# ADVANCED BASIC

## Part 4

## Interpreters, compilers and assemblers

**Phil Cohen**

Following its series on Back Door into BASIC, ETI has begun a series on Advanced BASIC for those who want to expand their knowledge further. Phil Cohen has already examined 'Sorting', 'Chaining' and 'Top-down programming' in earlier parts of the series, and this month his subject is 'Interpreters, compilers and assemblers'.

A FEW MONTHS ago I attended a meeting of an amateur computer society (which shall remain nameless) and heard the speaker (who shall also remain nameless) asked the question, "What's the difference between an interpreter and a compiler?". He thought for a few moments and then said, "Well, a compiler is non-interactive, isn't it?" (i.e: you can't enter data while the program is running).

This is of course totally wrong. The reason *why* it's wrong is, however, interesting. Compilers just aren't found in home systems yet. The reason for this is that they tend to need more RAM and are more difficult to design than interpreters. Thus they are usually found in mainframe systems only. Why, if they're so much trouble, should professional programmers use them? The answer is that they are extremely powerful — their advantages are well worth the extra RAM.

In order to describe what exactly a compiler (and, while we're at it, an assembler) is, we have to define a few terms:

SOURCE CODE: The program in BASIC (or whatever) as it is typed in, spaces and all.

OP CODES: Or 'operator codes' — a set of numbers (usually one byte each) which represent BASIC keywords. In the Commodore PET, the source code is translated into op codes as it is fed in (try putting graphics characters in a REM statement).

MACHINE CODE: Well, anyone who started reading this article should know what this is, but just in case — it's a set of numbers, each of which represents an instruction to the processor itself.

ASSEMBLY CODE: A set of mnemonics (usually three or four characters each) which are easier to remember than machine code, but mean the same — there is a 'one-to-one correspondence' between machine code and assembly code.

INTERMEDIATE CODE: Or, in the case of PASCAL, 'P-code'. This is a set of instructions to an imaginary processor. This is more useful than it sounds, as will be explained later.

INTERPRETER: A program which pretends to be a processor which will accept source code or op codes as instructions.

ASSEMBLER: A program which translates assembly code into machine code.

COMPILER: A program which translates source code into machine code. Some compilers translate source code into intermediate code, which is then run on an interpreter which pretends to be the intermediate code's processor. This is a compromise between a compiler and an interpreter.

EDITOR: A program which allows source code or assembly code to be input, changed, listed and generally messed about.

ASSEMBLER/EDITOR: A package containing one of each.

### Assemblers

What is required of an assembler? While on the one hand some very primitive ones just take in one assembly code mnemonic at a time and output one (or more) bytes of machine code, most assemblers do a lot more. Let's take the example of the following program segment:

```
ADDRESS
1          LDA 8
3          DEC
4          JNZ 3
```

The address is the position in RAM of the first byte of each instruction. LDA causes the next byte of code to be loaded into the accumulator. As the assembler knows that this is a two-byte instruction, it will look for a parameter for LDA — in this case 8. DEC decrements the accumulator. JNZ jumps to byte 3 if the accumulator is not zero. Fine. What if we want to change it to:

```
IN 1
INC
JNZ
DEC
```

In other words, input the accumulator value from port 1, add 1 to it, then loop as before.

How do we know the parameter to use for JNZ? — in a long program we won't know the address of DEC until it is assembled. We can get the assembler to save us the trouble of sorting this out by hand by adding a 'label' facility:

```
    IN 1
    INC
```

```
A: DEC
   JNZ A
```

The assembler will remember the address of point A and use it as the parameter of JNZ.

What if we want to write:

```
   IN 1
   INC
A: JNZ B
   DEC
   JMP A
B: RTS
```

One of the labels points further down the program. At point A the assembler won't know what the address of point B is. A decent assembler will solve this by using a 'multi-pass' technique. That is, it will go through the program more than once. The first time it will find the addresses of all the labels and the second time it will actually produce the machine code. Between passes it will store the label values in a 'symbol table', which will hold the label names and values. This table is very useful for debugging and may be printed out.

Another facility is the use of variable names. These will refer to areas of RAM which can be referred to by name rather than by location. These too will go into the symbol table.

Other facilities may include comment statements (like REMs) and other debugging aids.

## Compilers

In many ways compilers are very much like assemblers — the major difference being the source code. Compilers also produce symbol tables.

Trying to design a compiler is probably the best way to get to know how they work. There are a lot of problems which only appear while actually trying to design one. In an earlier article I said that entire books had been written about sorting algorithms. Well, *libraries* have been written about compilers!

The power of a compiler, as opposed to an interpreter, is due to three things:

a) The compiler checks *every* line of a program as it compiles it — there's no chance of a syntax error appearing while the program is running.

b) The machine code output runs about ten times faster than an interpreter.

c) It is easier to provide a 'structure' to the language — some compiled languages (e.g: ALGOL 68) do not need GOTO-like statements because the language is constructed to make it possible to do almost anything in a loop structure. For example, the BASIC statements:

```
10 FOR I = 1 TO 10
20 IF A(I) = B THEN 40
30 NEXT I
```

could be done in an ALGOL-like language simply as:

```
10 FOR I TO 10 WHILE A(I) ◇ B DO
20 NEXT I
```

I personally think that the sooner small compilers become available the better, especially for highly structured languages such as PASCAL or PL/I. Writing in BASIC implies the use of GOTO statements; some people would hold that this leads inevitably to 'bad habits' in program writing. Certainly, having programmed in both ALGOL and BASIC, I very much prefer the former. The recent rises in personal computing power *should* mean that structured-language compilers will soon be available — I only hope that BASIC is not too firmly embedded in the market to be removed quickly and painlessly.

The following program could form part of an interpreter or compiler. Basically what it does is to sort out bracketed statements in terms of precedence of execution. The program is 're-entrant'. This means that it uses *itself* as a subroutine. Say we have a program segment to strip leading spaces off a string. One way to structure it would be:

```
10 REM REMOVE SPACES
20 A$ = RIGHT$ (A$, LEN(A$) – 1)
30 IF LEFT$(A$, 1) = " " THEN ▶
```

```
    GOSUB 10
40  RETURN
```

Line 30 causes the routine to be re-entered unless there are no spaces left. While the above task *could* be done using a simple loop, the task of analysing a bracketed expression is such that intermediate results have to be stored in a particular order. The easiest way of doing this is to use a re-entrancy with a stack structure to hold the intermediate results. As the RETURN address of each GOSUB in a program is held in a stack also, there will be a correspondence between the two. The easiest way to understand this is to follow the program's action.

Given an input of, say, A(B)C, it will output the lowest priority part of the expression first and remove it from the input string. It will then re-enter with the remaining string:

input: A(B)C
output: C A B

input: A((B)C(D(E)))
output: A C B D E

First, the main program:

```
100 DIM S$(30)
```

S$ is the stack.

```
110 SB = 0 : SE = 30
```

SB marks the top of the stack, SE is the maximum stack height.

```
120 INPUT I$
```

I$ is the expression to be analysed.

```
130  GOSUB 1000 : REM ANALYSE
140  GOTO 120
```

Now for the analysis routine.

```
1000 REM ANALYSE
1010 IF I$ = "" THEN RETURN
```

checks for null input string — finished.

```
1030 J = 0
```

J is the current bracket depth.

```
1035 I = LEN(I$) + 1
1040 I = I - 1
```

This is part of a loop in which I goes from LEN(I$) to 1. This form (initialising I to the value above its starting value) allows the decrement of I to be at the start of the loop.

```
1045  IF I < 1 THEN 1080
```

end of loop check.

```
1047  IF J < 0 THEN STOP
```

negative J means too many "("s — an error.

```
1050  T$ = MID$ (I$, I, 1)
```

T$ is a temporary store to speed the program up — saves working out MID$() every time.

```
1055  IF T$ = ")" THEN J = J + 1 :
      GOTO 1040
1056  IF T$ = "(" THEN J = J - 1 :
      GOTO 1040
```

adjust J to be equal to the bracket depth. If T$ gets past 1055 and 1056 it must be a character for output — but only if it's outside the outer set of brackets.

```
1057  IF J<>0 THEN 1040
```

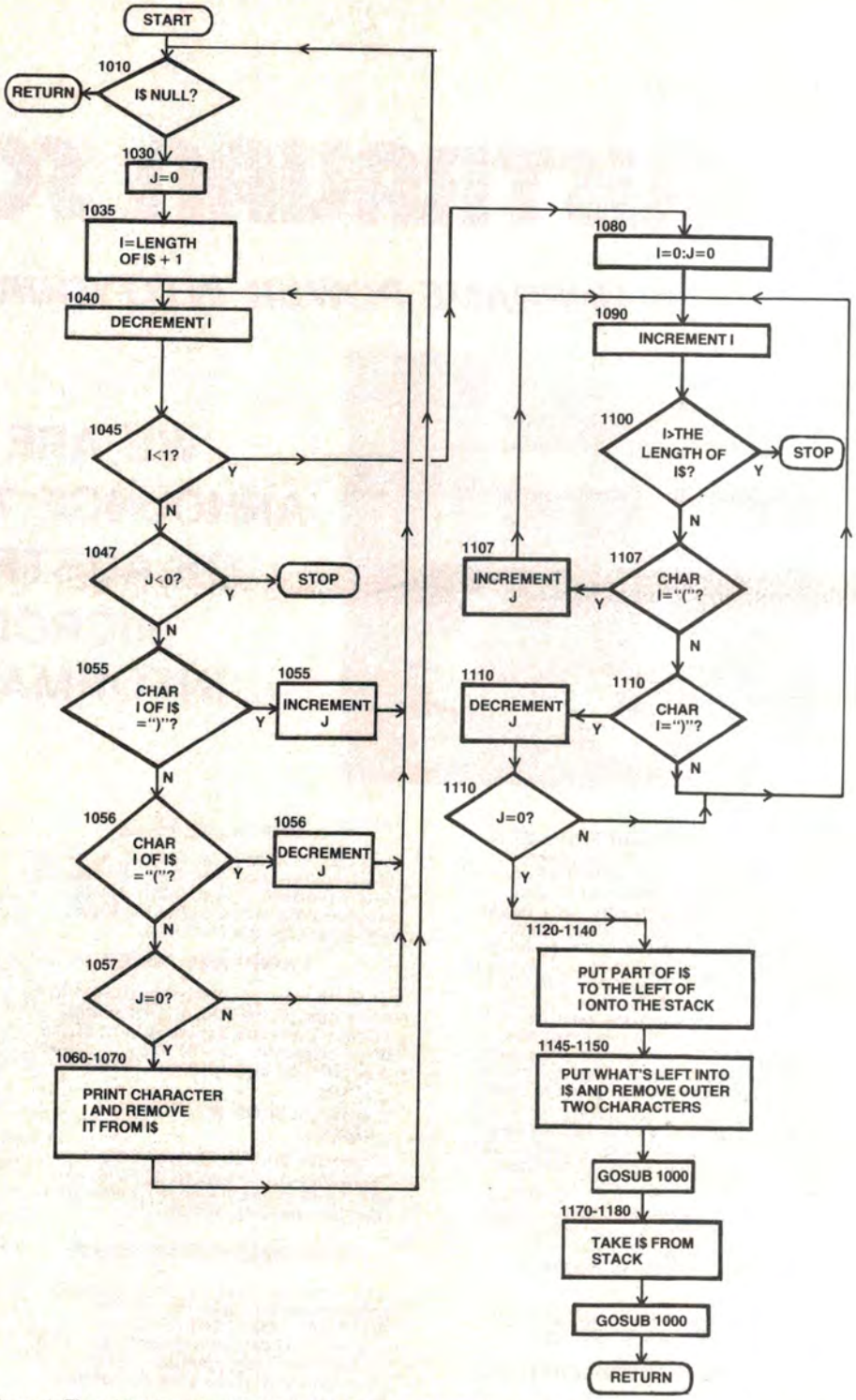


*Figure 1.* **Flowchart for a 'synthetic analyser' — a program to deal with bracketed expressions for a compiler. The input string is I$.**

Okay, T$ must be valid output if it gets to here.

```
1060  PRINT "-"; T$; "-"
```

The "-"s are to show T$ clearly even if it's a space. Now we have to remove T$ from I$:

```
1061  IF LEN(I$)=1 THEN I$=
      "":GOTO 1070
1062  IF I=1 THEN I$=RIGHT$
      (I$,LEN(I$)-1):GOTO 1070
1063  IF I=LEN(I$) THEN
      I$=LEFT$ (I$,LEN(I$)-1):
      GOTO 1070
1065  I$ = LEFT$(I$, I-1) +
      RIGHT$(I$ LEN(I$) - 1)
```

Lines 1061, 1062 and 1063 are necessary because some forms of BASIC (including, unfortunately, the one I'm working in) won't accept LEFT$(I$,0) or RIGHT$(I$,0).

```
1070  GOTO 1000
```

deals with the next character. Line 1045

(the end of the loop) points to:

```
1080 I = 0 : J = 0
```

I again points to a specific character in the string and J is again the bracket depth. This part of the program splits the string into a complete bracket pair and another part. It re-enters with both of them.

```
1090 I = I + 1
1100 IF I › LEN(I$) THEN STOP
```

Line 1100 stops the program if a string which got to line 1080 didn't have a bracket pair in it.

```
1105 T$ = MID$(I$, I, 1)
1107 IF T$ = "(" THEN J = J + 1 :
     GOTO 1090
1110 IF T$ = ")" THEN J = J – 1 : IF
     J=0 THEN 1120
```

if I points to the end of a bracketed expression, go to line 1120, otherwise:

```
1115 GOTO 1090
```

to try the next character.

```
1120 SB = SB + 1
1130 IF SB › SE THEN STOP
```

increments the stack pointer and checks for stack overflow.

```
1135 IF I = LEN(I$) THEN S$(SB) =
     " " : GOTO 1145
1140 S$ (SB) = RIGHT$(I$, LEN(I$)
     – I)
1145 I$ = LEFT$(I$, I)
```

splits I$ at character I. The value of I marks the end of a complete bracket pair. Line 1135 serves the same purpose as line 1061 to 1063.

```
1150 I$ = MID$(I$, 2, LEN(I$) – 2)
```

removes the outer two characters (which should be "(" and ")" ).

```
1160 GOSUB 1000
```

re-enters. This will (eventually) reduce I$ to " ".

```
1170 I$ = S$(SB)
```


```
1180 SB = SB – 1
```

takes the rest of I$ back off the stack.

```
1190 GOSUB 1000
```

re-enters with it.

```
1200 RETURN
```

finished. Jumps back to wherever it was called from, be it within the subroutine or (when finished) from line 140. ●

*(This article concludes Phil Cohen's 'Advanced Basic' series.)*