An infix syntax for Forth
(and its compiler)

Andrew Haley
Early Inspiration


- Very elegant, but closer to Pascal than to Forth – the resulting syntax is more restricted, and the control structures are those of Pascal, not Forth. Also, restricted to single-length integer expressions and arrays, no structures, etc, etc.
Previous efforts

- Forthwrite Dec ’86:

```forth
VARIABLE 'EXPRESSION : EXPRESSION 'EXPRESSION @EXECUTE ;
VARIABLE TEMP CREATE )
: ,C ( a) 2- , ;
: NEXT ( - a) ~' IF NUMBER TEMP ! 0 ELSE DROP THEN ;
: CHECK ( a a') - ABORT" not matched" ;

: FACTOR ( a - a') DUP ['] ( = IF DROP NEXT EXPRESSION ['] ) CHECK ELSE ?DUP IF ,C
ELSE TEMP @ [COMPILE] LITERAL THEN THEN NEXT ;
: TERM ( a - a') FACTOR BEGIN DUP ['] * = OVER ['] /=
OR WHILE NEXT FACTOR ,C REPEAT ;
: EXPRESSION ( a - a') TERM BEGIN DUP ['] + = OVER ['] -=
OR WHILE NEXT TERM SWAP ,C REPEAT ;

: INFIX  NEXT ['] ( CHECK NEXT EXPRESSION ['] ) CHECK ;
  IMMEDIATE ' EXPRESSION 'EXPRESSION !

Example of use:

44 CONSTANT FRED
: TEST ( -- n ) INFIX ( 3 * FRED / ( ( 3 + 5 ) / 2 ) ) ;
```
Previous efforts

- Forthwrite Dec ’86:
  - Uses recursive descent
  - Compile only – no use in interpreter
  - No LOCAL variables
  - Extremely simple
  - Only arithmetic expressions
  - Uses data stack
  - Uses – ’ (aka FIND) and , C (aka COMPILE,)
Previous efforts

comp.lang.forth Feb 2002, some details elided:

```
: op ( a) state @ if compile, else execute then ;
: lit  =number @ state @ if postpone literal then ;

ops[ relop > > < < = = ]
ops[ addop + + - - or or xor xor ]
ops[ mulop * * / / and and ]
ops[ unop - negate @ @ ]

\ These are the productions.

defer expr
: expr-list expr begin match , while token expr repeat ;

: parens expr-list match ) 0= abort" )" ;

: primary
    match# if lit token exit then
    match ( if token parens token exit then
        this >r token match ( if token parens token then r> op ;

: factor unop if >r token recurse r> op exit then primary ;

: term factor
    begin mulop while >r token factor r> op repeat ;

: simple-expr term
    begin addop while >r token term r> op repeat ;

: noname simple-expr
    begin relop while >r token simple-expr r> op repeat ;

is expr
```
Previous efforts

- comp.lang.forth Feb 2002:
  - Uses recursive descent
  - STATE-smart: allows interpretive use
  - Still extremely simple
  - Function calls: `FOO ( 1, BAR, 3 )`
  - Uses return stack for temporary storage of execution tokens that haven’t yet been used because they are of low precedence – much cleaner; means we can use data stack for interpretive expression evaluation
  - Written in almost Standard Forth
  - Still doesn’t allow LOCAL variables in expressions
The problem with locals

“Words that return execution tokens, such as ' (tick), [ ' ], or FIND, shall not be used with local names.”

This is a horrible restriction! Effectively it means that locals can never be used as factors. Locals cannot be used as part of an expression in this parser because it uses ' and COMPILE,
Designing the syntax

- Let’s ignore the implementation problems for a little while and look at the syntax we’d like to have. We’ll return to the implementation later.
Designing the syntax

- A word is any string of non-whitespace characters. Words are separated by spaces.
- Numbers are just words, so they don’t need to be treated specially. The syntax need make no special provision for them.
Designing the syntax

- Simple cases:
  - Basic Forth syntax is
    noun noun ... verb noun noun ... verb
    profanely,
    verb ( noun , noun , ... ) ; verb ( noun , noun , ... ) ;

- Control structures:
  - a b > if    becomes    if ( a > b )
  - 10 0 do    becomes    do ( 10 , 0 )
Designing the syntax

- More simple cases:
  - Arithmetic expressions:
    - Traditional operator precedence, defined by syntax
      
      \[
      b \negate b b \times 4 a \times c \times - \sqrt{2} / a \times +
      \]

      becomes

      \[
      - b + ( \sqrt{b \times b - 4 \times a \times c} / 2 \times a )
      \]

      The reserved tokens are

      + - \times / f+ f- f\times f/ ( ) < > = f< f> f= \text{or xor and @}

      Everything else is just a word, and can be used as a function or an argument.
Designing the syntax

- To allow multiple statements, we add the `;` operator:
  
  ```forth
  expr ; expr
  ```

- Local variables can be assigned with the `:=` operator:
  
  ```forth
  a b * to c becomes c := a * b
  ```

- `@` is a problem. We could just treat it as a function like any other Forth word, but then it would be cumbersome to use because of parentheses:
  
  ```forth
  @ ( a ) + @ ( b ) ...
  ```

  so we define `@` to be a high-precedence unary operator, which is much nicer:
  
  ```forth
  @ a + @ b ...
  ```

- We could arguably do the same with `!`, treating it as a binary operator
Designing the syntax

- A structure access, as per the Forth 200x structures RFD, is just the application of a function to a pointer.
  
  Given a struct, we can use its fields with no special treatment:

  ```forth
  struct point
    1 cells +field p.x
    1 cells +field p.y
  end-struct

  \ Draw a line from p1 to p2
  draw ( p.x ( p1 ) , p.y ( p1 ) , p.x ( p2 ) , p.y ( p2 ) );
  ```

- We could define a word \ as a postfix function operator, but that isn’t obviously a big improvement.
Designing the syntax

Because every statement is also an expression, we can have conditionals in expressions, so:

\[ a := b + ( \text{if} (c < 10) ; 1 ; \text{else} ; 2 ) \]

is equivalent to

\[ b \ c \ 10 < \text{if} \ 1 \ \text{else} \ 2 + \ \text{to} \ a \]
I'm still not certain about the absolute best syntax for arrays, but Smalltalk is a good place to start.

For array reads,
- `a at: i` produces `a i at:`

And for writes,
- `<expression> put: ( b , 2 )` produces `b 2 put:`
- `(Maybe b at: 2 put: <expression> would be better)`

With an additional shorthand (purely for familiarity's sake):
- `a [ i ]` is equivalent to `a at: i`
Designing the syntax

- Arrays are tricky. In profane languages *lvalues* are treated differently from *rvalues*: an lvalue is evaluated for its address, but an rvalue is evaluated for its value.

- For example,
  \[ a[i] := b[j] \]

- We can’t simply say that every array access on the LHS of an assignment is evaluated for its address, because of things like
  \[ a[b[i]] := b[j] \]
  where only the *outermost* array access is evaluated for its address.

- It’s difficult to do a mapping in a purely syntactical way. If we’re simply scanning from left to right we have no way to know that an assignment is imminent; that would require *backtracking*. 
Designing the syntax

- Parsing words are the biggest headache. Anything that acts as a prefix operator by using `PARSE` or `WORD` needs special treatment.

- String constants are easy enough, though:
  
  ```
  s" hello " type
  ```

  maps easily to
  
  ```
  type ( " hello " )
  ```

- I don’t think the lack of `. "` is important.
Escape to Forth

- If all else fails and there really is a Forth expression that cannot be rendered as infix in any way, there’s an escape:
  
  \[
  \text{[ ." Hello, world" ]}
  \]

- This also allows local declarations, etc:

  \[
  \text{[ LOCALS| a b c | ]}
  \]
The problem with TO

“An ambiguous condition exists if either POSTPONE or [COMPILE] is applied to TO.”

So TO can never be used as a factor either.

This is a very bad design decision: if Forth is about any single thing it’s factoring, and this is an important part of the language that forbids factoring.
Implementation

- The problem with TO not being allowed to be ticked or POSTPONED was, as it turned out, a big inspiration.
- We can’t use XTs, but we can use strings. So, instead of saving XTs on the return stack, we create a string stack and define $S$ and $S>$. Also, we create an output buffer and push into it words from the string stack.
- At any stage in the compilation, we only have to decide whether to push a word into the output buffer or onto the string stack.
Implementation

Source:
\[-b + \sqrt{b^2 - 4ac}\]

String stack: Output:

\[
\begin{array}{c}
- \\
\text{sqrt} \\
+
\end{array}
\]

\[
\ldots \ b \ \text{negate} \ b \ b \ *
\]
Implementation

A great benefit – arguably the greatest benefit – of doing this by using strings rather than XTs is that we no longer need to be STATE-smart. The infix code is rewritten to be postfix and then passed to INTERPRET. INTERPRET either compiles or interprets.
An example

Original FORTRAN:

do i = 1, dim1
  do j = 1, dim3
    C(i, j) = 0.
    do k = 1, dim2
      C(i, j) = C(i, j) + A(i, k)*B(k, j)
    enddo
  enddo
endo
An example

Infix Forth:

```
do ( dim1 , 1 )
  do ( dim3 , 1 )
    0.e0 put: C ( j , i )
  do ( dim2 , 1 )
    C [ k , j ] f+ A [ k , i ] f* B [ i , j ] put: C ( k , j )
  loop
loop
```

Loop

generates

```
dim1 1 do
  dim3 1 do
    0.e0 j i C put:
    dim2 1 do
      C k j at: A k i at: B i j at: * + k j C put:
    loop
  loop
loop
```
In summary

- Infix Forth is not a translator from some other language to Forth, but an infix form of the language that doesn’t change its semantics.
- Most Forth words can still be used and keep their glossary definitions.
- If we’re going to translate from FORTRAN, C, etc, to Forth for a standard algorithms library, this is a much better way to do it than translating from infix to postfix by hand. It’s easier to do and easier to check.