FUNCTIONALITY, THE PASCAL EXTENSIONS TO FORTH WHICH ARE THE SUBJECT OF THESE ARTICLES ARE TYPICALLY INTO THREE SECTIONS. THE FIRST TWO WERE COVERED IN PART ONE; THE LEXICAL ANALYSIS (FETCHING THE NEXT TOKEN FROM THE INPUT STREAM AND DECIDING WHAT IT IS), AND THE ARITHMETIC EXPRESSION TRANSLATION. HERE, THE THIRD SECTION OF STATEMENT TRANSLATION IS DESCRIBED. IT IS OFTEN THE CASE IN FORTH PROGRAMS THAT THE LATER SECTIONS MATCH EASIER THAN THE EARLIER ONES AND THIS IS CERTAINLY TRUE HERE. IN FACT, THE PASCAL CONTROL STRUCTURES ARE, WITH ONLY ONE EXCEPTION, ALMOST BARELY EXEMPLIFIED IN FORTH — THEREBY SUPPORTING THE CLAIM THAT FORTH IS A USEFUL LANGUAGE FOR IMPLEMENTING OTHER LANGUAGES (SOMETIMES REFERRED TO AS A "META-LANGUAGE"). IT IS MY HOPE THAT THIS SECTION WILL FURTHER THIS CLAIM BY PROVING OTHERS INTO IMPLEMENTING MORE SOPHISTICATED SCHEMES THAN THIS (NOT DIFFICULT, I SUSPECT).

**PASCAL STATEMENT SYNTAX**

In keeping with the technique established in part one, of taking the Pascal syntax graphs as the basis for the translation algorithm, I shall begin by setting out the statement syntax (as implemented here) in figs 1A and 1B. Eight different statements are covered: assignment, if, while, repeat, for, function call (which was described in part one) and begin (a begin statement is simply a collection of statements). Also a write statement is included. Strictly, this is a standard function call, but it is used so often to be worthy of special inclusion.

The syntax conforms to standard Pascal with only minor exceptions, notably the addition of an optional "by" clause in the for loop, to allow step values other than -1. (Standard Pascal limits for loop step values to 1 or -1.) Also the "string" printing facility of the write statement uses double quotes to delimit the string, rather than the usual Pascal single quotes. This is to enable the use of 'dot-quoted' ('') in the definition of write (see block 1989 in the listing at the end of this article).

The assignment statement, defined in block 2017, allows the use of single dimensional arrays - a facility already provided in the arithmetic expression translator (although I forgot to mention this in part one). These are integer arrays which must be created by CREATE and ALLOT before they are referenced in Pascal statements. Fig 1 of the last month's article and fig 3 here illustrate examples of this. Arrays should be used with a little care since there is no checking at array bounds at run-time (unless you do it yourself), but then Forth programmers are used to this sort of thing.

**CONTROL STRUCTURE MAGIC**

Writers of compilers and interpreters are often haunted by the problem of handling control structures, especially when translating fully structured languages such as Pascal. This was the case with me until I realized that the Pascal and Forth structures; while and repeat, are identical except that the Pascal structures are in infix and the Forth is Reverse Polish. An elementary observation? Certainly, but one that enables us to translate from Pascal into Forth with astonishing ease, with, of course, a little help from that magical 'forth' word [COLLAB].

Consider the 'while' loop. In Pascal the structure is:

```pascal
while <expr> do <statement>
```

In the same structure Forth is:

```forth
BEGIN <expression> WHILE <... FORTH words ..> REPEAT
```

So, we simply translate the Pascal word 'while' into the Forth word 'BEGIN', the Pascal 'do' into the Forth word 'WHILE' and terminate the compiled Forth version by REPEAT. All accomplished in the very simple definition outlined in fig 2, and listed in detail in block 2022. The definitions of 'if' and 'repeat' structures are equally straightforward.

The development of a 'for' loop poses a rather different problem because of the fundamental difference between the Pascal for loop and Forth DO loop. The Pascal loop uses a variable to hold the index value whereas the Forth loop uses the return stack. Thus making the index value easily accessible within a Pascal for loop would be difficult if the loop were based
Fig 1b: Detailed syntax diagrams for the various Pascal statements outlined in Fig 1a
upon the FORTH DO loop. It seems to me that the most elegant
solution to this problem is to construct a completely new FORTH
looping structure which more closely mirrors the Pascal for
loop and translate into this new structure — thus avoiding the
FORTH DO loop altogether. (Although I would welcome any alter-
native ideas on this.)

which means, not surprisingly, ‘: = expected here’. If the error
occurred while LOADING a disk block then ‘error’ will also
helpfully display the offending line of program before pointing to
the error. The following table lists all of the syntax errors which
can be detected by this compiler.

**Error number**  | **Meaning**
--- | ---
-1 | Variable expected
0 | Undeclared syntax error, possibly an undefined variable
1 | Statement cannot start like this
18 | ‘;’ expected
18 | Statement cannot end like this
21 | ‘;’ or ‘:end’ expected
23 | ‘:then’ expected
26 | ‘do;’ expected
26 | ‘until’ expected
35 | ‘:to’ expected
36 | ‘:;’ expected
39 | ‘:;’ expected

I should mention that a common cause of cryptic syntax errors is
forgetting to insert at least one space between each word in the
Pascal program. This is a bit of a serious limitation — but it did
simplify the ‘lexical’ analysis considerably.

**SOME PASCAL/FORTH EXAMPLES**

Only a single statement in the syntax graph sense is allowed
between ‘pascal end’, if multiple statements are required they
must be nested within ‘begin ... end’ or some other structure.
So, for example,

```
VAR ch
getch (pascal ch := KEY )
```

is correct, but to add another statement we need to use the
begin ... end structure.

```
readch (pascal begin
ch := KEY ;
EMIT (ch )
end );
```

These examples illustrate the way that FORTH words may be
called from Pascal as if they were standard functions or pro-
cedures. KEY requires no arguments but leaves a value on top of
the stack (the ASCII value of the key pressed) which is returned
correctly as the result of the expression and stored in the vari-
able ‘ch’. The FORTH word EMIT, on the other hand, needs one
argument (the ASCII value of the character to be printed) but
leaves nothing on the stack, hence it is called like a Pascal pro-
cedure. The distinction is based upon the stack offset of the
FORTH word. An example of a word with arguments and a result is

```
VAR a
( limit a to <= 100 )
: limits ( pascal a := MIN ( a 100 ) );
```

To enable Pascal arguments to pick up values from the stack we
can store them in variables before going into Pascal, e.g.

```
VAR tos and VAR n as
: test tos 1 not ( pascal ...)
```

but a nicer solution is to define a dummy FORTH word ‘STACK’
which has no effect except that of making Pascal think that an
expression is valid, for example.

```
: STACK ; VARIABLE i
```
PROGRAMMING PASCAL IN FORTH (PART 2)

( check that ASCII character on stack is a digit )
: digit? (pascal begin
: ch := STACK ;
: i := ( ch >= 48 ) and ( ch <= 57 )
end )
: i @ ; ( leave result on stack )

We might then use this function in a routine to insert
column numbers.

VARIABLE n VARIABLE neg
( ask for a number and leave it on the stack )
: readn (pascal begin
: n := 0 ;
: neg := 0 ;
: write ("?") ;
: readch ;
: if ch = 45 then begin neg := 1 ; readch end ;
: while digit? ch do begin
: n := n * 10 + ( ch - 48 ) ;
: readch ;
: end ;
: if neg then n := -n ;
: write (" ") ;
: n @ ;

An example of a program which makes use of 'readn' is the
following to test the Pascal 'for' loop:

VARIABLE Index
VARIABLE Start VARIABLE End VARIABLE Step
: testfor (pascal begin
: write ("Start") ;
: Start := readn ;
: write ("End") ;
: End := readn ;
: write ("Step") ;
: Step := readn ;
: for Index := Start to End by Step do
: write (Index)
end ) ;

These programs are included here primarily to demonstrate how
the Pascal extensions are used in practice, but they do serve
another purpose — as part of a compiler validation suite. This is
a set of programs with to the development of any new compiler
which is used to thoroughly check out the compiler at every
stage of development, and to ensure that no new modification
has any unexpected side effects.

ERATOSTHENES SIEVE

As a final example I present the Pascal/Forth version of a pro-
gram for generating prime numbers using the algorithm known
as Eratosthenes Sieve. This program has been adopted by Byte
as a benchmark for comparing different languages (see reference)
and is a useful test of speed as well as correctness.

Apart from the variable definitions which must be in Forth,
the Pascal/Forth version, listed in fig. 3, is virtually identical
to the pure Pascal equivalent.

The program does produce the correct answer of '1999 primes'
and runs in 29 seconds on my 4MHz Z80 system. The pure Forth and Basic (interpreted) versions of the same pro-
gram ran, on my system, in 8 seconds and 143 seconds respec-
tively. Why the difference between 2 seconds (Forth) and 29
seconds (Pascal/Forth)? Well, principally because the pure Forth
version of this program uses no variables at all, but uses the
stack instead — which is bound to be faster. Also the flags array
is cleared (see lines 0 and 1 of block 1003 above) in the Forth
version with the FILL operation. Another factor is that the new
FOR loop is about 6 times slower than the old DO loop (but it is a
much more powerful operator). In fact, taking all of this into
account, I'm really quite surprised that the Pascal/Forth version
runs as quickly as it does!

Sieve of Eratosthenes benchmark listing

```
( B6 0 Sieve of Eratosthenes, now BYT! Nov 1, 1982 )
1) BYT! COMPLXSIZE (n, m)
2) CREATE flags size 2^n NOT (array)
3)
4) ( create some variables... )
5) VARIABLE flag VARIABLE prime VARIABLE count
6) VARIABLE k VARIABLE j
7) k and useful constants...
8) 0 CONSTANT false 1 CONSTANT true
9) 0) count the number of primes between 3 and (1031)
10) primes (pascal)
11) begin
12) write (" * iteration only " )
13) count := 0 ;
14) count := 0 ;
15) ( prime counter )
```

Block 1003

9) for i := 0 to size do
10) flags [ i ] := true ;
11) if flags [ i ] then
12) begin
13) prime := i + 1 ;
14) write (" twice index ")
15) ( write prime )
16) if prime [ i ] then
17) flags [ i ] := prime ;
18) if prime is a multiple of k i
19) flags [ k * i ] := false ;
20) if flags [ i ] then
21) count := count + 1 ;
22) write (" total primes found ")
23) end
24) i := i + 1
25) end
26) count := count + 1 ;
27) write (" total primes found ")
28) end
29) i := i + 1
30) write (" total primes found ")
31) end
32) i := i + 1
33) write (" total primes found ")
34) end
35) i := i + 1
36) write (" total primes found ")
37) end
38) i := i + 1
39) write (" total primes found ")
40) end
41) i := i + 1
42) write (" total primes found ")
43) end
44) i := i + 1
45) write (" total primes found ")
46) end
47) i := i + 1
48) write (" total primes found ")
49) end
50) i := i + 1
51) write (" total primes found ")
52) end
53) i := i + 1
54) write (" total primes found ")
55) end
56) i := i + 1
57) write (" total primes found ")
58) end
59) i := i + 1
60) write (" total primes found ")
61) end
62) i := i + 1
63) write (" total primes found ")
64) end
65) i := i + 1
66) write (" total primes found ")
67) end
68) i := i + 1
69) write (" total primes found ")
70) end
71) i := i + 1
72) write (" total primes found ")
73) end
74) i := i + 1
75) write (" total primes found ")
76) end
77) i := i + 1
78) write (" total primes found ")
79) end
80) i := i + 1
81) write (" total primes found ")
82) end
83) i := i + 1
84) write (" total primes found ")
85) end
86) i := i + 1
87) write (" total primes found ")
88) end
89) i := i + 1
90) write (" total primes found ")
91) end
92) i := i + 1
93) write (" total primes found ")
94) end
95) i := i + 1
96) write (" total primes found ")
97) end
98) i := i + 1
99) write (" total primes found ")
100) i := i + 1
101) write (" total primes found ")
102) i := i + 1
103) write (" total primes found ")
104) i := i + 1
105) write (" total primes found ")
```

Block 1008

1) print natural expression, or string
2) print
3) IF
4) THEN
5) ELSE
6) print expression (as expression)
7) print
8) PRINT
9) PRINT
10) PRINT
11) PRINT
12) PRINT
13) PRINT
14) PRINT
15) PRINT
```

Block 1009

1) write statement
2) write
3) write
4) write
5) write
6) write
7) write
8) write
9) write
10) write
11) write
12) write
13) write
14) write
15) write
16) write

```
PROGRAMMING

PASCAL IN FORTH (PART 2)

Like so many programming exercises this one started out simply to see if it could be done. I can, of course, but whether the end result is useful or just of academic interest remains to be seen. I feel that I have presented here not so much a polished program, but more a demonstration of ideas and testing of techniques. There are many possibilities still to explore. One that I have looked at briefly is to redefine the sub-expression syntax to allow an entire statement to be written between the brackets, rather than just an expression. This gives an Algol-68 flavour to the language, allowing things like:

\[ A := B + ( \text{if } C <= 10 \text{ then } 1 \text{ else } 2 ) \]

Another interesting (but mind-boggling) problem is to extend the implementation of Pascal to allow the definition of data structures in Pascal rather than FORTH. The solution to this would involve, I suspect, devising a way of creating more than one dictionary entry simultaneously. Since the Pascal program under compilation is generating a new dictionary entry — the compiled version of itself — how do we make it generate other dictionary entries for the data structures at the same time? Food for thought I hope!

Reference