4 Expression Evaluation

One of the major sections of the BASIC interpreter is the expression evaluator. Virtually every statement uses it to get the number or numbers that it is going to work with. For example the ‘HIMEM’ statement uses it to find the new value that HIMEM is to be set to.

4.1 Operator precedence

When expressions are to be evaluated, some operators take precedence over others. For example, multiplication is always done before addition, unless the addition is surrounded by brackets. This makes expression evaluation somewhat more complex than it would otherwise be, as you can’t just scan along the line, doing every operation as you come across it.

In fact, many old electronic calculators *did* just scan along the line like this. If you pressed:

```
2 + 3 * 5 =
```

you would get the answer ‘25’. This is not particularly satisfactory for an expression evaluator in BASIC, because if ‘2 + 3 * 5’ appears as an expression, it is assumed that the multiplication will be done first, giving the answer ‘17’ . Somehow, BASIC must identify that the addition must be done after the multiplication, save the ‘2’ while the ‘3’ and ‘5’ are being multiplied together, and then add the ‘2’ on afterwards.

4.2 Top-down analysis

To get these operator priorities right, BASIC uses a method called *top-down analysis*, where the expression evaluation is divided up into several levels. The top levels deal with the low priority operators, and these call the bottom levels (which deal with the high priority operators) for the items to operate on. This means that the high priority operations will be performed first, by the bottom levels of the expression evaluator, before the results of those operations are passed back to the top levels, for the low priority operations to be performed.

Taking the example of ‘2 + 3 * 5’ again, the top level would deal with the addition, and call the bottom level to get the values for it to add. The bottom level would deal with the multiplication, before passing the result back to the top level.

If we call the top level **`<expression>`**, and the bottom level **`<term>`**, we can see how this would operate:

1 **`<expression>`** calls **`<term>`** to get the first item to operate on.

2 **`<term>`** gets the number ‘2’ from the line.

3 There is not a ‘*’ or a ‘/’ after the ‘2’, so <term> passes ‘2’ up to **`<expression>`**.

4 **`<expression>`** finds that there is a ‘+’ after the item that **`<term>`** had evaluated, so it saves the ‘2’ and calls **`<term>`** again to get the item to add to it.

5 **`<term>`** gets the number ‘3’ from the line.

6 There is a ‘*’ following the ‘3’, so **`<term>`** saves the ‘3’ and gets the number ‘5’ from the line.

7 The ‘5’ is multiplied by the saved ‘3’, to give the result ‘15’

8 There is not a ‘*’ or a ‘/’ after the last number just read (the ‘5’), so **`<term>`** passes the ‘15’ up to **`<expression>`**.

9 <expression> retrieves the ‘2’ that it had saved at stage 4, and adds it to the ‘15’ passed up from **`<term>`**, giving the result ‘17’.

10 There is not a ‘+’ or a ‘−’ after the item that **`<term>`** had evaluated (the ‘3*5’), so it passes the ‘17’ up as the result of the **`<expression>`**.

The levels in this simple expression evaluator can be expressed using *Backus-Naur Form*, or BNF (see appendix A). It is expressed as follows:

```
<expression> ::= <term> {+|- <term>}
<term> ::= <number> {*|/ <number>}
```

`::=`	means ‘is defined as’
`{}`	surround items which can appear zero or more times
`|`	separates alternatives

So an `<expression>` can consist of just a `<term>` or any number of `<term>`s with each one separated by a ‘+’ or a ‘−’. Similarly a `<term>` can be just a `<number>`, or it can be any number of `<number>`s with each one separated by a ‘*’ or a ‘/’ .

In the example ‘2 + 3 * 5’:

the `<expression>` is ‘2 + 3 * 5’

the first `<term>` is ‘2’
the second `<term>` is ‘3 * 5’

The BASIC program below shows a simple expression evaluator with the `<expression>`, `<term>`, and `<number>` levels .

FNexpr evaluates an `<expression>`, calling FNterm to get the `<term>`, and FNnumber is used to get the `<number>` . Spaces are not allowed in expressions evaluated by this program.

The program uses *one character look-ahead*, where the next character is always kept in the variable ‘char$’. This allows the character not recognised by **FNterm**, say, to be passed to **FNexpr** in case it was a ‘+’ or a ‘− ’. If this were not done, `<expression>` would have to re-read the character from the line, before testing it for one of its operators. If a character is recognised, the next one must be read into char$ before another routine is called (for example, on line 1030).

```
  5 REM Simple expression evatuator to demonstrate the
 10 REM "top-down" method of expression analysis
 15 REM (spaces not allowed in expressions)
 20 REM
 90 REM *** Main loop ***
100 REPEAT
110   INPUT"EXPRESSION :"line$
120   lptr = 1
130   PRINT"VALUE IS   :";FNexpr
140   UNTIL FALSE
```

```
 990
1000 DEF FNexpr       :REM Get <expression> from line
1005 PROCgetchar      :REM Get char into char$
1010 value = FNterm   :REM Call <term> to get first item
1015 REPEAT
1030   IF char$="+" THEN PROCgetchar:value =value+FNterm
1040   IF char$="-" THEN PROCgetchar:value =value-FNterm
1045   UNTIL char$<>"+" AND char$<>"-"
1050 =value           :REM Final result
2000 DEF FNterm       :REM Get <term> from line
2010 value = FNnumber :REM Call <number> to get first item
2025 REPEAT
2030   IF char$="*" THEN PROCgetchar:value =value*FNnumber
2040   IF char$="/" THEN PROCgetchar:value =value/FNnumber
2042   UNTIL char$<>"*" AND char$<>"|"
2050 =value           :REM Result of <term>
3000 DEF FNnumber     :REM Read in <number> from line
3020 IF char$>"9" OR char$<"0" PRINT "NO NUMBER":STOP
3035 number = 0
3040 REPEAT
3050   digit = ASC(char$)-&30
3060   number = number*10 + digit
3070   PROCgetchar
3090   UNTIL char$>"9" OR char$<"0"
3100 = number         :REM Value of <number>
4000 DEF PROCgetchar  :REM Get character from line
4030 char$ = MID$(line$,lptr,1)
4040 lptr = lptr+1
4060 ENDPROC
```

The expression evaluator in BASIC has eight levels, rather than Just the 2 in the simple model. The levels, and their associated operators, are as follows (lowest priority at the top): Operators

Level	Operators
`<testable-condition>`	`OR, EOR`
`<logical-expression>`	`AND`
`<relnl-expression>`	`=, <, <=, <>, >, >=`
`<expression>`	`+, -`
`<term>`	`*, /, MOD, DIV`
`<sub-term>`	`^`
`<factor>`	`+, - (unary operatoes)`
`<primitive>`	

Note that **`<testable-condition>`** is the same as **`<numeric>`** (see chapter 33 of the BBC *User Guide*, or chapter 25 of the *Electron User Guide*). Numbers, functions and variables appear at the **`<primitive>`** level. A **`<primitive>`** could also be a **`<testable-condition>`** in brackets, causing the expression evaluator to *recurse* down from the top level again. For a more complete definition of the expression evaluator, and the rest of BASIC, see appendix A.

Most functions enter the expression evaluator at the **`<factor>`** level rather than at the top; this means that variables or numbers can be given to a function without brackets, but an **`<expression>`** must be included in (round) brackets. So, for example, the expression ‘SIN2+5’ will be evaluated as ‘(SIN2)+5’.

When finished, each level of the expression evaluator leaves its result in IntA, FPA, or StrA (depending on the type), with the type in the 6502 accumulator. The type bytes are:

&00	real (floating point) number
&40	integer
&FF	string

Note that these are not the same as the variable types described in section 3.1.

Each level can check this type byte returned to it by a lower level, and do any conversions necessary (or generate an error if a type mismatch has occurred). The particular ROM routines in section 10.4 give more details of the use of these type numbers.

No check is made to see if the expression evaluator is running out of 6502 stack (due to all the subroutines it is calling). This means, for example, that if more that 17 levels of nested brackets are used, the stack will overflow, and the expression will not be evaluated properly (it may even generate an obscure error). In practice, this number of brackets is hardly ever used, so the problem never arises.