7/X)**. What if we had placed the operator on the line above, as in

$STACK	Notes
A FS>	\ <subject>
G+	\ operator
Z/(W-SIN(THETA*PI/180)/4)	\ <term>
-15.3E7*EXP(7/X)	\ <term>

Eventually we see this merely exchanges one problem for another of equal difficulty: How do we distinguish a <factor> or <term> that contains no more operators or functions – and is therefore ready to be emitted as code – from the operator **G+**, which contains a "+" sign? Now we need complex expression recognition, which will lead to a slow, complicated program.

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 issue has been overlooked. Here, we failed to distinguish FORTH words, **FS>** and **G+**, from FORTRAN expressions. 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[23]:

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 operator stack, which we call E/S and O/S for short. On two stacks,

E/S	O/S	Notes
A	FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G+	\ <term>
-15.3E7*EXP(7/X)	NOP	\ <term>

Why the **NOP**s ("no operation") on the O/S? We want to keep the stack levels the same (so we do not have to check when POPping off code strings); we thus have to put **NOP** on the O/S to balance a string on the E/S.

The TOS now contains a <term>, so we apply the rules

\ <function> -> <id> <arglist>
\ <term> -> <factor> | <factor> **%** <term>
\ <factor> -> <id> | <fp#> | (<expr'n>) | <func>

We note there is an operator at the " **%** " priority level (the "*" in the TOS). We split the top <term> at this point, issuing a **G***.

E/S	O/S	Notes
A	FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G+	\ <term>
EXP(7/X)	G*	\ <term>
-15.3E7	NOP	

23. Leo Brodie, *Thinking FORTH* (Prentice-Hall, Inc., NJ, 1984).

The parsing has now reached a turning point: the top string on the E/S can be reduced no further. The program must recognize this and emit the corresponding line of code (see Ch. 5):

```
% -15.3E7 REAL*4 F>FS
```

leaving

E/S	O/S	Notes
A	FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G+	\ <term>
NULL	G*	
EXP(7/X)	NOP	\ <function>

What is **NULL** and why have we pushed it onto the E/S? Simply, it is not yet time to emit the **G*** so we have to save it; however, we have another operator, **G+**, to associate with Z/(W-SIN(THETA*PI/180)/4) . Thus we have no choice but to define a placeholder for the E/S, analogous to **NOP** on the O/S.

TOS now contains a function. Assuming we can recognize it as such, we want to check that it is in the library and put the correct operator on the E/S. Thus we want to decompose to

E/S	O/S	Notes
A	FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G+	\ <term>
NULL	G*	
NULL	GEXP	\ <function>
(7/X)	NOP	\ <arglist>

The parentheses around the <arglist> on TOS serve no purpose, so drop them.

We see, once again, an operator of the priority-level **%** (the "/" between **7** and **X**), so we again apply the rule

\ <term> -> <factor> | <factor> **%** <term>

to obtain

E/S	O/S	Notes
A	FS >	\ < subject >
Z/(W-SIN(THETA*PI/180)/4)	G +	\ < term >
NULL	G*	
NULL	GEXP	
X	G/	\ < id >
7	NOP	\ < fp# >

Once again we can emit a number, so we do it:

```
% 7 REAL*8  F > FS
```

Wait! Why did we say REAL*4 with -15.3E7, but REAL*8 with 7 just now? Can't we make up our minds? The answer is that we want to respect precision over-rides *via* FORTRAN's E (single precision, so we say REAL*4) or D (double precision – REAL*8) exponent prefixes. However, where we are free to choose, it makes sense to keep maximum precision.

We continue, emitting the next simple items on the $stack:

```
X G/  GEXP  G*
```

leaving

E/S	O/S	Notes
A	FS >	\ < subject >
Z/(W-SIN(THETA*PI/180)/4)	G +	\ < term >

Once again we find the most exposed operator to be " / ", which we split with the rule

```
\ < term >   ->    < factor >  |  < factor >  %  < term >
```

E/S	O/S	Notes
A	FS >	\ < subject >
NULL	G +	\ < term >
(W-SIN(THETA*PI/180)/4)	G/	\ (< expr >)
Z	NOP	

Emit the TOS:

```
Z >FS
```

and apply the rule (first drop the parentheses)

```
\ <expr'n> ->    <term> | <term> & <expr'n>
```

E/S	O/S	Notes
A	FS>	\ <subject>
NULL	G+	\ <term>
NULL	G/	
-SIN(THETA*PI/180)/4	G+	
W	NOP	

Why did we issue **G+** and keep the leading "–" sign with **SIN**? 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

$$X - Y \equiv X + (-Y)$$

lets us issue only **G+** , 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 **%** ("treat the following number as floating point and put it on the 87stack") surely knows how to handle a unary "–" sign; whereas the second case will require us to issue a strategic **GNEGATE**.

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

$$-X^Y \stackrel{?}{=} -(X^Y) \text{ or } (-X)^Y$$

in favor of the former alternative. If the latter is intended, it must be specified with explicit parentheses.

After sending forth the phrase

```
W >FS
```

the leading "–" preceding **SIN(...)/4** must be dealt with. To preserve the proper ordering on emission we will want a word **LEADING–** that puts the token for **GNEGATE** on the O/S and moves the string **SIN(THETA*PI/180)/4** to the TOS, issuing a **NOP** on the E/S, obtaining

E/S	O/S	Notes
A	FS >	\ < subject >
NULL	G +	\ < term >
NULL	G/	
NULL	G +	
NULL	GNEGATE	
SIN(THETA*PI/180)/4	NOP	

The next exposed operator is at " **%** " -level. We apply **< term >** once more, to get:

E/S	O/S	Notes
...	...	\ ...
NULL	GNEGATE	
4	G/	
SIN(THETA*PI/180)	NOP	\ (< expr'n >)

After handling the function as before we find the successive stacks and FORTH code emissions

E/S	O/S	Notes
A	FS >	\ < subject >
NULL	G +	\ < term >
NULL	G/	
NULL	G +	
NULL	GNEGATE	
4	G/	
NULL	GSIN	
(THETA*PI/180)	NOP	

E/S	O/S	Notes
A	FS>	\ <subject>
NULL	G+	\ <term>
NULL	G/	
NULL	G+	
NULL	GNEGATE	
4	G/	
NULL	GSIN	
180	G/	
PI	G*	
THETA	NOP	

```
THETA  >FS PI  >FS  G*  % 180 REAL*8 F>FS
G/ GSIN  % 4 REAL*8 F>FS G/ GNEGATE G+
G/ G+  A FS> ok
```

§§4 Coding the FORmula TRANslator

We proceed in the usual bottom-up manner. The first question is how to define the $stack. In the interest of brevity, I chose not to push the actual strings on the E/S, but rather pointers to their beginnings and ends. By using a token to represent the operator, we can define a 6-byte wide stack which will point to the text of interest (which itself resides in a buffer), and will hold the token for the operator at the current level. This way only one stack is pushed or popped and the levels never get out of synchronization.

Again at the lowest level, we can develop the components that recognize patterns, *e.g.*, whether a piece of text is a floating point number. The word that does the latter is **fp#?**, already described in §2§§1.

A function is defined by the rule

```
\ <arglist>      -> ( <expr'n> { , <expr'n> } ^ )
\ <function>     -> <id>  <arglist>
```

We may therefore identify a function by splitting at the first left parenthesis,

```
: >(      ( $end $beg - - $end $beg')       \ find first "("
   1- BEGIN 1+ DUPC@ ASCII ( =  >R
        DDUP = R> OR UNTIL ;
```

and then applying appropriately defined FSMs to determine whether the pieces are as they should be.

```
: <function> ( $end $beg - - f )
   DUP>R  >(      ( - - $end $beg')
   UNDER 1- R> <id>
   -ROT SWAP <arglist> AND ;
```

The FSM **<arglist>** must be smart enough to exclude cases such as

```
SIN(A+B)/(C-D)
```

and

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

that is, compound expressions that might contain functions; it must also correctly recognize, *e.g.*,

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

as a function.

In FORTH there is no distinction between library functions and functions we define ourselves. In either case, the protocol defined in Chapter 8 will work fine. Thus the code generator for **<function>** emits the code

```
USE( fn.name arg1 arg2 ... argN  F(x)
```

What, however, do we do about translating standard FORTRAN names such as SIN, COS and EXP to their generic FORTH equivalents? The simplest method defines words with the names of the FORTRAN library functions. The FORTH-83 word **FIND** locates the code-field address of a name residing in a string. Thus we could have (**note**: **.GSIN** is a **CONSTANT** containing a token)

```
.GSIN   CONSTANT  SIN                    \ etc.
```

```
: LIBRARY? ( $end $beg - - cfa | 0 )
   -ROT UNDER - DUP>R
   PAD 1+ SWAP CMOVE R> PAD C! \ make $
   PAD FIND ( - - cfa n) 0= NOT AND ;
```

now we can define **function!** which, assuming pointers to the <id> and <arglist> are on the stack, rearranges the $stack like this:

E/S	O/S	Notes
...	...	...
(arg1, arg2, ..., argN)	.F(X)	\ an op.
NULL	.lib_name	\ if library function

or, if it is a user-defined function, like this:

E/S	O/S	Notes
...	...	...
(arg1, arg2, ..., argN)	.F(X)	\ an op.
name	NOP	\ user function

The code that does all this is

```
: function! ( $end2 $beg2 $end1 $beg1 - - )
   .F(X) $push                    \ push arglist
   DDUP LIBRARY? DUP 0=
   IF    DROP .NOP $push          \ user fn
   ELSE EXECUTE                   \ in lib.
        NULL ROT $PUSH DDROP
   THEN ." USE( " ;
```

For the program itself[24] we work from the last word, **<assignment>**, to the first (which we do not know the name of yet). We shall describe the program in pseudocode only, in the interest of saving space. Clearly,

```
: <assignment>          \ input $ assumed in buffer
   <subj> = <expr>      \ split at " = " ( - - f )
   IF subj! THEN        \ put subj and its code on $stack
   .NOP $push ;         \ then put expr on $stack
```

24. File FTRAN.FTH on the program diskette.

Certain decisions need to be made here: for example, do we want **F"** to be able to parse an expression that is *not* an assignment (that is, generating code which evaluates the expression and leaves the result on the ifstack)? We allowed this with the **IF...THEN**.

Next we pseudocode **< expression >**:

```
: < expression >  empty? IF EXIT THEN
   $pop
   < trm > & < expr >          ( - - f)  \ split at &
   IF trm&expr!  RECURSE
   ELSE  NULL  ROT  $push
         .NOP  $push  < term >
   THEN  ;
```

Defined recursively in this way, **< expression >** will keep working on the TOS until it has broken it up into term s.

We similarly define **< term >** recursively, so it will break up any term s into all their factors. It should also recognize < arglist > s. Thus:

```
: < term >  empty? IF EXIT THEN
   $pop  < arglist >
   IF arglist!  < expression >  EXIT THEN
   < factor > % < term >        ( - - f ) \ split at % = "*/"
   IF     fct%trm!  RECURSE
   ELSE  .NOP $push  < factor >    \ term = factor
   THEN  ;
```

And finally, we define **< factor >**, again recursively,

```
: < factor >  empty? IF EXIT THEN          \ done
   $POP   < fp# >                          \ fp#?
         IF  fp#!  RECURSE  EXIT THEN
   leading -?
         IF  leading-!  < expression >     \ forward ref.
         EXIT THEN
   < id > IF  id!         < expression >     \ forward ref.
         EXIT THEN
   < f > ^ < f >                           \ exponent?
         IF  f^f!  RECURSE  RECURSE
         EXIT THEN
\ cont'd ...
```

```
< function >
        IF  function!  < expression >       \ forward ref.
        EXIT  THEN
( < expression > )              \ expression inside ( )?
        IF  expose!  < expression >       \ forward ref.
        ELSE  ." Not a factor!" ABORT  THEN  ;
```

Note the forward references found in **< factor >**; since **< expression >** is defined later, we must use vectored execution or some similar method to permit this recursive call.

With this we conclude our discussion of rule-based programming. The complete code for the FORmula TRANslator is too lengthy to print, hence it will be found on the included diskette.