## NAME

snobol4 − SNOBOL4 interpreter

## SYNOPSIS

**snobol4** [ *options* . . . ] [ *file(s)* . . . ] [ *params* . . . ]

## DESCRIPTION

This manual page describes a port of the original Bell Telephone Labs (BTL) Macro Implementation of SNOBOL4 to machines with ILP32 (32-bit int/long/pointer) or LP64 (64-bit long/pointer) C compilers by Philip L. Budne. The language and its implementation are described in [1] and [2]. Extensions from Catspaw SNOBOL4+, SPITBOL and SITBOL have been added. This page discusses only the changes/extensions.

### Limitations

All aspects of the language are implemented except:

- Trapping of arithmetic exceptions.
- **LOAD()** can be used to access external functions on most platforms, but not all. External functions can be statically linked (poor man's loading) into the snobol4 executable on **ALL** platforms. See /usr/local/lib/snobol4/*version*/load.txt for more information.

### Changes

The following behaviors have been changed from the original Macro SNOBOL4;

- Listings are disabled by default. Default listing side (when enabled by **-LIST** or the **-l** command line option is **LEFT**. Listings are directed to standard output (or file specified by the **-l** command line option.
- Error messages, the startup banner and statistics are directed to standard error. Compilation error messages (including erroneous lines) appear on standard error as well as in the listings. Error messages now reference the source file name and line number.
- Character set (see below).
- The **PUNCH** output variable no longer exists (see **TERMINAL** variable below).
- I/O is not performed using FORTRAN I/O. The 3rd argument to the **OUTPUT()** and **INPUT()** functions is interpreted as a string of I/O options (see below).
- Control lines and comment characters are valid after the end of string (;) statement separator. Listing statement numbers show the statement number of the LAST statement on the line (rather than the first).
- Setting the **&ABEND** keyword causes a core dump upon termination!
- The value of the **&CODE** keyword determines the exit status of the **snobol4** application.
- The **DATE()** function returns strings of the form: *MM/DD/YYYY HH:MM:SS*. See extensions section for arguments to the **DATE()** function.
- Keyword **&STLIMIT** now defaults to -1. When **&STLIMIT** is less than zero there is no limit to the number of statements executed, and **&STCOUNT** is not incremented.
- **VALUE** tracing applies to variables modified by immediate value assignment ($ operator) and value assignment (. operator) during pattern matching.
- The **BACKSPACE()** function is not implemented. Use the **SET()** function instead.
- I/O unit numbers up to 256 can be used.
- Attempts to output on a closed unit generates a fatal "Output error".

**Character set**

**snobol4**(1) is 8-bit clean, and uses the native character set. Any 8-bit byte is accepted as a SNOBOL datum or in a string constant of a SNOBOL source program. The value of the SNOBOL protected keyword **&ALPHABET** is a 256-character string of all bytes from 0 to 255, in ascending order.

On ASCII-based systems, any character with the 8th bit set is treated as "alphabetic", and can start, or be used in identifiers and labels. This includes characters from the "upper half" of national character sets and all bytes resulting from the UTF-8 encoding of Unicode characters.

Programs may be entered in mixed case; By default lower case identifiers are folded to upper case (see **&CASE** and **−CASE** extensions below). Case folding is performed by using the C library **islower**(3) test, and then using **toupper**(3) to convert the lower-case characters to upper case. When using UTF-8 encoded characters in code, case folding should be disabled, to prevent any bytes which appear to be lower case in the current locale from being modified.

The following operator character sequences are permitted and represent a cross between PDP-10 Macro SNOBOL, SITBOL and Catspaw SPITBOL usage:

| | |
|---|---|
| Exponentiation: | ^ ** |
| Alternation: | \| ! |
| Unary negation: | ~ \ |
| Assignment: | = _ |
| Comment line: | * # \| ; ! |
| Continuation line: | + . |

Both square brackets ([]) and angle brackets (<>) may be used to subscript arrays and tables. The **TAB** (ASCII 9) character is accepted as whitespace. Note that the use of the pound sign for comments allows use of the shell interpreter sequence at the top of a file (e.g., "#!/usr/local/bin/snobol4 -b"). Underscore (_) and period (.) are legal within identifiers and labels.

**Extensions**

**ARRAY/TABLE** access

Multiple **ARRAY** and/or **TABLE** index operations may appear in a row, without having to resort to use of the **ELEMENT** function, so long as no intervening spaces (or line continuations) appear.

**BREAKX()**

The **BREAKX()** function is a pattern function used for fast scanning. **BREAKX(***str***)** is equivalent to **BREAK(***str***) ARBNO(LEN(1) BREAK(***str***))**. In other words BREAKX matches a sequence of ever larger strings terminated by a break set. BREAKX can be used as a faster matching replacement for **ARB**: **BREAKX('S') 'STRING'** always runs faster than **ARB 'STRING'** since it only attempts matching **'STRING'** at locations where an **'S'** has been detected.

Case folding

By default the compiler folds identifiers and directives to upper case, so programs can be entered in either case. To disable case folding use the directive **−CASE 0 or −CASE**. To re-enable case folding use directive **−CASE** *n* where *n* is a non-zero integer. The status of case folding may be examined and controlled from a running program by the unprotected system keyword **&CASE**.

**CHAR()**

The **CHAR()** function takes an integer from 0 to 255 and returns the nth character in **&ALPHABET**.

**DATE()**

For compatibility with new versions of Catspaw SPITBOL, **DATE(0)** returns strings of the form *MM/DD/YY HH:MM:SS*, and **DATE(2)** returns strings of the form *YYYY-MM-YY HH:MM:SS*. With any other arguments **DATE()** returns strings of the form *MM/DD/YYYY HH:MM:SS*.

**&DIGITS**

The protected keyword **&DIGITS** contains the string **0123456789** for compatibility with the Unicon/Icon languages!

**−ERROR/−NOERRORS**

Directives **−ERROR** and **−NOERRORS** control execution of program with compiler errors. If the **−ERROR** directive is given, the program will be executed (but any attempt to execute a statement with a compiler error will cause a fatal execution error). By default programs with compiler errors will not be started, this can be restored using **−NOERRORS**.

**&ERRTEXT**

After a statement with an error has been handled due to a non-zero value in **&ERRLIMIT**, the protected keyword **&ERRTEXT** will contain the error message. See the **DIAGNOSTICS** section below for explanations of errors, and whether they are fatal or not.

**−EXECUTE/−NOEXECUTE**

Directives **−EXECUTE** and **−NOEXECUTE** control execution of programs. If the **−NOEXECUTE** directive is given, the program will be not executed after compilation. **−EXECUTE** cancels any previous **−NOEXECUTE**.

**FILE_ABSPATH()/FILE_ISDIR()**

**FILE_ABSPATH(***string***)** is a predicate which returns the null string if its argument is an absolute file path, and returns failure if the path is relative. **FILE_ISDIR(***string***)** is a predicate which returns the null string if its argument is the pathname of a directory, and returns failure if not.

**FREEZE()/THAW()**

The **FREEZE()** function prohibits creation of new entries in the referenced table. This is useful once a table has been initialized to avoid creating empty entries on lookups that fail. This can greatly improve program speed, since frozen tables will not become clogged with empty entries. Lookups for uninitialized entries will return the null string. Attempts to assign to a non-existent entry will cause a "Variable not present where required" error. The **THAW()** function restores normal entry creation behavior.

**FUNCTION()**

The **FUNCTION()** predicate evaluates its argument as a string (with case folding), and returns the null string if a function with that name exists and fails if it does not. The **FUNCTION()** predicate exists in SITBOL.

**&GCTIME**

The protected keyword **&GCTIME** contains a **REAL** number of milliseconds of execution time spent in the garbage collector.

**&GTRACE**

If the unprotected keyword **&GTRACE** set to a non-zero value, each time a garbage collection is run, a trace message is output indicating the source file and line number of the current statement, how long the GC took, and how many units of storage are now free. If positive, the value of **&GTRACE** will be decremented after it is tested.

**-HIDE** The **-HIDE** directive stops listing, and assignment of new statement numbers. It is used to hide the source code for the **sdb**(1) source code debugger.

**HOST()**

A limited simulation of the SPITBOL **HOST()** function (with liberal implementation specific extensions) is included.

The **-INCLUDE** file **host.sno** contains symbolic defines for these (and **many** other) function codes.

**HOST()** with no parameters returns a string describing the system the program is running on. The string contains three parts, separated by colons. The first part describes the physical architecture, the second describes the operating system, and the third describes the language implementation name. NOTE! Architecture names come from the **uname**(3) library call, and may be different for the same hardware when running different operating systems. An example value: *i386:FreeBSD 10.0-ALPHA2:CSNOBOL4 1.5*

**HOST(0)** returns a string containing the command line parameter supplied to the **-u** option, if any. If no **-u** option was given, **HOST(0)** returns the concatenation of all user parameters following the input filename(s).

**HOST(1,*string*)** passes the string to the **system**(3) c library function, and returns the subprocess exit status.

**HOST(2,*n*)** for integer *n* returns the *n*'th command line argument (regardless of whether the argument was the command name, an option, a filename or a user parameter) as a string, or failure if *n* is out of range.

**HOST(3)** returns an integer for use with **HOST(2)** indicating the first command line argument available as a user parameter.

**HOST(4,*string*)** returns the value of the environment variable named *string*.

**−INCLUDE**

The **−INCLUDE** directive causes the compiler to interpolate the contents of the named file enclosed in single or double quotes. Any filename will be included only once, this can be overridden by appending a trailing space to the filename. Trailing spaces are removed from the filename before use. If the file is not found in the current working directory an attempt will be made to find it in the directories specified in the search path. **−COPY** is a synonym for **−INCLUDE** for compatibility with SPITBOL/370.

**IO_FINDUNIT()**

The **IO_FINDUNIT()** function returns an unused I/O unit number for use with the **INPUT()** or **OUTPUT()** functions. **IO_FINDUNIT()** is meant for use in subroutines which can be reused. **IO_FINDUNIT()** will never return a unit number below 20.

**LABEL()**

The **LABEL()** predicate evaluates its argument as a string (with case folding), and returns the null string if a label with that name has been defined, and fails if it does not. The **LABEL()** predicate was copied from Steve Duff's version of Macro SPITBOL. SITBOL has a **LABEL function that takes an optional line offset** from the label and returns a **CODE** pointer.

**&LINE/&FILE/&LASTLINE/&LASTFILE**

The **&LINE** and **&FILE** keywords can be used to determine the source file and file line associated with the current statement. The **&LASTLINE** and **&LASTFILE** return the source file and file line associated with the previous statement (and are useful in **TRACE** and **SETEXIT** handlers.

**−LINE**

The **−LINE** directive can be used to alter SNOBOL's idea of the current source file and line (ie; for use by preprocessors). **−LINE** takes a line number and an optional quoted string filename.

Lexical comparison

A full set of lexical (string) comparison predicates have been added to complement the standard **LGT()** function; **LEQ()**, **LGE()**, **LLE()**, **LLT()**, **LNE()**.

**LPAD()/RPAD()**

The **LPAD()** and **RPAD()** functions take the first argument (subject) string, and pad it out to the length specified in the second argument, using the first character of the optional third argument. If the third argument is missing, or is the null string, spaces will be used for padding. The subject will be returned unmodified if already long enough.

**&MAXINT**

The protected keyword **&MAXINT** contains the largest positive integer value that can be represented by the **INTEGER** data type.

Named files

Filenames can be supplied to the **INPUT()** and **OUTPUT()** functions via an optional fourth argument.

The filename - (hyphen) is interpreted as stdin on **INPUT()** and stdout on **OUTPUT()**.

If the filename begins with a vertical bar (|), the remainder is used as a shell command whose stdin (in the case of **OUTPUT()**) or stdout (in the case of **INPUT()**) will be connected to the file variable via a pipe. If a pipe is opened by INPUT() input in "update" mode, the connection will be bidirectional (on systems with socketpair and Unix-domain sockets). See below for how to associate a variable for I/O in both directions.

If the filename begins with two vertical bars (||) the remainder is used as a shell command executed with stdin, stdout and stderr attached to the slave side of a pseudo-terminal (pty), if the system C library contains the **forkpty**(3) routine. Use of ptys are necessary when the program to be invoked cannot be run without a "terminal" for I/O. See below on how to properly associate the I/O variable.

The magic filenames **/dev/stdin**, **/dev/stdout**, and **/dev/stderr** refer to the current process standard input, standard output and standard error I/O streams respectively regardless of whether those special filenames exist on your system.

The magic pathname **/dev/fd/***n*, opens a new I/O stream associated with file descriptor number *n*.

The magic pathname **/tcp/***hostname*/*service* can be used to open connection to a TCP server. If the path ends in the optional suffix **/priv** the local address will be bound to a port number below 1024, if process privileges allow. **/udp/***hostname*/*service* behaves similarly for UDP.

The magic pathname **/dev/tmpfile** opens an anonymous temporary file for reading and writing, see **tmpfile**(3).

On VMS, Win32, and MS-DOS the pathnames **/dev/null** and **/dev/tty** are magical, and refer to the null device, and the user's terminal/console, respectively.

**ORD()**

The **ORD()** function returns the ordinal value (zero to 255) of the first character in its string argument (the inverse of the **CHAR()** function).

**&PARM**

The entire command line is available via the **&PARM** protected keyword for compatibility with Catspaw SNOBOL4+. Use of the SPITBOL compatible **HOST()** function is preferable, as it makes it possible to tell if a positional parameter containing spaces was passed in using shell quote characters.

**&PI**

The protected keyword **&PI** contains value of the transendental number $\pi$.

**REAL** numbers in **INTEGER** contexts

**REAL** numbers (or strings convertible to **REAL**) are accepted in all contexts which previously required an **INTEGER** (or string convertible to **INTEGER**). Contexts include **TABLE()**, **ITEM()**, array indices, **INPUT()**, **OUTPUT(), SET()**, keyword values, **CHAR()**, **RPAD()**, **LPAD()**, **FIELD()**, **COLLECT()**, **DUMP()**, **DUPL()**, **OPSYN()**, **SUBSTR()**.

**REVERSE()**

**REVERSE()** returns its subject string in reverse order.

Scientific notation

**REAL** number syntax has been expanded to allow exponents of the form:
*ANY('Ee') ('+' | '-' | '') SPAN('0123456789')*. Exponential format reals need not contain a decimal point.

**SERV_LISTEN()**

The **SERV_LISTEN()** function makes SNOBOL4 into a network server process, and takes three STRING arguments: FAMILY, TYPE, SERVICE. FAMILY must be either "inet" for an Internet Protocol v4 socket, "inet6" for an Internet Protocol v6 socket, or "unix" for a local ("unix domain") socket. The second argument, TYPE must be "stream", and the third argument, SERVICE must be a port number or service name (for an internet socket), or a pathname (for a "unix" socket). SERV_LISTEN() listens for incoming requests, accepts them, then “forks” a child process and returns an integer file descriptor which can be opened for bidirectional I/O using a "/dev/fd/n" magic pathname. The original (“parent”) process never returns from the SERV_LISTEN() call. This function is only available on systems with the “fork” system call, which makes a child process which is an identical copy of the parent process.

**SET()**

The **SET()** function can be used to seek the file pointer of an open file. The first argument is an I/O unit number, the second is an integer offset. The third argument, an integer determines from **whence** the file pointer will be adjusted. If whence is zero the starting point is the beginning of the file, if whence is one, the starting point is the current file pointer, and if whence is two, the starting point is the end of the file. **SET()** returns the new file pointer value. On systems with 64-bit file pointers and 32-bit integers (ie; 4.4BSD on i386) the return value will be truncated to 32-bits, and only the first and last 4 gigabytes of a file can be accessed directly.

**SITBOL** file functions

**FILE(***string***)** is a predicate which returns the null string if its argument is the name of a file that exists, and fails if it does not. **DELETE(***string***)** is a predicate which tries to remove the file named by its argument, and fails if it cannot. **RENAME(***string1***,***string2***)** is a predicate which attempts to rename the file named by *string2* to the file named by *string1*. Unlike the **SITBOL** version, if the target file exists, it will be removed.

**SNOBOL4+** real functions

**EXP()**, **LOG()** and **CHOP()** functions are available for compatibility with SNOBOL4+. **EXP()** returns the value *e ** x*, **LOG()** returns the natural logarithm of its REAL argument, and **CHOP()** truncates the fractional part of its REAL argument (rounding towards zero), and returns a REAL. **LN()** is an alias for **LOG()**, for compatibility with Catspaw SPITBOL.

**SORT()/RSORT()**

The **SORT()** and **RSORT()** functions take two arguments. The first can be either an array or a table. If the first argument is an array, it may be singly-dimensioned in which case the second argument, if non-null should indicate the name of a field of a programmer defined data type to use to access the sort key. Otherwise the first argument should be a table or a doubly-dimensioned array, in which case the second argument may an integer indicating the array column on which to sort. If the second argument is null, it is taken to be 1. The array (or table) is not modified; a new array is allocated and returned. **SORT()** sorts elements in ascending order, while **RSORT()** sorts in descending order.

Example: for a table *TAB* of integers, indexed by strings being used to tabulate word counts *FREQ = RSORT(TAB,2)* returns an array such that *FREQ<1,1>* contains the most frequent word while *FREQ<1,2>* contains the number of occurrences of that word. While *WORDS = SORT(TAB,1)* returns an array with the rows by the lexicographical ordering of the words; *WORDS<1,1>* contains the lexicographically first word and *WORDS<1,2>* contains the number of occurrences of that word.

**SPITBOL** operators

The SPITBOL scan (?) and assignment (=) operators have been added. A pattern match can appear within an expression, and returns the matched string as its value. Similarly assignment can appear in an expression, and returns the assigned value. An assignment after a scan (ie; *STRING ? PATTERN = VALUE*) performs a scan and replace. Assignment is right associative, and has the lowest precedence, while scan is left associative and has a precedence just higher than assignment.

The SPITBOL selection/alternative construction can be used in any expression. It consists of a comma separated list of expressions inside parentheses. The expressions are evaluated until one succeeds, and its value is returned. Abuse of this construction may result in incomprehensible code.

The type **NUMERIC** with **CONVERT()** and the removal of leading spaces from strings converted to numbers (implicitly or explicitly) are also legal when SPITBOL extensions are enabled. SPITBOL extensions can be enabled and disabled using the **−PLUSOPS** directive. **−PLUSOPS 0** or **−PLUSOPS** disables SPITBOL operators, while **−PLUSOPS** *n* where *n* is a non-zero integer enables them. SPITBOL extensions are enabled by default.

SPITBOL **SETEXIT()** function

The argument to **SETEXIT()** is a label to which control is passed if a subsequent error occurs, providing that the value of the keyword **&ERRLIMIT** is non-zero. The value of **&ERRLIMIT** is decremented when The error trap occurs. A **SETEXIT()** call with a null argument causes cancellation of the intercept. A Subsequent error will terminate execution as usual with an error message. The result returned by **SETEXIT()** is the previous intercept setting (i.e., a label name or null if no intercept is set). This can be used to save and restore the **SETEXIT()** conditions in a recursive environment. The error intercept routine may inspect **&ERRTYPE** and take one of the following actions:

1. Terminate execution by transferring to the special label **ABORT**. This causes error processing to resume as though no error intercept had been set.

2. Branching to the special label **CONTINUE** causes execution to resume by taking the failure exit of the statement in error.

3. Branching to the special label **SCONTINUE** causes execution to resume at the point of interruption.

4. If the error occurred inside a function (**&FNCLEVEL** is non-zero), branch to label **RETURN**, **FRETURN**, **NRETURN**.

The occurrence of an error cancels the error intercept. The error intercept routine must reissue the **SETEXIT()** Error handlers cannot be nested: only one copy of the saved execution state is kept.

**SQRT()**

The **SQRT()** function is available for compatibility with SPARC SPITBOL. **SQRT()** fails if the argument is negative, but does not cause a fatal error.

**SSET()**

Experimental "scaled set" function. Takes arguments unit, offset, whence, and scale. The first three are analogous to the same arguments for the **SET()** function. The last parameter is used as a multiplicative scaling factor on the **offset** parameter, and as a divisor on the return value. When used in combination with relative **SET()** calls (whence of one), any file offset can be achieved, even when system file offsets are larger than can be represented in a SNOBOL4 **INTEGER**. Support for "Large Files" is enabled when available, but not all file systems support them.

**&STEXEC**

The protected keyword **&STEXEC** contains the number of statements executed, regardless of the value of **&STLIMIT**.

**SUBSTR()**

**SUBSTR()** takes a subject string as its first argument, and returns the substring starting at the position specified by the second argument (one-based) with a length specified by the third argument. If the third argument is missing or zero, the remainder of the string is returned.

**TERMINAL** I/O variable

The variable **TERMINAL** is associated with the standard error file descriptor for both input and output.

**TRACE()** function type argument

The second argument of the **TRACE()** function can be abbreviated to a single letter: **C** (CALL), **F** (FUNCTION), **K** (KEYWORD), **L** (LABEL), **R** (RETURN), or **V** (VALUE) as in Macro SPITBOL.

Trig functions

**ATAN()**, **SIN()**, **COS()** and **TAN()** functions are available for compatibility with SPARC SPITBOL. **SIN()**, **COS()** and **TAN()** take arguments in radians. **ATAN()** returns the principal value of the arc tangent of its REAL argument.

**&UCASE/&LCASE**

Protected keywords **&UCASE** and **&LCASE** contain upper and lower case characters respectively.

**VDIFFER()**

The **VDIFFER()** function takes two arguments, if they **DIFFER()**, the first argument's value is returned. This is intended to be used in contexts where *DIFFER(X) X* would otherwise have been used. The **VDIFFER()** function was copied from Steve Duff's version of Macro SPITBOL.

## I/O Associations

I/O is performed by associating a variable name with a numbered I/O unit using the **INPUT()** and **OUTPUT()** functions. The following associations are available by default;

| Variable | Unit | Association |
|---|---|---|
| **INPUT** | 5 | standard input |
| **OUTPUT** | 6 | standard output |
| **TERMINAL** | 7 | standard error (output) |
| **TERMINAL** | 8 | /dev/tty (input) |

## I/O Options

The third argument of the **INPUT()** and **OUTPUT()** functions is interpreted as a string of single letter options, commas are ignored. Some options effect only the I/O variable named in the first argument, others effect any variable associated with the unit number in the second argument.

*digits* A span of digits will set the input record length for the named I/O variable. This controls the maximum string that will be returned for regular text I/O, and the number of bytes returned for binary I/O. Record length is per-variable association; multiple variables may be associated with the same unit, but with different record lengths. The default record length for input is 1024; longer lines will be silently truncated.

**A** For **OUTPUT()** the unit will be opened for append access (no-op for **INPUT()**).

**B** The unit will be opened for binary access. On input newline characters have no special meaning; the number of bytes transferred depends on record length (see above). On output no newline is appended. For terminal devices, all I/O to this unit will be done without special processing for line editing or EOF, while characters which deliver signals (interrupt, kill, suspend) are still processed. Units opened on the same terminal device entry operate independently; some can use binary mode, while others operate in text mode.

**C** Character at a time I/O. A synonym for **B,1**.

**T** Terminal mode. No newline characters are added on output, and any newline characters are returned on input. Terminal mode effects only the referenced unit. Terminal mode is useful for outputting prompts in interactive programs.

**Q** Quiet mode. Turns off input echo on terminals. Effects only input from this unit.

**U** Update mode. The unit is opened for both input and output. Here is how to associate a variable for I/O in both directions:

```
unit = IO_FINDUNIT()
INPUT(.name, unit, 'U', 'filepath')
OUTPUT(.name, unit)
```

Useful situations for this when filepath is **/dev/fd/***n* where *n* is a file descriptor number returned by **SERV_LISTEN,** filepath specifies a pipe (|command) or pseudo-terminal (||command) paths. The above sequence is also useful with when combined with fixed record length, binary mode and the **SET()** function for I/O to preexisting files. Performing **OUTPUT()** first will create a regular file if it does not exist, but will also truncate a preexisting file!

**W** Unbuffered writes. Each output variable assignment causes an I/O transfer to occur, rather than collecting the data in a buffer for efficiency.

## OPTIONS

**-b** Toggle startup banner output (by default on).

**-d** *DDD* Allocate "dynamic storage" region of *DDD* descriptors for program code and data. A suffix of **k** multiplies the number by 1024, a suffix of **m** multiplies the number by 1048576. A larger dynamic region may result in fewer garbage collections (storage regenerations), however large values may cause execution to slow down when large amounts of garbage collect. Most programs do not need an increased dynamic region to run. If your program terminates with an "Insufficient storage to continue" message you need to increase the dynamic storage region size. The **-h** option displays the default dynamic region size.

**-f** Toggle folding of identifiers to upper case (see **−CASE** and **&CASE**).

**-g** Enable garbage collection tracing (sets **&GTRACE** to -1).

**-h** Give help. Shows usage message, includes default sizes for "dynamic region" and pattern match stack.

**-k** Toggle running programs with compilation errors (see **−ERROR** and **−NOERRORS** extensions). By default programs with compilation errors will not be run.

**-l** *LISTINGFILE* Enable listing output to the named file. The default listing side is **LEFT**.

**-n** Toggle running programs after compilation (see **−EXECUTE** and **−NOEXECUTE** extensions). By default programs are run after compilation.

**-p** Toggle SPITBOL extensions (also controlled by **−PLUSOPS**).

**-r** Toggle reading **INPUT** from input file(s) after **END** label. Otherwise **INPUT** defaults (back) to standard input after program compilation is complete.

**-s** Toggle termination statistics (off by default).

**-u** *params* specifies a parameter string available via **HOST(0)**.

**-v** Show version and exit.

**-z** Show directory search path in use (including **-I** options) and exit.

**--** Terminates processing items as options. Any remaining strings are treated as files or user parameters.

**-I** *directory* Appends directory argument to include search path (before elements from the **SNOLIB** or **SNOPATH** environment variables).

**-L** *file* Load source file before others on command line. May be used multiple times. Used to preload the **sdb**(1) source code debugger.

**-M** Specifies that all items left on the command line after option processing is complete are to be treated as filenames. The files are read in turn until an **END** statement is found (Any remaining data is available via the **INPUT** variable if the **-r** option is also given). A **--** terminates processing of arguments as files, and makes the remaining arguments available as user parameters (see the **HOST()** function).

**-P** *DDD* Allocate *DDD* descriptors for the pattern match stack. A suffix of **k** multiplies the number by 1024, a suffix of **m** multiplies the number by 1048576. The pattern match stack is used to save backtracking and conditional assignment information during pattern matching. If your program terminates with an "Overflow during pattern matching" message (Error 16) you need to increase the pattern match stack size. The **-h** option displays the default pattern match stack size.

**-S** *DDD* Allocate *DDD* descriptors for the interpreter stack. A suffix of **k** multiplies the number by 1024, a suffix of **m** multiplies the number by 1048576. The interpreter stack is used for saving data, and passing parameters to internal procedures. If your program terminates with an "Stack overflow" message (Error 21) you need to increase the interpreter stack size. A common reason for needing additional stack space is for tracing deeply nested DATA() structures during garbage collection. The **-h** option displays the default interpreter stack size.

**Directory Search List**

Files specified in **-INCLUDE** directives and **LOAD()** function calls are first checked for in the current working directory. If not found, the following directory search order is used: Any directories specified on the command line using **-I** options, in the order specified; The list of directories from the **SNOPATH** environment variable, if defined; If **SNOPATH** is not defined, the **SNODIR** environment directory (or a compiled in default) is used to add the following:

SNODIR/VERSION/local

SNODIR/VERSION

SNODIR/local

SNODIR

## ENVIRONMENT

**SNOPATH**

Is a list of directories delimited by colons (semi-colons on VMS and Windows) appended to the Directory Search List (see above). NOTE!! **SNOPATH**, **SNOBOL_PRELOAD_PATH** introduces security and portability issues!!

**SNOBOL_PRELOAD_PATH**

Is a list of source files delimited by colons (semi-colons on VMS and Windows) that will be read before the program source. Like **SNOPATH**, **SNOBOL_PRELOAD_PATH** introduces security and portability issues.

**SNOLIB**

(the sole search directory in versions of CSNOBOL4 prior to version 1.5) is used to establish the base libary path if **SNOPATH** is not defined. See Directory Search List above.

## SEE ALSO

**sdb**(1),**snobol4dbm**(3),**snobol4tcl**(3),**snobol4time**(3),**snolib**(3).

http://www.snobol4.org

All things SNOBOL4 related.

http://www.snobol4.com
Catspaw: commercial SPITBOL implementations, Free SNOBOL4+ for DOS.

http://www.snobol4.org/doc/burks/tutorial/contents.htm
SNOBOL4 language tutorial (from Catspaw Vanilla SNOBOL4)

http://www.snobol4.org/doc/burks/manual/contents.htm
Catspaw Vanilla SNOBOL4 manual.

ftp://ftp.snobol4.com/spitman.pdf
Catspaw Macro SPITBOL manual

[1] R. E. Griswold, J. F. Poage, and I. P. Polonsky
*The SNOBOL4 Programming Language*, 2nd ed., Prentice-Hall Inc., 1971.
*(aka the green book)*
ftp://ftp.cs.arizona.edu/snobol4/gb.pdf

[2] R. E. Griswold,
*The Macro Implementation of SNOBOL4*, W. H. Freeman and Co., 1972.
Book describing the implementation techniques used in Macro SNOBOL4.

http://www.snobol4.org/docs/arizona/
University of Arizona SNOBOL4 memos, formatted as PDF files by Scott Marovich. Includes memo s4d58, which describes each pseudo-instruction in the Snobol Implementation Language (SIL), corrigendae for [1], [2] and *String and List Processing in SNOBOL4*.

## AUTHOR

Philip L. Budne

with help from:

R. E. Griswold, J. F. Poage, and I. P. Polonsky

Mark Emmer (code from SNOBOL4+)

Viktors Berstis (code from Minnesota SNOBOL4)

## DIAGNOSTICS

SNOBOL4 errors shown with **&ERRTYPE** values. Errors marked *fatal* cannot be curtailed by setting **&ERRLIMIT**.

1 "Illegal data type"

2 "Error in arithmetic operation"

3 "Erroneous array or table reference"

4 "Null string in illegal context"

5 "Undefined function or operation"

6 "Erroneous prototype"

7 "Unknown keyword"

8 "Variable not present where required"

9 "Entry point of function not label"

10 "Illegal argument to primitive function"

11 "Reading error"

12 "Illegal i/o unit"

13 "Limit on defined data types exceeded"

14 "Negative number in illegal context"

15 "String overflow"

16 "Overflow during pattern matching" *fatal.* See the **-P** option.

17 "Error in SNOBOL4 system" *fatal.* Various internal errors.

18 "Return from level zero" *fatal.*

19 "Failure during goto evaluation" *fatal.*

20 "Insufficient storage to continue" *fatal.* See the **-d** option.

21 "Stack overflow" *fatal.* See the **-S** option.

22 "Limit on statement execution exceeded" *fatal.*

23 "Object exceeds size limit" *fatal.*

24 "Undefined or erroneous goto" *fatal.*

25 "Incorrect number of arguments" *fatal.*

26 "Limit on compilation errors exceeded" *fatal.*

27 "Erroneous END statement" *fatal.*

28 "Execution of statement with compilation error" *fatal.* Last error in standard SNOBOL4.

29 "Erroneous INCLUDE statement" *fatal.*

30 "Cannot open INCLUDE file" *fatal.*

31 "Erroneous LINE statement" *fatal.*

32 "Missing END statement" *fatal.*

33 "Output error"

34 "User interrupt" Interrupt character (SIGINT) was recieved.

35 "Not in a SETEXIT handler" Attempt to branch to **CONTINUE**, **SCONTINUE**, or **ABORT** when no unhandled error condition present.

## BUGS

I/O retains some record oriented flavor.

I/O is still tied to unit numbers.

“Dynamic” storage cannot be expanded after startup.

Integer math can never “fail”, even on overflow.

Oversize integer constants may not be detected.

This manual page is too long, and should be split up!

There should be a **snobol4func**(1) man page which describes all intrinsic functions (this page only includes extensions).