This is the first of two articles describing a set of Pascal-like extensions to Forth. The intention here is not for a full-blown Pascal compiler but to provide the Forth programmer with the option of writing those parts of a task which are best described with a Pascal-like syntax, in Forth. The philosophy behind this idea is in the widely held view that, while most languages are good for some things low, i.e. good at everything and instead of trying to design new all-embracing wonder-languages it might be more productive to develop multilingual programming environments. Whether you hold this view or not, the extension of Forth into Pascal is still an interesting exercise and a worthy test of power and flexibility of Forth.

Before going any further, perhaps I should give an example of the way I see this amalgamation of Forth and Pascal working in practice. Suppose that we need to sort a list of numbers into ascending order. A standard algorithm for doing this might be expressed very clearly in a Pascal/Forth hybrid as shown in Fig 1.

**BASIC IDEAS**

Initially one of the principal requirements was for complete interchangeability between Forth and Pascal, so that forth routines may call previously defined words written in Forth and vice versa. It is particularly useful for Pascal to be able to make use of the large set of standard Forth words. As an example, suppose we need to print a number, right justified, from within a Pascal routine. A convenient way would be to call the forth primitive 'R' as if it were a Pascal procedure, i.e.

```
R | i, 5
```

to print the contents of variable i, in a field width of 5. This should work very well if we can arrange that the expressions enclosed by brackets leave their results on the stack at run time, which is precisely where the Forth word 'R' will expect to find its input parameters.

Passing results back from Pascal defined words into Forth is not quite so straightforward, since a Pascal program cannot easily take values off (or leave them on) the stack. Pascal does all of its arithmetic with variables, so, it would seem sensible to allow Pascal and Forth to share the same variables, and communicate results through them. In Fig 1 the array list was created in Forth, but manipulated in 'sort-list' exactly as if it were a 1-dimensional Pascal ARRAY. A Forth definition to sort any list whose address is supplied on the stack might then be written as in Fig 2.

Fig 3 summarises the structure of a color definition incorporating both Forth and Pascal. The Pascal statements are enclosed by '{pascal} and '}' (the word 'pascal' is the new 'compiling' word which is really the subject of these articles). There may be any number of segments of Forth and Pascal in the same definition, although more than one or two might be confusing.

Possibly the most demanding design constraint is that the compiled Pascal should be as close as possible to the equivalent compiled Forth, so that there is little or no compromise on execution speed for routines written in Pascal. Thus, as an example, the Pascal statement:

```
F = 2 THEN i = j+k;
```

would be translated in Forth as

```
: F ( i j k )
  LITERAL 2 2eon
  OVER DUP ADD

  \ Sort any 100 element list into ascending order

  \ sortlist ( copy anylist into list )
  \ sortlist ( sort it )
  \ list SWAP 100 MOVE \ Copy back into anylist )

  \ Generate two lists and sort them ...
  CREATE list1 200 ALLOT list1 sort
  CREATE list2 200 ALLOT list2 sort

  \ Fig 2. A Forth definition to sort any list.

  \ new-word ... some FORTH words ...
  \ (pascal) ... some Pascal statements ...
  \ ... more FORTH words ...
  \ etc ...

  \ Fig 3. A Hybrid Forth/Pascal definition.
```
should, ideally, compile into the same internal form as the
Forth phase:

\[ a \cdot 2 \equiv F; @ k \equiv + 1 \text{ THEN} \]

Of course it is unlikely that a program written entirely in Forth
would make such a heavy use of variables as this, but the ability
to mix Pascal and Forth (and Assembler, on many forth systems)
should allow plenty of opportunity for optimizing any time-
critical part of a program.

THE ALGORITHM

Readers of the two excellent articles in issue 1 of SDT, 'Writing
Compilers in Basic' and 'Infix Maths in Forth' will already be
awake to the power and versatility of the technique known as
'recursive descent' and so take no umbrage for using this here.
In fact, structured languages like Pascal tend themselves par-

ticularly well to compilation by recursive descent, because their
syntax is often highly recursive — that is, defined in terms of
itself. An example of this in Pascal is that a 'statement' amongst

other things might consist of the reserved word 'BEGIN',
followed by any number of statements separated by 'END', and
terminated by 'END'. Thus, when the statement compiler encoun-
ters the word BEGIN it will simply call itself.

This structure is difficult to describe in words, but very much
easier to see when expressed as a 'syntax graph'. For example
fig 4 illustrates the syntax graph for 'begin-statement'.

The principle function of the syntax graph is to specify exactly
the syntax of a programming language. That is its primary use
in 'parsers' of the language. However, for compiler writers, syntax
graphs form a basis that might be directly related to a compilation algorithm.

THE FORTH BASICS

The total program splits very conveniently into two parts; the
arithmetic expression compiler, and the statement compiler, so
I shall describe the former here and the latter in the second part
of this article. Although I am at risk of re-inventing some wheels
here (see the second referenced) it does turn out that the
expression compiler is useful in its own right as an Infix
arithmetic compiler/interpreter, as I will show later.

First, however, we must establish the basic routines for
parsing the input stream and identifying Pascal reserved words.
Given the design constraints outlined earlier, we can see that a
Pascal program will contain five basic word types:

- Pascal reserved words
- Numbers
- References to Pascal variables
- References to Forth constants
- Other words to be treated as Forth functions or procedures

What we require is a basic routine to 'fetch the next thing from
the input stream' and classify it as one of these five basic types.

Since the final three types variable, constant or function, will all
be words in the dictionary when the Pascal program is com-
iled, it is clear that a dictionary search will be needed some-
where in this routine. Why not, therefore, create a special
vocabulary containing the Pascal reserved words and let the
Forth dictionary search word FIND do all the work for us.

When FIND is executed it will fetch the next word from the
input stream (delimited by 'space'), and search the CONTEXT
vocabulary for a matching word. If it successfully returns the
(unique) address of the dictionary entry for that word, or the
value zero if not. (This is the Code Field Address or CFA. All that

...
we need to do is in our basic parsing routine (which I call "next"), is to arrange that the Pascal reserved word library be CONS- TEXT when FIND is executed. This will ensure that it is searched before the rest of the forth dictionary, avoiding any conflict between similar Pascal and FORTH words. We can then use the CIA to classify the word into one of the five types above.

All of this happens in blocks 201-206; lines 8-30. The last of the line is the article. The FORTH reserved words are a set of empty definitions in the vocabulary 'andalso-clauses', in block 201-206. The classification of reserved words is achieved by attempting to match the FORTH word returned by FIND with one of the entries in a special table containing the reserved word FORTHs ("faa_table", block 202). If a match is found then 'next' (blocks 205-206) compares the FORTH with 'code' and 'code'; the FORTH for variable and constant, respectively. No further matching is necessary, since if all these tests fail the word must have been either a number, or a forth function or procedure call, and a value of zero returned by FORTH eight, of course, indicate a number (or a syntax error).

The final result of 'next' is returned in the variable 'type'; values of 1-3 indicate Pascal reserved words, 0-2, constants, 1-variables, 0-numbers, or any other value for references to forth words (defined by colon definitions). Thus, after loading blocks 201-206 into a forth system, we may test these by typing:

next type ? 1 ok (Pascal 'and')
next begin type ? 20 ok (and 'begin')
next ? type ? 0 ok (a number)

VARIABLES: 1-CONSTANT true
next true type ? -1 ok (true)
next true type ? -2 ok (true)
next true type ? 12524 ok (false)

Although it may come as a surprise, the development of the word 'next' was one of the most difficult problems of this entire application. Everything else falls into place with remarkable ease.

THE EXPRESSION/COMPILER

Blocks 209-2015 contain the arithmetic expression compiler, in usual forth fashion, with the lowest level definitions: 'number', 'variable' etc. and working up the highest level: 'expression' in block 2015. These closely follow, both in name and structure, the corresponding syntax graphs of fig 6A. A reasonably faithful subset of standard Pascal arithmetic syntax, as described in the third reference.

Now, while I do not propose to examine in detail each of the colon definitions in these blocks, it is important at this stage to establish what the 'output' of the compiler should be, and how to achieve it. If we consider as an example the simple arithmetic expression:

A * 2

the compiler should translate this into the equivalent forth expression:

A @ 2 *

We notice straight away that when the compiler comes across a variable it should generate the code to 'push the value of the variable onto the stack' and, likewise, when a number appears in the input expression, the compiler should generate the code to 'push the number'. This is precisely what 'variable' and 'number' do; if 'variable' is complicated a little by single...
sional array handling as well. Providing that A is a predefined forth variable, then A and 2 are both syntactically correct factors (see fig 5), and tracing a route through the syntax graphs, starting at 'expression', will eventually arrive at the starred position in the graph for 'term'. At this point we know that we must have had two valid factors, separated by **i otherwise this route would not have been followed, and the compiler may output the code for **i.

There is really very little structural difference between a compiler and an interpreter. The principal distinction is that an interpreter actually performs the operation indicated by the input expression during translation, whereas a compiler instead generates some code that will perform the same operation later, at run time. Combining this observation (which was a revelation to me when I first realized it) with the forth convention that anything inside a code definition is compiled, and anything else interpreted leads to the possibility of making the compiler into an interpreter as well. To make this happen we need only write, for example:

```forth
STATE @ IF
   ( if we are inside a code definition )
   COMPILE * ( ... then compile **i )
ELSE
   ( else we are not compiling )
   * ( ... do it now )
THEN
```

in the 'term' routine, after having picked up two 'factors' separated by **i. Since this is rather long winded I have defined a new compiling word 'omp/int' in block 20008, which allows us to write 'omp/int **i' with exactly the same effect.

**INFIX EXPRESSIONS**

An unexpected and useful byproduct of this quest for Pascal in Forth is that the arithmetic expression compiler/interpreter may be used as a stand-alone utility enabling us to write arithmetic expressions — in Forth — in infix notation rather than the usual Reverse Polish. To do this and I have defined the IMMEDIATE word **i, in block 20016, which simply calls 'expression'. New expression' will exit when a word is found that is recognized but is not syntactically correct (and all of the CASE comparisons fail, thus) is a suitable terminator. So, to write an infix expression, just place it between curly brackets, as in the following examples:

```
VARIABLE A ok
VARIABLE B ok
10 A! 20 B! ok
(A + 2 * B). 50 ok
((A + 2) * B). 240 ok

: testA<3 ( A < B ) IF "yes" THEN ; ok
  testA<3 ok

: formula ( A * A + B * B ). = ; ok
  formula = 500 ok
```

Note that the result of evaluating an infix expression remains on the stack where it may be used by subsequent forth operation in the normal way.

References: Pascal In Forth
Perr M, 'Writing your own Compiler', SOFT, June 1993.