Advanced BASIC — Part 3
Top-down programming

Phil Cohen

Following the series on Back Door into BASIC, ETI has begun a series on Advanced BASIC for those who want to follow up their elementary knowledge. Previous articles have explained 'Sorting' and 'Chaining'; this month Phil Cohen goes into 'Top-down programming'.

WHAT EXACTLY is 'top-down programming' — the almost mystical art which professional programmers are sometimes heard to refer to in awe? Essentially, it means that the program is written from the outside inwards — the main outline is programmed first, the details of the algorithm coming later in subroutines.

As an example of this, let's take a hypothetical chess program. Writing such a program from scratch in BASIC is a daunting task without the use of top-down methods. You have to start by knowing exactly what the program is required to do, and this takes a large amount of work on the definition of the task before programming can even begin.

With top-downing, the program is written *as you plan the task*. For instance, the main program may consist of only subroutine calls (i.e: GOSUBs), with the subroutine tasks only roughly defined. Let's make the explanation of this a bit easier by changing the language a little. We'll represent this:

```
10    GOSUB 1000
.
.
.
1000 REM ROUTINE TO ADD
.
.
.
1100 RETURN
```

by this:

```
10    add
.
.
.
1000 PROCEDURE add
.
.
.
1100 END
```

This nomenclature will be familiar to those who have met ALGOL or PASCAL — languages ideally suited to top-downing. The word 'add' (unless preceded by the reserved word 'PROCEDURE') means that control is to be passed to the line of the program at which 'PROCEDURE add' occurs. This is entirely analogous to 'GOSUB', except that a name is used instead of a number to define the subroutine's start. The word 'END' has the same effect as 'RETURN' in BASIC.

Let's start writing our chess program. The main program may consist of:

```
10 set-up-board
20 input-human's-move
30 check-human-move-legal
40 make-move
50 test-for-human-win
60 choose-best-move
70 make-move
80 test-for-machine-win
90 GOTO 20
```

Then the subroutines can be worked out in detail:

```
4000  PROCEDURE set-up-board
4010  change-memory
4020  print-results
4030  END

11000 PROCEDURE print-results
11010 FOR I=1 TO 8
11020 FOR J=1 TO 8
11030 print-square(I, J)
11040 NEXT J
11050 PRINT
11060 NEXT I
11070 END
```

In line 11030 we see parameter passing. This is also found in BASIC:

```
10 A = SIN (B)
```

the only differences being a) that here there is more than one parameter and b) that the procedure 'print-square(,)' delivers no result to the calling program. In this sense it is similar to the TAB() function in BASIC.

It can be seen from the above segment that this sort of top-down approach is easy in an ALGOL-like language — but how can we make use of it in BASIC?

Let's try re-writing the above example in BASIC:

```
10 GOSUB 1000 : REM SET UP
     BOARD
20 GOSUB 2000 : REM INPUT
     HUMAN'S MOVE
30 GOSUB . . .
```

and so on. Now for the subroutines:

```
4000 REM MAKE MOVE
4010 GOSUB 1000 : REM CHANGE
       MEMORY
4020 GOSUB 11000 : REM PRINT
       RESULTS
4030 RETURN

11000 REM PRINT RESULTS
11010 FOR I=1 TO 8
11020 FOR J=1 TO 8
```

Now here we hit a small problem. In BASIC there is no statement which will allow parameter passing to a subroutine, such as

```
11030 GOSUB(I, J) 12000 : REM
        PRINT SQUARE
```

and so we have to resort to

```
11030 GOSUB 12000 : REM PRINT
        SQUARE
11040 NEXT J
11050 PRINT
11060 NEXT I
11070 RETURN

12000 REM PRINT SQUARE
12010 P = B(I, J)
.
.
.
```

where P is the number representing the piece to be printed and B(,) is an 8 x 8 array which holds the current state of the board. Variables I and J are called 'global' variables in this context, because they have the same meaning inside and outside the subroutine.

Zoning

There's only one thing wrong with the above method in BASIC and that is that variables used in other routines (or in the main program) shouldn't change value in routine 'print-square'. This is all very well if you can keep track of *all* the variables you use in a large program. This is not always easy, however, and mistakes can take a very long time to find. Another point is that 'library' routines — useful ones which are going to be used in more than one program — will have to use non-overlapping sets of variables.

Languages such as ALGOL take care of these problems automatically — variables are defined on entry into a subroutine and will be destroyed on exit. Those which were used in the calling routine and re-defined on entry to the called routine will have their value stored on entry and will be set to their original value on exit.

Unfortunately, routines written in BASIC have to have this storage and retrieval added to them as they are written. One way to do this is to define a general-purpose stack:

```
10 DIM  S(100)  :  REM  (NO
     RELATION!)
20 SP = 0
30 SE = 100
```

and, on entry to each routine, to push onto the stack the values of the variables (except those which are parameters) which are used in the routine. These can then be pulled off the stack on exit:

```
12000 REM PRINT SQUARE
12010 IF SP = SE THEN PRINT
        "STACK      OVERFLOW":
        STOP
12020 SP = SP + 1
12030 S(SP) = P
12040 P = B(I, J)
.
.
.
12100 P = S(SP)
12110 SP=SP - 1
12120 RETURN
```

Lines 12010 to 12030 and 12100 to 12110 will be repeated (with a different variable each time) in *all* routines, and so we can shorten the above in the long term:

```
10  DIM S(100)
20  SP = 0
30  SE = 100
35  GOTO  500  :  REM  MAIN
      PROGRAM
40  REM PUSH
50  IF  SP  =  SE  THEN  PRINT
      "STACK OVERFLOW":STOP
60  SP = SP + 1
70  S(SP) = XX
80  RETURN
90  REM PULL
100 XX = S(SP)
110 SP = SP - 1
120 RETURN

12000 REM PRINT SQUARE
12010 XX = P : GOSUB 40 : REM
        PUSH
11020 P = B(I, J)
.
.
.
12100 GOSUB 90 : REM PULL
12110 P = XX
12120 RETURN
```

XX is a parameter to the push and pull routines and obviously shouldn't be used anywhere else.

Example

As an example of zoning, the following 'library' routine will print out the value of variable A in a format N characters wide. For example:

```
A = 1, N = 3 : 1.0
A = -0.01, N = 5 : -1E-2
A = 155, N = 4 : 2.E2
```

First, the stacks:

```
10 DIM S(100), S$(20)
```

two of them — one for numeric variables and one for strings. Also, two stack pointers:

```
20 SP = 0 : P2 = 0
```

and two end-of-stack indicators:

```
30 SE = 100 : E2 = 0
```

Jump to the main program:

```
35 GOTO 500
```

Now for the stack handling routines. The first two are as before, for the numeric variables:

```
40 REM PUSH
50 IF  SP  =  SE  THEN  PRINT
     "STACK OVERFLOW":STOP
60 SP = SP + 1
70 S(SP) = XX
```

▶

```
80 RETURN
90 REM PULL
100 XX = S(SP)
110 SP = SP - 1
120 RETURN
```

then the strings:

```
130 REM PUSH STRING
140 IF P2 = E2 THEN PRINT
    "STRING OVERFLOW":STOP
150 P2 = P2 + 1
160 S$(P2) = X$
170 RETURN
180 REM PULL STRING
190 X$ = S$ (P2)
200 P2 = P2 - 1
210 RETURN
```

Now for the main program. This one is just a test routine:

```
500 INPUT A
510 INPUT N
520 GOSUB 1000
530 STOP
```

Finally, the formatting routine itself. It begins by pushing all of the original values of the non-local variables onto the stacks:

```
1000 REM FORMAT
1010 XX = SA : GOSUB 40
1012 XX = SN : GOSUB 40
1014 XX = E : GOSUB 40
1016 XX = LL : GOSUB 40
1018 XX = L : GOSUB 40
1020 XX = K : GOSUB 40
1022 XX = I : GOSUB 40
1024 X$ = A$ : GOSUB 130
1026 X$ = B$ : GOSUB 130
```

SA and SN are stores for the input values of A and N. This is necessary because of the iterative nature of the routine, as will be explained later:

```
1060 SA = A
1070 SN = N
```

E is the 'rounding' figure. This is also part of the iterative procedure:

```
1080 E = 38
```

LL is a constant which is used to convert natural logs (as found in BASIC) to base 10 logs:

```
1090 LL = LOG(10)
```

Line 1097 is the start of a loop in the program, so A and N are re-initialised here:

```
1097 A = SA
1098 N = SN
```

as are A$ and B$ — A$ holds the mantissa of the output while B$ holds the exponent:

```
1099 A$ = "" : B$ = ""
```

Both are set to the null string. Check to see if A is zero. This would upset the algorithm and so it is 'trapped' here:

```
1110 IF A = 0 THEN A$ = "0" :
     GOTO 2040
```

Line 2040 is the exit from the routine. Now see if a "–" sign is required:

```
1120 IF A > 0 THEN 1160
1130 A$ = "–" : A = ABS(A)
1140 N = N - 1
1150 IF N = 0 THEN 2040
```

and if it is, add it — as we have to deal from here on in with a positive number, we can 'forget' whether A was negative or not, as long as we've added a "–" sign if it was required. N is decremented if the "–" character *was* added and if there's no more room (i.e: if N = 0) we're finished — 2040 is the exit.

Now for the rounding up (explanation later):

```
1160 A = A + 5/10↑E
```

Now to find the exponent part of the output:

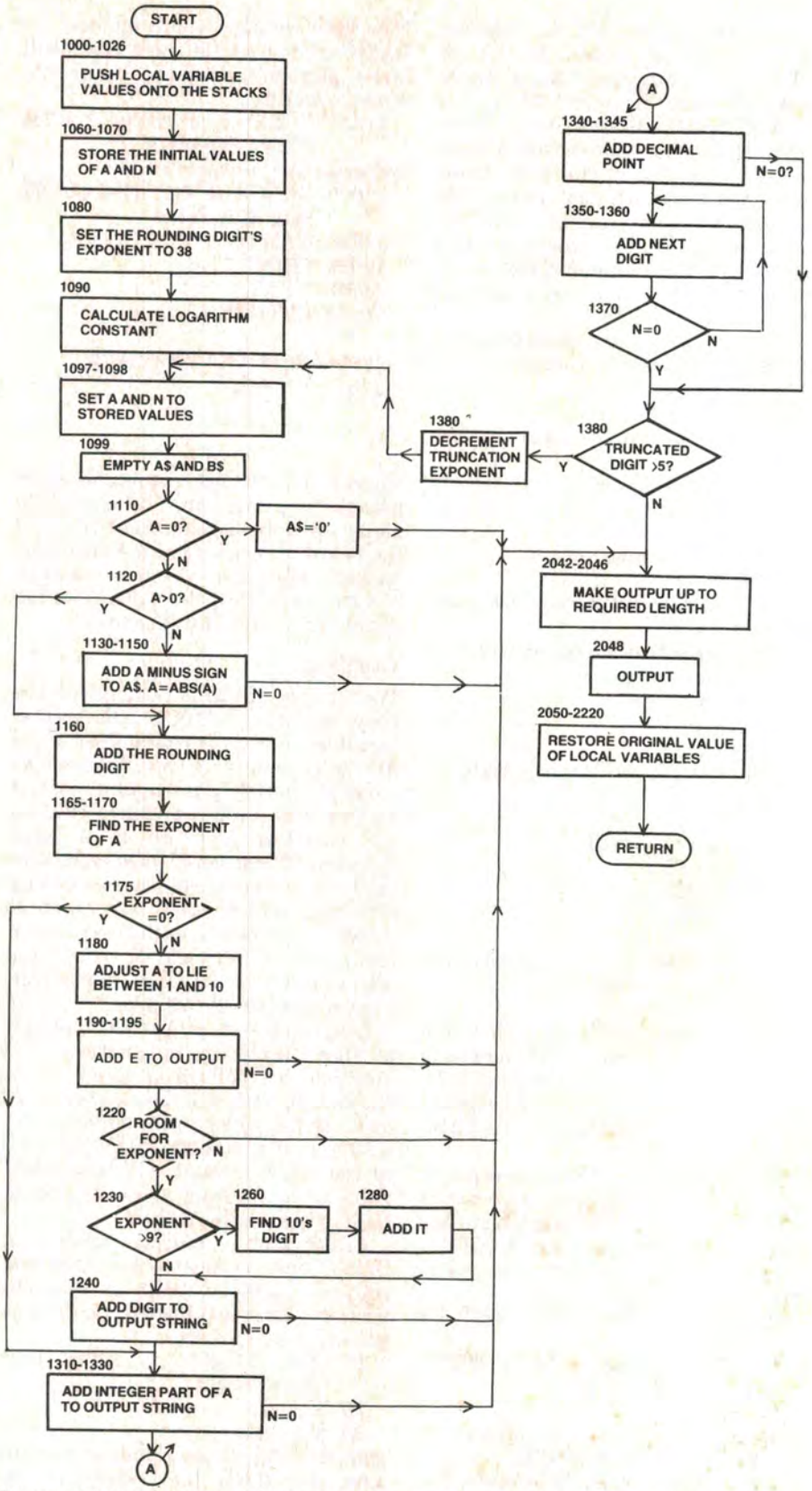

Figure 1. **A full flowchart for a routine to display the value of variable A in a print field of width N.**

```
1165 L = LOG(A)/LL
```
sets L to the base 10 log of A.
```
1170 L = INT(L)
```
makes L the required exponent.
```
1175 IF L = 0 THEN 1310
```
misses out the exponent output section entirely if an exponent is not required. 'Forget' the exponent part of A — we have it already in L:
```
1180 A = A/10↑L
```
Start putting the exponent into B$:
```
1190 B$ = "E" : N = N - 1
1195 IF N = 0 THEN 2040
```
Find out whether a "−" sign is needed in the exponent and add it if it is:
```
1200 IF L > = 0 THEN 1220
1210 B$ = B$ + "−" : N = N - 1 : L =
     ABS(L)
```
Check to see if there's room for the exponent (if it's two figures, having room for only one could be misleading):
```
1220 IF N = 0 OR (L > 9 AND N < 2)
     THEN 2040
1230 IF L > 9 THEN 1260
```
jumps if two figures are needed. If only one is required, add it:
```
1240 B$ = B$ + CHR$ (ASC("0") +
     L) : N = - 1
```
The above statement is based on two functions — ASC(), which delivers the ASCII value of a character, and CHR$(), which delivers the character equivalent of an ASCII value. As the ASCII numbers are in sequence, starting from 0, the above statement will add the character corresponding to the value of L. Now jump the next bit, which is only required for two-digit exponents:
```
1250 GOTO 1300
```
Find the first (most significant) digit:
```
1260 K = INT(L/10)
```
and the least significant:
```
1270 L = L - K*10
```
Add the first character:
```
1280 B$ = B$ + CHR$(ASC("0") +
     K) : N = N - 1
```
then jump back to add the next as if it were a one-digit exponent:
```
1290 GOTO 1240
```
After adding the exponent, start on the mantissa (A should be between 1 and 10 by now):
```
1300 IF N = 0 THEN 2040
1310 K = INT(A) : A = A - K : A =
     A*10
```
strips the most significant digit from A and puts it between 0 and 10.
```
1320 A$ = A$ + CHR$(ASC("0") +
     K) : N = N - 1
1330 IF N = 0 THEN 2040
```
Now add the decimal point:
```
1340 A$ = A$ + "." : N = N - 1
1345 IF N = 0 THEN 2040
```
Add the subsequent digits:
```
1350 K = INT(A) : A = A - K : A =
     A*10
1360 A$ = A$ + CHR$(ASC("0") +
     K) : N = N - 1
1370 IF N > 0 THEN 1350
```
Once it's got through that, N must be zero and A must be the value of the next figure — the one which is 'truncated' (Lit. 'cut off'). Let's say that A on entry to the routine was 1.46 and that all we had room for was 1.4. It would be nice to display 1.5 instead, which would be closer to the truth. So if A at this stage is greater than 5, it means that the number needs 'rounding up'. We can't just change the last digit of the output — say it was 9.999 and we only had 4 spaces for it (one for the decimal point, remember); if we put 4 digits in we get 9.99 — we find that the truncated part is a 9 and so we change the last digit — we get 9.90. This is obviously not correct. Even if we change *all* the digits, we get 10.00 — this is now 5 characters long!

No, by far the simplest (although certainly not the quickest) approach is to start again. How do we deal with the rounding up, though? We add a 5 in the truncated position. In this way, if the number in this position is 5 or over, it will overflow into the higher digits. Wait a minute, though, we don't know which will be the truncated digit until we try it. What we have to do is to start with it at the extreme right-hand end of the number (i.e: at 5E−38 − the smallest number in many BASICs) and, *if the truncated digit is over 5*, to move it up until it makes the truncated part overflow. This is what lines 1080 and 1160 are for. Find out if iteration is required:
```
1380 IF A > = 5 THEN E = E - 1 :
     GOTO 1097
```
Now print the result.
```
2040 A$ = A$ + B$
```
Make up the length to that required:
```
2042 IF LEN(A$) = SN THEN 2048
2046 FOR I = 1 TO SN - LN(A$) :
     A$ = A$ + " "
2047 NEXT I
2048 PRINT A$;
```
Now set all the local variables back to their original values *in the reverse order:*
```
2050 GOSUB 180
2060 B$ = X$
2070 GOSUB 180
2080 A$ = X$
2090 GOSUB 90
2100 I = XX
2110 GOSUB 90
2120 K = XX
2140 GOSUB 90
2150 L = XX
2160 GOSUB 90
2170 LL = XX
2180 GOSUB 90
2190 E = XX
2200 GOSUB 90
2210 SN = XX
2220 GOSUB 90
2230 SA = XX
2240 RETURN
```
And that's it!