Vanilla Snobol4

Tutorial and Reference Manual

Mark B. Emmer
[marke@snobol4.com]

Of what this is and how it came into being

This manual is the user documentation that accompanies Vanilla Snobol,[1] a slightly cut-down version of Snobol4+.[2] Both Vanilla Snobol and Snobol4+ are Catspaw Inc.'s[3] 16-bit MS-DOS implementations from the 1980s of the Snobol programming language. Snobol4+ used to be a commercial product, with Vanilla Snobol its free version.

The subsequent release of Snobol4+ also as a free program came with no documentation, but the programmers could still use the one from Vanilla Snobol. Indeed, the only truly essential feature that Vanilla Snobol omits with respect to Snobol4+ or the Snobol language itself, is floating-point numbers. The Vanilla Snobol manual was so well written that it became, and remained, very popular. Eventually, it has been acknowledged by the Snobol community as an excellent source on the language and its use – a source worth keeping available, even though those two particular Snobol implementations are no more widely used.

Within Vanilla Snobol, the manual is a single file in plain ASCII form, broken down into 144 pages, each with a header and footer.[4] The manual can be searched within and printed but is not particularly good for reading and browsing as it is entirely monotonous as regards typography.

Some years ago, HTML versions of the tutorial[5] and reference[6] parts were created. By breaking the manual into chapters, providing hyperlinks to chapters, sections and subsections, properly marking the titles of all levels, and highlighting displayed program code through showing it in a 'computer' (monospaced) font, the manual became much more easier to browse, read, and navigate in.

However, a demanding reader – one used to high typographical and usability standards – would want more, such as:

- full and correct bookmarking;
- better typographical quality, in particular – enhancing the visibility of all parts of the text by properly highlighting them; e. g., inlined program code should be set in a 'computer' font, just like displayed code;
- ability to search throughout the whole text automatically (lost in the HTML version as it consists of many files rather than one);
- ability to print the manual or parts of it, including breaking it down into pages (this, too, apparently lost in the HTML version); printed text should be scalable and reproducible with utmost quality.

All this was made possible by redoing the manual in LaTeX. A PDF output from LaTeX is equally good to read on a computer screen and print on paper, and can be properly bookmarked and hyperlinked.

Creating the LaTeX version of the Vanilla Snobol manual was much easier by using the HTML version as source than it would have been starting from the plain ASCII version. The transformation from HTML to LaTeX was done with `gnuhtml2latex`. Before applying it, the HTML files were run through `tidy`, to ensure the source markup was syntactically correct.

Even though the rough work for obtaining LaTeX files was thus automated, the result required a huge number of manual additions, corrections, improvements and other necessary changes. These mainly concerned markup, but also sometimes the content itself. For example, some ways of denoting certain entities were apparently devised for the ASCII version but were too rough to stick with in a finely typeset document. The result of manually, and thoroughly, reworking the manual was, on the one hand, removing many inconsistencies, inadequacies and plain defects ensuing from the HTML-to-LaTeX conversion or the preceding plain-text-to-HTML transformation, or originating from the plain text version; and, on the other hand, vastly improved readability.

[1] ftp://ftp.snobol4.com/vanilla.zip
[2] ftp://ftp.snobol4.com/snobol4p.zip
[3] http://www.snobol4.com
[4] Actually, there is a program which, when run, produces the manual
[5] http://www.math.bas.bg/bantchev/place/snobol/vtutor/contents.htm
[6] http://www.math.bas.bg/bantchev/place/snobol/vrefman/contents.htm

The original ASCII manual contains a part named *Getting started* that was dropped in the HTML release. In the present edition, it is back again. Talking of $5\frac{1}{4}$ diskettes and other bits that a today's reader might not have even heard of, it is perhaps not wery useful to them, but it is precious as a mark of the time when the document was written.

Those interested in the differences between VANILLA SNOBOL and SNOBOL4+ may wish to check an independently written document.[7]

Finally, it is worth noting that the present manual was the base of the Macro SPITBOL manual. Macro SPITBOL is a compiler of a dialect of SNOBOL4 and in many ways is a successor to SNOBOL4+. At present it is freely available,[8] along with the manual. The latter is much larger than the VANILLA SNOBOL's manual, but has the same structure and is of the same remarkable quality.

I hope the reader will find, as I do, the work on recreating this text worth the time and effort. Enjoy reading!

February 2011 Boyko Bantchev

[7] http://www.sachsdavis.clara.net/Snobol4Pdoc.html

[8] https://code.google.com/archive/p/spitbol, http://github.com/spitbol

Part I

Getting Started

Chapter 1

Getting started

Welcome to the world of SNOBOL4! It's a world where you can manipulate text and search for patterns in a simple and natural manner. SNOBOL4 is a completely general programming language, and its magic extends far beyond the world of text processing. Concise, powerful programs are easy to write. In addition, SNOBOL4's pattern programming provides a new way to work with computers. If you would like to add SNOBOL4 to your repertoire of problem-solving tools, and learn why so many people are excited about it, read on.

1.1 About this manual

This manual is divided into three parts. This part shows you how to create and run small programs with SNOBOL4.

Tutorial is addressed to the beginning SNOBOL4 programmer. It assumes a modest knowledge of general programming concepts, and experience with another high-level language, such as BASIC, C, FORTRAN, or PASCAL. Readers without any programming background may wish to consult books written with them in mind: *'A SNOBOL4 primer'* and *'SNOBOL programming for the humanities'*, listed in the file `SNOBOL4.DOC`.

Reference is a complete description of Vanilla SNOBOL4. If you are already familiar with the SNOBOL4 language, you may wish to skip the tutorial section and proceed directly to the reference section for specific details. Later, you can return to the tutorial section for fresh insight into the language's use.

1.2 Installing Vanilla Snobol4

1.2.1 System requirements

SNOBOL4 requires the following:

1. IBM PC, XT, AT, or any other 8086/88/186/286/386 family computer. Your computer need not be an IBM PC look-alike; SNOBOL4 requires MS-DOS compatibility only.
2. PC- or MS-DOS, Version 2.0 or above.
3. 105K bytes of free RAM memory.

1.2.2 Making a backup copy

The Vanilla SNOBOL4 distribution disk should never be used for production work. Always make a backup copy, and use it for your day-to-day activities:

1. Use the DOS `FORMAT` command to initialize a new, blank diskette.
2. If your system has two $5\frac{1}{4}$ inch diskette drives, place the SNOBOL4 diskette in drive `A`, and the new disk in drive `B`, and type:

```
DISKCOPY A: B:
```

3. If you have only one diskette drive, enter:

   ```
   DISKCOPY A: A:
   ```

   and follow the instructions for swapping diskettes. The Vanilla SNOBOL4 diskette is the *Source* diskette, while the newly formatted diskette is the *Target*.

If you have a fixed disk, you may create a subdirectory for SNOBOL4, and copy all of the SNOBOL4 disk to it.

1.2.3 Initial checkout

Place your backup disk in the default drive, and play a game of Tick-Tack-Toe. Our examples will assume a two-drive system, using drive `B` as the default drive. If you have a one-drive system, or are running SNOBOL4 from the fixed disk, your screen will display a different default drive letter (`A` or `C`). Enter:

```
B>SNOBOL4 TICTAC
```

The SNOBOL4 program should load, and compile the Tick-Tack-Toe program. The game will begin execution, and display instructions.

1.3 An example

Just to get a feel for where we're going, let's take a look at a small SNOBOL4 program. It produces a sorted list of the words in a file, along with a count of how many times each word appears. Don't be concerned if you don't understand the program; I just want to give you a taste of the language:

```
* Trim input, set up constants, and create table to
*  hold word counts
        &TRIM   =  1
        WRDPAT  =  BREAK(&LCASE) SPAN(&LCASE "-'") . WORD
        TALLY   =  TABLE()

* Read a line, convert upper case letters to lower case
READ    LINE     =  REPLACE(INPUT,&UCASE,&LCASE) :F(CONVERT)

* Get and remove next word from LINE, place in variable WORD
NEXTWRD LINE  WRDPAT =                           :F(READ)

* Increment the count for this word
        TALLY[WORD] =  TALLY[WORD] + 1           :(NEXTWRD)

* Convert the table to an array
CONVERT RESULT  =  CONVERT(TALLY, "ARRAY")       :F(NONE)

* Display the results
        OUTPUT  =  "Word Counts"
        I       =  1
PRINT   OUTPUT  =  RESULT[I,1] " - " RESULT[I,2]  :F(END)
        I       =  I + 1                         :(PRINT)
NONE    OUTPUT  =  "There aren't any words!"
END
```

Running the program with the sample text on the disk as input would produce a usage count like this:

```
Word Counts
hark - 2
the - 1
lark - 1
at - 2
heaven's - 1
...
```

Notice some of the things that seem to occur so effortlessly here. A word is defined to be any combination of lower case letters, hyphen, and apostrophe. Data from the file are converted to lower case. A table of word counts uses the words themselves as subscripts. The table is converted to an array in one statement, and printed without any knowledge of the array's size. Finally, because the definition of a word is contained in one succinct pattern, it's easy to modify the program to catalog other kinds of text patterns.

Excluding comments and the `END` statement, there are 12 working statements in this program – and this program uses only a fraction of SNOBOL4's power. How much work would it be to write such a program in any other language you are familiar with? Is it possible that there is something unique about SNOBOL4?

Let's go on now to write a simple first program.

Chapter 2

First program

2.1 A first program

For the following exercises, you should have SNOBOL4 available on your default disk drive, or in your default directory if a fixed disk is used. This manual assumes that drive `B` is your default disk drive, and will show the DOS prompt as `B>`. Users with other hardware configurations may see `A>` or `C>`.

We will begin with a very simple program, one that prints a greeting on your computer's display screen. It will familiarize you with the mechanics of running a SNOBOL4 program. Every line you enter from the keyboard (or 'console') should end by pressing the Enter key.

You start the system by typing SNOBOL4 `CON` at the DOS command prompt `B>`. SNOBOL4 displays two title lines and prompts you to enter your program with a question mark on each line:

```
B>SNOBOL4 CON

     Vanilla SNOBOL4      Version 2.14.
     (c) Copyright 1984,1988 Catspaw, Inc. All Rights Reserved.
     Enter program, terminate with "END"
     ?
```

Now enter the program. Use the tab character to begin the indented line, and be sure to place blanks on each side of the equal sign:

```
     ?       OUTPUT = 'Hello world!'
     ?END

     No errors

     Hello world!

     B>
```

As you enter each line, it is compiled into a compact internal notation. The first program line begins with a tab; the second is flush left. The word `END` is special; it signals SNOBOL4 that you have finished entering program lines. It must appear at the left margin to be recognized. After the `END` statement is entered, SNOBOL4 begins to run your program.

This program consists of one assignment statement. Assignment takes the value on the right side of the equals sign, and stores it in the variable on the left. The value on the right is the character string literal `'Hello world!'`. The variable's name is `OUTPUT`, which is a special name in SNOBOL4; values assigned to it are displayed on the screen. After the assignment statement is performed, control flows into the `END` statement and the program stops.

SNOBOL4 only provides DOS in-line editing as you enter your program. It is not a program editor, and does not save your program or let you correct mistakes in previous program lines. Usually, you'll want to prepare your program in a disk file.

Try creating a program file in DOS. The symbol `^Z` represents the DOS end-of-file character, which terminates the DOS `COPY` command. It is created by entering Ctrl-Z or pressing key F6.

```
B>COPY CON HELLO.SNO
        OUTPUT = 'Hello world!'
END
^Z
        1 File(s) copied
B>
```

Now you can have SNOBOL4 read and execute your program from file HELLO.SNO:

```
B>SNOBOL4 HELLO.SNO

Vanilla SNOBOL4      Version 2.14.
(c) Copyright 1984,1988 Catspaw, Inc. All Rights Reserved.

No errors

Hello world!

B>
```

Of course, the program file could also have been created with your program text editor. If you are using a word processor, remember to produce an unadulterated ASCII file, free of any special format controls.

SNOBOL4 assigns a unique number to each program statement. The statement number and line number are displayed whenever an error message is produced. To get a listing of your program with SNOBOL4's statement numbers, try:

```
B>SNOBOL4 HELLO /L=CON

Vanilla SNOBOL4      Version 2.14.
(c) Copyright 1984,1988 Catspaw, Inc. All Rights Reserved.
1               OUTPUT = 'Hello world!'
2       END

No errors

Hello world!

B>
```

The first line, on which you typed SNOBOL4, is called the command line. It may contain options that alter SNOBOL4's behavior. The option `/L=` tells SNOBOL4 to send a listing of your source file to the specified file or device. Another device, such as `PRN:`, would print a listing on your printer. Other command line options are discussed in the chapter *Running a* SNOBOL4 *Program* of the Reference Manual.

In this example we omitted the file name extension. SNOBOL4 will supply the `.SNO` extension for the source file if it is absent.

You've now run a simple SNOBOL4 program in two ways: by typing it in directly, and by creating a disk file.

2.2 Interactive statement execution

It's very helpful to try out simple statements as they are introduced in the text. There is a SNOBOL4 program called `CODE.SNO` on the distribution diskette to help you do this. Try it now with a few simple statements. Type `END` or Ctrl-Z to stop the program.

```
B>SNOBOL4 CODE

Vanilla SNOBOL4      Version 2.14.
(c) Copyright 1984,1988 Catspaw, Inc. All Rights Reserved.

No errors

Enter SNOBOL4 statements:
?       OUTPUT = 'HELLO AGAIN!'
HELLO AGAIN!
Success
?       OUTPUT = 16
16
Success
?END

B>
```

Feel free to experiment – you can't break anything by using this program. At most, you will get a SNOBOL4 error, and return to the DOS command prompt. In that case, just start SNOBOL4 and `CODE.SNO` over again.

Whenever you see examples in the text that begin with a question mark, they are meant to be tried with `CODE.SNO`. In the text I'll omit the word `Success` most of the time unless it is relevant to the concept being presented, although it will still appear on your display. I'll also try to restrict the examples to upper case, so you can set the Caps Lock mode on your computer, and type without using the shift key.

Let's now proceed to the tutorial.

Part II

A Snobol4 Tutorial

Chapter 3

Fundamentals

SNOBOL4 is really a combination of two kinds of languages: a conventional language, with several data types and a simple but powerful control structure, and a pattern language, with a structure all its own. The conventional language is not block structured, and may appear old-fashioned. The pattern language, however, remains unsurpassed, and is unique to SNOBOL4.

You should try to master the conventional portion of SNOBOL4 first. When you're comfortable with it, you can move on to pattern matching. Pattern matching by itself is a very large subject, and this manual can only offer an introduction. The sample programs accompanying Vanilla SNOBOL4, as well as the many SNOBOL4 books available from Catspaw can be studied for a deeper understanding of patterns and their application.

We'll begin by discussing data types, operators, and variables.

3.1 Simple data types

SNOBOL4 has several different basic types, but has a mechanism to define hundreds more as aggregates of others. Initially, we'll discuss the two most basic: integers and strings.

3.1.1 Integers

An integer is a simple whole number, without a fractional part. In SNOBOL4, its value can range from −32 767 to +32 767. It appears without quotation marks, and commas should not be used to group digits. Here are some acceptable integers:

```
14    -234    0    0012    +12832    -9395    +0
```

These are incorrect in SNOBOL4:

`13.4`	fractional part is not allowed
`49723`	larger than 32 767
`-`	number must contain at least one digit
`3,076`	comma is not allowed

Use the `CODE.SNO` program to test different integer values. Try both legal and illegal values. Here are some sample test lines:

```
Enter SNOBOL4 statements:
?       OUTPUT = 42
42
?       OUTPUT = -825
-825
?       OUTPUT = 73768
Compilation error: Erroneous integer, re-enter:
```

3.1.2 Reals

Vanilla SNOBOL4 does not include real numbers. They are available in SNOBOL4+, Catspaw's highly enhanced implementation of the SNOBOL4 programming language.

3.1.3 Strings

A string is an ordered sequence of characters. The order of the characters is important: the strings `AB` and `BA` are different. Characters are not restricted to printing characters; all of the 256 combinations possible in an 8-bit byte are allowed.

Normally, the maximum length of a string is 5 000 characters, although you can tell SNOBOL4 to accept longer strings. A string of length zero (no characters) is called the null string. At first, you may find the idea of an empty string disturbing: it's a string, but it has no characters. Its role in SNOBOL4 is similar to the role of zero in the natural number system.

Strings may appear literally in your program, or may be created during execution. To place a literal string in your program, enclose it in apostrophes (`'`)[1] or double quotation marks (`"`). Either may be used, but the beginning and ending marks must be the same. The string itself may contain one type of mark if the other is used to enclose the string. The null string is represented by two successive marks, with no intervening characters. Here are some samples to try with `CODE.SNO`:

```
?       OUTPUT = 'STRING LITERAL'
STRING LITERAL
?       OUTPUT = "So is this"
So is this
?       OUTPUT = ''

?       OUTPUT = 'WHO COINED THE WORD "BYTE"?'
WHO COINED THE WORD "BYTE"?
?       OUTPUT = "WON'T"
WON'T
```

3.2 Simple operators

If data is the raw material, operators are the tools that do the work. Some operators, such as `+` and `-`, appear in all programming languages, and pocket calculators. But SNOBOL4 provides many more, some of which are unique to the SNOBOL4 language. SNOBOL4 also allows you to define your own operators. We'll examine just a few basic operators below.

3.2.1 Unary vs. binary

SNOBOL4 operators require either one or two items of data, called operands. For example, the minus sign (`-`) can be used with one object. In this form, the operator is considered unary:

```
-6
```

or as a binary operator with two operands:

```
4 - 1
```

In the first case, the minus sign negates the number. The second example subtracts 1 from 4. The minus sign's meaning depends on the context in which it appears. SNOBOL4 has a very simple rule for determining if an operator is binary or unary: Unary operators are placed immediately to the left of their operand. No blank or tab character may appear between operator and operand.

[1] Apostrophe (single quote) should not be confused with the grave accent mark (`) which appears next to it on some computer keyboards. The grave accent may not be used as a string delimiter

Binary operators have one or more blank or tab characters on each side.

The blank or tab requirement for binary operators causes problems for programmers first learning SNOBOL4. Most other languages make these white space characters optional. Omitting the right hand blank after a binary operator will produce a unary operator, and while the statement may be syntactically correct, it will probably produce unexpected results. Fortunately, blanks and binary operators quickly become a way of SNOBOL4 life, and after some initial forgetfulness there are few problems.

3.2.2 Some binary operators

Operation: assignment
Symbol: `=`

You've already met one binary operator, the equals sign (`=`). It appeared in the first sample program:

```
OUTPUT = 'Hello world!'
```

It assigns, or transfers, the value of the object on the right (`'Hello world!'`) to the object on the left (variable `OUTPUT`).

Operation: arithmetic
Symbols: `** * / + -`

These characters provide the arithmetic operations – exponentiation, multiplication, division, addition, and subtraction respectively. Each is assigned a priority, so SNOBOL4 knows which to perform first if more than one appear in an expression. Exponentiation is performed first, followed by multiplication, division, and finally addition and subtraction. SNOBOL4 is unusual in giving multiplication higher priority than division; most programming languages treat them equally.

You may use parentheses to change the order of operations. Division of an integer by another integer will produce a truncated integer result; the fractional result is discarded. Try the following:

```
?       OUTPUT = 3 - 6 + 2
-1
?       OUTPUT = 2 * (10 + 4)
28
?       OUTPUT = 7 / 4
1
?       OUTPUT = 3 ** 5
243
?       OUTPUT = 10 / 2 * 5
1
?       OUTPUT = (10 / 2) * 5
25
```

When the same operator occurs more than once in an expression, which one should be performed first? The governing principle is called associativity, and is either left or right. Multiple instances of `*`, `/`, `+` and `-` are performed left to right, while `**`'s are performed right to left. Again, parentheses may be used to change the default order. Try a few examples:

```
?       OUTPUT = 24 / 4 / 2
3
?       OUTPUT = 24 / (4 / 2)
12
?       OUTPUT = 2 ** 2 ** 3
256
```

```
?       OUTPUT = (2 ** 2) ** 3
64
```

Here's the first bit of SNOBOL4 magic: what happens if either operand is a string rather than an integer or real number? The action taken is one which is widespread throughout the SNOBOL4 language; the system tries to convert the operand to a suitable data type. Given the statement

```
?       OUTPUT = 14 + '54'
68
```

SNOBOL4 detects the addition of an integer and a string, and tries to convert the string to a numeric value. Here the conversion succeeds, and the integers 14 and 54 are added together. If the characters in the string do not form an acceptable integer, SNOBOL4 produces the error message '`Illegal data type`'.

SNOBOL4 is strict about the composition of strings being converted to numeric values: leading or trailing blanks or tabs are not allowed. The null string is permitted, and converted to integer 0. Try producing some arithmetic errors:

```
?       OUTPUT = 14 + ' 54'
Execution error #1, Illegal data type
Failure
?       OUTPUT = 'A' + 1
Execution error #1, Illegal data type
Failure
```

Note: Error numbers are listed in chapter *System Messages* of the Reference Manual.

Operation: concatenation
Symbols: blank or tab

This is the fundamental operator for assembling strings. Two strings are concatenated simply by writing one after the other, with one or more blank or tab characters between them. There is no explicit symbol for concatenation (it is special in this regard), the white space between two objects serves to define this operator. The blank or tab character merely specifies the operation; it is not included in the resulting string.

The string that results from concatenation is the right string appended to the end of the left. The two strings remain unchanged and a third string emerges as the result. Try a few simple concatenations with `CODE.SNO`:

```
?       OUTPUT = 'CONCAT' 'ENATION'
CONCATENATION
?       OUTPUT = 'ONE,' 'TWO,' 'THREE'
ONE,TWO,THREE
?       OUTPUT = 'A'                'B'       'C'
ABC
?       OUTPUT = 'BEGINNING '   'AND '   'END.'
BEGINNING AND END.
```

The string resulting from concatenation can not be longer than the maximum allowable string size.

The concatenation operator works only on character strings, but if an operand is not a string, SNOBOL4 will convert it to its string form. For example,

```
?       OUTPUT = (20 - 17)  ' DOG NIGHT'
3 DOG NIGHT
?       OUTPUT = 19  (12 / 3)
194
```

In the first case, concatenation's right operand is the string `' DOG NIGHT'`, but the left operand is an integer expression (`20 - 17`). SNOBOL4 performs the subtraction, converts the result to the string `'3'`, and produces the final result `'3 DOG NIGHT'`. In the second example, the integer operands are converted to the strings `'19'` and `'4'`, to produce the result string `'194'`. This is not exactly good math, but it is correct concatenation.

You must be careful however. If you accidentally omit an operator, SNOBOL4 will think you intended to perform concatenation. In the example above, perhaps we omitted a minus sign and had really meant to say:

```
?       OUTPUT = 19 - (12 / 3)
15
```

It is always possible for concatenation to automatically convert a number to a string. But there is one important exception when SNOBOL4 doesn't try to do this: if either operand is the null string, the other operand is returned unchanged. It is not coerced into the string data type. If the first example were changed to:

```
?       OUTPUT = (20 - 17)  ''
3
```

the result is the integer 3. You'll find you'll use this aspect of null string concatenations extensively in your SNOBOL4 programming.

Before we proceed, let's think about the null string one more time as the string equivalent of the number zero. First of all, adding zero to a number does not change its value, and concatenating the null string with an object doesn't change it, either. Second, just as a calculator is cleared to zero before adding a series of numbers, the null string can serve as the starting place for concatenating a series of strings.

3.2.3 Some unary operators

There aren't many interesting unary operators at this point in your tour of SNOBOL4. Most of them appear in connection with pattern matching, discussed later. Note, however, that all unary operations are performed before binary operations, unless precedence is altered by parentheses.

Operation: arithmetic
Symbols: `+ -`

These unary operators require a single numeric operand, which must immediately follow the operator, without an intervening blank or tab. Unary minus (`-`) changes the arithmetic sign of its operand; unary plus (`+`) leaves the sign unchanged. If the operand is a string, SNOBOL4 will try to convert it to a number. The null string is converted to integer 0. Coercing a string to a number with unary plus is a noteworthy technique. Try unary plus and minus with `CODE.SNO`:

```
?       OUTPUT = -(3 * 5)
-15
?       OUTPUT = +''
0
```

3.3 Variables

A variable is a place to store an item of data. The number of variables you may have is unlimited, provided you give each one a unique name. Think of a variable as a box, marked on the outside with a permanent name, able to hold any data value or type. Many programming languages require that you formally declare what kind of entity the box will contain – integer, real, string, etc. – but SNOBOL4 is more flexible. A variable's contents may change repeatedly during program execution. The size of the box contracts or expands as necessary. One moment

it might contain an integer, then a 2 000 character string, then the null string; in fact, any SNOBOL4 data type.

There are only a few rules about composing a variable's name when it appears in your program:

- the name must begin with an upper- or lower-case letter;
- if it is more than one character long, the remaining characters may be any combination of letters, numbers, or the characters period (`.`) and underscore (`_`);
- the name may not be longer than the maximum line length (120 characters).

Here are some correct SNOBOL4 names:

```
WAGER     P23     VerbClause     SUM.OF.SQUARES     Buffer
```

Normally, SNOBOL4 performs case-folding on names. Lower-case alphabetic characters are changed to upper-case when they appear in names – `Buffer` and `BUFFER` are equivalent. Naturally, casefolding of data does not occur within a string literal. Casefolding can be disabled by the command line option `/C`.

In some languages, the initial value of a new variable is undefined. SNOBOL4 guarantees that a new variable's initial value is the null string. However, except in very small programs, you should always initialize variables. This prevents unexpected results when a program is modified or a program segment is reexecuted.

You store something in a variable by making it the object of an assignment operation. You can retrieve its contents simply by using it wherever its value is needed. Using a variable's value is nondestructive; the value in the box remains unchanged. Try creating some variables using `CODE.SNO`:

```
?       ABC = 'EGG'
?       OUTPUT = ABC
EGG
?       D = 'SHELL'
?       OUTPUT = abc d              ;* same as ABC D
EGGSHELL
?       OUTPUT = NONESUCH           ;* new variable is null
?       OUTPUT = ABC NULL D
EGGSHELL
?       N1 = 43
?       D = 17
?       OUTPUT = N1 + D
60
?       output = ABC D
EGG17
```

`OUTPUT` is a variable with special properties; when a value is stored in its box, it is also displayed on your screen. There is a corresponding variable named `INPUT`, which reads data from your keyboard. Its box has no permanent contents. Whenever SNOBOL4 is asked to fetch its value, a complete line is read from the keyboard and used instead. If `INPUT` were used twice in one statement, two separate lines of input would be read. Try these examples:

```
?       OUTPUT = INPUT
TYPE ANYTHING YOU DESIRE
TYPE ANYTHING YOU DESIRE
?       TWO.LINES = INPUT '-AND-' INPUT
FIRST LINE
SECOND LINE
?       OUTPUT = TWO.LINES
FIRST LINE-AND-SECOND LINE
```

SNOBOL4 variables are global in scope – any variable may be referenced anywhere in the program.

Chapter 4

Control Flow and Functions

4.1 Success and failure

Success and failure are as important in SNOBOL4 as they are in life. Success and failure are unmistakable signals; something either worked, or it didn't. Significant program conciseness is achieved by recognizing that data values and signals are fundamentally different entities.

The elements of a statement provide values and signals as computation proceeds. SNOBOL4 accumulates both, and stops executing a particular statement when it finds it cannot succeed. Program flow can be altered based upon this success or failure.

The success signal will have a value result associated with it. In situations in which the signal itself is the desired object, the result value may only be the null string. The failure signal has no associated value. (In some instances, it may be helpful to view failure as meaning 'failure to produce a result'.)

Previously, we introduced the variable `INPUT`, which reads a line from the keyboard. In general, `INPUT` can be made to read from any disk file. The line read may be any character string, including the null string if it is an empty line. If any string might appear, then there is no special value we can test for to detect end-of-file. Success and failure provide an elegant alternative to testing for special values.

When we retrieve a value from `INPUT`, we normally get a string and a success signal. But when end-of-file is encountered, we get a failure signal instead, and no value.

Since Ctrl-Z (or key F6) allows you to enter an end-of-file from the keyboard, we can easily demonstrate this type of failure. As you've noticed, the `CODE.SNO` program reports the success or failure of each statement. So far, all examples have succeeded. Now try this one:

```
?       OUTPUT = INPUT
^Z
Failure
```

Success and failure are control signals, and appear only during the execution of a statement. They cannot be stored in a variable, which holds values only.

There is much more which can be done with success and failure, but to understand their use, you'll need to know how SNOBOL4 statements are constructed.

4.2 A Snobol4 statement

In general, a SNOBOL4 statement looks like this:

Label *Statement body* :*Goto*

Label is optional, and is omitted by placing a blank or tab in the first character position. *Goto* is also optional, and can be eliminated simply by omitting it and the colon. In fact, even *Statement body* is optional. You can have a program line consisting of just a label or a Goto field.

4.2.1 The label field

SNOBOL4 normally executes the statements of a program in sequence. The ability to transfer control from one statement to another, perhaps conditionally, makes SNOBOL4 much more usable.

Labels provide names for statements. If present, they must begin in the first character position of a statement, and must start with a letter or number. Additional characters may be anything but blank or tab. Like variable names, lower-case letters are equivalent to upper-case when case-folding (the default).

4.2.2 The Goto field

Transfer of control is made possible by the Goto. It interrupts the normal sequential execution of statements by telling SNOBOL4 which statement to execute after the present one. The Goto field appears at the end of the statement, preceded by a colon (`:`), and has one of these forms:

```
:(label)
:S(label)
:F(label)
:S(label1) F(label2)
```

White space is required before the colon. `label` is the name given to the target statement, and must be enclosed in parentheses. If the first form is used, execution resumes at the referenced statement, unconditionally. In the second and third forms, transfer occurs only if the statement has succeeded or failed, respectively. Otherwise, execution proceeds to the next statement in line. If the fourth form is used, transfer is made to `label1` if the statement succeeded, or to `label2` if it failed. A statement with a `label` and a Goto would look like this:

```
COPY    OUTPUT = INPUT          :F(DONE)
```

Now let's write a short program which copies keyboard input to the screen, and reports the total number of lines. If you are an accurate typist, you can type it into SNOBOL4 directly. Otherwise, you should use your text editor to create a file containing the program text. First stop the `CODE.SNO` program by typing `END`:

```
?END

B>SNOBOL4 CON

Vanilla SNOBOL4      Version 2.14.
(c) Copyright 1984,1988 Catspaw, Inc. All Rights Reserved.
Enter program, terminate with "END"
?       N = 0
?COPY   OUTPUT = INPUT          :F(DONE)
?       N = N + 1               :(COPY)
?DONE   OUTPUT = 'THERE WERE ' N ' LINES'
?END

No errors

TYPE IN A TEST LINE
TYPE IN A TEST LINE

AND ANOTHER
AND ANOTHER

^Z
```

```
THERE WERE 2 LINES

B>
```

We start the line count in variable `N` at 0. The next statement has a label, `COPY`, a statement body, and a Goto field. It is an assignment statement, and begins execution by reading a line of input. If `INPUT` successfully obtains a line, the result is stored in `OUTPUT`. The Goto field is only testing for failure, so SNOBOL4 proceeds to the next statement, where `N` is incremented, and the unconditional Goto transfers back to statement `COPY`.

When an end-of-file is read, variable `INPUT` signals failure. Execution of this statement terminates immediately, without performing the assignment, and transfers to the statement labeled `DONE`. The number of lines is displayed, and control flows into the `END` statement, stopping the program.

4.3 Built-in functions

A function is analogous to an operator; it operates on data to produce a result. The data objects are called the arguments of the function. The result returned – the function of the arguments – may have two components: the success or failure signal; and for success, a value. The value may be any data type.

A function is used by writing its name and a list of arguments enclosed by parentheses:

$$function\_name(arg_1, arg_2, \ldots, arg_n)$$

It may appear in your program anywhere a constant is allowed – in expressions, patterns, even as the argument of another function. If the function has more than one argument, they should be separated by commas. If trailing arguments are omitted, SNOBOL4 will supply the null string instead. Some functions, such as one that returns the current date, have no arguments at all.

SNOBOL4 provides a large number of predefined functions, and allows you to define your own. The large repertoire of built-in functions makes SNOBOL4 programming easier. Most functions are concerned with pattern matching, input/output, and advanced features of the language. Here we'll introduce a few simple conditional, numeric, and string functions to give you an idea of the variety. Try them interactively with `CODE.SNO`.

4.3.1 Conditional functions

These functions fail or succeed depending upon their arguments. They are sometimes called predicate functions because the success of an expression using them is predicated upon their success. If they succeed, they return the null string as their value. Here are the built-in predicates and the conditions upon which they succeed:

`IDENT(S,T)`	`S` and `T` are identical. `S` and `T` may be constants or variables with any data type. To be identical, the arguments must have the same data type and value. Since omitted arguments default to the null string, `IDENT(S)` succeeds if `S` is the null string
`DIFFER(S,T)`	`S` and `T` are different. `DIFFER` is the opposite of `IDENT`. `DIFFER(S)` succeeds if `S` is not the null string
`EQ(X,Y)`	Integers `X` and `Y` are equal. `X` and `Y` must be integers, or strings which can be converted to integers
`NE(X,Y)`	Integers `X` and `Y` are not equal
`GE(X,Y)`	Integer `X` is greater than or equal to `Y`
`GT(X,Y)`	Integer `X` is greater than `Y`
`LE(X,Y)`	Integer `X` is less than or equal to `Y`

LT(X,Y)	Integer X is less than Y
INTEGER(X)	X is an integer, or a string which can be converted to an integer
LGT(S,T)	String S is lexically greater than string T using a character-by-character comparison.

Leading blanks may be used in front of a argument for readability. Here are some exercises for CODE.SNO:

```
?       N = 3
?       EQ(N, 3)
Success
?       IDENT(N, 3)
Success
?       EQ(3, "3")
Success
?IDENT(3, "3")                    ;* integer and string
Failure
?       EQ(N, 4)
Failure
?       NE(N, 4)
Success
?       INTEGER(N)
Success
?       INTEGER('47')
Success
?       DIFFER('ABC', 'abc')
Success
?       IDENT('a' 'b' 'c', 'abc')
Success
?       LGT('ABC', 'ABD')
Failure
```

When any of these functions succeed, they return a null string. Since other statement elements are not altered when concatenated with the null string, this provides an easy way to interpose tests and construct loops. Suppose we execute the statement:

```
N = LT(N,10) N + 1        :S(LOOP)
```

Function LT fails if N is 10 or greater. If the statement fails, the assignment is not performed, and execution continues with the next statement. However, if LT succeeds, its null string value is concatenated with the expression N + 1, and the result is assigned to N. This has the effect of increasing N by 1 and transferring to statement LOOP until N reaches 10.

If we concatenated several conditional functions together, and they all succeeded, the result would still be the null string. If any function failed, the entire concatenation would fail. This gives us a simple way to produce a successful result if a number of conditions are all true. For example, the expression:

```
INTEGER(N) GE(N,5) LE(N,100)
```

succeeds if N is an integer between 5 and 100.

4.3.2 Other functions

These functions always succeed; all but REMDR and SIZE return a string result.

DATE()	Return current date and time as a string
DUPL(S,N)	Duplicate string S, N times

REMDR(X,Y)	Produce the remainder (modulus) of X / Y
REPLACE(S1,S2,S3)	Return string S1 after performing the character replacements specified by strings S2 and S3. S2 specifies which characters to replace, and S3 specifies what to replace them with
SIZE(S)	Return the number of characters in string S
TRIM(S)	Return string S with trailing blanks removed.

Exercises for CODE.SNO:

```
?       OUTPUT = 'THE DATE AND TIME ARE: ' DATE()
THE DATE AND TIME ARE: 10-19-87 11:49:33.90
?       OUTPUT = DUPL('ABC', 20)
ABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABC
?       OUTPUT = SIZE('ZIPPY')
5
?       OUTPUT = SIZE('')
0
?       OUTPUT = TRIM('TRAILING BLANKS  ') 'GONE'
TRAILING BLANKSGONE
?       OUTPUT = REPLACE('spoon','po','PO')
sPOOn
```

Chapter 5

Input/Output and Keywords

5.1 Input/output

We've already performed simple input and output with variables `INPUT` and `OUTPUT`. In this chapter, you'll learn more about SNOBOL4's I/O capabilities.

SNOBOL4 can communicate with up to 16 different files at once. A 'file' is either a disk file or a device, such as a printer. Every file is identified by a unit number, which is an integer between 1 and 16. You chose the numbers when you select the files you wish to use. The particular numbers chosen have no special significance; they just distinguish one file from another.

Actual input or output of data is performed by associating a variable with a unit number and a direction. When a statement tries to use the variable's value, a line is read from the associated file. When a value is stored in the variable, a line is written to the associated file. `INPUT` and `OUTPUT` are variables whose association with the keyboard and screen were preset by SNOBOL4. For historical reasons, they use unit numbers 5 and 6 respectively.

Strings are the only data type which can be transferred to and from files. A successful input operation always returns a string. During output, nonstring objects, such as integers, are automatically converted to their string form.

The functions `INPUT` and `OUTPUT` (not to be confused with the variables `INPUT` and `OUTPUT`) are provided to attach a unit number to a variable, and optionally, to a particular file. Their names are distinguished from the variables of the same names by appearing as functions, that is, with parentheses and an argument list.

5.1.1 Associating file names and units

There are two ways to tell SNOBOL4 what file will be used with a particular unit number:

- As an option on the SNOBOL4 command line, like this:

  ```
  B>SNOBOL4 PROGRAM /2=ADDRESS.TXT /8=RESULT.DAT
  ```

  Here, unit number 2 is associated with the file named `ADDRESS.TXT`, and unit number 8 with file `RESULT.DAT`. It will still be necessary to use the `INPUT` or `OUTPUT` function described below to associate variables with these unit numbers. This method works best when different files will be used each time the program is run.

- Use a string containing the file name as the fourth argument to the `INPUT` or `OUTPUT` function, as in:

  ```
  INPUT(..., 2, ..., 'ADDRESS.TXT')
  ```

  This method is better when the file name will not change, or is a string derived from a dialogue with the user, or is produced from a string calculation.

A file name consisting of a single hyphen (-) is reserved, and specifies the MS-DOS standard input file when used with the INPUT function, or the standard output file when used with the OUTPUT function. These standard input or output files may be redirected on the command line using the MS-DOS redirection operators (< *filename* and > *filename*).

5.1.2 Input

This function associates a variable with data read from a file:

```
INPUT('variable', unit, length, 'file')
```

It succeeds and returns the null string if the file was found and successfully opened, and fails otherwise. Length is an optional integer that specifies the line length. If the file name argument is omitted, SNOBOL4 consults the command line to find the file to use with this unit.

For example, to open file TEXT.IN for input as unit 1, and associate variable READLINE with it, we would say

```
        INPUT('READLINE', 1, , 'TEXT.IN')    :S(OK)
        OUTPUT = 'Could not find file'       :(END)
OK       ...
```

If the file name were specified on the command line as /1=TEXT.IN, we only need the first two arguments to INPUT:

```
        INPUT('READLINE', 1)             :S(OK)
        OUTPUT = 'Could not find file'   :(END)
OK      ...
```

To read a line from the file, we simply use READLINE in a statement. The statement fails when the end-of-file is read:

```
LINE = READLINE                  :F(END.OF.FILE)
```

Each file-associated variable will have a line length associated with it (80 characters unless SNOBOL4 is told otherwise in the length argument). Normally, reading stops at each end-ofline character (carriage return). If more than the line length has been read, the extra characters are discarded. If a short line is encountered, SNOBOL4 pads the line with blanks to produce the full line length. The end-of-line character is not included in the string returned.

Blank padding is another historic feature from the days when most input was on punch cards. The next section, *Keywords*, will show you how to disable it. You can also use the TRIM function to remove superfluous trailing blanks. The previous statement then becomes:

```
LINE = TRIM(READLINE)       :F(END.OF.FILE)
```

When READLINE encounters the end-of-file, its failure signal is propagated outward, causing function TRIM to fail. This failure is detected in the Goto field in the usual manner.

5.1.3 Output

This function associates a variable with data written to a file. If the file does not exist, it is created. If it already exists, its previous contents are discarded.

```
OUTPUT('variable', unit, length, 'file')
```

The function succeeds and returns the null string if the file was successfully opened for output, and fails otherwise.

We write data to the file by assigning it to the associated variable. In this example, we will use a variable called PRINT, and the DOS device PRN: with a line length of 132 characters:

```
OUTPUT('PRINT', 2, 132, 'PRN:')   :S(PRTOK)
OUTPUT = 'Could not attach printer'  :(END)
PRINT  = 'Text Listing -- ' DATE()
...
```

If the string assigned to an output variable is longer than the line length, SNOBOL4 will output as many lines as necessary of the standard line length to accommodate the string. SNOBOL4 supplies the carriage return and line feed characters at the end of each line.

Once again, the output file name could be given on the command line (`/2=PRN:`). The function call would then look like this:

```
OUTPUT('PRINT', 2, 132)           :S(PRTOK)
 ...
```

5.1.4 Changing I/O defaults

Having `INPUT` and `OUTPUT` associated with the keyboard and screen may be altered in the SNOBOL4 command line. A surprising number of programs can be written this way, using only the variables `INPUT` and `OUTPUT` for I/O. The command line phrase `/I=FILENAME`, associates `INPUT` with the named file, and `/O=FILENAME` does the same for `OUTPUT`. SNOBOL4 makes all the associations for you; no call to the `INPUT` or `OUTPUT` function is required.

SNOBOL4 also provides the pre-associated variable `SCREEN`. Using `SCREEN` allows your program to post messages to the display even if `OUTPUT` has been redirected elsewhere.

If we have a program written in terms of variables `INPUT` and `OUTPUT`, it can be run without alteration with different data files. For example, the following program will copy `INPUT` to `OUTPUT`, and place the line length and a blank in front of each line:

```
LOOP    S = TRIM(INPUT)                 :F(END)
        OUTPUT = SIZE(S) ' ' S          :(LOOP)
END
```

Suppose we associate file `TEXT.IN` with `INPUT`, and `TEXT.OUT` with `OUTPUT`. We've supplied the morning song from Shakespeare's Cymbeline in file `TEXT.IN`, and the program above in file `LENGTH.SNO`. You can run it like this:

```
B>SNOBOL4 LENGTH /I=TEXT.IN /O=TEXT.OUT

Vanilla SNOBOL4      Version 2.14.
(c) Copyright 1984,1988 Catspaw, Inc. All Rights Reserved.

No errors

B>TYPE TEXT.OUT
44 Hark! hark! the lark at heaven's gate sings,
...
```

SNOBOL4 will supply the default file name extensions `.IN` and `.OUT` for the `/I` and `/O` options, so the command line could be shortened to:

```
B>SNOBOL4 LENGTH /I=TEXT /O=TEXT
```

5.2 Keywords

Input/Output allows your program to communicate with the outside world. Your program may also communicate with the SNOBOL4 system itself. Keywords allow you to modify SNOBOL4's behavior, and to obtain information from the system. A keyword consists of the ampersand character (`&`) followed by an alphabetic name. They are used in a statement in the same way

as a variable. They either provide values or have values assigned to them. Numeric keywords are restricted to integer values.

- `&TRIM` Remove trailing blanks

  Normally, short lines read from a file are padded with blank characters to the standard line length. In `LENGTH.SNO`, we used the function `TRIM(INPUT)` to remove those blanks. A simpler method assigns an integer value to keyword `&TRIM` to control padding. If `&TRIM` is set to a nonzero value, blanks are not appended, and any trailing blanks are removed. A statement to do this looks like this:

  ```
  &TRIM = 1
  ```

  Since trailing blanks are usually not desired, you'll often see this statement at the beginning of many SNOBOL4 programs.

- `&MAXLNGTH` Maximum string length

  This keyword controls the maximum permissible string length. Its initial value is 5 000, but it may be set to any positive integer from 0 to 32 767. Setting it to 0 is going to severly restrict what you can do, since only the null string will be available to you!

- `&DUMP` Termination dump of variables

  This keyword is useful for debugging programs because it tells SNOBOL4 to display the values of your variables when your program terminates. Setting `&DUMP` to a positive, nonzero integer causes the variable names to be sorted alphabetically. A negative integer produces a unsorted dump. Zero is the default value, inhibiting the dump. Only variables with nonnull values are displayed.

- `&ALPHABET` Complete character set

  This keyword contains a 256 character string, the computer's entire character set in ascending sequence. It is called a protected keyword because it cannot be modified by your program.

- `&LCASE` Lower case letters

  This keyword contains the 26 lower case alphabetic characters, `"abcdefghijklmnopqrstuvwxyz"`.

- `&UCASE` Upper case letters

  This keyword contains the 26 upper case alphabetic characters, `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`.

5.3 Programs without pattern matching

You now have the ingredients to create some simple programs. However, if this were all of the SNOBOL4 language, there would be very little reason to use it. We'll get to pattern matching shortly, where you'll find many new, challenging concepts. First, however, you should be comfortable with the preceding material.

Take a few minutes to examine and run the following programs.

5.3.1 File counts – `FCOUNTS.SNO`

This program counts the number of characters and lines in a file. Because real numbers are not available in Vanilla SNOBOL4, you should only use this program with input files smaller than 32 767 characters.

```
        &TRIM  = 1
        CHARS  = 0
NEXTL   CHARS  = CHARS + SIZE(INPUT)     :F(DONE)
        LINES  = LINES + 1               :(NEXTL)
```

```
DONE    OUTPUT = CHARS ' characters'
        OUTPUT = +LINES ' lines read'
END
```

In such a small program, it's permissible to rely upon the fact that the system initializes `LINES` to the null string. The first use of the statement:

```
LINES  = LINES + 1              :(NEXTL)
```

converts `LINES` from the null string to an integer value. We used the expression `+LINES` in the last statement to produce an integer 0 (instead of the null string), if the input file were empty. To count the characters and lines in a file, use the `/I=` option, as in:

```
B>SNOBOL4 FCOUNTS.SNO /I=TEXT.IN
```

5.3.2 Formatting text – TRIPLET.SNO

This program reformats a file by centering the lines and arranging them in groups of three. Note that statements containing an asterisk in column one are considered comments by SNOBOL4.

```
* Trim input, count input lines:
        &TRIM = 1
        N = 0

* Read next input line, all done if end-of-file.
LOOP    S = INPUT                       :F(END)

* Precede with blanks to center within 80 character line:
        OUTPUT = DUPL(' ', (80 - SIZE(S)) / 2) S

* Increment count, but reset to 0 every third line.
* Also, output a blank line when count resets:
        N = REMDR(N + 1, 3)
        OUTPUT = EQ(N, 0)               :(LOOP)
END
```

This program uses the `DUPL` function to produce the leading blanks required to center a line. A simple calculation based on each line's width determines the number of blanks needed.

The last two statements break the file lines into triplets. The `REMDR` function returns the integer remainder (modulus) when the first argument is divided by the second. In this case, assigning the result to variable `N` causes `N` to continually cycle through the values 0, 1, 2, 0, 1, ... When `N` is 0, the last statement assigns the null string to `OUTPUT`, producing a blank line. If `N` is 1 or 2, `EQ` fails, and the assignment fails.

Try running the program with the sample text file:

```
B>SNOBOL4 TRIPLET /I=TEXT
```

Chapter 6

Pattern Matching

6.1 Introduction

Pattern matching examines a subject string for some combination of characters, called a *pattern*. The matching process may be very simple, or extremely complex. For example:

- The subject contains several color names. The pattern is the string `"BLUE"`. Does the subject string contain the word `"BLUE"`?
- The subject contains a nucleic acid (DNA) sequence. The pattern searches for a subsequence that is replicated in two other places in the string.
- The subject contains a paragraph of English text. The pattern describes the spacing rules to be applied after punctuation. Does the subject string conform to the punctuation rules?
- The subject string represents the current board position in a game of Tick-Tack-Toe. The pattern examines this string and determines the next move.
- The subject contains a program statement from a prototype computer language. The pattern contains the grammar of that language. Is the statement properly formed according to the grammar?

Most programming languages provide rudimentary facilities to examine a string for a specific character sequence. SNOBOL4 patterns are far more powerful, because they can specify complex (and convoluted) interrelationships. The colors of a painting, the words of a sentence, the notes of a musical score have limited significance in isolation. It is their relationship with one another which provides meaning to the whole. Likewise, SNOBOL4 patterns can specify context; they may be qualified by what precedes or follows them, or by their position in the subject.

6.1.1 Knowns and unknowns

Patterns are composed of known and unknown components.

Knowns are specific character strings, such as the string `"BLUE"` in the first example above. We are looking for a yes/no answer to the question: 'Does this known item appear in the subject string?'

Unknowns specify the kind of subject characters we are looking for; the specific characters are not identifiable in advance. We might want to match only characters from a restricted alphabet, or any substring of a certain length, or some arbitrary number of repetitions of a string. If the pattern matches, we can then capture the particular subject substring matched.

6.2 Specifying pattern matching

A pattern match requires a subject string and a pattern. The subject is the first statement element after the label field (if any). The pattern appears next, separated from the subject by

white space (blank or tab). If SUBJECT is the subject string, and PATTERN is the pattern, it looks like this:

```
label   SUBJECT PATTERN
```

The pattern match succeeds if the pattern is found in the subject string; otherwise it fails. This success or failure may be tested in the Goto field:

```
label   SUBJECT PATTERN            :S(label1) F(label2)
```

A real point of confusion is the distinction between pattern matching and concatenation. How do you tell the difference? Where does the subject end and the pattern begin? In this case, parentheses should be placed around the subject, since SNOBOL4 always uses the first complete statement element as the subject. In the statement

```
X Y Z
```

X is the subject, and Y concatenated with Z is the pattern. Whereas

```
(X Y) Z
```

indicates the subject is string X concatenated with string Y, and the pattern is Z.

6.3 Subject string

The subject string may be a literal string, a variable, or an expression. If it is not a string, its string equivalent will be produced before pattern matching begins. For example, if the subject is the integer 48, integer to string conversion produces the character string "48". Remember, if the subject includes concatenated elements, they should be enclosed in parentheses.

6.4 Pattern subsequents and alternates

Arithmetic expressions are composed of elements and simpler subexpressions. Similarly, patterns are composed of simpler subpatterns which are joined together as subsequents and alternates. If P1 and P2 are two subpatterns, the expression

```
P1 P2
```

is also a pattern. The subject must contain whatever P1 matches, immediately followed by whatever P2 matches. P2 is the subsequent of P1. The white space (blank or tab) between P1 and P2 is the same binary concatenation operator previously used to join strings; its use with patterns is completely analogous. The preceding pattern matches pattern P1 followed by pattern P2.

The binary alternation operator is the vertical bar (|). As it is a binary operator, it must have white space on each side. The pattern

```
P1 | P2
```

matches whatever P1 matches, or whatever P2 matches. SNOBOL4 tries the various alternatives from left to right.

Normally, concatenation is performed before alternation, so the pattern

```
P1 | P2 P3
```

matches P1 alone, or P2 followed by P3. Parentheses can be used to alter the grouping of subpatterns. For example:

```
(P1 | P2) P3
```

matches P1 or P2, followed by P3.

When a pattern successfully matches a portion of the subject, the matching subject characters are bound to it. The next pattern in the statement must match beginning with the very next subject character. If a subsequent fails to match, SNOBOL4 backtracks, unbinding patterns until another alternative can be tried. A pattern match fails when SNOBOL4 cannot find an alternative that matches.

The null string may appear in a pattern. It always matches, but does not bind any subject characters. We can think of it as matching the invisible space between two subject characters. One possible use is as the last of a series of alternatives. For example, the pattern

```
ROOT ('S' | 'ES' | '')
```

matches the pattern in ROOT, with an optional suffix of 'S' or 'ES'. If ROOT matches, but is not followed by 'S' or 'ES', the null string matches and successfully completes the clause. Its presence gives the pattern match a successful escape.

The conditional functions of the previous chapter may appear in patterns. If they fail when evaluated, the current alternative fails. If they succeed, they match the null string, and so do not consume any subject characters. They behave like a gate, allowing the match to proceed beyond them only if they are true. This pattern will match 'FOX' if N is 1, or 'WOLF' if N is 2:

```
EQ(N,1) 'FOX' | EQ(N,2) 'WOLF'
```

Parentheses may be used to factor a pattern. The strings 'COMPATIBLE', 'COMPREHENSIBLE', and 'COMPRESSIBLE' are matched by the pattern:

```
'COMP' ('AT' | 'RE' ('HEN' | 'S') 'S') 'IBLE'
```

6.5 Simple pattern matches

Here are examples of pattern matches using a string literal or variable for the subject. The patterns consist entirely of known elements. Use the CODE.SNO program to experiment with them:

```
?       'BLUEBIRD' 'BIRD'
Success
?       'BLUEBIRD' 'bird'
Failure
?       B = 'THE BLUEBIRD'
?       B 'FISH'
Failure
?       B 'FISH' | 'BIRD'
Success
?       B ('GOLD' | 'BLUE') ('FISH' | 'BIRD')
Success
```

The first statement shows that the matching substring ('BIRD') need not begin at the start of the subject string. This is called *unanchored matching*. The second statement fails because strings are case sensitive, unlike names and labels. The third statement creates a variable to be used as the subject. The fifth statement employs an alternate: we are matching for 'FISH' or 'BIRD'.

The last statement uses subsequents and alternates. We are looking for a substring in B that contains 'GOLD' or 'BLUE', followed by 'FISH' or 'BIRD'. It will match 'GOLDFISH', 'GOLDBIRD', 'BLUEFISH' or 'BLUEBIRD'. If the parentheses were omitted, concatenation of 'BLUE' and 'FISH' would be performed before alternation, and the pattern would match 'GOLD', 'BLUEFISH', or 'BIRD'.

6.6 The pattern data type

If we execute the statement

```
?       COLOR = 'BLUE'
```

the variable COLOR contains the string 'BLUE', and could appear in the pattern portion of a statement:

```
?       B COLOR
Success
```

Even though it is used as a pattern, COLOR has the string data type. However, complicated patterns may be stored in a variable just like a string or numeric value. The statement

```
?       COLOR = 'GOLD' | 'BLUE'
```

will create a structure describing the pattern, and store it in the variable COLOR. COLOR now has the pattern data type. The preceding example can now be written as:

```
?       CRITTER = 'FISH' | 'BIRD'
?       BOTH = COLOR CRITTER
?       B BOTH
Success
```

6.7 Capturing match results

If the pattern match

```
B BOTH
```

succeeds, we may want to know which of the many pattern alternatives were used in the match. The binary operator *conditional assignment* assigns the matching subject substring to a variable. The operator is called conditional, because assignment occurs only if the entire pattern match is successful. Its graphic symbol is a period (.). It assigns the matching substring on its left to the variable on its right. Note that the direction of assignment is just the opposite of the statement assignment operator (=). Continuing with the previous example, we'll redefine COLOR and CRITTER to use conditional assignment:

```
?       COLOR = ('GOLD' | 'BLUE') . SHADE
?       CRITTER = ('FISH' | 'BIRD') . ANIMAL
?       BOTH = COLOR CRITTER
?       B BOTH
Success
?       OUTPUT = SHADE
BLUE
?       OUTPUT = ANIMAL
BIRD
```

The substrings that match the subpatterns COLOR and CRITTER are assigned to SHADE and ANIMAL respectively. The statement

```
BOTH = COLOR CRITTER
```

had to be reexecuted because its previous execution captured the old values of COLOR and CRITTER, without the conditional assignment operators. The redefinition of COLOR and CRITTER was not reflected in BOTH until the statement was reexecuted.

Conditional assignment may appear at any level of pattern nesting, and may include other conditional assignments within its embrace. The pattern

```
(('B' | 'F' | 'N') . FIRST 'EA' ('R' | 'T') . LAST) . WORD
```

matches 'BEAR', 'FEAR', 'NEAR', 'BEAT', 'FEAT', or 'NEAT', assigning the first letter matched to FIRST, the last letter to LAST, and the entire result to WORD.

The variable OUTPUT may be used as the target of conditional assignment. Try:

```
?       'B2' ('A' | 'B') . OUTPUT (1 | 2 | 3) . OUTPUT
B
2
Success
```

6.8 Unknowns

All of the previous examples used patterns created from literal strings. We may also want to specify the qualities of a match component, rather than its specific characters. Using unknowns greatly increases the power of pattern matching. There are two types, primitive patterns and pattern functions.

6.8.1 Primitive patterns

There are seven primitive patterns built into the SNOBOL4 system. The two used most frequently will be discussed here. Chapter *Advanced Topics* introduces the remaining five.

- REM Match remainder of subject

  REM is short for the remainder pattern. It will match zero or more characters at the end of the subject string. Try the following:

  ```
  ?       'THE WINTER WINDS' 'WIN' REM . OUTPUT
  TER WINDS
  Success
  ```

  The subpattern 'WIN' matched its first occurrence in the subject, at the beginning of the word 'WINTER'. REM matched from there to the end of the subject string – the characters 'TER WINDS' – and assigned them to the variable OUTPUT. If we change the pattern slightly, to:

  ```
  ?       'THE WINTER WINDS' 'WINDS' REM . OUTPUT
  Success
  ```

  then 'WINDS' matches at the end of the subject string, leaving a null remainder for REM. REM matches this null string, assigns it to OUTPUT, and a blank line is displayed.

  The pattern components to the left of REM must successfully match some portion of the subject string. REM begins where they left off, matching all subject characters through the end of string. There are no restrictions on the particular characters matched.

- ARB Match arbitrary characters

  ARB matches an arbitrary number of characters from the subject string. It matches the shortest possible substring, including the null string. The pattern components on either side of ARB determine what is matched. Try the statements

  ```
  ?       'MOUNTAIN' 'O' ARB . OUTPUT 'A'
  UNT
  Success
  ?       'MOUNTAIN' 'O' ARB . OUTPUT 'U'
  Success
  ```

In the first statement, the ARB pattern is constrained on either side by the known patterns 'O' and 'A'. ARB expands to match the subject characters between, 'UNT'. In the second statement, there is nothing between 'O' and 'U', so ARB matches the null string. ARB behaves like a spring, expanding as needed to fill the gap defined by neighboring patterns.

6.8.2 Cursor position

During a pattern match, the cursor is SNOBOL4's pointer into the subject string. It is integer-valued, and points between two subject characters. The cursor is set to zero when a pattern match begins, corresponding to a position immediately to the left of the first subject character. As the pattern match proceeds, the cursor moves right and left across the subject to indicate where SNOBOL4 is attempting a match. The value of the cursor will be used by some of the pattern functions that follow.

The *cursor position* operator assigns the current cursor value to a variable. It is a unary operator whose graphic symbol is the 'at' sign (@). It appears within a pattern, preceding the name of a variable. By using OUTPUT as the variable, we can display the cursor position on the screen. For instance:

```
?       'VALLEY' 'A' @OUTPUT ARB 'E' @OUTPUT
2
5
Success
?       'DOUBT' @OUTPUT 'B'
0
1
2
3
Success
?       'FIX' @OUTPUT 'B'
0
1
2
Failure
```

Cursor assignment is performed whenever the pattern match encounters the operator, including retries. It occurs even if the pattern ultimately fails. The element @OUTPUT behaves like the null string – it doesn't consume subject characters or interfere with the match in any way.

6.8.3 Integer pattern functions

These functions return a pattern based on their integer argument. The pattern produced can be used directly in a pattern match statement, or stored in a variable for later retrieval.

- LEN(integer) Match fixed-length string

  LEN(I) produces a pattern which matches a string exactly I characters long. I must be an integer greater than or equal to zero. Any characters may appear in the matched string. For example, LEN(5) matches any 5-character string, and LEN(0) matches the null string. LEN may be constrained to certain portions of the subject by other adjacent patterns:

```
?       S = 'ABCDA'
?       S LEN(3) . OUTPUT
ABC
?       S LEN(2) . OUTPUT 'A'
CD
```

The first pattern match had only one constraint – the subject had to be at least three characters long – so `LEN(3)` matched its first three characters. The second case imposes the additional restriction that `LEN(2)`'s match be followed immediately by the letter `'A'`. This disqualifies the intermediate match attempts `'AB'` and `'BC'`.

Using keyword `&ALPHABET` as the subject provides a simple way to convert a decimal character code between 0 and 255 to its one character equivalent. For example, by consulting an ASCII character code chart we find that the BEL character is decimal 7. We can load that character into variable `BEEP` with one statement:

```
?       &ALPHABET LEN(7) LEN(1) . BEEP
```

and produce five beeps on the speaker with:

```
?       OUTPUT = DUPL(BEEP,5)
```

`&ALPHABET` contains all 256 members of the computer's character set, in ascending order. `LEN(7)` matches the first seven characters (codes 0-6), leaving `BEL` as the next match position for `LEN(1)`. This operation is analogous to the `CHR$` function in BASIC.

The inverse operation, obtaining the numerical value of a character code, is also possible. If variable `CHAR` contains a one character string, variable `N` will be set to its decimal equivalent with the second statement below:

```
? CHAR = 'A' ? \&ALPHABET @N CHAR ? OUTPUT = N 65
```

In chapter *Program-defined Objects*, I'll demonstrate how you can define your own functions to encapsulate each of these operations.

- `POS(integer)`, `RPOS(integer)` Verify cursor position

  The `POS(I)` and `RPOS(I)` patterns do not match subject characters. Instead, they succeed only if the current cursor position is a specified value. They often are used to tie points of the pattern to specific character positions in the subject.

  `POS(I)` counts from the left end of the subject string, succeeding if the current cursor position is equal to `I`. `RPOS(I)` is similar, but counts from the right end of the subject. If the subject length is `N` characters, `RPOS(I)` requires the cursor be `N - I`. If the cursor is not the correct value, these functions fail, and SNOBOL4 tries other pattern alternatives, perhaps extending a previous substring matched by `ARB`, or beginning the match further along in the subject.

  Continuing with `CODE.SNO`:

```
?       S = 'ABCDA'
?       S POS(0) 'B'
Failure
?       S LEN(3) . OUTPUT RPOS(0)
CDA
?       S POS(3) LEN(1) . OUTPUT
D
?       S POS(0) 'ABCD' RPOS(0)
Failure
```

  The first example requires a `'B'` at cursor position 0, and fails for this subject. `POS(0)` anchors the match, forcing it to begin with the first subject character. Similarly, `RPOS(0)` anchors the end of the pattern to the tail of the subject. The next example matches at a specific mid-string character position, `POS(3)`. Finally, enclosing a pattern between `POS(0)` and `RPOS(0)` forces the match to use the entire subject string.

At first glance these functions appear to be setting the cursor to a specified value. Actually, they never alter the cursor, but instead wait for the cursor to come to them as various match alternatives are attempted. This, in turn, allows other patterns in the statement to be processed in an orderly fashion. You can demonstrate this waiting for the cursor behavior like this:

```
?       S @OUTPUT POS(3)
0
1
2
3
Success
```

- `TAB(integer)`, `RTAB(integer)` Match to fixed position

  These patterns are hybrids of `ARB`, `POS()`, and `RPOS()`. They use specific cursor positions, like `POS` and `RPOS`, but bind (match) subject characters, like `ARB`. `TAB(I)` matches any characters from the current cursor position up to the specified position `I`. `RTAB(I)` does the same, except, as in `RPOS()`, the target position is measured from the end of the subject.

  `TAB` and `RTAB` will match the null string, but will fail if the current cursor is to the right of the target. They also fail if the target position is past the end of the subject string.

  These patterns are useful when working with tabular data. For example, if a data file contains name, street address, city and state in columns 1, 30, 60, and 75, this pattern will break out those elements from a line:

```
P = TAB(29) . NAME TAB(59) . STREET TAB(74) . CITY REM . STATE
```

  The pattern `RTAB(0)` is equivalent to primitive pattern `REM`. One potential source of confusion is just what it is that `RTAB` matches. It counts from the right end of the subject, but matches to the left of its target cursor. Try:

```
?       'ABCDE' TAB(2) . OUTPUT RTAB(1) . OUTPUT
AB
CD
Success
```

  `TAB(2)` matches `'AB'`, leaving the cursor at 2, between `'B'` and `'C'`. The subject is 5 characters long, so `RTAB(1)` specifies a target cursor of $5 - 1$, or 4, which is between the `'D'` and `'E'`. `RTAB` matches everything from the current cursor, 2, to the target, 4.

6.8.4 Character pattern functions

These functions produce a pattern based on a string-valued argument. Once again, the pattern may be used directly or stored in a variable.

- `ANY(string)`, `NOTANY(string)` Match one character

  Each function produces a pattern which matches one character based upon the subject string. `ANY(S)` matches the next subject character if it appears in the string `S`, and fails otherwise. `NOTANY(S)` matches a subject character only if it does not appear in `S`. Here are some sample uses of each:

```
?       VOWEL = ANY('AEIOU')
?       DVOWEL = VOWEL VOWEL
?       NOTVOWEL = NOTANY('AEIOU')
?       'VACUUM' VOWEL . OUTPUT
```

```
A
?       'VACUUM' DVOWEL . OUTPUT
UU
?       'VACUUM' (VOWEL NOTVOWEL) . OUTPUT
AC
```

The argument string specifies a set of characters to be used in creating the ANY or NOTANY pattern. It may contain duplicate characters, and the order of characters in S is immaterial.

- SPAN(string), BREAK(string) Match a run of characters

  These are multicharacter versions of ANY and NOTANY. Each requires a nonnull argument to specify a set of characters.

  SPAN(S) matches one or more subject characters from the set in S. SPAN must match at least one subject character, and will match the longest subject string possible.

  BREAK(S) matches up to but not including any character in S. The string matched must always be followed in the subject by a character in S. Unlike SPAN and NOTANY, BREAK will match the null string.

  These two functions are called stream functions because each streams by a series of subject characters. SPAN is most useful for matching a group of characters with a common trait. For example, we can say an English word is composed of one or more alphabetic characters, apostrophe, and hyphen. The statements

```
?       LETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ'-"
?       WORD = SPAN(LETTERS)
```

  produce a suitable pattern in WORD. To match the material between words (white space, punctuation, etc.), use the pattern:

```
?       GAP = BREAK(LETTERS)
```

  SPAN and BREAK are two of the most useful SNOBOL4 functions. Try some examples using CODE.SNO:

```
?       'SAMPLE LINE' WORD . OUTPUT
SAMPLE
?       'PLUS TEN DEGREES' ' ' WORD . OUTPUT
TEN
?       GAPO = GAP . OUTPUT
?       WORDO = WORD . OUTPUT
?       ': ONE, TWO, THREE' GAPO WORDO GAPO WORDO
:
ONE
,
TWO
        DIGITS = '0123456789'
?       INTEGER = (ANY('+-') | '') SPAN(DIGITS)
?       'SET -43 VOLTS' INTEGER . OUTPUT
-43
?       REAL = INTEGER '.' (SPAN(DIGITS) | '')
?       'SET -43.625 VOLTS' REAL . OUTPUT
-43.625
?       S = '0ZERO,1ONE,2TWO,3THREE,4FOUR,5FIVE,'
?       S 4 BREAK(',') . OUTPUT
FOUR
```

If you require a version of SPAN which will match the null string, or a BREAK which will not match the null string, you can use the following constructions:

```
(SPAN(S) | '')
(NOTANY(S) BREAK(S))
```

We need to introduce one more fundamental concept – replacement – before we can write some meaningful programs.

6.9 Pattern matching with replacement

Pattern matching identifies a subject substring with a particular trait, specified by the pattern. We used conditional assignment to copy that substring to a variable. Replacement moves in the other direction, letting you alter the substring in the subject. The space occupied by the matching substring may be enlarged or contracted (or removed entirely), leaving adjacent subject characters undisturbed. If the pattern matched the entire subject, replacement behaves like a simple assignment statement.

Replacement appears in a form similar to assignment:

```
SUBJECT PATTERN = REPLACEMENT
```

First, the pattern match is attempted on the subject. If it fails, execution of the statement ends immediately, and replacement does not occur. If the match succeeds, any conditional assignments within the pattern are performed. The replacement field is then evaluated, converted to a string, and inserted in the subject in place of the matching substring. If the replacement field is empty, the null string replaces the matched substring, effectively deleting it. Try a few examples with CODE.SNO:

```
?       T = 'MUCH ADO ABOUT NOTHING'
?       T 'ADO' = 'FUSS'
Success
?       OUTPUT = T
MUCH FUSS ABOUT NOTHING
?       T 'NOTHING' =
Success
?       OUTPUT = T
MUCH FUSS ABOUT
?       'MASH' 'M' = 'B'
Execution error #8, Variable not present where required
Failure
```

The first replacement searches for 'ADO' in the subject string, replacing it with 'FUSS'. The second replacement has a null string replacement value, and deletes the matching substring. The last example demonstrates that a variable must be the subject of replacement. Variables can be changed; string literals – like 'MASH' – cannot.

The following will replace the 'M' in 'MASH' with a 'B':

```
?       VERB = 'MASH'
?       VERB 'M' = 'B'
?       OUTPUT = VERB
BASH
```

If the matched substring appears more than once in the subject, only the first occurrence is changed. The remaining substrings must be found with a program loop. For example, a statement to eliminate all occurrences of the letter 'A' from the subject looks like this:

```
ALOOP   SUBJECT 'A' =                   :S(ALOOP)
```

Here ALOOP is the statement label, SUBJECT is some variable containing the subject string, 'A' is the pattern, and the replacement field is empty. If an 'A' is found, it is deleted by replacing it with the null string, and the statement succeeds. The success Goto branches back to ALOOP, and another search for 'A' is performed. The loop continues until no 'A's remain in the subject, and the pattern match fails. Of course, the pattern and replacement can be as complex as desired.

Simple loops like this can be tried in CODE.SNO by appending a semicolon after the Goto field. (Semicolon is used with Goto in CODE.SNO only; you would not use it in normal programs.) Continuing with the previous example:

```
?       VOWEL = ANY('AEIOU')
?VL     T VOWEL = '*'                    :S(VL);
?       OUTPUT = T
M*CH F*SS *B**T
```

Since conditional assignment is performed before replacement, its results are available for use in the replacement field of the same statement. Here's an example of removing the first item from a list, and placing it on the end:

```
?       RB = 'RED,ORANGE,YELLOW,GREEN,BLUE,INDIGO,VIOLET,'
?       CYCLE = BREAK(',') . ITEM LEN(1) REM . REST
?       RB CYCLE = REST ITEM ','
Success
?       OUTPUT = ITEM
RED
?       OUTPUT = RB
ORANGE,YELLOW,GREEN,BLUE,INDIGO,VIOLET,RED,
```

Pattern CYCLE matches the entire subject, placing the first color into ITEM, bypassing the comma with LEN(1), and placing the remainder of the subject into REST. REST and ITEM are then transposed in the replacement field, and stored back into RB.

6.10 Sample programs

I've introduced a lot of concepts in this chapter; it's time to see how they fit together into programs. They're supplied on the Vanilla SNOBOL4 diskette.

6.10.1 Word counting

The first program counts the number of words in the input file. Lines with an asterisk in the first column are comment lines – their contents are ignored by SNOBOL4.

```
*   Simple word counting program, WORDS.SNO.
*
*   A word is defined to be a contiguous run of letters,
*   digits, apostrophe and hyphen.  This definition of
*   legal letters in a word can be altered for specialized
*   text.
*
*   If the file to be counted is TEXT.IN, run this program
*   by typing:
*       B>SNOBOL4 WORDS /I=TEXT
*
        &TRIM  =  1
        WORD   =  "'-"  '0123456789' &UCASE &LCASE
        WPAT   =  BREAK(WORD) SPAN(WORD)
```

```
NEXTL   LINE   =  INPUT                          :F(DONE)
NEXTW   LINE WPAT =                              :F(NEXTL)
        N      =  N + 1                          :(NEXTW)

DONE    OUTPUT =  +N ' words'
END
```

After defining the acceptable characters in a word, the real work of the program is performed in the three lines beginning with label `NEXTL`. A line is read from the input file, and stored in variable `LINE`. The next statement attempts to find the next word with pattern `WPAT`. `BREAK` streams by any blanks and punctuation, stopping just short of the word, which `SPAN` then matches. Both the word and any preceding punctuation are removed from `LINE` by replacement with the null string.

When no more words remain in `LINE`, the failure transfer to `NEXTL` reads the next line. If the match succeeds, `N` is incremented, and the program goes back to `NEXTW` to search for another word. When the end-of-file is encountered, control transfers to `DONE` and the number of words is displayed.

It's simple to alter pattern `WPAT` to search for other things. For instance, if we wanted to count occurrences of double vowels, we could use:

```
WPAT = ANY('AEIOUaeiou') ANY('AEIOUaeiou')
```

To count the occurrences of integers with an optional sign character, use:

```
WPAT = (ANY('+-') | '') SPAN('0123456789')
```

Perhaps we want to count violations of simple punctuation rules: period with only one blank, or comma and semicolon followed by more than one blank:

```
WPAT = '. ' NOTANY(' ') | ANY(',;') ' ' SPAN(' ')
```

Notice how closely `WPAT` parallels the English language description of the problem.

6.10.2 Word crossing

This program asks for two words, and displays all intersecting letters between them. For example, given the words `LOOM` and `HOME`, the program output is:

```
 H
LOOM
 M
 E

  H
LOOM
  M
  E

   H
   O
LOOM
   E
```

A pattern match like this would find the first intersecting character:

```
HORIZONTAL ANY(VERTICAL) . CHAR
```

However, we want to find all intersections, so will have to iterate our search. In conventional programming languages, we might use numerical indices to remember which combinations were tried. Here, we'll use place-holding characters like '*' and '#' to remove solutions from future consideration. As seems to be the case with SNOBOL4, there are more comments than program statements:

```
* CROSS.SNO - Print all intersections between two words

        &TRIM = 1

*  Get words from user
*
AGAIN   OUTPUT = 'ENTER HORIZONTAL WORD:'
        H      = INPUT                          :F(END)

        OUTPUT = 'ENTER VERTICAL WORD:'
        V      = INPUT                          :F(END)

*       Make copy of horizontal word to track position.
        HC     = H

*  Find next intersection in horizontal word.  Save
*  the number of preceding horizontal characters in NH.
*  Save the intersecting character in CROSS.
*  Replace with '*' to remove from further consideration.
*  Go to AGAIN to get new words if horizontal exhausted.
*
NEXTH   HC @NH ANY(V) . CROSS = '*'              :F(AGAIN)

*  For each horizontal hit, iterate over possible
*  vertical ones.  Make copy of vertical word to track
*  vertical position.
*
        VC     = V

*  Find where the intersection was in the vertical word.
*  Save the number of preceding vertical characters in NV.
*  Replace with '#' to prevent finding it again in that
*  position.  When vertical exhausted, try horizontal again.
*
NEXTV   VC @NV CROSS = '#'                       :F(NEXTH)

*  Now display this particular intersection.
*  We make a copy of the original vertical word,
*  and mark the intersecting position with '#'.
*
        OUTPUT =
        PRINTV = V
        PRINTV POS(NV) LEN(1) = '#'

*  Peel off the vertical characters one-by-one.  Each will
*  be displayed with NH leading blanks to get it in the
*  correct column.  When the '#' is found, display the full
*  horizontal word instead.
*  When done, go to NEXTV to try another vertical position.
```

```
*
PRINT   PRINTV LEN(1) . C =                    :F(NEXTV)
        OUTPUT = DIFFER(C,'#') DUPL(' ',NH) C :S(PRINT)
        OUTPUT = H                             :(PRINT)
END
```

6.11 Anchored and unanchored matching

Most of the examples above match substrings which do not begin at the first subject character. This is the unanchored mode of pattern matching. Alternately, we can anchor the pattern match by requiring it to include the first subject character. Setting keyword `&ANCHOR` to a nonzero value produces anchored matching. Anchored matching is usually faster than unanchored, because many futile attempts to match are eliminated.

Even when the desired item is not at the beginning of the subject, it is often possible to simulate anchored matching by prefixing the pattern with a subpattern which will stream out to the desired object. The stream function spans the gap from the first subject character to the desired item. Use `CODE.SNO` to experiment with `&ANCHOR`:

```
?       DIGITS = '0123456789'
?       &ANCHOR = 1
?       'THE NEXT 43 DAYS' BREAK(DIGITS) SPAN(DIGITS) . N
```

This will assign substring `'43'` to `N`, even in anchored mode. In unanchored mode, the test lines:

```
?       &ANCHOR = 0
?       'THE NEXT 43 DAYS' SPAN(DIGITS) . N
```

would ultimately succeed, but only after `SPAN` failed on each of the characters preceding the `'4'`. The efficiency difference is more pronounced if the subject does not contain any digits. In the first formulation, `BREAK(DIGITS)` fails and the anchored match then fails immediately. The second construction fails only after `SPAN` is tried at each subject character position.

When your program first begins execution, SNOBOL4 sets keyword `&ANCHOR` to zero, the unanchored mode. If you can construct all your patterns as anchored patterns, you should set `&ANCHOR` nonzero for anchored matching. Setting and resetting `&ANCHOR` throughout your program is error prone and not advised. Another alternative is to leave `&ANCHOR` set to 0, but to 'pseudo-anchor' patterns by using `POS(0)` as the first pattern element.

It always takes less time for a pattern to succeed than to fail. Failure implies an exhaustive search of all combinations, whereas success stops the pattern match early. You should try to construct patterns with direct routes to success, such as the use of `BREAK` above. Wherever possible, impose restrictions on the number of alternatives to be tried. Combinatorial explosion is the price of loose pattern programming.

Chapter 7

Additional Operators and Data Types

In this chapter we will explore some additional SNOBOL4 operators and data types. Many of these concepts are entirely absent from other programming languages. Far from being esoteric, they fit quite naturally into SNOBOL4, and add to its conciseness and power of expression. In the following examples, we will continue to use the `CODE.SNO` program to illustrate each new idea.

7.1 Indirect reference

In conventional programming languages, a variable's name may be specified only at the time the program is written. In fact, once the run-time storage has been allocated, the textual form of the name can be discarded. This is not the case in SNOBOL4; you can create new variables during execution, and reference existing ones from names specified in character strings.

The unary operator dollar sign (`$`) is the *indirect reference operator*. By applying it to a variable you instruct SNOBOL4 to use its contents as the name of another variable, and to continue on to reference that variable. SNOBOL4 goes through the operand to reach the variable. Try the following simple example:

```
?       DOG = 'BARK'
?       CAT = 'MEOW'
?       ANIMAL = 'CAT'
?       OUTPUT = $ANIMAL
MEOW
?       ANIMAL = 'DOG'
?       OUTPUT = $ANIMAL
BARK
```

These statements make their indirect reference through the string contained in variable `ANIMAL`. `ANIMAL`'s contents are treated as a pointer to the final destination. That is, using `ANIMAL` by itself retrieves the string `'DOG'`, while `$ANIMAL` refers to the variable `DOG`.

New variables may also be created by using an indirect reference as the object of an assignment. Here, `$DOG` causes variable `BARK` to be created, and assigned the string `'RUFF'`:

```
?       $DOG = 'RUFF'
?       OUTPUT = BARK
RUFF
```

Indirect referencing may proceed to any depth, provided the null string is never encountered as a variable name:

```
?       OUTPUT = $ANIMAL '-' $$ANIMAL
```

```
BARK-RUFF
?       OUTPUT = $RUFF
Execution error #4, Null string in illegal context
```

In the first example, `$ANIMAL` produces the contents of variable `DOG`, while `$$ANIMAL` refers to the variable `BARK`. The second example attempts to go through `RUFF` – which was not previously defined – and obtains the null string. Of course, the null string is not a valid variable name.

7.1.1 Associative programming

Indirect referencing provides a means of programming by association. Suppose we want to write a program that allows the user to enter a state name and receive the state's capital in response. We've provided a data file called `CAPITAL.DAT`, in which each line contains a state name, comma, and the capital. The first part of the program will read the file and set up an associative data base:

```
*  Trim input, attach data file to variable INFILE
        &TRIM = 1
        INPUT('INFILE', 1, , 'CAPITAL.DAT')       :F(ERR)

*  Read a line from file.  Start querying upon EOF
READF   LINE = INFILE                             :F(QUERY)

*  Break out state and capital from line
        LINE BREAK(',') . STATE LEN(1) REM . CAPITAL :F(ERR)

*  Convert state name into a variable, and assign the
*  capital city string to it.  Then read next line.
        $STATE = CAPITAL                          :(READF)

ERR     OUTPUT = 'Illegal data file'              :(END)
QUERY    ...
```

We attach the file, and associate variable `INFILE` with it. Successive file lines are read into variable `LINE`. Pattern matching assigns the state name and capital city to variables `STATE` and `CAPITAL` respectively. We use an indirect reference through `$STATE` to create a new variable with the state's name, and assign the capital city to it. For example, the file line `'COLORADO,DENVER'` creates variable `COLORADO`, containing `'DENVER'`.

Having established a data base, completing the program to access it is trivial:

```
*  Read state name, access it as a variable
QUERY   OUTPUT = $INPUT                           :S(QUERY)
END
```

An input line is read from the user, and used for an indirect reference. If the user types a state name, treating it as a variable name obtains the state capital. An invalid state name would reference a new variable, whose value is the null string, and a blank line would be output. A more complete program might test for this null string and produce an error message.

The addition of one statement to the program loop creating the data base allows us to enter either the state name or capital city, and obtain the other:

```
$STATE = CAPITAL
$CAPITAL = STATE                            :(READF)
```

How would we solve this problem in a language like BASIC? States and capitals could be stored in an array. We would then use a loop to sequentially compare the user's input string with the array elements. If a match were found, the result would be displayed from another array element. In SNOBOL4, we did it all with one statement: `OUTPUT = $INPUT`. Associative programming can often replace a conventional linear search.

7.1.2 Variable names

Earlier I said that variable names were composed of letters, digits, and the characters period and underscore. These restrictions apply only to variables which appear in program text. Variable names created or referenced with the indirect reference operator may be composed of any nonnull string of characters, and may be as long as any other string. If we set keyword `&DUMP` nonzero, we would see a list of states and capitals when the program terminated. The variable names created by `$STATE` are in the left column, and their string contents in the right column:

```
ALABAMA = 'MONTGOMERY'
ALASKA = 'JUNEAU'
      ...
NEW HAMPSHIRE = 'CONCORD'
```

The dump reveals a variable named `NEW HAMPSHIRE`, which contains a blank within its name. Clearly, you cannot directly say:

```
NEW HAMPSHIRE = 'CONCORD'
```

since SNOBOL4 sees this as a pattern match statement: variable `NEW` is the subject, and variable `HAMPSHIRE` contains the pattern. To reference this variable, we must use:

```
$'NEW HAMPSHIRE' = 'CONCORD'
```

Try `CODE.SNO` with some unconventional variable names:

```
?       $'"' = 'DOUBLE QUOTE'
?       $'$#@!*' = 53
?       OUTPUT = $'$#@!*' $'"'
53 DOUBLE QUOTE
```

7.1.3 Indirect Gotos

Indirect referencing is not restricted to the main body of a statement. It may be used in the Goto field to transfer control to a label specified by a variable. Suppose variable `OP` held the one-character string `'+'`, `'-'`, `'*'`, or `'/'`. This Goto would transfer to one of four statements, labeled `L+`, `L-`, `L*`, or `L/`:

```
        statement                              :($('L' OP))
L+      statement
L-      statement
```

The string in `OP` is appended to string `'L'`, and the result is used with an indirect reference to obtain the final label name.

Indirect referencing in the Goto field is a more powerful version of the computed Goto which appears in some languages. It allows a program to quickly perform a multiway control branch based on an item of data. Of course, the computed label name must be defined in the program. SNOBOL4 provides an error message if your program transfers to an undefined label.

Indirect referencing may not be used in a statement's label field. Dynamically changing the name of a statement during execution is excessive even by SNOBOL4 standards.

7.2 Unevaluated expressions

The pattern data type appears when a pattern structure is stored in a variable for subsequent use in a pattern match. For example, a pattern to capture the next `N` characters after a colon, and store them in variable `ITEM` could be written as:

```
NPAT = ':' LEN(N) . ITEM
```

Notice that a definition such as this is static. NPAT captures the value of variable N at the time of pattern construction. If we subsequently alter N in the program, NPAT retains N's original value. One way to use the current value of N is to explicitly specify the pattern each time it is needed:

```
SUBJECT ':' LEN(N) . ITEM
```

Now the pattern is being constructed anew whenever the statement is executed. But reconstructing a pattern whenever it is used is inefficient, so a one-time definition is preferable.

The *unevaluated expression* operator allows us to obtain the efficiency of the NPAT formulation, yet use the current value of N when NPAT is referenced. It is a unary operator, whose graphic symbol is the asterisk (*). Now we would specify NPAT like this:

```
NPAT = ':' LEN(*N) . ITEM
```

The pattern is only constructed once, and assigned to NPAT. The current value of N is ignored at this time. Later, when NPAT is used in a pattern match, the unevaluated expression operator tells SNOBOL4 to fetch the current value of N.

The unevaluated expression operator may be used with the argument of the pattern functions ANY, BREAK, LEN, NOTANY, POS, RPOS, RTAB, SPAN, or TAB. It may also be applied to an alternate or subsequent clause or to an entire pattern. Here's an example:

```
?       PAT = TAB(*I) . OUTPUT SPAN(*S) . OUTPUT
?       SUB = '123AABBCC'
?       I = 4
?       S = 'AB'
?       SUB PAT
123A
ABB
?       I = 3
?       SUB PAT
123
AABB
```

It's worth noting that I and S were undefined when PAT was first constructed. Later, we will apply this technique to construct recursive patterns.

7.3 Immediate assignment

Our examples have made extensive use of the conditional assignment operator to capture matched substrings after a successful pattern match. The *immediate assignment* operator allows us to capture intermediate results during the pattern match.

Immediate assignment occurs whenever a subpattern matches, even if the entire pattern match ultimately fails. Immediate assignment is a binary operator whose graphic symbol is the dollar sign ($). Like conditional assignment, the matching substring on its left is assigned to the variable on its right. Here are examples with CODE.SNO where we use variable OUTPUT to reveal the work of the pattern matcher:

```
?       S = 'ABCDEFG'
?       S 'A' ARB $ OUTPUT 'E'

B
BC
BCD
Success
?       S ('B' LEN(2) | 'C' LEN(3)) $ OUTPUT 'G'
```

```
BCD
CDEF
Success
?
```

7.3.1 Immediate assignment and unevaluated expressions

As useful as immediate assignment is for revealing the inner workings of a pattern match, a more powerful use is possible. It can be used with the unevaluated expression operator to develop a new class of patterns. An interesting substring at the beginning of the subject is immediately assigned to a variable, and the variable is then subsequently used in the very same pattern.

Suppose a number at the beginning of the subject specifies the length of a variable width field that follows. We would like to capture the number into variable N, then use it with the `LEN` function to transfer the data into variable `FIELD`. When used with `LEN`, `N` must be preceded by the unevaluated expression operator, so that its new value is retrieved. For instance:

```
?       FPAT = SPAN('0123456789') $ N LEN(*N) . FIELD
?       '12ABCDEFGHIJKLMNOPQ' FPAT
Success
?       OUTPUT = FIELD
ABCDEFGHIJKL
```

`SPAN` matched the field length, 12, and immediately assigned it to `N`. `LEN(*N)` then matched the next 12 characters. Another subject, with a different field length, would update `N` appropriately. Type conversion was working quietly behind the scenes here: `N` was assigned the string `'12'`, yet it appeared as integer 12 to the `LEN` function.

Now here is an example which provides a glimpse of just how powerful Snobol4's pattern matching can be. Problem: Examine a subject for an arbitrary three-character substring which appears twice in a row, or bracketed in parentheses. Solution:

```
?       TWOPAT = LEN(3) $ X . OUTPUT *(X | "(" X ")")
?       'ABCDECDEFGH' TWOPAT
CDE
Success
?       'ABCDE(CDE)BA' TWOPAT
CDE
Success
```

As you experiment with these types of patterns, you may discover some which fail when they should succeed. The problem is that Snobol4 stops matching when it believes further match attempts would be futile. These heuristics are normally invisible, and speed program execution. At this time, we'll defer discussing heuristics, and simply mention that they can be disabled with the statement:

```
&FULLSCAN = 1
```

Let's take a break from pattern matching, and examine some other Snobol4 data types.

7.4 Arrays

7.4.1 Array concepts

Arrays in Snobol4 are similar to arrays in other programming languages. They allow a single variable name to specify more than one data element; integer subscripts distinguish the individual members of an array. Each array element may contain any data type, independent of the types in other array elements.

A one-dimensional array is a vector; it is simply a list of I items. A two-dimensional array is a grid composed of several adjacent vectors – an I by J array has I rows and J columns. A three-dimensional array, I by J by K in size, is a rectangular solid consisting of K adjacent grids. There's no limit to the number of dimensions allowed, but such arrays become increasingly difficult to visualize.

In keeping with SNOBOL4's pliability, an array is defined during program execution, rather than at compilation time. Its size and shape is specified by a string. The definition of an array may be changed at any time, or the array may be deleted and its memory reused when it is no longer needed.

7.4.2 Array creation

Arrays are created by the SNOBOL4 function ARRAY. A program calls this function with a prototype string which specifies the number of dimensions and their sizes. The function returns an array pointer, which is stored in a variable; the array elements are referenced by applying subscripts to this variable. Here are two statements for use with CODE.SNO. They create oneand two-dimensional arrays named LIST and BOX respectively:

```
?       LIST = ARRAY('25')
?       BOX = ARRAY('12,3')
```

LIST points to a vector of 25 elements. BOX points to a grid, 12 rows high and 3 columns wide, containing 36 elements. The ARRAY function initializes all array elements to the null string.

7.4.3 Array referencing

Array subscripts are integer valued, and are specified by angular or square brackets (<> or []). Subscript values range from 1 to the size of each dimension. If you attempt to use a subscript outside this range, the array reference will fail, and the failure may be detected in the Goto portion of the statement. Try some array references with CODE.SNO:

```
?       LIST<3> = 'MAPLE'
?       BOX[10,2] = 3
?       LIST[33] = 4
Failure
?       OUTPUT = LIST[3] LIST[4] BOX<10,2>
MAPLE3
```

Angular and square brackets are interchangeable. The reference to LIST[33] failed because the largest subscript allowed for that array is 25. LIST[4] produced its initialized value, the null string, and had no effect on the concatenation. The array pointer in LIST can be assigned to another variable:

```
?       B = LIST
?       OUTPUT = B[3]
MAPLE
?       B<3> = 'WILLOW'
?       OUTPUT = LIST<3>
WILLOW
```

Assigning the pointer in LIST to B made both variables point to the same array. Since there's but one actual array, array references made using LIST or B are equivalent. The COPY function (see chapter *Built-in Functions* of the Reference Manual) creates a duplicate copy of an entire array.

Array elements may be used anywhere a variable name is allowed – expressions, patterns, function arguments, etc. The fact that an array reference fails if a subscript is out-of-bounds can be used in a simple and natural way when scanning an array. Rather than having to know

an array's size, we simply loop until an array reference fails. A program segment to display the members of an array SCORE might look like this:

```
I = 0
I = I + 1
OUTPUT = SCORE[I]                    :S(PRINT)
...
```

7.4.4 Array initialization

Arrays may be created with an initial value other than the null string. ARRAY accepts a second argument which specifies this initial value. We can create a three-dimensional array with all elements initialized to the string 'PA-18' as follows:

```
?       A = ARRAY('2,3,4','PA-18')
?       OUTPUT = A[1,2,3]
PA-18
```

7.4.5 Other array bounds

Ordinarily, subscripts range from 1 to the size of each dimension. However, if you find it more convenient, other subscript ranges may be used. The prototype string for ARRAY's first argument has the general form:

```
'L1:H1,L2:H3,...,Ln:Hn'
```

The Ls and Hs are integers specifying the lower and upper bounds of each dimension. If the lower bound and colon are omitted from any dimension, the integer 1 is assumed. Here is a five element vector, with allowed subscripts −2, −1, 0, 1 and 2:

```
?       A = ARRAY('-2:2','PIPER')
?       OUTPUT = A[-1]
PIPER
?       OUTPUT = A[3]
Failure
```

Arrays are a traditional computer programming concept. Now we'll see how SNOBOL4 takes the idea one important step further, with the concept of tables.

7.5 Tables

7.5.1 Table creation and referencing

A table is similar to a one-dimensional array, with two important differences. First, a table's size is not fixed; it extends itself automatically whenever a new element is added to it. Second, table subscripts are not limited to integers, but may be any SNOBOL4 data type. Strings and patterns may be used as subscripts. Tables combine the idea of associative programming with the data grouping of arrays.

Tables are created by the SNOBOL4 function TABLE. No arguments are required, since a table's size is not fixed. The function returns a table pointer, which you store in a variable. Like arrays, table elements are referenced by applying subscripts to the variable. Try this example with CODE.SNO:

```
?       T = TABLE()
?       T['ROSE'] = 'RED'
?       T['N'] = 6
?       OUTPUT = T['N'] T['THE'] T['ROSE']
```

```
6RED
?       FLOWER = 'ROSE'
?       T[FLOWER] = T[FLOWER] ',THORNS'
?       OUTPUT = T[FLOWER]
RED,THORNS
```

Here, strings have been used as table subscripts. The concept of an out-of-bounds subscript does not exist with tables. The reference to `T['THE']` created a new entry, and assigned it the null string. Unlike arrays, no initial value for new entries may be specified in the call to `TABLE`; new table entries are always initialized to the null string.

7.5.2 Conversion between tables and arrays

In the above example, we know what values were used as table subscripts. But if the table were constructed from data in a file, how can we determine what items were placed in the table? We need to know the subscripts to view the table, but the subscripts themselves are part of the table. If this were an array, we could run an integer subscript over the array to see the data. Applying integer subscripts to a table only creates more entries.

SNOBOL4 provides a simple solution to this dilemma – a method to convert a table to an array. An `N` row by 2 column array can be created from a table. The first array column contains the subscripts that were used to create the table. The second column contains the data items stored with the corresponding table subscript. `N` is the number of table entries with nonnull values.

Once the table is in array form, integer subscripts can be applied to the array to display the subscripts and their values. A table is converted to an array with the `CONVERT` function, which accepts a table argument and the word `'ARRAY'`, and returns a pointer to the new array. Continuing with the previous example:

```
?       A = CONVERT(T, 'ARRAY')
Success
?       OUTPUT = A[1,1] '-' A[1,2]
ROSE-RED,THORNS
?       OUTPUT = A[2,1] '-' A[2,2]
N-6
```

As you would expect with SNOBOL4, the inverse operation – conversion of an array to a table – is also possible. The array must be rectangular, `N` rows by 2 columns. The array entries in the first column become the table subscripts. The array's second column becomes the table entry values:

```
?       W = CONVERT(A, 'TABLE')
Success
?       OUTPUT = W['ROSE']
RED,THORNS
```

7.5.3 Counting word usage with a table

Tables are useful when we want to record a number of pair associations, where each half of the pair might have any data type. A classic example of a table's utility is a word usage program. Earlier, we developed a program to count the total number of words in a file. We will modify that program to count the number of times each unique word appears. The program begins like this:

```
*       Simple word usage program, WORDU.SNO.
*
*  A word is defined to be a contiguous run of letters,
*  digits, apostrophe and hyphen.  This definition of legal
```

```
*  letters in a word can be altered for specialized text.
*
*  If the file to be counted is TEXT.IN, run as follows:
*       B>SNOBOL4 WORDU /I=TEXT
*
        &TRIM  = 1

*  Define the characters which comprise a 'word'
        WORD   = "'-"  '0123456789' &LCASE

*  Pattern to isolate each word as assign it to ITEM:
        WPAT   = BREAK(WORD) SPAN(WORD) . ITEM

*  Create a table to maintain the word counts
        WCOUNT = TABLE()

*  Read a line of input and obtain the next word
NEXTL   LINE   = REPLACE(INPUT, &UCASE, &LCASE)   :F(DONE)
NEXTW   LINE WPAT =                               :F(NEXTL)

*  Use word as subscript, update its usage count
        WCOUNT<ITEM> = WCOUNT<ITEM> + 1           :(NEXTW)
DONE     ...
```

We'll convert the input to lower case, so words like `'The'` and `'the'` are counted together. `WPAT` has been changed to store each word in variable `ITEM`. When a word is identified, it is used as a subscript for table `WCOUNT`. When `ITEM` contains a new word, the first reference to `WCOUNT<ITEM>` creates a new table entry and returns the null string. Integer 1 is added to the null string, and the result, 1, is stored back into `WCOUNT<ITEM>`. If the same word is encountered again, `WCOUNT<ITEM>` for that word will be incremented to 2.

The program reads the input file, building a table with entries for each unique word. When end-of-file is read, control transfers to label `DONE`, where we display the words and their respective counts. We convert `WCOUNT` to an array, and use integer subscripts to retrieve the words and their counts. Conversion fails if the table is empty. Continuing with this program:

```
*  Convert table to array.  Fail if table is empty
DONE    A = CONVERT(WCOUNT, 'ARRAY')              :F(EMPTY)

*  Scan array, printing words and counts
        I = 0
PRINT   I = I + 1
        OUTPUT = A<I,1> '--' A<I,2>     :S(PRINT) F(END)

EMPTY   OUTPUT = 'No words'
END
```

The table subscripts were the file's words, and have been placed in the first column of the array, `A<I,1>`. The count for each word was the table entry, now in the second column, `A<I,2>`. Tables are very convenient for recording information about data items, while conversion to an array makes it easy to systematically examine the recorded information.

7.6 The name operator

The unary *name* operator provides the address or location in memory where a variable is stored. Its graphic symbol is the period (`.`). We'll introduce it here through an example.

Consider the indirect reference operator mentioned earlier. Suppose we want to use a variable to point to different elements of an array or table. If we try the following, we immediately discover a problem:

```
?       A = ARRAY('10,10')
?       A[4,2] = 'DOG'
?       V = 'A[4,2]'
?       OUTPUT = $V
Success
```

The indirect reference operator treats the string `'A[4,2]'` as a variable name, rather than an array element. Remember, any character sequence can be used indirectly to create a variable. SNOBOL4 creates a variable called `A[4,2]` that has absolutely no connection with array `A`. The fact that this character sequence happens to look like an array reference to us is purely coincidental from SNOBOL4's point of view.

To make this work, the name operator is applied to `A[4,2]` to obtain the address of that array element. The address can be stored in variable `V`, and referenced with the indirect operator:

```
?       V = .A[4,2]
?       OUTPUT = $V
DOG
```

The name operator provides a general method for specifying the name of an object. Both of these statements are correct for specifying the first argument to the `INPUT` function:

```
INPUT('INFILE', 1, , 'CAPITAL.DAT')
INPUT(.INFILE,  1, , 'CAPITAL.DAT')
```

Either form, `'INFILE'` or `.INFILE`, tells the `INPUT` function the name of the variable to be input associated. However, using the name operator allows us to associate a file with an array or table element:

```
INPUT('A[4,2]', 1, , 'CAPITAL.DAT')  (incorrect)
INPUT(.A[4,2],  1, , 'CAPITAL.DAT')
```

Note that alternate use of the indirect reference and name operators cancel one another, so

```
?       OUTPUT = $(.($(.A[4,2])))
DOG
```

is simply a reference to `A[4,2]`.

Chapter 8

Program-defined Objects

SNOBOL4 is a very large and rich language, providing a diverse assortment of built-in features. It is also an extensible language; it allows you to define new data types, functions, and operators. You can, by creating your own entities, obtain another level of conciseness and power of expression.

We will begin with program-defined functions because they allow a program to be partitioned into smaller, more manageable segments. As functions tend to be just a few lines long, transfers of control within them are usually obvious and manageable. If your main program has complex, intertwined Gotos, consider how the use of functions would clarify things.

Functions also allow us to postpone the complete development of an algorithm. We can design the overall program structure, using function names for components which will be developed later. Furthermore, if a particular function proves inefficient, it can be replaced later with an improved version.

8.1 Program-defined functions

The concept of a function should be clear from all the examples of SNOBOL4's built-in functions. A function accepts some number of arguments, performs a computation based on their values, and returns a result and a success signal. A function can also signal failure, and not return any value.

8.1.1 Function definition

We can define a new function by specifying its name and arguments. The definition will be composed of dummy arguments – place holders that show how the arguments are to be used in the function. Later, when the function is called, the actual arguments will replace the dummy arguments in the computation.

We define a new function in SNOBOL4 by using the built-in function `DEFINE`. We call it with a prototype string containing the new function's name and arguments. `DEFINE` makes the new function's name known to SNOBOL4, so it can be used subsequently.

Suppose we want to create a new function called `SHIFT`, which would circularly rotate a string through a specified number of character positions. We'll define all rotations as being to the left – characters removed from the front of the string are placed back on the end. For example, `SHIFT('ENGRAVING',3)` would return the string `'RAVINGENG'`.

We will begin by defining the function name and its dummy arguments, `S` and `N`. Any names of your choosing can by used for dummy arguments. In a program, it would look like this:

```
DEFINE('SHIFT(S,N)')
```

It is important to realize that the `DEFINE` function must be executed for the definition to occur. Most other programming languages process function definitions when a program is compiled.

SNOBOL4's system is more flexible; the prototype string can itself be the result of other run-time computations. In an extreme case, data input to a program could determine the names and kinds of functions to be defined.

8.1.2 The function body

Having declared the function name and dummy arguments, we need to provide the statements which will implement the function. A very simple convention applies:

When the function is used, SNOBOL4 transfers control to a statement label with the same name as the function.

In this case, the first statement of the function would be labeled `SHIFT`. There is no limit to the number of statements comprising the function body.

8.1.3 Returning function results

First, a function may return a value by assigning it to a variable with the same name as the function. If no assignment occurs, the result is the null string.

Second, the function must tell SNOBOL4 that it is finished, and that control should return back to the caller. It does this by transferring to the special label `RETURN`.

The label `RETURN` should not appear anywhere in your program. It is a special name, reserved by SNOBOL4 for just this purpose.

With this information, we can now write our `SHIFT` function. We will remove the first `N` characters from the beginning of the argument string, and place them on the end. The function body looks like this:

```
SHIFT   S LEN(N) . FRONT REM . REST
        SHIFT = REST FRONT                      :(RETURN)
```

Each time `SHIFT` is called, the particular arguments used are placed in `S` and `N`. The first statement splits `S` into two parts, assigning them to variables `FRONT` and `REST`. The second statement reassembles them in the shifted order, and assigns them to variable `SHIFT`, to be returned as the function result. The Goto then transfers to label `RETURN` to return back to the caller.

8.1.4 Function failure

What happens if we try the function call `SHIFT('PEAR',7)`? As the function is defined above, the pattern match would fail, since `LEN(7)` is longer than the subject string. The assignment to `FRONT` and `REST` would not take place, and the function would return an erroneous result.

Now we could extend the definition of `SHIFT` to cycle the argument string multiple times. In general, though, we want to develop a convenient method that allows a function to signal an exceptional condition back to the caller. Function failure allows us to do just that. Another convention is provided:

Transferring to the special label `FRETURN` returns from a function signaling failure to the caller. No value is returned as the function result.

We can now rework the function body to signal failure when `N` is too large. In this case, the pattern match fails, and we detect the failure in the Goto field:

```
SHIFT   S LEN(N) . FRONT REM . REST          :F(FRETURN)
        SHIFT = REST FRONT                   :(RETURN)
```

In general, the transfer to `FRETURN` does not need to be the result of the failure of a particular statement. Any success or failure could be tested to produce a transfer to `FRETURN`. For example, if we decided to explicitly test the length of `S`, the function could begin with:

```
SHIFT   GT(N, SIZE(S))                       :S(FRETURN)
        ...
```

8.1.5 Local variables

FRONT and REST were used in this function as temporary variables to rearrange the argument string. If they had appeared elsewhere in your program, their old values would be destroyed. Such inadvertent conflicts become harder to avoid as your function library grows. The prototype string used with DEFINE can specify local variables to be protected when the function is called. For our SHIFT function, the call would look like this:

```
DEFINE('SHIFT(S,N)FRONT,REST')
```

The local variables appear after the argument list. When SHIFT is called, any existing values for FRONT and REST will be saved on a pushdown stack. FRONT and REST are set to the null string, and control is transferred to the first statement of the function body. When the function returns, FRONT and REST are restored to their previous values.

Since the same potential problem exists for dummy arguments S and N, SNOBOL4 automatically saves their values before assigning the actual arguments to them. And just like local variables, when the function returns, the dummy arguments are restored to their original values.

8.1.6 Using functions

Once a function has been defined, it may be used in exactly the same manner as a built-in function. It may appear in a statement anywhere its value is needed – in the subject, pattern, or replacement fields. If used with the indirect reference operation, functions may even be used in the Goto field. Of course, a function may be used as the argument of another function.

The value returned by a function is not restricted to strings. Any SNOBOL4 data type, including patterns, may be returned. Earlier, in the *Pattern Matching* chapter, we showed how simple patterns could be tailored to our needs by using them in more complicated clauses. The specific example was a variation of the BREAK pattern which would not match the null string. Let's use a programdefined function to create a new function, BREAK1, with this property. The definition statement might look like this:

```
DEFINE('BREAK1(S)')
```

and the function body, like this:

```
BREAK1  BREAK1 = NOTANY(S) BREAK(S)          :(RETURN)
```

This function can now be used directly in a pattern match. For example, BREAK1('abc') constructs a pattern which matches a nonnull string, up to the occurrence of the letters 'a', 'b', or 'c'. Of course, the pattern returned by a function can be as complex as desired, giving us an elegant method to define our own pattern matching primitives.

8.1.7 Organizing functions

SNOBOL4 does not know or care which statements belong to a particular function. There is no explicit END statement for individual functions. To keep programs readable, we'll have to impose some discipline of our own. Also, having to execute the DEFINE function is a mixed blessing. It offers tremendous flexibility, but requires us to place all our DEFINE's at the beginning of a program. Here is the system proposed by Gimpel, which I like to use to manage functions and their definitions:

We keep the function definition, any one-time initialization, and the function body together as a unit. A Goto transfers control around the function body after the definition and initialization statements are executed. Also present are comments describing its use and any exceptional conditions. Rewriting the SHIFT function in this form, and taking this opportunity to avoid rebuilding the pattern each time the function is called, it looks like this:

```
* SHIFT(S,N)  --  Shift string S left N character positions.
*  As characters are removed from the left side of the
*  string, they are placed on the end.
*
*  The function fails if N is larger than the size of S.

        DEFINE('SHIFT(S,N)FRONT,REST')
        SHIFT_PAT = LEN(*N) . FRONT REM . REST   :(SHIFT_END)

SHIFT   S SHIFT_PAT                              :F(FRETURN)
        SHIFT = REST FRONT                       :(RETURN)
SHIFT_END
```

Now this group of lines can be incorporated as a unit into the beginning of any program that wants to use it. When execution begins, the first statement defines the `SHIFT` function. Next we define a pattern, called `SHIFT_PAT`, for use when the function is called. The pattern definition is only executed once, so we use the unevaluated expression operator (`*N`) to obtain the current value of `N` on each function call. After defining the pattern, we jump around the function body, to label `SHIFT_END`. (Remember, we are defining the function now, not executing it; falling into the function body would be an error.) The function is now defined, and ready to be used.

In general, functions should be prepared in this form:

```
* fname  -- description of use
        DEFINE('fname(arg1,arg2,...,args)local1,local2,...,localt')
        ...
*       a one-time initialization for fname
        ...                                      :(fname_END)
fname   function body
        ...
fname_END
```

If you place your functions in individual disk files, they can be included in new programs as necessary. By preparing functions in this form, they will all be defined and initialized when execution begins.

When discussing pattern matching, we used a pattern to convert a character to its ASCII decimal value. In BASIC, two functions are provided for similar operations: `ASC` and `CHR$`. We can create SNOBOL4 equivalents like this:

```
* ASC(S) -- Return the ASCII code for the first character of
*          string S.
*
*       The value returned is an integer between 0 and 255.
*       The function fails if S is null.

        DEFINE('ASC(S)C')
        ASC_ONE = LEN(1) . C
        ASC_PAT = BREAK(*C) @ASC             :(ASC_END)

ASC     S ASC_ONE                            :F(FRETURN)
        &ALPHABET ASC_PAT                    :(RETURN)
ASC_END

* CHR(N) -- Converts an integer ASCII code to a one character
*          string.
*
*       The argument N is an integer between 0 and 255.
```

```
*       The function fails if N is greater than 255.

        DEFINE('CHR(N)')
        CHR_PAT = TAB(*N) LEN(1) . CHR       :(CHR_END)

CHR     &ALPHABET CHR_PAT              :S(RETURN) F(FRETURN)
CHR_END
```

Note that both functions were written to work correctly regardless of the anchoring mode in use by the calling program.

(The `CHR` function is shown here as an example only. Vanilla SNOBOL4 provides a built-in function, `CHAR(N)`, for this purpose. See chapter *Built-in Functions* of the Reference Manual.)

8.1.8 Call by value and call by name

Function calls in SNOBOL4 transmit the value of the argument to the function. Variables used in the function call cannot be harmed by the function. This type of function usage is referred to as 'call by value'. Occasionally, we might want the function to access the argument variables themselves. The name (`.`) operator introduced in the previous chapter provides this ability. The function call still transmits a value, but the value used is the name of a variable.

Consider a function called `SWAP`, which will exchange the contents of two variables. If we wanted to exchange the contents of variables `COUNT` and `OLDCOUNT`, we would say `SWAP(.COUNT, .OLDCOUNT)`. The function looks like this:

```
* SWAP(.V1, .V2) -- Exchange the contents of two variables.
*  The variables must be prefixed with the name operator
*  when the function is called.

        DEFINE('SWAP(X,Y)TEMP')              :(SWAP_END)

SWAP    TEMP = $X
        $X = $Y
        $Y = TEMP                            :(RETURN)
SWAP_END
```

The name operator allows us to access the argument variables. If we had not used it, the function would be called with the variables' values, with no indication of where they came from. Calls to `SWAP` are not limited to simple variable arguments. Anything capable of receiving the name operator, such as array and table elements, could be used: `SWAP(.A<4,3>, .T<'YOU'>)`.

There are certain situations where call by name occurs implicitly. If the argument is an array or table name, or a programdefined data type (discussed below), it points to the actual data object, which can then be modified by the function. For example, if `FILL` were a function which loads an array with values read from a file, the statements

```
A = ARRAY(25)
FILL(A)
```

would cause array `A` to be altered.

8.1.9 Functions and `CODE.SNO`

The `CODE.SNO` program was provided to allow interactive experiments with SNOBOL4 statements. If you create functions using the preceding format, they also can be tested using `CODE.SNO`.

Use your text editor to create a disk file containing the `SHIFT` function. (Be certain to include the Goto that transfers around the function body.) Call the file `SHIFT.SNO`. Now, start the `CODE.SNO` program, and type the following:

```
?       SLOAD('SHIFT.SNO')
Success
?       OUTPUT = SHIFT('COTTON',4)
ONCOTT
?       OUTPUT = SHIFT('OAK',4)
Failure
```

8.1.10 Recursive functions

The statements that comprise a function are free to call any functions they choose, including the function they are defining. Of course, for this to make sense, they must call themselves with a simplified version of the original problem, or an endless loop would result. Eventually, the function calls itself with an argument so simple that it can return an answer without any further recursive calls. It's like winding a clock spring up. The central, non-recursive answer to the innermost call provides an answer to the next turn out, with the recursive calls unwinding until the original problem can be solved.

There is no explicit declaration for recursion; any function can be used recursively if it is designed properly. However, all local variables should be declared in the `DEFINE` function so they will be saved and restored during recursive calls.

Sometimes, recursion can produce dramatically smaller programs. *'Algorithms in* SNOBOL4*'* provides a example with the recursive function, `ROMAN`. It convert's an integer in the range 0 to 3 999 to its Roman numeral equivalent. Two premises are required:

1. We know the Roman numerals for the numbers 0 to 9 (null, I, II, ..., IX), and can perform this conversion with a simple pattern match.

2. We can use the `REPLACE` function to multiply a number in Roman form by 10 by replacing `I` by `X`, `V` by `L`, `X` by `C`, etc.

The function uses these two rules to produce a recursive solution for some integer `N`. The algorithm looks like this: The rightmost digit is removed from the argument and converted by premise 1. Removing the digit effectively divides the argument by 10, simplifying the problem.

The reduced argument is then converted by calling `ROMAN` recursively and multiplying the result by 10 according to premise 2.

The previously converted unit's digit is appended to the result.

Here's the function (note that a plus sign in column one allows a statement to be continued over several lines):

```
* ROMAN(N) -- Convert integer N to Roman numeral form.
*
*  N must be positive and less than 4000.
*
*  An asterisk appears in the result if N >= 4000.
*
*  The function fails if N is not an integer.

        DEFINE('ROMAN(N)UNITS')              :(ROMAN_END)

*  Get rightmost digit to UNITS and remove it from N.
*  Return null result if argument is null.
ROMAN   N RPOS(1) LEN(1) . UNITS =           :F(RETURN)

*  Search for digit, replace with its Roman form.
*  Return failing if not a digit.
        '0,1I,2II,3III,4IV,5V,6VI,7VII,8VIII,9IX,'  UNITS
+         BREAK(',') . UNITS                 :F(FRETURN)
```

```
*  Convert rest of N and multiply by 10.  Propagate a
*  failure return from recursive call back to caller.
        ROMAN = REPLACE(ROMAN(N), 'IVXLCDM', 'XLCDM**')
+               UNITS            :S(RETURN) F(FRETURN)
ROMAN_END
```

The first call to ROMAN may have an integer argument. The statement labeled ROMAN causes N to be converted to a string, and subsequent recursive calls use a string argument. The recursive calls cease when reducing N finally produces a null string argument – the match at statement ROMAN fails, and the function returns immediately with a null result.

8.2 Program-defined data types

With the exception of arrays and tables, a variable may have only one item of data in it at a time. In many applications, it is convenient if several data items can be associated with a variable. For example, if we wanted to work with complex numbers, a variable should contain two numbers – the real and imaginary parts. In an inventory system, an individual product might require values such as name, price, quantity, and manufacturer.

Program-defined data types enlarge SNOBOL4's repertoire to include new objects such as COMPLEX or PRODUCT. SNOBOL4 only provides a system for managing these new types; defining a data type does not magically invest SNOBOL4 with a knowledge of complex arithmetic or inventory accounting. It is still up to you to provide the computational support for each new type.

8.2.1 Data type definition

A program-defined data type will consist of a number of fields, each containing an individual data element. We begin by selecting names for the data type and fields. An inventory system might use the data type name PRODUCT, and field names NAME, PRICE, QUANTITY, and MFG.

A data type is defined by providing a prototype string to the built-in DATA function. The prototype assumes a form similar to a function call, with the data type taking the place of the function name, and field names for the arguments. The form of the prototype string is:

$$typename(field_1, field_2, \ldots, field_n)$$

Blanks are not permitted within a prototype. Try creating a new data type using the CODE.SNO program:

```
?       DATA('PRODUCT(NAME,PRICE,QUANTITY,MFG)')
Success
```

The DATA function tells SNOBOL4 to define a four-argument object creation function with the new data type's name:

```
PRODUCT(...)
```

This new function can be called whenever we wish to create a new object with the PRODUCT data type. Its arguments are the initial values to be given to the four fields which comprise a PRODUCT. The function returns a pointer to the new object, which can be stored in a variable, array, or table. Try creating two new objects as follows:

```
?      ITEM1 = PRODUCT('CAPERS', 2, 48, 'BRINE BROTHERS')
?      ITEM2 = PRODUCT('PICKLES', 1, 72, 'PETER PIPER INC.')
```

8.2.2 Data type use

The defining call to the DATA function also created several field reference functions. In this case, they are:

```
NAME()    PRICE()    QUANTITY()    MFG()
```

The argument used with each function is an object created by the PRODUCT function. Try accessing ITEM1's fields:

```
?       OUTPUT = MFG(ITEM1)
BRINE BROTHERS
?       OUTPUT = PRICE(ITEM1) * QUANTITY(ITEM1)
96
```

We can alter the value of a field after an object is created. Field reference functions can also be used as the object of an assignment, so:

```
?       QUANTITY(ITEM2) = QUANTITY(ITEM2) - 12
```

changes the QUANTITY field of ITEM2 from 72 to 60.

8.2.3 Copying data items

It is important to recognize that variables like ITEM1 and ITEM2 contain pointers to the data. Assigning ITEM1 to another variable, say LASTITEM, merely copies the pointer; both variables still point to the same physical packet of data in memory. Altering the QUANTITY field of ITEM1 would alter the QUANTITY field of LASTITEM. This is the same behavior observed earlier for array and table names.

The built-in COPY function creates a unique copy of an object – one which is independent of the original. Try using it with CODE.SNO:

```
?       LASTITEM = COPY(ITEM1)
?       QUANTITY(ITEM1) = 24
?       OUTPUT = QUANTITY(LASTITEM)
48
```

8.2.4 Creating structures

Our inventory example used string and integer values as the field contents. In fact, any SNOBOL4 data type may be stored in a field, including pointers to other program-defined types. Complex structures, such as queues, stacks, trees, and arbitrary graphs may be created.

For example, if we wanted to link together all products made by the same manufacturer, PRODUCT could be defined with an additional field. We won't go through the exercise with CODE.SNO, but will sketch out the changes:

```
DATA('PRODUCT(NAME,PRICE,QUANTITY,MFG,MFGLINK')
```

As each product is defined, we will determine if we have another product from the same manufacturer. If so, MFGLINK is set to point to that other product. If not, it is set to the null string. A table M provides a convenient way to keep track of manufacturers. Assume variable COMPANY contains the manufacturer's name as each product is defined. Then all of the requisite searching and linking can be accomplished in one statement:

```
M<COMPANY> = PRODUCT(..., ..., ..., COMPANY, M<COMPANY>)
```

If this is the company's first appearance, it is not in the table, and the last argument to the PRODUCT function sets MFGLINK to the null string. The assignment statement uses the company as the table subscript, and the entry points to the current product.

If another product definition uses the same company, MFGLINK will point to the previous product, and the table will be updated to point to the current product. In this manner, all products from a manufacturer will be threaded together. Each thread starts with a table entry, and goes through each product's MFGLINK field, ending with a null string in the last product's MFGLINK.

Now if we wanted to display all products supplied by a particular manufacturer, we select and follow the appropriate thread:

```
        X       =  M<COMPANY>
LOOP    OUTPUT  =  DIFFER(X) NAME(X)          :F(DONE)
        X       =  MFGLINK(X)                 :(LOOP)
DONE
```

8.2.5 The DATATYPE function

The DATATYPE function allows you to learn the type of data in a particular variable. It is useful when the kind of processing to be performed depends on the data type. The formal data type name is returned as an upper-case string:

```
?       OUTPUT = DATATYPE(54)
INTEGER
?       OUTPUT = DATATYPE(ITEM1)
PRODUCT
```

8.3 Program-defined operators

If you can define new functions and data types, why not new operators too? Indeed, SNOBOL4 provides this feature, although most programs can be written without it. For the sake of completeness, we'll provide a brief discussion.

8.3.1 Operators and functions

Unary or binary operators can be thought of as functions of one or two arguments. For example, A + B can be written in functional form as PLUS(A,B), where PLUS is some function which implements addition. Operators can be redefined by specifying a function to replace them. We still write our program in terms of the operator's graphic symbol, but SNOBOL4 will use the specified function whenever the operator must be performed.

The built-in function OPSYN creates synonyms and new definitions for operators. Synonyms permit different names or symbols to be used in place of a function or operator. The general form of OPSYN is:

OPSYN(*new name*, *old name*, *i*)

The new name is defined as a synonym of the old name. The third argument is 0, 1, or 2 if we are defining functions, unary operators, or binary operators respectively.

8.3.2 Function synonyms

We can make the name LENGTH a synonym for the SIZE function:

```
?       OPSYN('LENGTH', 'SIZE', 0)
?       OUTPUT = LENGTH('RABBIT')
6
```

The word synonym is not quite an accurate description of OPSYN. The name LENGTH becomes associated with the code that implements the SIZE function, not with the word SIZE per se. If SIZE was subsequently redefined – perhaps as a program-defined function – LENGTH would continue to return the number of characters in a string.

8.3.3 Operator synonyms

Take a moment to examine the tables in chapter *Operators* of the Reference Manual. Note that in each table there are a number of operator symbols whose definition is `<none>`.

If you use an undefined binary operator, you'll get an error:

```
?       OUTPUT = 1 # 1
Execution error #5, Undefined function or operation
```

However, we could make this operator synonymous with the `DIFFER` function (which also uses two arguments) and use it instead:

```
?       OPSYN('#', 'DIFFER', 2)
?       OUTPUT = 1 # 2
Failure
```

Conversely, we can define a function in place of an operator:

```
?       OPSYN('PLUS', '+', 2)
?       OUTPUT = PLUS(4, 5)
9
```

Unary operators can be similarly treated, using 1 as the third argument:

```
?       OPSYN('!', 'ANY', 1)
?       'ABC321' !'3C' . OUTPUT
C
```

Operators can be created to maintain a stack, or navigate around a tree. The full generality of functions and programdefined data types are available to your operators. Through this technique you can make SNOBOL4 speak the language of your particular problem.

Chapter 9

Advanced Topics

The material presented so far allows you to write powerful SNOBOL4 programs. In this chapter, we will examine other interesting and useful features of the language.

9.1 The ARBNO function

This function produces a pattern which will match zero or more consecutive occurrences of the pattern specified by its argument. As its name implies, `ARBNO` is useful when an arbitrary number of instances of a pattern may occur. For example, `ARBNO(LEN(3))` matches strings of length 0, 3, 6, 9, ... There is no restriction on the complexity of the pattern argument.

Like the `ARB` pattern, `ARBNO` is shy, and tries to match the shortest possible string. Initially, it simply matches the null string. If a subsequent pattern component fails to match, SNOBOL4 backs up, and asks `ARBNO` to try again. Each time `ARBNO` is retried, it supplies another instance of its argument pattern. In other words, `ARBNO(PAT)` behaves like

```
( '' |  PAT | PAT PAT | PAT PAT PAT | ... )
```

Also like `ARB`, `ARBNO` is usually used with adjacent patterns to draw it out. Let's consider a simple example. We want to write a pattern to test for a list. We'll define a list as being one or more numbers separated by comma, and enclosed by parentheses. Use `CODE.SNO` to try this definition:

```
?       ITEM = SPAN('0123456789')
?       LIST = POS(0) '(' ITEM  ARBNO(',' ITEM) ')' RPOS(0)
?       '(12,345,6)' LIST
Success
?       '(12,,34)' LIST
Failure
```

`ARBNO` is retried and extended until its subsequent, `')'`, finally matches. `POS(0)` and `RPOS(0)` force the pattern to be applied to the entire subject string.

Alternation may be used within `ARBNO`'s argument. This pattern matches any number of pairs of certain letters:

```
?       PAIRS = POS(0) ARBNO('AA' | 'BB' | 'CC') RPOS(0)
?       'CCBBAAAACC' PAIRS
Success
?       'AABBB' PAIRS
Failure
```

9.2 Recursive patterns

This is the pattern analogue of a recursive function – a pattern is defined in terms of itself. The unevaluated expression operator makes the definition possible.

Suppose we wanted to expand the previous definition of a list to say that a list item may be a span of digits, or another list. The definition proceeds as before, except that the unevaluated expression operator is used in the first statement; the concept of a list has not yet been defined:

```
?       ITEM = SPAN('0123456789') | *LIST
?       LIST = '(' ITEM  ARBNO(',' ITEM) ')'
?       TEST = POS(0) LIST RPOS(0)
?       '(12,(3,45,(6)),78)' TEST
Success
?       '(12,(34)' TEST
Failure
```

Recursion occurs because `LIST` is defined in terms of `ITEM`, which is defined in terms of `LIST`, and so on. Note that functions `POS(0)` and `RPOS(0)` were moved out one level, to `TEST`, because `LIST` must now match substrings within the subject.

In our previous discussion of recursive functions, we said they work because successive calls present the function with progressively simpler problems, until the problem can be solved without further recursion. Similarly, patterns `ITEM` and `LIST` are applied to successively smaller substrings, until `ITEM` can use its `SPAN()` alternative instead of invoking `LIST` again.

In general, you will need an alternative somewhere in the recursive loop to allow the pattern matcher a way out. Also, you should place recursive objects last in a series of alternatives, so that the simpler, nonrecursive patterns are attempted first and recursive plunges can be avoided.

SNOBOL4 saves information on a pattern stack during the pattern match process. Heavily recursive patterns and long subject strings can sometimes result in stack overflow. If this occurs, you should break the problem apart into several smaller pattern matches.

As recursive patterns use the unevaluated expression operator, it is sometimes necessary to disable SNOBOL4's heuristics by setting `&FULLSCAN = 1`.

9.3 Quickscan and fullscan

Pattern matching can be very time-consuming because of the number of possibilities which must be attempted. In the normal quickscan mode, SNOBOL4 stops searching for a match when it thinks further efforts would be futile. The heuristics are complex, but can be summarized as follows: pattern matching fails when there are insufficient subject characters to satisfy the remaining pattern components.

The cursor operator can be used to demonstrate at what point SNOBOL4 gives up. For example, in the pattern match

```
?       'ABCD' @OUTPUT 'X' LEN(3)
0
Failure
```

SNOBOL4 does not attempt to match `'X'` against `'B'` because fewer than 3 subject characters remain after it, and `LEN(3)` could never succeed.

A second type of heuristic is the 'one character assumption' for unevaluated expressions. SNOBOL4 assumes that unevaluated expressions will match at least one character. This heuristic was originally provided to break recursive loops, but can cause programming problems when an unevaluated expression must match the null string. Consider a pattern which succeeds if `'B'` is at least 4 character positions beyond an `'A'` in the subject:

```
?       P = 'A' ARB $ X 'B' *GE(SIZE(X), 4)
?       'A12345BC' P
Success
?       'A12345B' P
Failure
```

The characters between 'A' and 'B' are matched by ARB, and immediately assigned to X. The size of X is then compared to 4 by the GE function, which succeeds and returns the null string. This null string result should not interfere with the pattern match, but we find the pattern misbehaves when 'B' is the last character of the subject. The unevaluated expression operator made SNOBOL4 assume a one character length for the GE function, and matching 'B' against the last subject character was never attempted.

For most pattern matching, heuristics are invisible. However, there are circumstances when we would like SNOBOL4 to be exhaustive in its match attempts. We can disable heuristics and enter fullscan mode by setting keyword &FULLSCAN nonzero:

```
?       &FULLSCAN = 1
?       'A12345B' P
Success
?       'ABCD' @OUTPUT 'X' LEN(3)
0
1
2
3
4
Failure
```

The quickscan mode can be reinstated by setting &FULLSCAN = 0.

9.4 Other primitive patterns

We can accomplish quite a lot with just the primitive patterns ARB and REM. However, there are five additional patterns which you should be aware of:

- ABORT End pattern match

  The ABORT pattern causes immediate failure of the entire pattern match, without seeking other alternatives. Usually a match succeeds when we find a subject sequence which satisfies the pattern. The ABORT pattern does the opposite: if we find a certain pattern, we will abort the match and fail immediately. For example, suppose we are looking for an 'A' or 'B', but want to fail if '1' is encountered first:

  ```
  ?       '--AB-1-' (ANY('AB') | '1' ABORT)
  Success
  ?       '--1B-A-' (ANY('AB') | '1' ABORT)
  Failure
  ```

  The last example may be confusing because the ANY function appears as the first alternative, fostering the illusion that it will find the 'B' in the subject before the other pattern alternative is tried. However, that is not the order of pattern matching; all pattern alternatives are tried at cursor position zero in the subject. If none succeed, the cursor is advanced by one, and all alternatives are tried again. When the cursor is in front of subject character '1', ANY still does not match, but the second alternative now does. As the '1's match, ABORT is reached, causing failure.

- BAL Match balanced string

  The BAL pattern matches the shortest nonnull string in which parentheses are balanced. (A string without parentheses is also considered to be balanced.) These strings are balanced:

  ```
  (X)      Y      (A!(C:D))      (AB)+(CD)      9395
  ```

  These are not:

```
)A+B      (A*(B+)      (X))
```

BAL is concerned only with left and right parentheses. The matching string does not have to be a well-formed expression in the algebraic sense; in fact, it needn't be an algebraic expression at all. Like ARB, BAL is most useful when constrained on both sides by other pattern components.

- FAIL Seek other alternatives

  The FAIL pattern signals failure of this portion of the pattern match, causing the pattern matcher to backtrack and seek other alternatives. FAIL will also suppress a successful match, which can be very useful when the match is being performed for its side effects, such as immediate assignment. For example, in unanchored mode, this statement will display the subject characters, one per line:

  ```
  SUBJECT LEN(1) $ OUTPUT FAIL
  ```

  LEN(1) matches the first subject character, and immediately assigns it to OUTPUT. FAIL tells the pattern matcher to try again, and since there are no other alternatives, the entire match is retried at the next subject character. Forced failure and retries continue until the subject is exhausted.

- FENCE Prevent match retries

  Pattern FENCE matches the null string and has no effect when the pattern matcher is moving left to right in a pattern. However, if the pattern matcher is backing up to try other alternatives, and encounters FENCE, the match fails.

  FENCE can be used to lock in an earlier success. Suppose we want to succeed if the first 'A' or 'B' in the subject is followed by a plus sign. In the following example, the 'A's match, we go through the FENCE, and find '+' does not match the next subject character, 'B'. SNOBOL4 tries to backtrack, but is stopped by the FENCE and fails:

  ```
  ?       '1AB+' ANY('AB') FENCE '+'
  Failure
  ```

  If FENCE were omitted, backtracking would match ANY to 'B', and then proceed forward again to match '+'.

  If FENCE appears as the first component of a pattern, SNOBOL4 cannot back up through it to try another subject starting position. This results in an anchored pattern, even if the &ANCHOR keyword specifies unanchored mode:

  ```
  ?       'ABC' FENCE 'B'
  Failure
  ```

- SUCCEED Match always

  This pattern matches the null string and always succeeds. If the scanner is backtracking when it encounters SUCCEED, it reverses and starts forward again. Placing a pattern between SUCCEED and FAIL causes the pattern matcher to oscillate.

9.5 Other functions

I'd like to briefly point out a few more built-in functions. See chapter *Built-in Functions* of the Reference Manual for a complete description of their use.

- APPLY

  Allows an indirect call to a function through a variable

- `CONVERT`

  Provides explicit conversion from one data type to another. Chapter *Data Types and Conversion* of the Reference Manual describes the conversions possible

- `ENDFILE`

  Closes a file and detaches all variables associated with it

- `ITEM`

  Allows an indirect reference to an array or table

- `LPAD` & `RPAD`

  These are padding functions, which will pad a string on its left or right side with blanks or a given character. Padding is provided to a specified width, and is useful when producing columnar output.

9.6 Other unary operators

Operation: negation
Symbol: `~` (tilde)

The negation operator, or tilde (`~`), inverts the success or failure result of its operand. If the expression `X` succeeds, then `~X` fails. Conversely, if `X` fails, `~X` succeeds and returns the null string.

Operation: interrogation
Symbol: `?`

Unary question mark is called the interrogation operator, although 'value annihilation' might be more descriptive. If `X` is an expression which fails, `?X` also fails. However, if `X` succeeds, `?X` also succeeds, and returns the null string. In other words, any value component of `X` is replaced by the null string.

9.7 Run-time compilation

The two functions described below are among the most esoteric features, not just of SNOBOL4, but of all programming languages in existence. While your program is executing, the entire SNOBOL4 compiler is just a function call away.

A SNOBOL4 program is nothing more than a string of characters. The functions `EVAL` and `CODE` let you supply the compiler with character strings from within the program itself.

9.7.1 The `EVAL` function

This function is used to evaluate an expression. Its argument may take a number of forms:

- If the argument is an integer, or a number in string form, the number is returned as the function result:

```
?       OUTPUT = EVAL(19)
19
```

- If the argument is an unevaluated expression, it is evaluated using current values for any variables it might contain. `EVAL` returns the expression's value as its result:

```
?       E = *('N SQUARED IS ' N ** 2)
?       N = 15
?       OUTPUT = EVAL(E)
N SQUARED IS 225
```

This is similar to our earlier use of unevaluated expressions with patterns. In this case, however, the unevaluated expression operator (*) must be applied to the entire expression to create an object with the EXPRESSION data type.

- If the argument is a string (other than a simple number), EVAL tries to compile it as a SNOBOL4 expression. Only an expression is permitted – not an entire SNOBOL4 statement:

```
?       OUTPUT = EVAL('3 * N + 2')
47
```

If the string compiles without error, EVAL then evaluates the expression and returns the result.

It is this last use of EVAL – to compile a string – which is the most interesting. Here is a trivial program which behaves like a simple desk calculator.

```
LOOP    OUTPUT = EVAL(INPUT)                    :S(LOOP)
END
```

You can easily try it by placing a semicolon after the Goto to protect it from CODE.SNO's own machinations:

```
?LOOP   OUTPUT = EVAL(INPUT)                    :S(LOOP);
4 * (5 - 2) / 2
6
N + 1
16
^Z
```

The program reads a line of input, compiles and evaluates it, and displays the result. Each expression you enter must be wellformed according to SNOBOL4's syntax rules. In particular, this means there must be blanks around the binary operators.

The BNF program included with Vanilla SNOBOL4 demonstrates that EVAL's power is useful even if the input data does not conform to SNOBOL4 syntax. It reads a definition of a grammar from a file, and converts it to SNOBOL4 patterns.

EVAL fails if evaluation of the argument fails, or if the argument contains a syntax error. The SNOBOL4 keyword &ERRTEXT will contain a string describing the error.

The expressions used with EVAL may return any SNOBOL4 data type, not just numbers. For instance, the expression might construct a new pattern, and return it as the result:

```
ITEM = EVAL('SPAN("0123456789") | *LIST')
```

Note that EVAL can only call the compiler with a string argument. If we used a pattern as the argument, we would produce an execution error:

```
ITEM = EVAL(SPAN("0123456789") | *LIST)  (incorrect)
```

9.7.2 The CODE function

CODE accepts a string argument containing one or more statements to be compiled. Multiple statements are separated by semicolons (;). Statements may be labeled, and can include all the usual components – subject, pattern, replacement, and Goto. However, comment and continuation statements are not permitted.

The CODE function compiles the statements, and returns a pointer to the resulting object code. It fails if any statement contains an error, and places an error message in &ERRTEXT.

There are two ways to execute the new object code.

- Transfer to a label which is defined in the new code:

```
*  Compile a sample piece of code:
        S = 'L OUTPUT = N; N = LT(N,10) N + 1  :S(L)F(DONE)'
        CODE(S)
*  Transfer to a label in it:
                                                :(L)
*  Come here when the new code transfers back.
DONE     ...
```

Notice how we placed a Goto from the new code back to label `DONE` in the main program. If we had not done this, SNOBOL4 would terminate when execution fell out of the bottom of the new code block.

- The pointer returned by the `CODE` function can be used in a direct Goto to transfer to the first statement in the code block. A direct Goto is performed by enclosing the pointer in angular brackets in the Goto field:

```
*  Compile a sample piece of code:
        S = 'L OUTPUT = N; N = LT(N,10) N + 1  :S(L)F(DONE)'
        C = CODE(S)
*  Transfer to the first statement in the block:
                                                :<C>
DONE     ...
```

Labels contained in the new program fragment override any labels of the same name in your main program. This provides the ability to write self-modifying SNOBOL4 programs, and makes the division between code and data far less distinct than in other high-level languages.

Chapter 10

Debugging and Program Efficiency

10.1 Debugging and tracing

You are probably well aware of the diversity of potential errors when writing computer programs. They range from simple typographical errors made while entering a program, to subtle design problems which may only be revealed by unexpected input data.

Debugging a SNOBOL4 program is not fundamentally different than debugging programs written in other languages. However, SNOBOL4's syntactic flexibility and lack of type declarations for variables produce some unexpected problems. By way of compensation, an unusually powerful trace capability is provided.

Of course, there may come a time when you can't explain your program's behavior, and decide 'the system' is at fault. No guarantee can ever be made that SNOBOL4 is completely free of errors. However, its internal algorithms have been in use in other SNOBOL4 systems since 1967, and all known errors have been removed. Often the problem is a misunderstanding of how a function works with exceptional data, and a close reading of the reference section clears the problem up. In short, suspect the system last.

10.1.1 Compilation errors

Compilation errors are the simplest to find; SNOBOL4 displays the erroneous line on your screen with its statement number, and places a marker below the point where the error was encountered. The source file name, line number, and column number of the error are displayed for use by your text editor. Only the first error in a statement is identified, so you should also carefully check the remainder of the statement. A typical line looks like this:

```
32              ,OUTPUT = CNT+ 1
                 ^
test.sno(57,10) : Compilation Error : Erroneous statement
```

Here, the comma preceding the word `OUTPUT` is misplaced. The message indicates that `,OUTPUT` is not a valid language element.

Programs containing compilation errors can still be run, at least until a statement containing an error is encountered. When that happens, SNOBOL4 will produce an execution error message, and stop.

A complete description of error messages is provided in chapter *System Messages* of the Reference Manual.

10.1.2 Execution errors

Once a program compiles without error, testing can begin. Two kinds of errors are possible: SNOBOL4 detectable errors, such an incorrect data type or calling an undefined function, and program logic errors that produce incorrect results.

With the first type of error, you'll get a SNOBOL4 error message with statement and line numbers. Inspecting the offending line will often reveal typing errors, such as a misspelled function name, keyword, or label. If the error is due to incorrect data in a variable – such as trying to perform arithmetic on a non-numeric string – you'll have to start debugging to discover how the incorrect data was created. Placing output statements in your program, or using the trace techniques described below, will usually find such errors.

Here are some common errors to look for first:

- Setting keywords `&ANCHOR`, `&FULLSCAN`, and `&TRIM` improperly. We may have written a program with anchored pattern matching in mind, but let an unanchored match slip in inadvertently. Forgetting to set `&TRIM` to 1 causes blanks to be appended to input lines, and they usually interfere with pattern matching and conversion of a string to an integer.

- Misspelled variable names. Using `PUTPUT` instead of `OUTPUT`, as in:

  ```
  PUTPUT = LINE1
  ```

  creates a new variable and assigns `LINE1` to it. Worse still is using a misspelled name as a value source, since it will return a null string value.

  The first type of error is relatively easy to find – produce an end-of-run dump by using the SNOBOL4 command line option `/D`. You can study the list of variables for an unexpected name. The second type of error is naturally much harder to find, because variables with null string values are omitted from the end-of-run dump. In this case, you will have to study the source program closely for misspellings.

- Spurious spaces between a function name and its argument list. A line like:

  ```
  LINE = TRIM (INPUT)
  ```

  is not a call to the `TRIM` function. The blank between `TRIM` and the left parenthesis is interpreted as concatenating variable `TRIM` with the expression `(INPUT)`. `TRIM` used as a variable is likely to be the null string, so `INPUT` is returned unchanged.

- No blank space after a binary operator. SNOBOL4 sees a unary operator instead, with completely unexpected results. For instance:

  ```
  X = Y -Z
  ```

  concatenates `Y` with the expression `-Z`.

- Confusion occurring when a variable contains a number in string form. When used as an argument to most functions, conversion from string to number is automatic, and proper execution results. However, functions `IDENT` and `DIFFER` do not convert their arguments, and seemingly equal values are thought to be different. For example, if we want to test an input line for the number 3, the statements:

  ```
  N = INPUT
  IDENT(N,3)                                    :S(OK)
  ```

  are not correct. `N` contains a string, which is a different data type from the integer 3. This could be corrected by using `IDENT(+N,3)`, or `EQ(N,3)`. Once again, `&TRIM` should be 1, or the blanks appended to `N` will prevent its conversion to an integer.

- Omitting the assignment operator when we wish to remove the matching substring from a subject, resulting in a program which loops forever. For example, our word-counting program replaced each word with the null string:

  ```
  NEXTWRD LINE WRDPAT =                         :F(READ)
  ```

  However, by omitting the equal sign we would repeatedly find the same first word in `LINE`:

  ```
  NEXTWRD LINE WRDPAT                           :F(READ)
  ```

- Unexpected statement failure, with no provision for detecting it in the Goto field. For example, the CONVERT function fails if the table being converted is empty:

```
RESULT = CONVERT(TALLY, "ARRAY")
```

  RESULT will not be set if CONVERT fails, and a subsequent array reference to RESULT would produce an execution error.

- Failure can be detected but misinterpreted when there are several causes for it in a statement. This statement fails when an end-of-file is read, or if the input line does not contain any digits:

```
INPUT SPAN('0123456789') . N          :F(EOF)
```

  In the latter case, if we want to generate an error message, the statement should be split in two:

```
N = INPUT                             :F(EOF)
N SPAN('0123456789') . N              :F(WARN)
```

- Using operators such as alternation (|) and conditional assignment (.) for purposes other than pattern construction. Using them in the subject field will produce an 'Illegal data type' error message. Using them in the replacement field produces a pattern, intended for subsequent use in a pattern match statement. For example, this statement sets N to a pattern; it does not replace it with the words 'EVEN' or 'ODD', as was probably intended:

```
N = EQ(REMDR(N,2),0) 'EVEN' | 'ODD'
```

  We note in passing that SNOBOL4+, Catspaw's professional SNOBOL4 package, provides language extensions that allow just that:

```
N = (EQ(REMDR(N,2),0) 'EVEN', 'ODD')
```

- Forgetting that functions like TAB and BREAK bind subject characters. This won't matter for simple pattern matching, but for matching with replacement, problems can appear. For example, suppose we wanted to replace the 50th character in string S with '*'. If we used:

```
S TAB(49) LEN(1) = '*'
```

  we would find the first 50 characters replaced by a single asterisk. Instead, we should say:

```
S POS(49) LEN(1) = '*'
```

  or, even more efficiently:

```
S TAB(49) . FRONT LEN(1) = FRONT '*'
```

- Omitting the unevaluated expression operator when defining a pattern containing variable arguments. For example, the pattern

```
NTH_CHAR = POS(*N - 1) LEN(1) . CHAR
```

  will copy the Nth subject character to variable CHAR. The pattern adjusts automatically if N's value is subsequently changed. Omitting the asterisk would capture the value of N at the time the pattern is defined (probably the null string).

10.1.3 Simple debugging

These simple methods should find a majority of your bugs:

- Set keyword `&DUMP` nonzero, or use command line option `/D` to get an end-of-run dump. Examine it closely for reasonable values and variable names. Dumps can also be produced at any time during execution by calling the built-in function `DUMP`.
- Use keyword `&STLIMIT` to end execution after a fixed number of statements.
- Use the keyboard Ctrl-C key to interrupt a program which is looping endlessly, and record the statement number.
- Use `:F(ERROR)` to detect unexpected failures and data errors. Do not define the label `ERROR` – SNOBOL4 will display the statement number of the error if an attempt is made to transfer to label `ERROR`.
- Assign values to `OUTPUT` to monitor data values. Use immediate assignment and cursor assignment (to `OUTPUT`) to observe the operation of a pattern match.
- Produce end-of-run statistics with the command line option `/S`. Are the number and kind of operations reasonable?
- Use the `CODE.SNO` program to setup simple test cases. This is particularly useful when pattern-matching statements do not behave as expected.

More subtle errors can be pinpointed using SNOBOL4's trace facility, described below.

10.2 Execution tracing

Tracing the flow of control and data in a program is usually the best way to find difficult problems. SNOBOL4 allows tracing of data in variables and some keywords, transfers of control to specified labels, and function calls and returns. Two keywords control tracing: `&FTRACE` and `&TRACE`.

10.2.1 Function tracing

Keyword `&FTRACE` is set nonzero to produce a trace message each time a program-defined function is called or returns. The trace message displays the statement number where the action occurred, the name of the function, and the values of its arguments. Function returns display the type of return and value, if any. Each trace message decrements `&FTRACE` by one, and tracing ends when `&FTRACE` reaches zero. A typical trace messages looks like this:

```
STATEMENT 39: LEVEL 0 CALL OF SHIFT('SKYBLUE',3),TIME = 140
STATEMENT 12: LEVEL 1 RETURN OF SHIFT = 'BLUESKY',TIME = 141
```

The level number is the overall function call depth. The program execution time in tenths of a second is also provided.

10.2.2 Selective tracing

Keyword `&TRACE` will also produce trace messages when it is set nonzero. However, the `TRACE` function must be called to specify what is to be traced. Tracing can be selectively ended by using the `STOPTR` function. The `TRACE` function call takes the form:

`TRACE(`*name* `,` *type* `,` *string* `,` *function*`)`

The name of the item being traced is specified using a string or the unary name operator. Besides variables, it is also possible to trace a particular element of an array or table:

```
TRACE('VAR1', ...)
TRACE(.A<2,5>, ...)
TRACE('SHIFT', ...)
```

Type is a string describing the kind of trace to be performed:

'VALUE'	Trace whenever name has a value assigned to it. Assignment statements, as well as conditional and immediate assignments within pattern matching will all produce trace messages
'CALL'	Produce a trace whenever function name is called
'RETURN'	Produce a trace whenever function name returns
'FUNCTION'	Combine the previous two types: trace both calls and returns of function name
'LABEL'	Produce a trace when a Goto transfer to statement name occurs. Flowing sequentially into the labeled statement does not produce a trace
'KEYWORD'	Produce a trace when keyword name's value is changed by the system. The name is specified without an ampersand. Only keywords &ERRTYPE, &FNCLEVEL, &STCOUNT, and &STFCOUNT may be traced.

If omitted, a VALUE trace is assumed.

When the first argument is specified with the unary name operator, the third argument, *string*, will be displayed to identify the item being traced:

```
TRACE(.T<"zip">, "VALUE", "Table entry 'zip'")
```

The last argument, *function*, is usually omitted. Its use is described in the next section.

The form of trace message displayed for each type of trace is listed in chapter *System Messages* of the Reference Manual.

Each time a trace is performed, keyword &TRACE is decreased by one. Tracing stops when it reaches zero. Tracing of a particular item can also be stopped by function STOPTR:

```
STOPTR(name , type)
```

10.2.3 Program trace functions

Normally, each trace action displays a descriptive message, such as:

```
STATEMENT 371: SENTENCE = 'Ed ran to town',TIME = 810
```

Instead, we can instruct SNOBOL4 to call our own programdefined function. This allows us to perform whatever trace actions we wish. We define the trace function in the normal way, using DEFINE, and then specify its name as the fourth argument of TRACE. For example, if we want function TRFUN called whenever variable COUNT is altered, we would say:

```
&TRACE = 10000
TRACE('COUNT', 'VALUE', , 'TRFUN')
DEFINE('TRFUN(NAME,ID)')              :(TRFUN_END)
...
```

TRFUN will be called with the name of the item being traced, 'COUNT', as its first argument. If a third argument was provided with TRACE, it too is passed to your trace function, as ID. (Here the argument was omitted.)

To use trace functions effectively, we must pause to describe a few more SNOBOL4 keywords:

`&LASTNO`	The statement number of the previous SNOBOL4 statement executed
`&STCOUNT`	The total number of statements executed. Incremented by one as each statement begins execution
`&ERRTYPE`	Error message number of the last execution error
`&ERRLIMIT`	Number of nonfatal execution errors allowed before SNOBOL4 will terminate.

The first three keywords are continuously updated by SNOBOL4 as a program is executed.

Now, let's consider debugging a program where variable `COUNT` is inexplicably being set to a negative number. Continuing with the previous example, the function body would look like this:

```
        &TRACE = 10000
        TRACE('COUNT', 'VALUE', , 'TRFUN')
        DEFINE('TRFUN(NAME,ID)TEMP')                    :(TRFUN_END)

TRFUN   TEMP = &LASTNO
        GE($NAME, 0)                                    :S(RETURN)
        OUTPUT = 'COUNT negative in statement ' TEMP    :(END)
TRFUN_END
```

The first statement of the function captures the number of the last statement executed – the statement that triggered the trace. We then check `COUNT`, and return if it is satisfactory. If it is negative, we print an error message and stop the program.

When a trace function is invoked, keywords `&TRACE` and `&FTRACE` are temporarily set to zero. Their values are restored when the trace function returns. There is no limit to the number of functions or items which may be traced.

Tracing keyword `&STCOUNT` will call your trace function before every program statement is executed.

Program `CODE.SNO` traces keyword `&ERRTYPE` to trap nonfatal execution errors from your sample statements, and produce an error message. Keyword `&ERRLIMIT` must be set nonzero to prevent SNOBOL4 from terminating when an error occurs.

10.3 Program efficiency

To a greater extent than other languages, SNOBOL4 programs are sensitive to programming methods. Often, there are many different ways to formulate a pattern match, and some will require many more match attempts than others.

As you work with SNOBOL4, you will develop an intuitive feel for the operation of the pattern matcher, and will write more efficient patterns. I can, however, start you off with some general rules:

- Try to use anchored, quickscan, and trim modes when possible. If operating unanchored, artificially anchor whenever possible by using `POS(0)` or `FENCE` as the first subpattern.
- Try to use `BREAK` and `SPAN` instead of `ARB`.
- Use `ANY` instead of an explicit list of one-character strings and the alternation operator.
- `LEN`, `TAB` and `RTAB` are faster than `POS` and `RPOS`. The former step over subject characters in one operation; the latter continually fail until the subject cursor is positioned correctly. But be careful of misusing them with replacement and replacing more than you expected.
- Use conditional assignment instead of immediate assignment in pattern matching.

- Use `IDENT` and `DIFFER` to compare strings for equality, instead of pattern matching. Since each unique string is stored only once in SNOBOL4, these functions merely compare one-word pointers, regardless of string length. By contrast, pattern matching and functions such as `LGT` must perform character by character comparisons.
- Avoid `ARBNO` and recursion if possible.
- Pattern construction is time-consuming. Preconstruct patterns and store them in variables whenever possible.
- Keep strings modest in length. Although SNOBOL4 allows strings to be thousands of characters long, operating upon them is very time-consuming. They use large amounts of memory, and force SNOBOL4 to frequently rearrange storage.
- Use functions to modularize a program and make it easier to understand and maintain.
- Avoid algorithms that make a linear search of an array or list. The algorithms can usually be rewritten using tables and indirect references for associative programming.

Efficiency should not be measured purely in terms of program execution time. With the relatively low cost of microcomputers, the larger picture of time spent designing, coding, and debugging a program also must be considered. A direct approach, emphasizing simplicity, robustness, and ease of understanding usually outweighs the advantages of tricky algorithms and shortcut techniques. (But we admit that tricky pattern matching is fun!)

Chapter 11

Concluding Remarks

For much of this tutorial we've been concerned with the detailed mechanics of pattern matching – the functions, primitive patterns, and heuristics of applying a pattern to a character string. SNOBOL4 provides so many primitive functions and operations that it's easy to get lost in the forest. Let's step back and consider SNOBOL4's larger significance.

It would be a mistake to think of SNOBOL4 only as a text processing language. For example, programmers in the artificial intelligence field think in terms of lists, and have used the LISP language for some time. As Shafto demonstrates, SNOBOL4 can be made to emulate LISP, and go well beyond it, using pattern matching, backtracking, and associative programming (see file `SNOBOL4.DOC` for information on Shafto's report on AI SNOBOL4 programming.)

SNOBOL4's pattern matching provides a very powerful and completely general recognition system, in which character strings happen to be the medium of expression. Other recognition problems can be solved by mapping the object to be examined into a subject string, and the recognition criteria into SNOBOL4 patterns.

In the past, use of SNOBOL4 has been hindered by the high cost and inconvenience of running it on mainframe computers. Now it's on your desk top, with computer time essentially free.

What new insights can SNOBOL4 bring to your problems? Can you find other general applications for SNOBOL4's unique abilities?

The future of the language is in your hands.

Part III

Vanilla Snobol4 Reference Manual

Introduction

The reference section describes the SNOBOL4 system. It will tell you how to create and run SNOBOL4 programs, and catalogs all the standard language features. The tutorial section can be consulted for illustrative uses of various functions and operators.

SNOBOL4 is a full implementation of the powerful development language SNOBOL4 for the IBM PC and the entire 8086/286/386 family of computers. It has all the features of mainframe SNOBOL4, plus numerous useful extensions. Compatibility with mainframe SNOBOL4 is achieved by basing this product on the Macro Implementation used on such mainframes as the IBM 370 and the CDC 7600. Thus, it incorporates a thoroughly tested implementation in its entirety. All SNOBOL4 string and pattern matching facilities available in the mainframe environment are now available to the personal computer user.

The SNOBOL4 program contains both a compiler and interpreter. They are inseparable, and share many common routines. Your source program is compiled into a compact internal notation, which is interpreted during execution. More information on the internal code may be found in Griswold's *'The Macro Implementation of* SNOBOL4*'*; see file `SNOBOL4.DOC` for ordering information.

Language background

In 1962, several researchers at Bell Telephone Laboratories (BTL) were applying computers to problems such as factoring multivariate polynomials and symbolic integration. Available tools were the Symbolic Communication Language (SCL), an internal BTL product for processing symbolic expressions, and COMIT, designed for natural-language analysis. Both proved inadequate, and frustration with them led the researchers to attempt the design of a new language.

The original SNOBOL was developed by David J. Farber, Ralph E. Griswold, and Ivan P. Polonsky, and was first implemented on an IBM 7090 computer in 1963. The name, SNOBOL, came after the implementation, and ostensibly stands for 'StriNg Oriented symBOlic Language'.

It was soon discovered that SNOBOL was applicable to a much wider range of problems. In fact, the language proved more interesting than the problems it was intended to solve. As more people used it, new features such as recursive functions were added, and its generality grew. By 1964, it had become SNOBOL3, and was available on such machines as the IBM 7094, CDC 3600, SDS 930, Burroughs 5500, and the RCA 601. Because these implementations were all written from scratch, each machine introduced its own dialect of the language.

SNOBOL3 had only one data type, the string. The desire for additional data types, more complex pattern matching, and other features led to a major redesign of the language in 1966, by Ralph Griswold, Jim Poage, and Ivan Polonsky. The new language – SNOBOL4 – was also designed to be portable to other machines. Most of SNOBOL4 was completed by 1967, although some features, such as operator redefinition, did not appear until 1969. Portability was achieved by writing the system in a macro assembly language for an abstract machine, hence the name 'Macro Implementation of SNOBOL4.' By 1970 it was available on nine different types of mainframes. Currently, it is available on most large- and medium-scale computers.

The SNOBOL4 language evolved on computers whose primary input/output devices were the card reader, card punch, and line printer. The current breed of microcomputers are interactive, rather than batch-oriented. Thus, SNOBOL4 contains slight alterations of the language to conform to the personal computer environment. For example, the preassigned output keyword

`PUNCH` has been replaced by `SCREEN`. Experienced SNOBOL4 programmers will find little incompatibility with familiar implementations. Most existing SNOBOL4 programs should operate correctly using SNOBOL4 with little or no change.

Chapter 12

Running a Snobol4 Program

12.1 Basic command line format

The format for the command line is:

```
SNOBOL4  file  options  ;comments
```

Options are specified by a slash (/) or minus sign (-), and one or more option letters. When the option requires a file name, an equal sign may be used between the option letter and file name for readability.

`File`
: The source file contains your SNOBOL4 program. If no file is specified, `CON:` is assumed, and programs may be entered directly from the keyboard. Disk files will have extension `.SNO` supplied if none is specified.

The source and input files may be assigned to any disk file or valid input device. The listing, output, and error message files may be assigned to any disk file or valid output device. If the output disk file does not exist, it will be created.

`/I=`*file*
: The input file is associated with the variable `INPUT` when execution begins, as I/O unit 5. The default is `CON:`, your keyboard. Disk files will have extension `.IN` supplied if none is specified.

`/L=`*file*
: The listing file receives a listing of your program, with assigned statement numbers. Default is `NUL:`, that is, the listing is discarded. If `/L` appears without a file name, the source program file name will be used, with the extension changed to `.LST`.

`/O=`*file*
: The output file is associated with the variable `OUTPUT` when execution begins. This will be I/O unit 6. The default is `CON:`, which is usually your computer's display screen. Disk files will have extension `.OUT` supplied if none is specified. Execution dumps and tracings are sent to I/O unit 6.

`/E=`*file*
: A list of compilation and runtime error messages is written to this file. Default is `CON:`, that is, error messages are displayed on the screen. If `/E` appears without a file name, the source program file name will be used, with the extension changed to `.ERR`.

In addition to the `/I` and `/O` options, the `INPUT` and `OUTPUT` variables may also be assigned to files by using the MS-DOS redirection operators `<` and `>` on the command line.

Other I/O files may be specified explicitly within the `INPUT` and `OUTPUT` functions, or on the command line with a unit number:

/n = file
: The specified file becomes associated with unit number n, which must be in integer between 1 and 16. If your program calls the `INPUT` or `OUTPUT` function without a file name, the file specified here will be used. This command line option merely makes an association; the file is not opened or created until the `INPUT` or `OUTPUT` function is called.

File names may be a disk file, or any DOS device, such as `NUL:`, `CON:`, `LPT2:`, etc.
The remaining option switches alter SNOBOL4's behavior:

`/B` Termination messages and statistics are normally displayed via I/O unit 7 (`SCREEN`). The `/B` (batch) option instead directs them to I/O unit 6 (`OUTPUT`).

`/C` SNOBOL4 defaults to case-folding, making lower and upper case alphabetics equivalent for names and labels. Specifying this option inhibits case-folding: upper and lower case names are unique and distinct.

`/D` Sets the `&DUMP` keyword to 1. This is useful when you decide you want an end-of-run variable dump, and don't want to edit the source file.

`/H` Displays summary of options and Vanilla SNOBOL4 license information.

`/NX` No execution after compilation.

`/NP` Suppress column position information in error messages.

`/P` Displays additional product information.

`/S` Provide statistics upon termination.

Vanilla SNOBOL4 works very nicely with text editors that allow a program to be compiled from within the editor. If a compilation or runtime error occurs, you are returned to your editor with the cursor positioned on the troublesome statement. To use with your editor, you will need to use the command line option `/BE-`. This writes errors messages to standard output, where they can be captured by your text editor.

12.2 Providing your own parameters

The keyword `&PARM` contains the command line string. It begins with the blank following the word SNOBOL4, and contains all characters up to the terminating carriage return. Since SNOBOL4's command processor ignores all characters after a semicolon, comments placed there can easily communicate additional instructions to your program. Break them out with the statement:

```
&PARM ';' REM . INSTRUCTIONS
```

12.3 Command line examples

The command line:

```
SNOBOL4 PROG
```

will compile and run a source program from file `PROG.SNO`, discard the listing, and run it with keyboard input and screen output. The command line:

```
SNOBOL4 CONVERT /I=DATA /O=RESULT /2=STYLE.DAT ;DRAFT
```

will run a program that presumably transforms input file `DATA.IN` to output file `RESULT.OUT` according to program option `'DRAFT'`. I/O unit number 2 is associated with the file `STYLE.DAT`. The program can use the variable `SCREEN` to post error and status messages to the user, regardless of the reassignment of the input and output files.

```
SNOBOL4 SOURCE /I=SOURCE.SNO /L=OUTPUT /O=OUTPUT.LST /BCS
```

sets up a conventional batch job, with source program and input data on file `SOURCE.SNO` (following the `END` statement), listing and program output to `OUTPUT.LST`, no case-folding, and end-of-run statistics.

Chapter 13

Statements

Each line of input to SNOBOL4 consists of a sequence of ASCII characters, terminated by a carriage return.

Comment and control statements are always one line long. However, a program statement may occupy several lines if necessary. A continuation mark (`+` or `.`) is placed in the first column of the additional lines.

13.1 Comment statements

An asterisk (`*`) in character position one denotes a comment card. All text through the end-of-line is copied to the listing file, but is otherwise ignored by SNOBOL4.

13.2 Control statements

Control statements provide instructions to the SNOBOL4 compiler. They begin with a minus (`-`) in character position one. Controls may be specified in upper- or lower-case, regardless of the current state of case-folding. Unrecognized controls are ignored.

`-CASE` n
: Fold lower-case names to upper-case if n is nonzero. Treat upper- and lower-case names as distinct if n is zero or absent.

`-EJECT`
: Start a new page on the listing file.

`-LIST`
: Equivalent to `-LIST LEFT`.

`-LIST`
: Left turn on list output, produce statement numbers at left end of line.

`-LIST`
: Right turn on list output, produce statement numbers at right end of line.

`-UNLIST`
: Turn off list output. Errors are not shown on the screen.

SNOBOL4 defaults to `-LIST LEFT` and `-CASE 1`.

13.3 Program statements

If a line is not a control or comment statement, it is considered SNOBOL4 program text. A SNOBOL4 statement may have up to five components. The general form of a statement is:

Label Subject Pattern = Replacement :Goto

Statement elements are separated by blank or tab.

Ignoring the *Label* and *Goto* fields for a moment, the remaining elements may appear in various combinations to create different types of statements:

Subject **evaluate expression**

The expression comprising the subject is evaluated. It may invoke primitive and program-defined functions.

Subject = Replacement **assignment statement**

The value on the right is assigned to the variable on the left. If failure occurs when evaluating the subject or replacement components, the assignment does not occur.

Subject Pattern **pattern match**

The subject and pattern expressions are evaluated, and the specified pattern is applied to the subject string, producing success or failure.

Subject Pattern = Replacement **pattern match with replacement**

If the pattern match succeeds, the replacement expression is evaluated and replaces the portion of the subject matched. Only the matched portion is replaced; characters adjacent to the matching substring are not disturbed.

If the equal sign (=) is present but the replacement field is absent, the null string is assumed as the value of the replacement field.

The Goto field provides two-way branching to test the success or failure of the preceding statement elements.

13.3.1 Label field

If a label is present, it must begin with the first character of the line. Labels provide a name for the statement, and serve as the target for transfer of control from the Goto field of any statement. Labels must begin with a letter or digit, optionally followed by an arbitrary string of characters. The label field is terminated by the character blank, tab, or semicolon. If the first character of a line is blank or tab, the label field is absent.

If case-folding is in effect, lower-case letters are converted to upper-case before defining the label.

13.3.2 Subject field

The subject field specifies the string which will be the subject of pattern matching. It also specifies the left side of a simple assignment statement if pattern matching is absent.

In an assignment statement, the subject must be a variable name, an unprotected keyword, or a field-reference function from a program-defined data type. If a string is produced by evaluating an expression, the indirect ($) operator must be used to reference the underlying variable.

If the subject appears in pattern matching without replacement, the subject must evaluate to a string. The string is scanned left to right during the pattern match. If the subject evaluates to an integer, it is automatically converted to a string. If replacement is present, the same subject restrictions of assignment statements apply. Thus, a literal string is a valid subject only if replacement is absent.

If the expression comprising the subject contains the concatenation operator, the subject must be surrounded by parenthesis. This allows SNOBOL4 to distinguish concatenation blanks within the subject from the blank between subject and pattern.

13.3.3 Pattern field

The pattern may be a simple string, or a complex expression involving primitive pattern functions. The pattern specifies one or more strings which are systematically searched for in the subject. The pattern match succeeds if a match is found, and fails otherwise. The `&FULLSCAN` keyword determines whether the search is exhaustive, or if heuristics will be applied to prevent futile match attempts.

The pattern may assign various matching components to variables with the binary assignment operators dot (`.`) and dollar sign (`$`).

13.3.4 Replacement field

In an assignment statement, there are very few restrictions on the replacement field. If the subject is an unprotected keyword, the replacement field must evaluate to an integer value. If the subject is a variable, the replacement field is assigned directly to it, without type conversion.

If there is pattern matching on the left side of the statement, the replacement field must evaluate to a string, so that it may be inserted into the matched portion of the subject string.

Replacement occurs only if evaluation of the subject, pattern, and replacement succeed. Primitive functions which return success or failure may be used in the replacement field as predicate functions. Since they return the null string, they do not alter the replacement value. However, their failure can prevent replacement from occurring, and can be tested in the Goto field.

13.3.5 Goto field

Statement execution normally proceeds sequentially from one statement to the next. The Goto field allows this flow to be altered by directing the SNOBOL4 system to continue execution elsewhere. The Goto field is set off from the preceding statement elements by blank or tab, and colon (`:`). It may assume three forms: unconditional, conditional, and direct.

The unconditional Goto causes control to be transferred to the specified labeled statement. The label is enclosed in parenthesis, and may be a name, or the result of evaluating an expression and applying the indirect operator (`$`). Transfer is made to the labeled statement regardless of the success or failure outcome of the earlier parts of the statement.

The conditional Goto similarly specifies control transfer to a labeled statement, but it depends on the success or failure of the statement. The letter `S` precedes the parenthesized label where control goes next if the statement succeeds. The letter `F` specifies the branch to be taken if the statement fails. For example:

`:S(LOOP)`	branches to label `LOOP` if the statement succeeds
`:F(ERROR)`	branches to label `ERROR` if the statement fails
`:S(OK) F(NOGO)`	branches to label `OK` on success, to `NOGO` on failure
`:(AGAIN)`	unconditionally transfers control to label `AGAIN`
`:($('VAR' N))`	branches to the label obtained by concatenating the string `'VAR'` with the value of variable `N`

The direct Goto is used to branch to a block of code compiled with the `CODE` function. If the code contains labels, a regular Goto could branch to the label and begin execution in the code block. The direct Goto will branch to the start of the code block, labeled or not. A direct Goto is specified by placing in angle brackets the name of the variable which points to the code block.

Direct Gotos may be made conditional by preceding them with `S` or `F`. They may also appear with regular Gotos:

```
VAR = CODE(string)          :S<VAR> F(COMPILE_ERROR)
```

The lower-case letters `s` and `f` may be used interchangeably with `S` and `F`, regardless of case-folding.

The Goto field may appear on a line without any subject, pattern, and replacement. The absent SNOBOL4 statement is assumed to have succeeded.

13.4 Continuation statements

A SNOBOL4 statement may be divided across several lines by placing a plus (+) or period (.) in character position one of the successive lines. There is no limit to the number of continuation statements allowed. The statement must be divided at a point where a blank or tab could appear as an operator or separator; it cannot be split in the middle of a name or quoted string.

Very long strings may be entered on multiple lines, using the implicit blank between lines as a concatenation operator:

```
      LONG_STRING = "This is an example of a very long "
+     "string that wends its way across multiple continua"
+     "tion statements.  There is an implicit blank at the "
+     "beginning of each line that provides the concatenation"
+     " operator between segments."
```

13.5 Multiple statements

The semicolon character may be used to place several statements on one line. Each semicolon terminates the current statement and behaves like a new column one for the statement which follows. Only program statements are permitted after the semicolon; control and continuation statements are not allowed. Here are some examples:

```
I = 1;      J = 2;       S PAT = 'HENRI'        :S(YES)
I = 1;OUT   OUTPUT = A<I>   :F(END);   I = I + 1 :(OUT)
```

Because of its poor readability, placing labels in the middle of a statement is strongly discouraged.

As a language extension, Vanilla SNOBOL4 permits a comment statement after the semicolon. This provides a simple device for end-of-line comments:

```
PARA    NEXT = GETNEXT() :F(FRETURN) ;* return if EOF
        IDENT(NEXT)      :S(RETURN)  ;* return on empty line
        PARA = PARA NEXT :(PARA)     ;* splice line
```

13.6 The END statement

The last statement in a program must be an END statement. The word END appears in the label field, beginning in column one. Normally, it is the only word on the line:

```
        ...
        OUTPUT = 'All done'
END
```

After reading the END statement, compilation ends, and execution begins immediately with the very first program statement. When the program is done, it should flow into the END statement, or use a Goto to transfer to it.

Occasionally, we would like to begin execution at other than the first statement. If we place a statement label in the subject field of the END statement, execution will begin there. For example, this statement will cause execution to begin at the statement labeled START:

```
END     START
```

Chapter 14

Operators

Following are lists of all the unary and binary operators in SNOBOL4. Unused operators may be attached to program-defined functions using the `OPSYN` function. Unary operators have equal precedence among themselves, and higher precedence than binary operators. Operators of higher precedence are performed first, unless reordered by parentheses. Where several instances of operators with the same priority appear, associativity specifies which one is performed first.

14.1 Unary operators

All unary operators are left-associative: if several appear together, they are performed left-to-right.

Graphic	Name	Definition
`+`	plus	arithmetic positive
`-`	minus	arithmetic negative
`.`	period	name of object (address)
`$`	dollar sign	indirect reference through object
`*`	asterisk	unevaluated expression
`&`	ampersand	keyword
`~`	tilde	negation of success/failure
`?`	question mark	interrogation
`@`	at sign	cursor position assignment
`/`	slash	`<none>`
`^ !`	caret, exclamation	`<none>`
`%`	percent	`<none>`
`#`	pound sign	`<none>`
`|`	vertical bar	`<none>`

14.1.1 Indirect reference and case-folding

The indirect reference operator (`$`) converts a string to a variable name. When case-folding is in effect, the string characters are treated as upper-case letters when producing the name. The string itself is not modified. Thus,

```
$('abc')
```

references variable `ABC` when case-folding, and variable `abc` when not.

14.2 Binary operators

Graphic	Name	Definition	Precedence	Associates
~	tilde	<none>	12	right
?	question mark	<none>	12	left
.	period	conditional assignment	11	left
$	dollar sign	immediate assignment	13	left
^ !	caret, exclamation	exponentiation	12	right
**	double asterisk	exponentiation	12	right
%	percent	<none>	11	left
*	asterisk	multiplication	10	left
/	slash	division	9	left
#	pound sign	<none>	8	left
+	plus	addition	7	left
-	minus	subtraction	7	left
@	at sign	<none>	6	left
blank	blank	concatenation	5	left
tab	tab	concatenation	5	left
|	vertical bar	alternation	4	left
&	ampersand	<none>	3	left
=	equal sign	assignment	1	right

Chapter 15

Keywords

Keywords allow a program to communicate with SNOBOL4. Their names are set apart from other variables by the unary operator ampersand (`&`). Protected keywords cannot be changed by a program, while unprotected keywords can.

Several protected keywords can be traced using the `TRACE` function: `&ERRTYPE`, `&FNCLEVEL`, `&STCOUNT`, and `&STFCOUNT`. Tracing occurs each time SNOBOL4 alters their value. For example, tracing keyword `&STCOUNT` produces a trace after every SNOBOL4 statement is executed.

15.1 Protected keywords

Among these keywords are several which serve as read-only repositories of fundamental system patterns and values, such as `&ARB`. The nonkeyword form (`ARB`) may be changed by a program, and later restored to its original value by assigning it the corresponding keyword.

`&ABORT`
: The primitive pattern `ABORT`.

`&ALPHABET`
: String of 256 ASCII character values in ascending order.

`&ARB`
: The primitive pattern `ARB`.

`&BAL`
: The primitive pattern `BAL`.

`&ERRTEXT`
: String containing most recent system generated error text.

`&ERRTYPE`
: Integer code of the last execution error to occur. This keyword may be traced with function `TRACE()`.

`&FAIL`
: The primitive pattern `FAIL`.

`&FENCE`
: The primitive pattern `FENCE`.

`&FNCLEVEL`
: Integer depth of program-defined function calls. It is initially zero, and incremented by one for each function call, and decremented for each function return. This keyword may be traced.

`&LASTNO`
: Integer statement number of the previous statement executed.

`&LCASE`
: The 26 lower-case alphabetic letters.

`&PARM`
: The command string used to invoke SNOBOL4. Begins with the blank following the word SNOBOL4.

`&REM`
: The primitive pattern `REM`.

`&RTNTYPE`
: Contains a string describing the type of return most recently made by a program-defined function, either `'RETURN'`, `'FRETURN'`, or `'NRETURN'`.

`&STCOUNT`
: Integer count of the number of statements executed. This keyword may be traced. Since integers are 16-bit quantities, executing more than 32 767 statements will cause this keyword to overflow. No harm results, and the keyword may still be traced, but its value will be a large negative number.

`&STFCOUNT`
: Integer count of the number of statements which failed. This keyword may be traced. The same overflow problem discussed for `&STCOUNT` occurs with this keyword.

`&STNO`
: Integer statement number of the current statement being executed.

`&SUCCEED`
: The primitive pattern `SUCCEED`.

`&UCASE`
: The 26 upper-case alphabetic letters.

15.2 Unprotected keywords

These keywords may be set to integer values to modify SNOBOL4's behavior.

`&ANCHOR`
: Nonzero for anchored pattern match. Initially 0, unanchored.

`&CASE`
: Zero to prevent case-folding during compilation with the functions `CODE` and `EVAL`. Initially 1, causing case-folding to occur.

`&CODE`
: The end-of-job code is an integer value in the range 0 to 255 returned to the operating system. It can be tested with the DOS Batch condition `ERRORLEVEL`. Initially 0.

`&DUMP`
: Nonzero to list unprotected keywords and variables with nonnull values at program termination. A positive value causes the list to be sorted; negative values leave them unsorted. Initially 0. The dump is produced to I/O unit 6 (`OUTPUT`).

`&ERRLIMIT`
: Determines the number of conditionally fatal execution errors permitted before terminating a program. The *Execution error messages* section of chapter *System Messages* describes the errors which are conditionally fatal. Initially 0, causing SNOBOL4 to stop if any error occurs.

`&FTRACE`
: Nonzero value causes each call and return of a program-defined function to be listed. Decremented for each trace. Initially 0.

`&FULLSCAN`
: Nonzero to disable pattern matching heuristics. Initially 0, the quickscan mode of pattern matching.

`&INPUT`
: Zero to disable all input. When disabled, using variable `INPUT` (or other input-associated variables) does not read data from the file. Initially 1, input is enabled.

`&MAXLNGTH`
: Maximum string length. Initially 5 000, maximum value is 32 767. Memory limitations in Vanilla SNOBOL4 will limit actual strings to a smaller size.

`&OUTPUT`
: Zero to disable all output. When disabled, assigning data to `OUTPUT` or `SCREEN` (or other output-associated variables) does not write data to the file. Initially 1, output is enabled.

`&STLIMIT`
: The number of statements allowed to execute. If positive, it is decremented for each statement executed; execution terminates when it reaches 0. If negative, there is no limit, and it is not decremented. Initially -1.

`&TRACE`
: Nonzero to permit tracing with the `TRACE` function. Initially 0, it is decremented for each trace performed.

`&TRIM`
: Nonzero to strip trailing blanks from lines read from ASCII files. This is faster than using the `TRIM` function. It does not strip trailing tab characters. Initially 0: blanks are not removed and short records are blank padded to the file's standard record length.

15.3 Special names

The following names have special meaning to SNOBOL4. If casefolding is in effect, they may appear with any combination of upper- and lower-case letters.

`END`
: This is a special label which denotes the last statement of the user's program. An optional label may follow the word `END` (in the subject field) to denote where program execution is to begin. A program should terminate execution by transferring to label `END`.

`FRETURN`
: Transfer to this label to return from a program-defined function with a failure indication.

`INPUT`
: Variable associated with input from unit number 5.

`NRETURN`
: Transfer to this label to return successfully from a program-defined function by name, rather than by value. The function name should be assigned a name result (usually with the period (`.`) unary operator). This permits a function call to be the object of an assignment operation.

`OUTPUT`
: Variable associated with output to unit number 6.

`RETURN`
: Transfer to this label to return from a program-defined function with a success indication. A value may be returned as the function's result; simply assign it to a variable with the same name as the function before transferring to `RETURN`.

Chapter 16

Data Types and Conversion

Most other programming languages require the user to explicitly declare the type of data to be stored in a variable. In SNOBOL4, any variable may contain any data type. Furthermore, the variable's type may be freely altered during program execution. SNOBOL4 remembers what kind of data is in each variable.

16.1 Data type names

The formal name of a data type is specified by an upper-case string (or lower-case if case-folding is in effect), such as `'INTEGER'`, or `'ARRAY'`. It is used with the `CONVERT` function to specify the data type conversion desired. The formal name is also the string returned when the `DATATYPE()` function is used to determine an object's type.

`ARRAY` ***N*-dimensional array**

The primitive function `ARRAY()` creates an array storage area, and returns a pointer with this data type. If this pointer is stored in a variable, the variable is said to be of type `ARRAY`, and may then be subscripted to access the elements of the array.

`CODE` **compiled Snobol4 code**

The primitive function `CODE()` compiles a string containing SNOBOL4 statements, and returns a pointer to the resulting object code block. If this pointer is stored in a variable, the variable is said to be of type `CODE`. The variable may then be used with a direct Goto by enclosing it in angle brackets.

`EXPRESSION` **unevaluated expression**

When the unevaluated expression operator (`*`) is applied to an expression, the result has the data type `EXPRESSION`. Such expressions are not evaluated when they are defined, only when they are referenced.

```
E = *(LEN(K) POS(M))
```

defines `E` as an unevaluated expression. When this statement is executed, the code to concatenate two function calls is compiled, but not executed. It is only when `E` is referenced in a subsequent pattern match or appears as the argument of the `EVAL` function that the code is executed to produce a pattern.

The unevaluated expression operator must be at the outermost level to create an object of type `EXPRESSION`. If buried with the expression, the execution results may appear to be similar, but the object's data type is different. That is, the two statements

```
P = *LEN(N)
P = LEN(*N)
```

produce identical results when `P` is used in a pattern match (if `LEN` is not redefined). However, the first statement produces `P` as type `EXPRESSION`, while the second produces P as type `PATTERN`. Expressions may also be produced explicitly with the `CONVERT()` function (see below).

`EXTERNAL` **created by external function**

External assembly language functions may create new data types whose structure is known only to them. This feature is only available in SNOBOL4+, Catspaw's enhanced implementation of the SNOBOL4 language.

`INTEGER` **integer number**

A decimal number in the range −32 767 to +32 767. No fractional part may appear. One computer word (16 bits) is used to contain an integer value.

`NAME` **name of a variable**

When the unary name operator (`.`) is applied to a variable, two results are possible. If the variable's name is a simple string (a 'natural variable'), such as `ABC`, the variable's name is returned as type `STRING`. For example, `.ABC` has the value `'ABC'`. However, if the variable is a created variable, such as a table or array element, the `NAME` data type results. In either case, the result of the name operator can be thought of as the address or storage location of the variable. When the indirect reference operator (`$`) is applied to such a result, the original, underlying object is obtained. That is, `$(.A)` is the same as using the variable `A`.

For natural variables, SNOBOL4 has the surprising property that the string `'XYZ'` is the address (or name) of variable `XYZ`, so `$'XYZ'` is equivalent to `XYZ`.

`PATTERN` **pattern match structure**

A pattern is created by an expression containing any of the following: other patterns, primitive patterns, pattern functions, the alternation operator (`|`), the conditional or immediate assignment operator (`.` or `$`), or the cursor position operator (`@`). A simple string is not a pattern data type, even though it may appear in the pattern portion of a statement. The following are examples of the pattern data type:

```
POS(0) "A" LEN(1)
"COLUMN A" | "COLUMN B"
"ZIP" . X
"MATCH" @Y
```

Program-defined data type **created by `DATA()` function**

Up to 899 new data types may be created with the primitive function `DATA`. The name specified in the prototype string becomes a new data type in SNOBOL4. Any object created with the data type's creation function is given this name as its data type.

```
DATA('COMPLEX(REAL,IMAG)')      ;* define new type COMPLEX
NUM = COMPLEX(2,-4)             ;* create a COMPLEX object
OUTPUT = DATATYPE(NUM)          ;* print string 'COMPLEX'
```

`REAL` **real number**

A floating-point decimal number in the range 2.3E−308 to 1.7E+308. Reals are only available in SNOBOL4+, Catspaw's enhanced implementation of the SNOBOL4 language.

`STRING` **character string**

A sequence of characters. Each character occupies one memory byte, and may contain any of the 256 possible bit combinations. A string of length zero is called the null string. Maximum length of a string is determined by the keyword `&MAXLNGTH` (default 5 000). Memory restrictions in Vanilla SNOBOL4 will limit the longest string possible to less than the 32 767 characters allowed in SNOBOL4+, Catspaw's enhanced SNOBOL4 implementation.

`TABLE` **associatively referenced table**

The primitive function `TABLE()` creates a table storage area, and returns a pointer with this data type. If this pointer is stored in a variable, the variable is said to be of type `TABLE`. The variable may then be subscripted to access the elements of the table. A table may be thought of as a one dimensional array in which the array subscripts may be any SNOBOL4 data type. Arrays require integer subscripts, but table subscripts such as `T<"TALLY">` or `T<13.52>` are acceptable.

16.2 Data type conversion

Data may be implicitly or explicitly converted from one type to another.

16.2.1 Implicit conversion

Implicit conversion occurs automatically when SNOBOL4 requires a certain data type, and your program provides it in another form. Conversion to the correct data type will be attempted, and an error message given if conversion is not possible.

16.2.2 Explicit conversion

A program may use the `CONVERT()` function to explicitly convert an object to another data type. The first argument is the object to be converted; the second is a string containing the formal name of the desired data type. The formal name must be in uppercase (lower-case allowed if case-folding). If conversion is possible, the function succeeds and returns the converted object. If not, the function fails. The call looks like this:

```
NEWTYPE = CONVERT(OBJECT, "DESIRED TYPE")
```

16.2.3 Permissible conversions

`ARRAY` → `STRING`
: The formal name `"ARRAY"` is produced. The defining array dimension string is appended if less than 20 characters:

```
A = ARRAY('1:50,6')
OUTPUT = A
```

produces the string `"ARRAY('1:50,6')"`.

`CODE` → `STRING`
: The formal name `"CODE"` is produced:

```
C = CODE(' PIT2 = .OPPIT4 :(RETURN)')
OUTPUT = C
```

displays the string `"CODE"`.

`EXPRESSION` → `PATTERN`
: This occurs implicitly within a pattern match, or by using the `EVAL` function. The deferred expression is evaluated, using current values for any variables which appear. Example:

```
LASTN = *(RTAB(N) REM . LCHARS)
...
N = 4
SUBJECT LASTN                          :F(TOO_SHORT)
```

`EXPRESSION` → `STRING`
: The formal name `"EXPRESSION"` is produced. For example,

```
LASTN = *(RTAB(N) REM . LCHARS)
OUTPUT = LASTN
```

produces the string `"EXPRESSION"`.

`INTEGER` → `PATTERN`
: This only occurs implicitly within a pattern match. The integer is converted to a string, and the string converted to a pattern. Example:

```
SUBJECT 19 = ''
```

INTEGER → STRING

Leading zeros are suppressed, and a minus sign appears if the integer was negative. Integer zero is converted to the string "0". For example,

```
A = -23;  B = 0;  C = 92
OUTPUT = A B C
```

produces the string "-23092".

NAME → STRING

The formal name "NAME" is produced:

```
N = .A[2]
OUTPUT = N
```

displays the string "NAME".

PATTERN → STRING

The formal name "PATTERN" is produced. For example,

```
WPAT = BREAK(LETTERS) SPAN(LETTERS) . WORD
OUTPUT = WPAT
```

produces the string "PATTERN".

Defined data type → STRING

The formal name from the defining DATA function call is returned.

```
DATA('COMPLEX(REAL,IMAG)')
R1 = COMPLEX(2, 3)
OUTPUT = R1
```

produces the string "COMPLEX".

STRING → INTEGER

The string must not have any leading or trailing blanks. A leading plus or minus sign is allowed, but must be followed by at least one digit. Leading zeros are allowed, and the resulting value must be in the legal range for integer values. A null string is converted to integer zero.

```
RESULT = ("-14" + "") / "2"
```

stores integer -7 in RESULT.

STRING → PATTERN

This only occurs implicitly within a pattern match. The pattern created will match the specified substring:

```
SUBJECT "HOPE"
```

TABLE → ARRAY

This only occurs when using the CONVERT function. The table is converted to a two dimensional array. Example:

```
T = TABLE(100)
...
A = CONVERT(T, "ARRAY")          :F(EMPTY)
```

The table is converted to a rectangular array. Null table entries are omitted, and there must be at least one nonnull entry or the function fails. An $N \times 2$ array is created, where N is the number of nonnull table values. The first array column contains the table subscripts, the second column contains the entry values.

TABLE → STRING

The formal name "TABLE" is returned with the present size of the table and its expansion increment. For example,

```
        T = TABLE(10,10)
        ...
* insert 45 nonnull elements into T
        ...
        OUTPUT = T
```

produces the string `"TABLE(50,10)"` (because table segments in this case are allocated in multiples of 10).

The following matrix indicates conversions with `CONVERT()`. Rows represent argument types, result types are along columns:

	STRING	INTEGER	PATTERN	NAME	ARRAY	TABLE	CODE	EXPRESSION	DEFINED
STRING	*	I	P				C	E	
INTEGER	S	*	P						
PATTERN	F		*						
NAME	F			*					
ARRAY	A				*	1			
TABLE	T				2	*			
CODE	F						*		
EXPRESSION	F		P					*	
DEFINED	F								*

`*` The argument object is returned unchanged.

`A` The formal data type name `"ARRAY"` is returned with the defining prototype string if it is less than 20 characters.

`C` `CONVERT(string,"CODE")` behaves exactly like `CODE(string)`.

`E` Produces an unevaluated expression, that may be subsequently used in a pattern, or evaluated with the `EVAL()` function.

`F` The formal data type name is returned.

`I` Numeric conversion is conditioned on magnitude and syntax restrictions. No leading or trailing blanks are permitted.

`P` Occurs implicitly within a pattern match.

`S` A number may always be converted to its string form.

`T` The string `"TABLE"` is returned with the present size of the table and its expansion increment: `"TABLE(50,10)"`.

`1` The array must be rectangular, with a second dimension of 2 (N rows by 2 columns). A table with N entries is created. The table subscripts are taken from the first column of the array; the table values are copied from the second column.

`2` The table is converted to a rectangular array. Null table entries are omitted, and there must be at least one nonnull entry or the function fails. An $N \times 2$ array is created, where N is the number of nonnull table values. The first array column contains the table subscripts, the second column contains the entry values.

Chapter 17

Patterns and Pattern Functions

The SNOBOL4 pattern matcher is called *the scanner*. The *cursor* is the scanner's pointer into the subject string; it points between subject characters (no relation to your CRT cursor). It is initially zero when positioned to the left of the subject, and is incremented as the scanner moves to the right in the subject.

17.1 Primitive patterns

These variables initially contain the primitive patterns of the same name. They may be set to other values by a program, and restored to their original value from the corresponding protected keywords.

`ABORT`
: Causes immediate failure of the entire pattern match, without seeking alternatives.

`ARB`
: Matches zero or more characters of the subject string. It matches the shortest possible substring.

`BAL`
: Matches any nonnull string which is balanced with respect to parentheses. A string without parentheses is considered balanced. `BAL` matches the shortest string possible.

`FAIL`
: Causes failure of this portion of the pattern match, causing the scanner to backtrack and try alternatives.

`FENCE`
: Matches the null string and succeeds when the scanner is moving left to right in a pattern, but fails if the scanner has to back up through it, seeking alternatives.

`REM`
: Matches zero or more characters from the current cursor position to the end of the subject string.

`SUCCEED`
: Matches the null string and always succeeds.

Altering these primitive patterns can produce very confusing programs, unless the new value encompasses the old, like this:

```
ARB = &ARB . OUTPUT
```

17.2 Primitive pattern functions

These functions produce a pattern based on the argument supplied. The argument data type is shown below – other data types or expressions will be converted to the required type if possible.

Pattern functions may be combined with other primitive patterns, functions, and strings using the alternation and concatenation operators to produce larger patterns.

`ANY(`*string*`)` **match one character from set**
Matches exactly one character from the set of characters specified by the argument string.

`ARBNO(`*pattern*`)` **match repeated pattern**
Matches zero or more consecutive occurrences of the string matched by the argument pattern. `ARBNO` matches the shortest string possible – initially the null string – and only tries to match pattern if other pattern components in the statement require it.

`BREAK(`*string*`)` **match characters not in set**
Matches zero or more characters provided they are not in the set of characters in the argument string. That is, it matches up to, but not including, a character from the argument string.

`LEN(`*integer*`)` **match fixed length string**
Matches a string of the specified length. There are no restrictions on the subject string characters. An argument of zero will match the null string.

`NOTANY(`*string*`)` **match one character not in set**
Matches exactly one character provided it is not in the set of characters specified by the argument string.

`POS(`*integer*`)` **verify scanner position**
Succeeds if the scanner's current cursor position in the subject string is equal to the specified integer value. This function merely verifies scanner position – it does not consume or match any subject characters. `POS(0)` as the first component of a pattern produces an anchored pattern match.

`RPOS(`*integer*`)` **verify scanner position from end**
Succeeds if the scanner's current cursor position in the subject string is the specified number of characters from the end of the string. Like `POS()`, it verifies scanner position but does not consume any characters. `RPOS(0)` as the last component of a pattern forces the pattern to match to the end of the subject string.

`RTAB(`*integer*`)` **match through position counting from end**
Matches all characters from the current cursor position up to the specified cursor position, counting from the end of the subject string. `RTAB(N)` matches characters up to, but not including, the final `N` characters of the subject.

`SPAN(`*string*`)` **match characters in set**
Matches one or more characters from the set of characters specified by the argument string. `SPAN` will not match the null string; at least one character from the argument string must be found in the subject.

`TAB(`*integer*`)` **match through fixed position**
Matches all characters from the current cursor position up to the specified cursor position. `TAB(N)` matches characters up to, and including, the initial `N` characters of the subject. `TAB` will match the null string if the target position and current cursor position are the same. The function fails if the current scanner position is to the right of the target position.

Chapter 18

Built-in Functions

In this chapter, the following items are used to indicate the required argument type. Other types may be used, and will be automatically converted to the required type, if possible. Integer suffixes will be used to distinguish multiple arguments of the same type.

- ***d*** a generic argument of any SNOBOL4 data type
- ***a*** an array
- ***i*** an integer number
- ***n*** the name of a variable, function or label, such as `.VAR` or `'VAR'`; when case-folding, `'VAR'` and `'var'` are equivalent as names
- ***s*** any SNOBOL4 string
- ***t*** a table
- ***u*** I/O unit: an integer between 1 and 16

If an argument is omitted in a function call, SNOBOL4 supplies the null string instead.

`APPLY(`$n, d_1, d_2, \ldots, d_n$`)` **indirect call to a function**

Call function name with the specified arguments. Since n may be a variable containing a function name, it allows an indirect call to a function, similar to the `:($VAR)` construct in the Goto field.

`ARG(`n, i`)` **get dummy argument name from function definition**

Returns a string which is the ith argument from the formal definition of program-defined function n. `ARG` fails if i is greater than the number of arguments in n's definition. `ARG` is useful when one function is used to trace another. The trace function can access the actual argument used with the function being traced with an indirect reference: `$ARG(`n`,`i`)`.

`ARRAY(`s, d`)` **create an array**

S is a prototype which specifies the dimensions of the array created, and the optional d is the value used to initialize all array elements. The form of the prototype string is:

```
"L1:H1,L2:H2,...,Ln:Hn"
```

where `L` and `H` are integers giving the lower and upper bounds of each dimension. Blanks are not permitted. If the lower bound and colon are omitted from any dimension, `'1:'` is assumed. `ARRAY` returns a pointer to the new array, which should be assigned to a variable. The variable can then be subscripted to access the array elements.

A common error when defining a multidimensional array is to use integers instead of a string for the prototype:

```
ARRAY(3,4)
```

instead of

```
ARRAY("3,4")
```

The first example defines a 3-element, one-dimensional array, with elements initialized to integer 4. The second defines a rectangular array, 3 rows by 4 columns.

`CHAR(`i`)` **convert integer to ASCII character**

Converts an integer ASCII code to a one-character string. The argument must be in the range 0 to 255, otherwise the function fails.

`CLEAR()` **clear all variables**

The null string is assigned to all variables in the system (including primitive patterns, such as `ARB`. These patterns and names may be restored from the protected keywords with the same names (e. g., `ARB = &ARB`).

`CLEAR` does not modify variables which are currently saved on the function call stack.

`CODE(`s`)` **compile a string**

Returns a pointer to the object code compiled from the SNOBOL4 statements in string s. This pointer can be assigned to a variable, say `C`, and the code executed with the direct Goto `:<C>`.

`CODE` fails if it finds a syntax error, and places an error message string in keyword `&ERRTEXT`. Individual statements in s are separated by a semicolon (`;`). The first character following a semicolon must be a blank, tab, the start of a label, or a comment. Control and continuation statements are not allowed in s. Statements may be any length; the 120 character limit when compiling from a file does not apply. Case-folding of names is controlled by keyword `&CASE`.

`COLLECT(`i`)` **regenerate storage**

This function calls SNOBOL4's garbage collection routine, which reclaims all unused storage. It returns an integer result that is the number of free descriptors remaining in the work space (a descriptor contains 5 bytes of storage). If there are less than i free descriptors after regeneration, the function fails. SNOBOL4 automatically calls `COLLECT` whenever memory becomes full.

`CONVERT(`d, s`)` **convert to specified data type**

The argument is converted to the specified data type and returned as the value of the function. If conversion is not possible, the function fails. s is a data type name string, such as `'STRING'`, `'TABLE'`, etc. Data type names may be lower case if case-folding is active. Chapter *Data Types and Conversion* lists allowable conversions.

`COPY(`d`)` **make copy of argument**

Returns a distinct copy of d. The argument may be an array, code block, pattern, or program-defined data type. If `A` is an array, the statement

```
B = COPY(A)
```

creates a new array `B`, whose initial contents are the same as array `A`. Their elements are independent; altering element `A<I>` does not affect element `B<I>`. In contrast, the assignment `B = A` makes `A` and `B` alternate names for the same array.

`DATA(`s`)` **create new data type**

Defines a new data type according to the prototype in string s. The prototype assumes a form similar to a function call, with the data type taking the place of the function name, and the field names replacing the arguments. The form of the prototype string is

$$newtype(field_1, field_2, \ldots, field_n)$$

The `DATA` function implicitly defines a new function and n new field variables:

$newtype(\ldots)$	object creation function of n arguments
$field_1(x)$	reference to field variable 1
...	
$field_n(x)$	reference to field variable n

where x is an object created with the $newtype$ function.

The fields may be of any data type, including pointers to other program-defined data items.

`DATATYPE(`d`)` **get data type of argument**

Returns a string specifying the data type of the argument. Some typical arguments and their data types are:

```
12                  INTEGER
'ABCD'              STRING
POS(2) 'C' LEN(3)   PATTERN
.Q<3>               NAME
*PAT                EXPRESSION
```

If the argument is a program-defined data type, the name from the creating `DATA()` function is returned.

`DATE()` **get current date and time**

Returns a 20-character string of the form:

```
'MM-DD-YY HH:MM:SS.CC'
```

representing month, day, year, hour, minute, second, and centisecond respectively. The centisecond field can only be approximated, since many personal computer clocks are only updated every 55 milliseconds.

`DEFINE(`s, n`)` **create program-defined function**

This function creates a new, program-defined function. s is a prototype string specifying the function's name, arguments, and local variables, if any. n is optional, and specifies a label as the first statement of the function body. If absent, a label with the same name as the function is the assumed entry point. The form of the prototype string is

$$\mathit{fname}\,(n_1, n_2, \ldots, n_s)\; l_1, l_2, \ldots, l_t$$

where fname is the name of the function, and n_i are names of formal arguments to the function. Blanks are not permitted in the prototype. The values of the local variables l_j are saved prior to function entry, and restored upon function return.

Functions may return a value or variable name by assigning the result to a variable with the same name as the function. Functions return by transferring to one of the reserved labels `RETURN`, `NRETURN`, or `FRETURN` to return by value, by name, or to fail respectively.

`DETACH(`n`)` **remove I/O association**

Removes any input or output unit associated with the variable name. The underlying file is not affected in any way. Remember that n is the address of the variable (e. g. `.X` or `'X'`), not the variable itself.

`DIFFER(`d_1, d_2`)` **check if arguments are different**

Succeeds and returns the null string if and only if d_1 and d_2 are different. Strings and integers are different if they have unequal values. Other data types contain pointers to the actual data object, and differ only if the pointers are different. If d_2 is omitted, `DIFFER` succeeds if d_1 is not null.

`DUMP(`i`)` **dump variables**

This function causes all natural variables with nonnull values to be listed on the file associated with I/O unit 6 (normally `OUTPUT`). If i is zero, the dump does not occur.

`DUPL(`s, i`)` **duplicate string**

Returns the argument string s repeated i times. The function returns the null string if $i = 0$, and fails if $i < 0$.

`ENDFILE(`u`)` **close file**

The file attached to the specified I/O unit is closed, and the file buffer is flushed and released. All variables which have been associated with this unit have their association removed. Upon program termination, SNOBOL4 will automatically perform an `ENDFILE` function on all open units.

`EQ(`i_1, i_2`)` **equality test for numbers**

This function succeeds and returns the null string if the two integer arguments are equal. i_1 and i_2 must evaluate to integer values. The function fails if i_1 is not equal to i_2.

`EVAL(`s or n`)` **compile and evaluate expression**

If the argument is a string, it should contain a valid SNOBOL4 expression to be compiled and evaluated. The evaluation result is returned as the value of the function. `EVAL` fails and sets `&ERRTEXT` to an error message string if s contains a syntactic error. If the argument is a number, i, it is returned unchanged. If the argument is an unevaluated expression, it is evaluated, and the result returned.

`FIELD(`s, i`)` **get field name of defined data type**

Returns a string which is the Ith field name from the formal definition of the program-defined data type whose name is in string s. `FIELD` fails if i is greater than the number of fields in the data type's definition.

`GE(`i_1, i_2`)` **greater than or equal test for numbers**

This function succeeds and returns the null string if the two integer arguments satisfy the relationship $i_1 \geq i_2$. i_1 and i_2 must evaluate to integer values. The function fails if $i_1 < i_2$.

`GT(`i_1, i_2`)` **greater than test for numbers**

This function succeeds and returns the null string if the two integer arguments satisfy the relationship $i_1 > i_2$. i_1 and i_2 must evaluate to integer values. The function fails if $i_1 \leq i_2$.

`IDENT(`d_1, d_2`)` **check if arguments are identical**

Succeeds and returns the null string if and only if d_1 and d_2 are identical. Strings and integers re identical if they have the same values. Other data types contain pointers to the actual data object, and are identical only if they point to the same object. If d_2 is omitted, `IDENT` succeeds if d_1 is the null string.

`INPUT(`n, u, i, s`)` **open file for input**

This function opens a file for input, and associates it with a variable. Data may then be read from the file by using the variable in an expression or an assignment statement.

The file designated by string s is opened for input and given the specified unit number. i is an optional record length. The variable specified by n is associated with this unit.

The first argument, n, specifies a SNOBOL4 variable, typically as a quoted string or with the unary period operator:

```
INPUT('IN', ...)
INPUT(.IN, ...)
X = 'IN'
INPUT(X, ...)
```

The second argument, u, must evaluate to an integer value in the range 0 to 16 inclusive. $u = 0$ (or omitting the u argument) will select the default input unit, 5.

The third argument, i, contains the record length in characters. $0 < i \leq$ `&MAXLNGTH`. If omitted, the default is 80.

The fourth argument, s, is a string containing the name of the file to be opened. If the file is a disk file, s may contain an optional drive letter and pathname in addition to the filename. Besides disk files, MS-DOS device names such as `NUL:`, `CON:`, `COM2:`, etc., are permitted.

If s is absent or null, and this unit is not currently open, the SNOBOL4 command line is searched for a file to use with this unit (`/n:file`). If s is absent, but the unit is already open, the `INPUT` call serves only to establish another association between a variable and the unit. If s is not null, any file previously associated with this unit number is first closed by SNOBOL4 with an implicit `ENDFILE(`u`)`.

An error message is generated for an illegal unit number. The `INPUT` function fails (with no printed error message) if the file cannot be opened.

The record length i (or its default value, 80), determines the number of characters returned in a string when the associated variable is referenced. ASCII files will return i characters or less if an end-of-line condition is encountered. End-of-line is defined as either a carriage return, or a carriage return followed by a line feed. If i characters are read from an ASCII file without encountering an end-of-line, additional characters are read from the file and discarded until the end-of-line character(s) are found. That is, long lines are truncated. If less than i characters are read from an ASCII file, and keyword `&TRIM` is zero, the line will be padded with blank characters until length i is obtained.

A read operation will terminate on the last character of a disk file, returning a short record. Reading past the end-of-file will cause statement failure. If the file is ASCII, reading a Ctrl-Z character will be treated as an end-of-file.

Note: When program begins execution, the variable `INPUT` is associated with unit 5. Unit 5 is normally device `CON:`, the keyboard, unless redirected elsewhere by the `/I=file` command line option, or the MS-DOS redirection operation (`<file`).

`INTEGER(`d`)` **check if argument is an integer**

Succeeds and returns the null string if d is an integer, or a string which can be converted to an integer. If the argument is not an integer, the function fails.

`ITEM(`$a, i_1, i_2, \ldots, i_n$`)` **get array element**

`ITEM(`t, d`)` **get table element**

Returns the specified array or table element. i_1, i_2, ..., i_n are array subscripts, and d is a table subscript. Since the first argument may be a function which returns an array or table name, it allows an indirect reference in situations that would not be syntactically valid. `ITEM` is an analog of the `APPLY` function. For example, if `F(X)` is a program-defined function which returns an array name,

```
ITEM(F(X),20)
```

references the 20th element of that array, whereas `F(X)<20>` is not acceptable.

`LE(`i_1, i_2`)` **less than or equal test for numbers**

This function succeeds and returns the null string if the two integer arguments satisfy the relationship $i_1 \leq i_2$. i_1 and i_2 must evaluate to integer values. The function fails if $i_1 > i_2$.

`LGT(`s_1, s_2`)` **lexically greater than test for strings**

This function succeeds and returns the null string if s_1 is lexically greater than s_2 (according to their alphabetic ordering). The two strings are compared left to right, character by character. If one string is exhausted before the other – with all characters equal – the longer string is lexically greater than the shorter string. The null string is lexically less than any other non-null string. If there is a character mismatch at the same position in both strings, the relationship between the characters determines the relationship of the strings. Strings are equal only if they are the same length, and are identical character by character.

`LOCAL(`n, i`)` **get local variable name from function definition**

Returns a string which is the Ith local variable from the formal definition of program-defined function name. `LOCAL` fails if i is greater than the number of local variables in n's definition. `LOCAL` is useful when one function is used to trace another. The trace function can access the local variables used with the funcion being traced with an indirect reference: `$LOCAL(`n`,`i`)`.

`LPAD(`s_1, i, s_2`)` **pad left end of string**

This function is useful for right-justifying columnar output. It returns s_1 padded on its

left end until its total size is i characters. The pad character used is the first character of s_2 if present, otherwise a blank is used if s_2 is absent or null. If i is less than or equal to the length of s_1, s_1 is returned unchanged.

`LT`(i_1, i_2) **less than test for numbers**

This function succeeds and returns the null string if the two integer arguments satisfy the relationship $i_1 < i_2$. i_1 and i_2 must evaluate to integer values. The function fails if $i_1 \geq i_2$.

`NE`(i_1, i_2) **not equal test for numbers**

This function succeeds and returns the null string if the two integer arguments are not equal. i_1 and i_2 must evaluate to integer values. The function fails if $i_1 = i_2$.

`OPSYN`(s_1, s_2, i) **create operator synonym**

The function or operator name s_1 becomes a synonym for s_2. If i is absent or 0, both strings are assumed to be function names. If i is 1 or 2, then the strings are assumed to be unary or binary operators, respectively. Other values for i are illegal. Operators are specified by using their graphic symbol in a quoted literal, such as:

```
OPSYN('#', '/', 2)
```

The concatenation operator is specified as a one-character string containing a blank: `' '`. The implicit pattern match operator between subject and pattern cannot be `OPSYN`ed.

`OUTPUT`(n, u, i, s) **open file for output**

This function opens a file for output, and associates it with a variable. Data may then be written to the file by assigning values to the variable.

The description of the `OUTPUT` function parallels that of the `INPUT` function, and will not be duplicated here. The following differences are noted below.

If the output file already exists, it is deleted and recreated anew. Facilities for updating existing files (direct-access files) are not present in Vanilla SNOBOL4; they are contained in SNOBOL4+, Catspaw's enhanced implementation of the SNOBOL4 language.

When an output variable is assigned a string value, the string is written to the associated file. A carriage return and line feed appended to the string. If the string is longer than the record length (i, or the default, 80), a carriage return and line feed will be inserted every i characters. That is, long strings will create multiple output lines.

Note: When a program begins execution, the variable `OUTPUT` is associated with unit 6. Unit 6 is normally device `CON:`, the display, unless redirected elsewhere by the /O: command line option or the MS-DOS redirection operation (`>file`). The variable `SCREEN` is associated with unit 7, which is also attached to device `CON:`.

`PROTOTYPE`(a) **get prototype which created an array**

Returns the prototype string of dimensions used to create the specified array. If the array was created by the `ARRAY` function, then the string returned is identical to the first argument of the original `ARRAY` function call. If the array was produced from a table by the `CONVERT` function, the string has the form `'N,2'`, where `N` is the integer number of rows in the array.

`REMDR`(i_1, i_2) **get remainder after division**

`REMDR` returns the integer remainder resulting from i_1 divided by i_2, that is, i_1 modulus i_2. The result has the same sign as i_1.

`REPLACE`(s_1, s_2, s_3) **replace characters in string**

This function returns s_1 transformed according to a translation specified by s_2 and s_3. Each character of s_1 found in s_2 is replaced by the corresponding character in s_3. s_2 and s_3 must be the same length. If duplicate characters appear in s_2, the rightmost one is used to obtain the mapping character from s_3. Normally, s_2 and s_3 are thought of as parameters, and `REPLACE` performs character substitutions on the variable s_1. For instance:

```
REPLACE(S, 'aeiouAEIOU', '1234512345')
```

replaces all upper- and lower-case vowels in `S` with the digits 1 through 5. It is possible to use `REPLACE` as a transposition function if s_1 and s_2 are considered parameters, and s_3 allowed to vary. If s_1 and s_2 are the same length, a simple positional transformation results. For example,

```
REPLACE('123456', '214365', S)
```

returns the six character string `S` with adjacent pairs of characters interchanged (`'ABCDEF'` becomes `'BADCFE'`). s_1 and s_2 can be different lengths – only s_2 and s_3 must be the same size. If s_2 contains characters not in s_1, the corresponding characters in s_3 are dropped from the result. If s_1 contains characters not in s_2, they will appear in the result. The function call

```
REPLACE('Yy/Mm/Dd', 'Mm-Dd-Yy xx:xx:xx.xx', DATE())
```

returns the date in the form YY/MM/DD (e. g., 87/07/28). Duplicate characters in s_1 are permitted, so:

```
REPLACE('aaabbbccc', 'abc' '(1)')
```

produces `'(((111)))'`.

`RPAD(`s_1, i, s_2`)` **pad right end of string**

This function is useful for left-justifying columnar output. It returns s_1 padded on its right end until its total size is i characters. The pad character used is the first character of s_2 if present, otherwise a blank (ASCII character 32) is used if s_2 is absent or null. If i is less than or equal to the length of s_1, s_1 is returned unchanged.

`SIZE(`s`)` **get length of string**

The function `SIZE` returns an integer value which is the number of characters in its argument string. A null string argument returns 0.

`STOPTR(`n`,type)` **stop trace**

Discontinues the type of trace of the named item. Consult the `TRACE()` function for a list of tracing types available.

`TABLE(`i_1, i_2`)` **create a table**

A table is similar to a one-dimensional array, but the subscripts may be any SNOBOL4 data type. The `TABLE` function creates a table and returns a pointer to it. The integer i_1 specifies the initial number of entries in the table. Integer i_2 specifies the size by which the table is increased whenever it becomes full, and additional table space is required. If either is omitted, 10 is used as a default value.

`TIME()` **get execution time**

Returns the number of tenths of a second elapsed since the start of program execution, including all I/O wait time.

`TRACE(`$n_1, type, s, n_2$`)` **trace an entity**

The item n_1 is traced according to the action specified by type. Trace output is written to the file associated with I/O unit 6.

n_1 is a the name of a variable, function, statement label, or keyword. It may appear as a string, or specified with the unary name operator (`.`).

type is a string that determines the type of trace desired. It must be one of these values:

`'VALUE'`	when value of n_1 is changed (default if type omitted)
`'CALL'`	when function n_1 is called
`'RETURN'`	when function n_1 returns
`'FUNCTION'`	when function n_1 is called, or returns
`'LABEL'`	when control is transferred to label n_1
`'KEYWORD'`	when the value of keyword `&`n_1 is changed; note that the ampersand character (`&`) is not included in the first argument, n_1.

s is an optional identifying tag that is added to the trace output line when n_1 is a created object, such as an array or table element.

n_2 is an optional name of a program-defined function.

Instead of producing a trace output line, this function is called when the trace action occurs. The function is called with n_1 as the first argument, and string s as the second argument.

Tracing will only occur when the keyword `&TRACE` is nonzero. Each trace will decrement `&TRACE` by one. Tracing ends when it becomes zero.

`TRIM(`s`)` **remove trailing blanks**

Returns the argument string with trailing blanks removed. Trailing tab characters are not affected. If the argument string was read from an input file, it is more efficient to set keyword `&TRIM` nonzero than to use `TRIM(INPUT)`.

By combining function `TRIM` with `REPLACE`, any trailing character can be removed. The desired character is temporarily exchanged with blank, trimmed, then exchanged back. For example, this expression returns string `S` with trailing zeros removed:

```
REPLACE(TRIM(REPLACE(S,'0 ',' 0')),'0 ',' 0')
```

`UNLOAD(`n`)` **remove function definition**

The function n becomes undefined.

`VALUE(`n`)` **get value of an object**

The `VALUE` function returns the value of the variable n, behaving like the unary indirect operator (`$`).

Chapter 19

System Messages

This chapter lists all messages displayed by SNOBOL4.

19.1 Initial messages

When SNOBOL4 begins execution, this title is displayed:

```
Vanilla SNOBOL4      Version 2.14.
(c) Copyright 1984,1988 Catspaw, Inc. All Rights Reserved.
```

Additional messages which may appear:

`Cannot open file:  name`
: The file specified in the command line cannot be opened.

`Command line error:`
: A syntactic error was detected in the SNOBOL4 command line. The command line is displayed on two lines. The line break shows where the error occurred.

`Errors detected in source program`
: There were compilation errors in the source program. Execution will proceed until a statement with a compilation error is encountered.

`Insufficient storage for initialization`
: Not enough memory was available to initialize the SNOBOL4 system.

`No errors`
: Compilation is complete, and without error. Execution begins immediately.

19.2 Termination messages

Termination messages are normally produced on I/O unit 7, which defaults to the user's display screen. If the `/B` option was used in the invoking command line, they are produced on I/O unit 6, associated with variable `OUTPUT`. Dump messages are always produced to unit 6.

`Normal termination at level LL`
: The program transferred to the label `END`. `LL` is the current program-defined function call depth. This message is produced only if the `/S` command line option (statistics) was used.

`filename(XXX) : Last statement executed was NNN`
: `NNN` is the statement number of the last statement executed, `XXX` is its source line number. It is the statement that transferred to the `END` statement. If this was a normal termination, it is only displayed if the `/S` option was used.

`filename(XXX) : Warning: Interrupted in statement NNN at level LL`
: Execution was interrupted when you pressed the `BREAK` or Ctrl-C key. The interruption occurred before the specified statement was executed. `LL` is the current call depth of program-defined functions.

`Incomplete storage regeneration. Terminal dump not possible`
: Stack overflow occurred during storage regeneration, and the `&DUMP` keyword was nonzero. Memory is in an indeterminate form, and a dump listing cannot be produced.

`Dump of variables at termination`
: Natural variables Unprotected keywords These headings will appear if a termination dump was requested by setting the `&DUMP` keyword nonzero. Variables are listed only if they contain a nonnull value. The variable names will be sorted if the `&DUMP` keyword is positive; they are unsorted if it is negative.

19.2.1 Job statistics

End-of-run statistics on program execution are provided if the `/S` command line option is used. Compilation and execution times are in tenths of a second. Times are wall-clock values, and include all I/O wait time, such as delays for keyboard input:

```
SNOBOL4 statistics summary:
NN tenths of a second compilation time
NN tenths of a second execution time
NN statements executed, NN failed
NN arithmetic operations performed
NN pattern matches performed
NN regenerations of dynamic storage
NN reads performed
NN writes performed
```

19.3 Compilation messages

SNOBOL4 syntax errors are detected during compilation. Statement compilation ceases at the point where the error was detected. The error message contains a marker which indicates the valid portion of the statement accepted by the compiler – the error occurred after this point. Only the first error in a statement is detected. The erroneous statement is compiled with an internal error code which produces an error message if the statement is executed. Compilation resumes with the next statement. Compilation ceases and SNOBOL4 terminates if more than 50 errors are found.

When compiling without a list file (`/L`: command line option), the compiler will attempt to display the erroneous line on your screen. If a statement is continued over several lines, only the line in error is displayed. Several errors cannot be detected until the absolute end-of-statement is found. This may require reading the next line, and finding it is not a continuation statement. In this case, the single line displayed will be the next line, with the error marker in the first character position.

The `CODE` function may be used to compile SNOBOL4 statements that have been concatenated into a long string. The `CODE` function fails if a syntax error is found, and the keyword `&ERRTEXT` contains the error message string for the error encountered.

`Binary operators must be surrounded by blanks`
: Omitting a blank will often cause this error. An illegal or undefined binary operator will also produce this error.

`Error in Goto`
: There is a syntactic error in the Goto field.

`Erroneous END statement`
: The `END` statement contains a syntactic error, or the label specified in the subject field for initial transfer could not be found.

`Erroneous integer`
: An integer number appears which is too large for the SNOBOL4 system. The allowable range for magnitude values is 0 to 32 767.

`Erroneous label`
: The first character of a statement must be blank, tab, alphanumeric, `*` (comment), `+` or `.` (continuation), or `-` (control).

`Erroneous or missing break character`
: A character which separates language elements occurs in an illegal context, or an expression is not balanced with respect to parentheses.

`Erroneous subject`
: A compiler break character appears before the statement subject field. A break character is any of the following: `, = ) ] >`.

`Illegal character in element`
: A character was found which was incorrect for the type of language object being compiled. This often occurs when a blank is omitted between elements, causing them to run together.

`Improperly terminated statement`
: The source statement terminated with an incomplete language construction.

`Limit on compilation errors exceeded`
: More than 50 compilation errors were found in the source program.

`No END statement in source file`
: End-of-file was encountered in the source file without an `END` statement.

`Previously defined label`
: A duplicate label appears. The first definition is retained; subsequent definitions are discarded.

`Unclosed literal`
: The closing quotation mark from a literal string is missing. This error also occurs if the closing quotation mark (single or double) was different from the opening mark.

19.4 Execution error messages

Most program logic errors can only be detected during program execution. Some are unconditionally fatal, and cause the SNOBOL4 system to terminate. Others are conditionally fatal – the system terminates if the value of the keyword `&ERRLIMIT` is zero. If `&ERRLIMIT` is nonzero, the keyword `&ERRTYPE` is set to the error message number, `&ERRTEXT` is set to the message text, `&ERRLIMIT` is decremented, and execution continues.

The protected keyword `&ERRTYPE` may be traced, permitting a program-defined function to gain control when a conditional error occurs. The program `CODE.SNO` provides an example of how to do this. The initial value of the unprotected keyword `&ERRLIMIT` is zero, forcing program termination upon any error.

Errors 1-16 are conditionally fatal. Errors 17-28 are unconditionally fatal. When execution terminates due to an error, the following is displayed:

```
filename(XXX) : Error NN, -- description --
In statement NNN, at level LL
```

NN is the error number below. NNN is the statement number assigned in the compiler list file, XXX is the absolute line number in the source file. LL specifies the current programdefined function call depth (0 is the normal main-program level).

1. Illegal data type
   The data type of an operand was incorrect for the type of operation attempted. This occurs most frequently with arithmetic operations, when one operand is a string which cannot be converted to a number.

2. Error in arithmetic operation
   An arithmetic operation upon integer values produced a result which was out of range, or was undefined, such as division by zero.

3. Erroneous array or table reference
   An array or table reference was made to a variable which did not contain an array or table pointer.

4. Null string in illegal context
   The null string appeared where it is not permitted, such as the object of an indirect reference.

5. Undefined function or operation
   A function was called before it was defined, or an undefined operator was used.

6. Erroneous prototype
   A syntactic error occurred in the prototype string used with the functions ARRAY, DATA or DEFINE. Note that the blank and tab characters are not permitted within the prototype string.

7. Unknown keyword
   The keyword specified is unknown to the SNOBOL4 system.

8. Variable not present where required
   A variable name must be used as the subject of an assignment statement, or as the argument of the unary cursor, name, or keyword operator (@, ., &), or the binary pattern match assignment operators (., $).

9. Entry point of function not label
   At the time a program-defined function was first called, its entry point label did not appear as the label of any SNOBOL4 statement.

10. Illegal argument to primitive function
    An illegal value was used as an argument to the function ARG, FIELD, LOCAL, OPSYN, STOPTR, or TRACE, or an illegal value was specified in the third argument to INPUT or OUTPUT.

11. Reading error
    An error condition was returned when reading from a file.

12. Illegal I/O unit
    Allowable unit numbers are 1 through 16 (inclusive). (Unit 0 is allowed in functions INPUT and OUTPUT, and is converted to units 5 and 6 respectively.)

13. Limit on defined data types exceeded
    SNOBOL4 allows 899 different program-defined data types.

14. Negative number in illegal context
    A negative number was used incorrectly as the argument of the function LEN, POS, TAB, or RTAB.

15. `String overflow`
The program attempted to create a string larger than `&MAXLNGTH` characters.

16. `Overflow during pattern matching`
The internal SNOBOL4 stack overflowed during pattern matching. This can happen when a recursive or looping pattern is incorrectly specified.

17. `Error in SNOBOL4 system`
This message indicates an internal SNOBOL4 system error.

18. `Return from level zero`
An attempt was made to transfer to the function return label `RETURN`, `FRETURN`, or `NRETURN` outside of any function call.

19. `Failure during Goto evaluation`
The expression used for an indirect transfer within the Goto field failed when evaluated.

20. `Insufficient storage to continue`
All available memory has been used. Vanilla SNOBOL4 is limited to 30K bytes for program and data. SNOBOL4+, Catspaw's enhanced version, allocates 300K bytes for program and data.

21. `Stack overflow`
The SNOBOL4 internal stack has overflowed. This may be caused by excessive function recursion, or occur during memory garbage collection.

22. `Limit on statement execution exceeded`
The number of statements executed was greater than the value in the keyword `&STLIMIT`. `&STLIMIT` is initially -1, specifying unlimited execution.

23. `Object exceeds size limit`
The program attempted to create an object larger than the maximum size allowed.

24. `Undefined or erroneous Goto`
A transfer was attempted to an undefined label, or an expression in a Goto field evaluated to a string, rather than a label name – usually the result of omitting the indirect operator (`$`).

25. `Incorrect number of arguments`
A primitive function was called with too many arguments.

28. `Execution of statement with compilation error`
Execution proceeded to a statement that contained a compilation error.

19.5 Execution trace messages

Tracing is provided for variables, certain keywords, label transfers, and function calls and returns. A trace message is output to I/O unit 6 for each trace occurrence. Program execution time, in tenths of a second, is appended to each message.

`Tracing normally occurs only if the keyword &TRACE is nonzero.`
However, another keyword, `&FTRACE`, may be set nonzero to trace all function calls and returns independently of keyword `&TRACE`.

`STATEMENT NN: <vname> = <value>,TIME = TT`
Value trace; produced by the function call `TRACE('vname', 'VALUE')`, where vname is the name of the variable to be traced.

`STATEMENT NN: &<keyname> = <value>,TIME = TT`

Keyword trace; produced by the function call `TRACE('keyname', 'KEYWORD')`, where keyname is the upper case keyword name, without the leading ampersand.

`STATEMENT NN: TRANSFER TO <labname>,TIME = TT`

Label trace; produced by the function call `TRACE('labname', 'LABEL')`, where labname is the desired label name. Tracing only occurs on a transfer of control; it does not occur if the labeled statement is flowed into.

`STATEMENT NN: LEVEL LL CALL OF <fname>(arg1,...,argn),TIME = TT`

Call trace; produced by the function call `TRACE('fname', 'CALL')`, where `fname` is the name of the function to be traced. The function's arguments at the time of the call are evaluated and displayed.

`STATEMENT NN: LEVEL LL RETURN OF <fname> = <value>,TIME = TT`

`STATEMENT NN: LEVEL LL NRETURN OF <fname> = <value>,TIME = TT`

`STATEMENT NN: LEVEL LL FRETURN OF <fname>,TIME = TT`

Return trace; produced by the function call `TRACE('fname', 'RETURN')`, where `fname` is the name of the function whose return is to be traced. The type of return that occurred is displayed in the trace message.

`***Print request too long***`

An internal buffer is used to display trace messages, and variable values during dumps. If the required display is longer than 1 800 characters, this error message is produced instead.