SNOBOL 4+

Michael Davis

Information on the use of the version of SNOBOL4 known as SNOBOL 4+.

Contents

Copyright

Introduction

This is unofficial documentation for the version of the SNOBOL programming language published by Catspaw under the name SNOBOL4+. SNOBOL4+ is now available free, but the documentation from Catspaw is no longer available.

Catspaw also published a cut-down version of SNOBOL4+ called "Vanilla SNOBOL". The documentation for this is available free, and it covers most of the features of SNOBOL4+. It seemed to me pointless to repeat everything that is given there, so I have written this on the assumption that you have the Vanilla SNOBOL documentation, and this document carries on from there. Thus the Vanilla documentation and this document together constitute an almost complete documentation of SNOBOL4+.

SNOBOL4+ is available for download at ftp://ftp.snobol4.com/snobol4p.zip.

A tutorial for Vanilla SNOBOL is at:
http://burks.bton.ac.uk/burks/language/snobol/catspaw/tutorial/contents.htm

and the Vanilla SNOBOL reference manual at:
http://burks.bton.ac.uk/burks/language/snobol/catspaw/manual/contents.htm

If you want Vanilla SNOBOL itself, it is available at:
ftp://ftp.snobol4.com/vanilla.zip

Note: Previous versions of this page stated that the documentation was included with Vanilla SNOBOL. Either this was an error, or it was once true but is no longer.

The following assumes that you have these Vanilla SNOBOL documents, and describes only matters not covered there, or in some respect not the same as described there.

I have tested almost all the features I describe, and most of my description is based on my own experience. In a small number of cases I have simply passed on information from other sources without checking. In another very small number of cases I have corrected errors in one of those sources. Unfortunately previous editions of this document contained various errors. I have eradicated all the errors that I know of, but if I have missed any errors I apologize. If you find any I shall be extremely grateful if you let me know. I can be contacted at michaeldavismd (at) yahoo (dot) co (dot) uk.
In the past there were two versions of this document available: a PDF version, and an HTML version. However, I have decided that the effort of maintaining and updating both versions is not worthwhile, so now the only version is this one, at:
http://www.sachsdavis.clara.net/Snobol4Pdoc.pdf

Acknowledgements

I am indebted to two sources. They provided a starting point, which I have then built on with my own experimentation.

(1) Mark Emmer and Edward Quillen's Macro SPITBOL manual, which lists a number of features of SNOBOL4+ in order to indicate differences between SNOBOL4+ and SPITBOL.

(2) Gordon Peterson produced a database of differences between Vanilla SNOBOL and SNOBOL4+, and posted it on the Yahoo SNOBOL4 group.

SNOBOL4+ Features

(1) Control statements

(a) -INCLUDE This control statement enables a program file to include the code contained in another file. For example, suppose you have the following program:

```
OUTPUT = 'Here is the first line,'
-INCLUDE "subprog.sno"
OUTPUT = 'and here is the third line.'
END
```

and suppose the file "subprog.sno" contains the following:

```
OUTPUT = 'this is in the subprogram, '
```

then running the first program is equivalent to running:

```
OUTPUT = 'Here is the first line,'
  OUTPUT = 'this is in the subprogram,'
  OUTPUT = 'and here is the third line.'
  END.
```

(b) -FAIL A compiler directive that turns off 'NOFAIL' mode. See under –NOFAIL below.

(c) -LINE A compiler directive that allows macro preprocessors to place their line numbers into a source file. It is used as: -LINE linenumber 'filename'.

(d) -NOFAIL This is a compiler directive which sets 'NOFAIL' mode. This means that any statement which fails and does not have a conditional goto will terminate the program with an error message. An unconditional goto does not prevent this. This mode can also be set by the /NF command-line option. In either case the mode can be turned off by –FAIL.

(e) -OPTION Followed by a string of options, this produces the same effect as putting the options on the command line. Thus –OPTION '/W /i=text.txt' causes input to be taken from the named file, and a SAV file to be written. Options specified on the command line override this. Options /E, /K, /L, /M, and /NN are not allowed.

(f) -PLUSOPS Some of the additional features in SNOBOL4+ which go beyond standard SNOBOL 4 are available only if the control statement '-PLUSOPS 1' (or any other positive integer) is used. The same effect is achieved by the command-line option /P.

(2) 'Real' Numbers

In addition to integers, SNOBOL4+ can deal with 'real' (i.e. floating point) numbers. In most respects, real numbers work much as in many other languages. Thus 'a = 5' will give a an integer value, whereas 'a = 5.0' or 'a = 5.' will it give a 'real' value. However, since '.' is used as an operator in SNOBOL, it is *not* sufficient to write 'a = .7' ; it is necessary to write 'a = 0.7'. Numbers can be entered in exponential form, so that 1.23456E2 = 123.456. A 'D' can be used in place of 'E'. One might expect this to give double precision, but as far as I can determine the two are synonyms. I believe that they are included for compatibility with other versions of SNOBOL4 in which E gave a 32-bit representation, and D gave 64 bits. In SNOBOL4+ 64 bits are always used.

'Real' numbers include three values which are certainly not real numbers as normally understood: INFINITY, -INFINITY, and NAN ('not a number'). I make no attempt to give a full account of how they work, but the following examples illustrate the basic ideas:

1.0 / 0.0 gives the result INFINITY

5.0 - INFINITY gives the result -INFINITY
INFINITY - INFINITY gives the result NAN
0.0 / 0.0 gives the result NAN.

`ln(0)` gives the result `-INFINITY`
`ln(-1)` gives the result `NAN`

The `CONVERT()` function can be used to convert a string to a type '`NUMERIC`', which means either integer or real as appropriate, integer if possible.

The real functions `ln()` (i.e. natural logarithm) and `exp()` (i.e. e to the power) are provided. There is also a function `chop()`, which truncates a real number to its integer part, i.e. it rounds towards zero. Thus `chop(3.47)` and `chop(3.97)` both return the result `3.0`, and `chop(-3.47)` returns `-3.0`. Note that the result is a *real* number, not an integer. This should be compared with `convert(-3.47, 'integer')`, which produces the result `-3`, an *integer*.

(3) Keywords

SNOBOL4+ includes the following keywords:

(a) `&ABEND.` (**unprotected** keyword) My understanding is that this does nothing in SNOBOL4+ for DOS, and is included for compatibility with versions of SNOBOL which run on other platforms. It stands for 'Abnormal Ending'.

(b) `&line`, `&file`. (**protected** keywords) These give the line number within the current program source file of the current line, and the filename of the current program source file.

(c) `&lastline`, `&lastfile`. (**protected** keywords) These give the line number of the *last* line that was executed, and the filename of the program source file containing that line. This may be different from the current file if '`-INCLUDE`' has been used.

(d) `&MARB.` (**protected** keyword) This contains the `MARB` pattern. See `MARB` under 'Patterns & pattern-matching functions' for details.

(e) `&PARM.` This keyword gives the parameters from the MS-DOS command line that ran the program. It ignores any additional text at the end of the command line that SNOBOL has ignored. For example, the command line

```
snobol4 newtest /o=out.txt /p /e=eee.txt /c > out2.txt
```

results in `&PARM` having the value `NEWTEST /O=out.txt /P /E=eee.txt /C`.

(f) `&PRECISION.` (**unprotected** keyword) This sets the number of significant figures displayed for real numbers, from a minimum of 1 up to a maximum of 16. A value of 0 or more than 16 is treated as 16. The default is 7.

(g) `&REAL.` (**unprotected** keyword) If this is negative, real numbers are displayed in exponential notation, e.g. `1.2345E2` rather than `123.45`.

(h) `&TRIM.` This keyword works slightly differently in SNOBOL4+ from how it works in Vanilla SNOBOL. As in Vanilla SNOBOL, a positive value strips trailing blanks, and a zero value adds trailing blanks up to the record length for the input file. However, in SNOBOL4+ a *negative* value of `&TRIM` causes the input to be left as it is, with no change to the number of trailing blanks, if any.

(4) Embedded assignments and Multiple assignments (only with `-PLUSOPS 1`)

(a) Embedded assignments

An assignment (subject = object) can be used wherever an expression could be used. If the assignment succeeds, it returns the value assigned.

```
i = 5
```

```
a[i = i + 1] = 7
```

assigns the value 6 to `i`, and 7 to `a[6]`. The assignment `i = i + 1` is embedded in the assignment `a[...] = 7`.

(b) Multiple assignments

```
a[i] = b = c = 7
```

assigns the value 7 to all 3 of the destinations given.

In more complex examples care may be required over the order of the assignments. For example, consider the following:

```
i = 2
  a[i] = b[i = i + 1] = c[i] = d[i = i + 2] = i = i + 4
```

What is the value of `i` used in `c[i]`? In other words, which, if any, of the three assignments to `i` has already taken place when the address of `c[i]` is determined? And what value is eventually assigned to `c[i]`? The *original* value of `i` with 4 added? Or the value of `i` when all three assignments have been made? Or what?

The answer is that these assignments work as follows: Firstly, the addresses of the assignment destinations are evaluated in order, from left to right, and stored. Any assignments embedded inside the expressions will take place as they are evaluated. (Thus, in the example above, the addresses are stored for `a[2]`, `b[3]`, `c[3]`, `d[5]`, and `i`. By this time `i` will have the value 5.) Secondly, the right- hand expression is evaluated. (In the above example `i + 4` becomes 9.) Thirdly, this value is assigned to each of the destinations at the previously stored addresses.

So the effect of the above example is the same as:

```
a[2] = 9
b[3] = 9
c[3] = 9
d[5] = 9
i = 9
```

(5) Alternative evaluation (only with `-PLUSOPS 1`)

A list of expressions can be bracketed and separated by commas. The expressions are evaluated in order until one succeeds. The value of this expression then becomes the value of the whole bracketed expression. The remaining expressions are not evaluated.

This is an *extremely* useful feature. Only experience of its use will show the full extent of its power, but here are some simple examples:

Example 1:

```
(GT(A,B) A, B)
```

First the expression `GT(A,B) A` is evaluated. If it succeeds, then the value returned (i.e. The value of `A`) is returned as the value of the whole expression. If, on the other hand, `GT(A,B) A` fails, then `B` is evaluated, and its value is returned as the value of the whole expression. The effect of this is similar to "if A>B then A, else B", i.e. it produces the larger of `A,B`.

Example 2:

```
PAT1 = (break('x') . output)
PAT2 = (break('z') . output)
SUBJECT = 'The quick brown fox jumps over the lazy dog.'
SUBJECT (EQ(N,1) PAT1, EQ(N,2) PAT2, ABORT) = ''
```

```
   OUTPUT = SUBJECT
```

Here the first thing that happens is the evaluation of `EQ(N,1) PAT1`. If `N` is equal to 1, then `EQ(N,1)` succeeds, `PAT1 = ''` is applied to `SUBJECT`.

If `N` is not equal to 1, then `EQ(N,1)` fails, and so the same process is applied to `EQ(N,2) PAT2`.

If `N` is equal to neither 1 nor 2, then `ABORT` is reached.

The effect of this is that, depending on the value of `N` the sentence is sent to output as:

```
The quick brown fo
x jumps over the lazy dog.
```

(If `N` is 1)

or as:

```
The quick brown fox jumps over the la
zy dog.
```

(If `N` is 2)

or , finally, as:

```
The quick brown fox jumps over the lazy dog.
```

(If `N` is anything else).

Example 3:

```
   Another use of alternative evaluation is to prevent an
expression from failing. Thus (expression, ) will always succeed:
if expression fails, then the empty expression after the comma
will be evaluated, and will succeed. The return value of
(expression, ) will be either the value of expression, or null.
```

(6) String-handling functions

(a) `ASC(`*string*`)` Returns the code number representing the first character of the *string*. Thus `ASC('ABCD')` returns 65, the ASCII code for 'A' The function returns an integer from 0 to 255. If used only for single-character strings, this function is inverse to `CHAR(I)`. `ASC('')` causes the program to abort with an error message.

(b) `CHAR(`*integer*`)` This actually works as in Vanilla SNOBOL. However, there is an error in the description of this function in the Vanilla reference manual. There it is stated that a value of the *integer* outside the range 0-255 causes the function to fail: in fact it causes the program to abort with an error message.

(c) `REMOVE(`*string1*`, `*string2*`)` Returns string *string1* minus any occurrences of characters in string *string2*. Thus `REMOVE('first string', 'second')` returns `'firt trig'`.

(d) `REVERSE(`*string*`)` Returns the *string* reversed. So `REVERSE('This string.')` returns `'.gnirts sihT'`.

(e) `SUBSTR(`*string*`, `*integer1*`, `*integer2*`)` Returns a substring of *string*. *integer1* gives the position in *string* to start from, and *integer2* gives the length of the substring. `SUBSTR('abcdef', 3, 2)` returns `'cd'`. If *integer2* is absent or zero, the substring from position *integer1* to the end of the string is returned. For example `substr('abcdef', 3)` returns `'cdef'`.

(f) `LEQ(`*string1*`, `*string2*`), LNE(`*string1*`, `*string2*`), LLE(`*string1*`, `*string2*`),`

LGE(*string1*, *string2*), LLT(*string1*, *string2*) These work like LGT(*string1*, *string2*) [*string1* is lexically greater than *string2*], and stand for the following:

LEQ(*string1*, *string2*):	lexically equal to
LNE(*string1*, *string2*):	lexically not equal to
LLE(*string1*, *string2*):	lexically less than or equal to
LGE(*string1*, *string2*):	lexically greater than or equal to
LLT(*string1*, *string2*):	lexically less than.

(7) Patterns & pattern-matching functions

(a) ATAB(*integer*) (only with -PLUSOPS 1) works like TAB(*integer*), except that it is possible to use a value of *integer* which is to the left of the current cursor position.

e.g. `abcdefg f ATAB(2) . X`

This will give X the value 'cdef'.

(b) ARTAB(*integer*) (only with -PLUSOPS 1) works like RTAB(*integer*), except that it is possible to use a value of *integer* which is to the left of the current cursor position, in a similar way to ATAB(*integer*).

(c) BREAKX(*string*) works like BREAK(*string*), except that if it fails it will continue to search .

e.g. Consider the following:

```
s = 'abcXdefXghiX'
  pat = 'a' BREAK('X') LEN(1) 'g'
  s pat . Q
```

This will fail, because BREAK(X) will match up to the *first* occurrence of X, and g will not be found. On the other hand the same match with BREAKX instead of BREAK will succeed, and give Q the value 'abcXdefXg', because, when the match fails after the first X, breakx(X) will try again, and match up to the second X, and now g *is* found.

BREAKX(*string*) can be very useful, but in unanchored mode can slow things down considerably, as backtracking can cause it to repeatedly scan the same part of a string looking for matches.

(d) LEN(*integer*) In SNOBOL4+ (with -PLUSOPS 1) the parameter to this function can be negative. For example, if *integer* is 4, the cursor will move 4 spaces to the left, matching a string of 4 characters. The characters matched are kept in their left-to-right order. Be careful of using LEN with a negative parameter when QUICKSCAN heuristics are enabled, as it can result in unwanted failure of pattern matching.

(e) MARB (only with -PLUSOPS 1) This pattern behaves like ARB except that it matches the *maximum* number of characters it can, rather than the *minimum*. For example,

```
abcXdefXghiXjk' ARB 'X' LEN(1) . OUTPUT
```

will send 'd' to output, whereas

```
abcXdefXghiXjk' MARB 'X' LEN(1) . OUTPUT
```

will send 'j' to output.

The protected keyword &MARB contains the MARB pattern. This can be used to restore MARB if its value is altered.

(f) "?" A question mark can be used as a binary operator which performs pattern matching. Its

simplest use is as an alternative to white space, so that the last example could be written as

```
abcXdefXghiXjk' ? MARB 'X' LEN(1) . OUTPUT
```

but its real strength lies in the possibility of using it within expressions. For example:

```
x = ('abcdef' ? 'b' len(2)) ('uvwxyz' ? len(3) 'z').
```

This "returns the modified value of the subject, failure, or (without replacement) the portion of the subject matched by the pattern". [Gordon Peterson]

(8) The Functions 'INPUT' & 'OUTPUT'

The format of these functions is:

INPUT(*name, unit, options, filename*) *and* OUTPUT(*name, unit, options, filename*)*, unless the /NC command-line option has been specified, for 'Non-Compatible' input and output mode; in this case the format for the INPUT and OUTPUT functions becomes* INPUT *(name, unit, filename, options)*.

Alternatively, the file options can be given in square brackets after *filename* (the fourth parameter to INPUT or OUTPUT). Then a null third parameter must be included.

The parameters are as in Vanilla SNOBOL, except:

(1) *Unit* is not restricted to the values 0 to 16. I find that no error message is produced at least for values up to 100. Up to 32 files can be open at one time.

(2) *Options* can contain various option letters, as well as an integer record size. Thus the following are possible:

```
INPUT('READIN', 1, , 'INFILE')
INPUT('READIN', 1, 100, 'INFILE')
INPUT('READIN', 1, 'B', 'INFILE')
INPUT('READIN', 1, 'BWR25', 'INFILE')
```

The integer part of *options* is a record size, as in Vanilla SNOBOL. If it is omitted a default value of 80 is taken. For input, except in ebinaryf mode, this is a *maximum* record size, in the sense that the record size will be less if an end of record marker is encountered.

The meanings of the option letters are as follows:

If no option letter is included, OUTPUT puts 'Carriage return, Linefeed' (Characters 13, 10) at the end of each output record. INPUT takes either 'Carriage return, Linefeed' or 'Carriage return' alone as an end of record marker. Also character 26 is treated as an end of file marker by INPUT, so that nothing beyond this character will be read.

'L' [Linefeed] indicates that 'Linefeed, Carriage return' are to be used instead of 'Carriage return, Linefeed'. In INPUT, ' Linefeed ' is also read as an end of record.

'O' [One Character] alone uses 'Carriage return' alone instead of 'Carriage return, Linefeed' as an end of record marker. 'LO' causes 'Linefeed' alone to be used. 'LO' follows Unix file convention.

'Z' used in INPUT indicates that character 26 (ctrl-Z) should be read as it is, like any other character, and not treated as an end of file marker and then discarded. 'Z' used in OUTPUT causes character 26 to be appended to the end of the file. This may seem confusing: in INPUT the default is that character 26 is used an end of file marker, option Z indicating it is not so used;in OUTPUT the default is that character 26 is *not* used an end of file marker, option Z indicating it is so used.

'B' [Binary mode] means that all characters are accepted as they are: Characters 10, 13, & 26 are given no special status. In reading a file the number of characters specified by the record size will

always be read, unless the physical end-of-file is reached.

'R' [Read Mode] 'W' [Write Mode] & 'RW' [ReadWrite Mode]:

If you use

```
INPUT('READIN', 1, 'W', 'INFILE')
OUTPUT('WRITEOUT', 2, 'R', 'INFILE')
```

both for the same file, the file will be opened for both reading and writing, so that it can be updated. I cannot find any other useful way of deploying these modes. Even with the 'W' mode set, `INPUT` still works only for reading, as far as I can tell, and likewise even with `'R'`, `OUTPUT` still works only for writing. If anyone has any information on other ways to use these options, I should be grateful if they could let me know.

'E' [End of File] in `OUTPUT` causes an existing file to be opened for update, starting at the end, i.e. to append data to an existing file. In `INPUT` causes the file to be opened with the cursor at the end of the file, though I can see no very obvious use for this.

'Q' [Quiet mode] No screen echo of input characters. (Useful only for binary use of file CON:).

'T' [Tab compression or expansion] Does nothing to binary files. Otherwise for `INPUT` tabs are replaced by spaces (tab stops every 8 characters). For `OUTPUT` strings of successive blanks are replaced by tabs where appropriate.

(9) Other file-handling functions etc

(a) `BACKSPACE(`*unit*`)` Moves the cursor back one record in the file opened as *unit*. In the case of a file opened as a text file (e.g. by `INPUT('IN',1, '20', 'FILENAME')`) the cursor is moved back to the start of a line. In the case of a file opened in raw ('binary') mode (e.g. by `INPUT('IN',1, 'B20', 'FILENAME')`), the cursor is moved back by the number of bytes specified as the record size (in this example, 20 bytes).

(b) `PATHNAME(`*unit*`)` Returns the file name that was used in opening the file opened as *unit*. Note that the name is given in the form used on opening the file. Thus the following two statements might well both open the same file:

```
INPUT('in',1,,'FILE')
```

and

```
INPUT('in',2,,'C:\Programs\SNOBOL\file')
```

but then `PATHNAME(1)` would return `'FILE'`, while `PATHNAME(2)` would return `'C:\Programs\SNOBOL\file'`.

If Units 5, 6, and 7 have been left with their default assignments, then `PATHNAME` will return respectively `'STDIN'`, `'STDOUT'`, and `'STDERR'`.

(c) `REWIND(`*unit*`)` Moves the cursor back to the beginning of the file opened with the given unit number.

(d) `SEEK(`*unit*`,` *integer*`)` Moves the cursor in the file opened with the given unit number to byte number *integer* in the file.

It is possible to use `SEEK` *(integer)* with a value of *integer* beyond the current end of file. If you subsequently write to the file, the file will be extended to the length required. Gordon Peterson says: "SEEK() allows you to seek past the physical end of file. If you then write to the file, it will be extended with random characters of data (in fact, typically, these will be the data formerly occupying those sectors, if any... (a good way to recover files which have been deleted

inadvertently but still are physically present on your hard drive)." However, I have been unable to produce this effect: in my experiments the extra length has always been made up by spaces.

(e) TELL(*unit*) Returns the position of the cursor in the file opened with the given unit number. The position is given as the number of bytes from the beginning of the file.

(f) TERMINAL This is a name that is initially associated with the Screen for output, and the Keyboard for input. For output it effectively duplicates 'SCREEN'. It was not included in early versions of SNOBOL4+, but it is in the version that has been made freely available.

(g) TRUNCATE(*unit*) Used to truncate a portion of a file opened for output. This deletes everything beyond the current position in the file.

(10) Program control functions

(a) COLLECT(*integer*) Normally, forces garbage collection and fails if less than *integer* descriptors of free storage remain. If *integer* is negative, no regeneration occurs but the function merely returns the amount of free storage presently available. Does NOT include storage that will be made available by the next garbage collection. [Information from Gordon Peterson]

(b) EXECUTE(*command*) Executes *command* as a DOS command line. *Command* may be an OS command, or another program. If SNOBOL is running under Windows in a DOS window, it may be a windows program. The SNOBOL program is resumed when the *command* has finished. If *command* is null, then the user is returned to the DOS command line. The user can then enter EXIT to resume the SNOBOL program. There is also a version EXECUTE(*command, path*) , in which *path* is an executable program file to run, and command is its command line. In this case the return value of the executed program is passed on as the return value of EXECUTE.

(c) SAVE(*unit*) Where *unit* is an output unit number, saves the program object code, program data, etc., so that it is possible subsequently to resume the program, without the source code. The program can then be resumed by typing 'SNOBOL4 *SaveFileName*' at the command line. If any of the command line options /C, /D, /K, /M, /NC, /NF, /NH, /P, /R, /S are in use, they will be recorded in the SAVE file and take effect when it is run.

(d) SETBREAK(*function_name*) Sets the given function to work as a handler for Control+C. If Control+C is typed, the given function is called. An example of the use of 'SETBREAK' can be seen in the version of "CODE.SNO" that comes with SNOBOL4+.

(e) STATEMENTS() If the parameter is zero or absent, this function returns the number of statements that have been executed, including the one that calls this function. It does not appear to me to do anything useful with other parameters.

(11) Arrays & tables

(a) As in SPITBOL, but not standard SNOBOL4, the notation A<I><J> or A[I][J] can be used instead of item(a<I>,J). Likewise F(X)<I> instead of ITEM(F(X),I). $A<I><J> means ($(A<I>))<J>.

(b) FREEZE() and THAW(). FREEZE() is used to prevent the creation of new table entries, and THAW() re-enables creation of new table entries. Normally, searching a table for an entry which does not exist will cause the entry to be added to the table, and given a null-string as the value. FREEZE() prevents this from happening.

(c) RSORT(*array*) works as SORT(*array*) (see below) but returns an array sorted into descending order.

(d) SORT() If *array* is a 1-dimensional array then SORT(*array*) will return the array sorted into ascending order.

If *table* is a table then SORT(*table*) will return an array. The dimensions of the array are (n, 2), where n is the number of non-null entries in the table. For each entry in the table, the array will contain an element indexed by [i,1] which is the table key, and an element indexed by [i,2] which is the corresponding table value. The array is sorted into order of the values.

(e) TABLE(*i1,i2,default*) An attempt to read a value not yet stored in the table, returns the third parameter, instead of the null string. Also, when a table is converted to an array, entries containing the default value will not be included.

(12) Command-line options & environment variables

(a) /A Shows 'product information'.

(b) /K*n*, where *n* is an integer, controls the maximum stack size. *n* must be from 1 to 34, and the default is 6.

(c) /M*n*, where *n* is an integer, limits the amount of memory the program can use to *n* kBytes. *n* must be at least 54 and not more than 320. The default is *n* = 320. It may sometimes be useful to use a smaller number to make memory available to other programs if using EXECUTE().

(d) /NC 'Non-Compatible' input and output mode. The format for the INPUT and OUTPUT functions becomes

INPUT(*name, unit, filename, options*)

instead of

INPUT(*name, unit, options, filename*).

(e) /NF This sets 'NOFAIL' mode. This means that any statement which fails and does not have a conditional goto will terminate the program with an error message. An unconditional goto does not prevent this. This mode can also be set by the –NOFAIL compiler directive within a program. In either case the mode can be turned off by –FAIL.

(f) /NN Prevents use of a numeric floating-point coprocessor, even if there is one.

(g) /P In Vanilla SNOBOL, the command-line option /P shows 'product information'. In SNOBOL4+ this function has been transferred to /A , and /P is used instead to enable "PLUSOPS" features, like the -PLUSOPS 1 compilation directive.

(h) /R changes the behaviour of the program if integer overflow occurs, i.e. an integer exceeds 32767. Without this option the program produces an error. With this option, the program converts the numbers to real and does not produce an error condition.

(i) /W=filename causes a 'Save' file to be saved after compilation. It has a similar effect to the use of the function 'SAVE()' inside the program. If the filename is not specified, the name of the snobol program source file is used, with the extension 'SAV'.If there is no source filename (which may be the case with entry from the keyboard) then the filename SNOBOL4.SAV is used.

(j) According to Gordon Peterson it is possible to set default command-line options, which will then be used in compiling all SNOBOL programs. This is done from the DOS command-line by using SET SNOCMD=options, e.g. SET SNOCMD=/L /P However, in my experience setting any options at all in this way results in SNOBOL simply printing out product information, without running the program. Perhaps I have misunderstood how to do this, though I have tried everything I can think of. If anyone can enlighten me, I shall be grateful.

(k) `SET SNOLIB=`*directory* specifies a directory which will be searched for `INCLUDE` modules and `LOAD()` modules if they are not found in the current directory.

(l) `ENVIRONMENT(`*keyword*`)` This function is used to look up a DOS environment variable. Thus ENVIRONMENT('PATH') returns the DOS PATH string. The name of the DOS variable (e.g. PATH) must be given in upper case, otherwise the function fails.

(13) Miscellaneous

(a) Normally a string which is to be converted to a numerical value cannot have leading or trailing blanks. With `-PLUSOPS` active, leading and trailing blanks are permitted.

(b) SNOBOL4+ returns an ERRORLEVEL of 1 on compilation errors, or 2 if the source file could not be found.

(c) `LOAD()` This is used to call an external function. This function will have been written in another language and compiled into machine code.

(d) In Vanilla SNOBOL, the total size of the object program and data cannot exceed 30K bytes. In SNOBOL4+ this limit is raised to 320K bytes.