MACRO

SPITBOL

The High-Performance SNOBOL4 Language

Tutorial and Program Reference Manual

Manual text by Mark B. Emmer and Edward K. Quillen
with additional material by Robert B.K. Dewar

Catspaw SPITBOL™ program by
Robert B.K. Dewar, Mark B. Emmer, and Robert E. Goldberg

Catspaw, Inc.
P.O. Box 1123
Salida, CO 81201
U.S.A.
719/539-3884
FAX: 719/539-4830
support@SNOBOL4.com

**For technical assistance, call 719-539-3884,
Monday-Friday, 8:30 a.m. to 5:00 p.m., Mountain Time,
or FAX 719-539-4830.**

**Electronic mail: support@SNOBOL4.com
Web address: www.SNOBOL4.com**

Originally published in 1989 by Catspaw, Inc.
Revised September, 1991 for OS/2 2.0.
Revised May, 1995 for SPITBOL-8088 and AIX SPITBOL
Revised Februrary, 1998 for Windows NT.
Revised September, 2000.

Printed in the United States of America

Preface

What's in a name? Apparently a great deal when the name is "SPITBOL." The name elicits either confusion, derision, or admiration.

Confusion comes from those who think "SPITBOL" is a dialect of "COBOL," even though there's absolutely no similarity between the two. True, they do share "BOL" in their name, and Robert Dewar, SPITBOL's creator, has also written a COBOL compiler, but that's as close as they get. SPITBOL and COBOL were designed for very different types of data processing.

Derision comes because of SPITBOL's homophone, "spitball," which my dictionary defines as "1. a small ball or lump of chewed paper used as a missile. 2. *Baseball,* a pitch, now illegal, made to curve by moistening one side of the ball with saliva or perspiration." Hardly the stuff of which modern marketing images are made.

Admiration comes from those who have used other implementations of SPITBOL. Mainframe and minicomputer SPITBOLs have always had a following among programming cognoscenti. They know that calling it a "SPeedy ImplemenTation of snoBOL" is an understatement. Whatever connotations SNOBOL4 has as a powerful but slow language are discredited with an implementation that typically runs six to ten *times* faster than other SNOBOLs.

We briefly considered creating a more modern name, like "Athena," appropriately unfocused and non-descriptive for the current age; a name that could be applied equally to software or shampoo. But in the end we decided to wear the name proudly: SPITBOL—raw speed and programming power for non-numerical problem solving.

In today's climate of structured and object-oriented programming, the SNOBOL4 language may look archaic. Gotos? In the 1990's? But go a little deeper, and you'll find many features that are advanced by any standard: absence of type declarations, automatic memory management with dynamically-sized objects, associative data structures, patterns as objects, first-class objects, and runtime compilation of new program material.

I've used various incarnations of the SNOBOL language for over twenty-five years. Don't think me immodest when I say that using SPITBOL on a modern PC or workstation is exceedingly pleasant. The computer speed and memory now available on your desktop means never again having to apologize for SNOBOL4's performance. But don't take my word for it. Dig in and get started. You'll be surprised at how quickly you can write programs that would be difficult to formulate in other languages.

And try to have fun while you're at it. As one user told me: "Programming in SNOBOL4 sometimes leaves me euphoric."

Acknowledgements

The research that produced the SNOBOL series of languages took place at the Bell Telephone Laboratories during the 1960s. Bell Labs is to be commended for allowing the fruit of that research to enter the public domain.

Ralph Griswold, one of the original designers of SNOBOL, has kept the language alive through the SNOBOL4 Project at the University of Arizona. Dr. Griswold also gave permission for many of his SNOBOL4 programs from *String and List Processing in SNOBOL4* to be distributed here.

Robert Dewar created Macro SPITBOL in the mid 1970's, and showed the world that a high-performance version of SNOBOL4 was possible. His Macro SPITBOL compiler forms the kernel of this implementation. As an example of "aggressive" assembly-language programming, it remains a programming tour de force.

SNOBOL4 and SPITBOL evolved in a special world that is more collegial than competitive. Several SPITBOL loyalists have given generously of their time over the years to refine and improve Macro SPITBOL, among them Anthony McCann, Steve Duff, Robert Goldberg, and David Shields.

Portions of SPITBOL's I/O system and operating system interface were created by Robert Goldberg, based upon the ideas of Andrew Koenig.

Ed Quillen, local novelist, political columnist, and SNOBOL enthusiast, co-authored this manual. He combined various terse SPITBOL documents from other systems with Catspaw's SNOBOL4+ manual, while providing more complete explanations of some concepts. Any observed clarity in this manual is due to Ed, while the more opaque portions can be blamed on me.

My wife, Nancy R. Vickery, has adjusted to life with a SPITBOL enthusiast. She also proofread the manuscript and provided design assistance. Without her support, sustenance, and encouragement, SPITBOL would have remained a distant dream.

Finally, I'd like to recognize and thank the many users of SNOBOL4+ for their support and continued loyalty to the SNOBOL4 way. Your suggestions and comments have helped keep the SNOBOL4 flame alive.

Mark B. Emmer

Table of Contents

PART I
Getting Started

Chapter 1 Installation

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.

SPITBOL is an extremely high-performance implementation of the SNOBOL4 language. If you would like to add it to your repertoire of problem-solving tools, and learn why other users swear by it, read on.

About This Manual

Scope

This manual covers all implementations of Catspaw SPITBOL. At this writing it covers these architectures:

- Intel 80x86-based systems
- Motorola 680x0-based systems
- Sun SPARC RISC systems
- IBM POWER (RS/6000) and PowerPC-based systems
- MIPS R-3000 systems

and these operating systems:

- MS- and PC-DOS
- OS/2

- Windows 95 and Windows NT
- various Unix flavors: AIX, BSD, IRIX, System V, SunOS, and Solaris

The Intel 80x86-version is further divided into a 16-bit SPITBOL-8088 and a 32-bit SPITBOL-386. SPITBOL-8088 will run on all 80x86-family computers, from 8088s to Pentium Pros. SPITBOL-386 requires a 80386-, 80486-, or Pentium-family CPU. The combined nature of the manual reflects our desire to standardize on one version of SPITBOL and make it available on a variety of platforms and operating systems. Because of the very different work environment provided by the Apple Macintosh, a different manual is provided with Catspaw's MaxSPITBOL system.

MS-DOS, Windows 95, Windows NT

If you have purchased SPITBOL-386, you will find one SPITBOL executable file on the distribution disk. This file will run by itself as a 32-bit program in a command window under Windows 95 and Windows NT. Under native MS-DOS or Windows 3.1, you will also need the *DOS Extender* provided on the distribution disk to run SPITBOL as a protected-mode 32-bit program.

Organization

This manual is divided into four parts. This part, "Getting Started," shows you how to create and run small programs with SPITBOL. It consists of Chapters 1 and 2. Beginners and experts alike should take the time to read these two small chapters.

Part II, "Tutorial," is addressed to the beginning SNOBOL4 programmer; it comprises Chapters 3–11. It assumes a modest knowledge of general programming concepts, and experience with another programming language, such as BASIC, C, or Pascal. Readers without any programming background may wish to consult a book written with them in mind: *SNOBOL Programming for the Humanities,* listed in the bibliography.

Part III, "Reference," is a complete description of the SPITBOL product; it comprises Chapters 12–20. If you are already familiar with the SNOBOL4 language, you may wish to skip the tutorial and proceed to the reference section for specific details about SPITBOL. Later, you can return to the tutorial section for fresh insight into the language's use.

Part IV holds the appendices. Appendices A and B describe the programs on the SPITBOL distribution media. Appendix C lists differences between SPITBOL, standard SNOBOL4, and SNOBOL4+. Appendix D lists all compilation and run-time error messages. Appendix E lists HOST functions provided for machine-specific programming. Appendix F describes the external function interface for those implementations that provide it. Finally, there is a bibliography of SPITBOL-related material, and an index.

Installing SPITBOL

Create a new sub-directory on your hard disk called **spitbol**. (All file and directory names will be shown as bold lower case in this manual. Some systems such as MS-DOS will automatically convert such names to upper-case. No harm comes from entering them in lower case however). Make **spitbol** your current sub-directory.

Installing from disk

Copy the distribution disk(s) into the new sub-directory. MS-DOS and OS/2 users can simply use the xcopy command:

```
xcopy   a:*.*   /s
```

Windows users may prefer to use the file manager to drag the floppy disk icon to the subdirectory.

Sun 4/SPARC users will receive a “bar”-format floppy disk, created with compression. The method of reading depends upon the operating system. If you are running SunOS 4.x (Solaris 1), use:

```
bar   xvfZ   /dev/rfd0
```

If you are running Solaris 2.x, use the following two-command sequence:

```
volcheck
cpio -i -Hbar -d </vol/dev/rdiskette0/unlabeled
```

By special arrangement, AIX and Unix users can receive their SPITBOL system on an MS-DOS formatted diskette. They should use local system utilities to transfer the files and directory structure through their network to the target system.

Installing from tape

Users who receive SPITBOL as a “tar”- format cartridge tape can copy all files off the tape with either of these commands:

```
tar   xvf   /dev/rst0       (SunOS 4.x)
tar   xvf   /dev/rmt/0      (Solaris 2.x)
```

You may have to adjust the tape device name for your particular system.

If you are short on disk space, you may wish to only retain those files needed to run SPITBOL and work through the tutorial. Those files are **spitbol** (or **spitbol.exe**), **code.spt**, **readme.txt, tictac.spt, faustus**, **asc.inc**, **capitals.dat**, **fact.inc** and **roman.inc**. MS-DOS and Windows 3.1 users will need the DOS Extender files **32rtm.exe**, **dpmi32vm.ovl** and **windpmi.386**.

The other files on the distribution media are sample programs and functions which will be useful after you’ve used the tutorial.

Unix users may wish to copy the **spitbol** file to /usr/local/bin, or other appropriate directory on their path. The **spitbol** file must be given read and execute privileges (chmod +xr spitbol).

OS/2, Windows 95, Windows NT and Unix users should skip ahead to the Checkout section. Placing the SPITBOL executable somewhere on your execution path is all that is required. However, MS-DOS and Windows 3.1 users must read the following material describing the installation of the DOS Extender.

DOS-Extended SPITBOL-386

SPITBOL-386 for Windows NT can be run under MS-DOS or Windows 3.1 using the supplied DOS Extender. A DOS Extender allows SPITBOL to operate in 32-bit mode, and to break the 640K barrier common to MS-DOS systems. This DOS Extender is from Borland International, and is commonly known as their "PowerPack for MS-DOS."

Previous versions of SPITBOL-386 contained bound-in DOS Extenders from PharLap Software and Intel Corporation. These versions have been discontinued in favor of the present system — a native Win32 application and a separate, stand-alone DOS Extender for those environments that require it.

In addition to running in native MS-DOS environments, this DOS Extender is compatible with DPMI[*]-compliant hosts, such as a DOS Shell under Windows 3.1 Enhanced mode or under OS/2.

MS-DOS

For both MS-DOS and Windows 3.1, the two files **32rtm.exe** and **dpmi32vm.ovl** must be present in the current directory or on your path. We suggest placing them in your **c:\dos** directory. **32rtm.exe** is the 32-bit run-time manager, and **dpmi32vm.ovl** is a DPMI server that is installed by **32rtm** if DPMI services are not provided by your system. If DPMI is present, **32rtm** uses the existing services.

Windows 3.1

For Windows 3.1 (or Windows for Workgroups), one additional step is necessary. The file **windpmi.386** should be copied to the same directory where you placed **32rtm.exe**. Now go to your Windows directory, and edit the file **system.ini** using Notepad or some other editor. Look for the section that begins "[386Enh]" and add a line like this:

```
Device=c:\dos\windpmi.386
```

*DPMI - DOS Protected Mode Interface, a second-generation protected mode standard. Version 0.9 is supported by the DOS Shell in Windows 386 Enhanced mode and OS/2. It is likely to be supported in future versions of 80386 Unix.

where "c:\dos\" should be adjusted to reflect the path where you placed the **windpmi.386** file. You'll need to restart Windows to have this change take effect.

SPITBOL-8088

Throughout this manual, we'll use small sidebars like this to point out differences between SPITBOL-8088 and the other, 32-bit versions

The Intel 8088 version of SPITBOL is supplied in two forms, **spitbols.exe**, and **spitboll.exe**. The difference is whether integers are stored as 16- or 32-bit values. **spitbols.exe** provides *s*hort, 16-bit integers, while **spitbol*l*.exe** provides *l*ong, 32-bit integers.

The 8088 CPU has 16-bit wide data paths and registers, so 16-bit arithmetic is natural and fast. Even later processors, such as the Pentium, are operating under PC/MS-DOS with 16-bit registers unless a DOS Extender is active. 32-bit integer arithmetic will be slower, both because it must be emulated using the chip's 16-bit facilities, and because an additional two bytes of storage are required. The additional storage requirements also reduce the amount of data that can be stored in SPITBOL-8088's workspace.

For the purposes of this tutorial, we suggest using **spitboll.exe**. Its 32-bit integers are identical to those of its "big brother" SPITBOL versions, so examples will work exactly as presented. Throughout this tutorial, type "spitboll" instead of "spitbol," or simply rename **spitboll.exe** to **spitbol.exe**.

Checkout

Once SPITBOL is installed, play a game of Tick-Tack-Toe. Enter

```
spitbol tictac
```

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

File readme.txt

We put a file called **readme.txt** on the distribution media. It contains last-minute information that became available after this manual was printed. To view this file, load it into your text editor for inspection.

An Example

Just to get a feel for where we're going, let's take a look at a small program on the next page. 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; we just want to give you a taste of the language.

Running the program with some file of text as input would produce a sorted usage count like this:

```
Word  Counts
a  –  147
able  –  18
above  –  3
aren't  –  2
…
```

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 a sorted 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 11 working statements in this program—and this program uses only a fraction of SPITBOL'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 SPITBOL?

Sample program to produce a list of unique words in a file

```
*  Program wordlist.spt
*
*  To run:     spitbol wordlist <datafile
*
* Trim input, set up constants, create table to hold word counts
           &trim          = 1
           wrdpat         = break(&lcase) span(&lcase "–'") . word
           tally          = table(1000)

* Read a line, convert upper-case letters to lower-case
read       line           = replace(input, &ucase, &lcase)     :f(sort)

* 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, and sort the words
sort       result         = sort(tally)

* Display the results
           output         = “Word Counts”
           n              = 1
print      output         = result[n,1] “ – ” result[n,2]        :f(end)
           n              = n + 1
           :(print)
end
```

Chapter 2
First Program

This chapter will do three things:

1. It will take you through the steps needed to create and run a very simple SPITBOL program.
2. It will show you how to use the interactive execution features of the **code.spt** program. This handy program allows you to "try out" various SPITBOL statements and constructions. You'll find it useful whenever you use SPITBOL.
3. For the experts in the audience who want to skip the tutorial in Part II, we'll point out a few key things you need to know when converting programs from other SNOBOL4 systems.

A First Program

We will begin with a very simple program, one that displays a simple message on your computer's display screen. It will familiarize you with the mechanics of running a SPITBOL program.

In the examples that follow, we'll use the convention of showing text that you type in boldface. Every line you enter from the keyboard should end by pressing the Enter or Return key.

You start the system by typing **spitbol** and a hyphen at the system command prompt. SPITBOL displays two title lines and waits for you to enter your program:

```
spitbol –
SPITBOL-386    Release 3.7(ver 1.29)   Serial 20001
(c) Copyright 1987-1995 Robert B. K. Dewar and Catspaw, Inc.
```

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 = 'Real programmers use SPITBOL!'
END
Real programmers use SPITBOL!
```

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 SPITBOL that you have finished entering program lines. It must appear at the left margin to be recognized. After the END statement is entered, SPITBOL 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 'Real programmers use SPITBOL!'. The variable's name is OUTPUT, which is a special name in SPITBOL; 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.

Capitalization of names is largely historic. You're free to use any convention that you're comfortable with, such as Output or end.

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

Use your text editor to create the same two line program:

```
        OUTPUT = 'Real programmers use SPITBOL!'
END
```

and save it is a file called **real.spt**. If you are using a word processor, remember to produce an unadulterated ASCII file, free of any special format controls. Now you can have SPITBOL read and execute your program from a file:

```
spitbol real.spt
SPITBOL-386   Release 3.7(ver 1.29)   Serial 20001
(c) Copyright 1987-1995 Robert B. K. Dewar and Catspaw, Inc.
Real programmers use SPITBOL!
```

We substituted the program name for the hyphen on the command line (the hyphen told SPITBOL to read the file "keyboard").

SPITBOL assigns a unique number to each program statement. The statement number and line number are displayed whenever an error message is produced. Statement numbers may differ from the line numbers in your text editor. To get a listing of your program with SPITBOL's statement numbers, try (that's a lower-case "L" in the line below):

```
spitbol –l real
SPITBOL-386   Release 3.7(ver 1.29)   Serial 20001
(c) Copyright 1987-1990 Robert B. K. Dewar and Catspaw, Inc.
Real programmers use SPITBOL!
```

The command line allows options to be specified between the word "spitbol" and your program name. The option –l tells SPITBOL to produce a listing of your source file. It writes it to a file with the same name as your program but with the extension **.lst**.

Now display the file **real.lst** using your system's **type** (MS-DOS, Windows, OS/2) or **cat** (Unix) command:

```
type real.lst
```

or

```
cat real.lst

Macro SPITBOL Release 3.7(ver 1.29)
80386   05/22/95 14:36:00


1             OUTPUT = "Real programmers use SPITBOL!"
2          END
```

Other command line options are discussed in Chapter 13, "Running SPITBOL." In this example we omitted the file name extension. SPITBOL will supply the **.spt** extension for the source file if it is absent.

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

Interactive Statement Execution

Normally, you'll create a program in your text editor, save the file, and then run it.

But when first learning SPITBOL, it's helpful if you can test simple statements as they are introduced in the text. If an idea is unclear, you can try different variations to see what's happening. There is a SPITBOL program called **code.spt** on the distribution media to help you do this.

The code.spt program

To use **code.spt**, make sure it is in your current directory, and start up SPITBOL with **code** as the program name. Type END to stop the program:

```
spitbol code
SPITBOL-386    Release 3.7(ver 1.29)   Serial 20001
(c) Copyright 1987-1995 Robert B. K. Dewar and Catspaw, Inc.
Enter SPITBOL statements:
?          OUTPUT = 'Hello World!'
Hello World!
Success
?          OUTPUT = 16
16
Success
```

Feel free to experiment—you can't break anything by using this program. At most, you will get an error message. When you're done, stop the program by typing end:

```
?end
```

Whenever you see examples in the text that begin with a question mark, they are meant to be tried with **code.spt**. In the rest of this manual, we'll omit the bold type and the word Success unless it is relevant to the concept being presented.

What's next?

If you're just learning the SNOBOL4 language, start working through the tutorial in Part II. If you're already a SNOBOL4 programmer, and are anxious to get some existing programs running right away, read the following section, and then turn to Chapter 13, "Running SPITBOL." It's full of detailed information on the many features we've added to make the system more flexible.

Experienced Users

If you're familiar with the SNOBOL4 programming language, this section will give you the absolute minimum needed to start writing programs immediately. If you're not in that category, skip this section, and we'll proceed at a more leisurely pace.

Converting to SPITBOL

If you have existing SNOBOL4 programs that you want to run, here's a quick checklist of things that may have to be changed:

- Calls to the INPUT() and OUTPUT() functions. SNOBOL4+ and Robert Dewar's original PC-SPITBOL provide different processing options, and SNOBOL4+ places file names in a fourth argument. Full details are in the description of the INPUT function, see page 224. Catspaw SPITBOL uses the format:

  ```
  INPUT(.Variable, Channel, "filename[options]")
  ```

  where Channel can be either a simple integer, or the name of a variable.

- SPITBOL does not provide the &STFCOUNT keyword. The &FULLSCAN keyword is always set to 1 — pattern matching always takes place in fullscan mode.

- SPITBOL HOST functions are machine dependent. See Appendix E for specific differences.

- Some systems pre-declare a SCREEN variable to write data to the console. SPITBOL uses an equivalent variable called TERMINAL.

A complete list of differences and enhancements is given in Appendix C, particularly the section "SPITBOL for SNOBOL4+ Users," page 268.

PART II
Tutorial

Chapter 3 Fundamentals

SPITBOL 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 SPITBOL and its immediate relatives, such as SNOBOL4.

You should try to master the conventional portion of SPITBOL 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 book can only offer an introduction. The sample programs on the distribution diskette can be studied for a deeper understanding of patterns and their application. In addition, the book *Algorithms in SNOBOL4* (by James Gimpel, available from Catspaw) provides an extensive discussion of the theory of pattern-matching.

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

Simple Data Types

All computer languages provide certain basic types of data that are used in programs to produce meaningful results. SPITBOL has nine different basic types, but has a mechanism to define many more as aggregates of others. Initially, we'll discuss three of the most basic: integers, reals, and strings.

Integers

*Exception: In the short integer version of SPITBOL-8088 (**spitbols.exe**), integers are restricted to the range −32768 to +32767.*

An integer is a simple whole number, without a fractional part. In SPITBOL, its value can range from −2147483648 to +2147483647. 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 not integers in SPITBOL:

13.4	fractional part is not allowed
3145926535	larger than 2147483647
–	number must contain at least one digit
3,076	comma is not allowed

Use the **code.spt** program to test different integer values. Try both legal and illegal values. Here are some sample test lines:

```
Enter SPITBOL statements:
?         OUTPUT = 42
42
?         OUTPUT = −825
−825
?         OUTPUT = 5356295413
Error #231, Syntax error: Invalid numeric item
```

Reals

Unlike integers, both versions of SPITBOL-8088 support the full range and precision of real numbers.

Reals are numbers with a decimal point; they are sometimes called *floating-point* numbers. Reals provide a much wider range of values than integers, but at the expense of lost precision and longer program execution time. A real number has either a decimal point or an exponent (it must have at least one for SPITBOL to recognise it as a real number). There must be at least one digit to the left of the decimal point. The real number also has an optional sign.

Here are some examples of acceptable real numbers:

12.9543	
−0.0053294	
123456789012345678.901	
1.0E+7	(10000000.0 or 1 times 10 to the 7th power)
1E+7	(Has no decimal point, but has exponent)
2E7	(Sign for exponent is optional if positive)
8.4E−7	(0.00000084 or 8.4 times 10 to the −7th power)

SPITBOL maintains accuracy to 15 decimal places on real numbers (IEEE 64-bit, double-precision format). When converting to a string, the mantissa is displayed with 9 significant digits. The 10th digit is rounded; 6 and above in the 10th place will increment the 9th digit when displayed. Therefore, the long number above would be displayed by SPITBOL as 0.12345789E+18.

Note that if a real has more than 9 digits to the left of the decimal point, SPITBOL scales the mantissa between 0 and 1, and adjusts the exponent accordingly. Reals less than 1.0 are always handled this way.

These are incorrect real numbers:

.63	no digit to the left of the decimal point
1.23E–2.3	exponent not an integer

If a real number is larger than 0.18E+309, it causes an error. Generally, full precision is maintained for values as small as 0.22E–309, then there is a gradual loss of precision down to 0.49E–322. Numbers smaller than that are converted to 0. (In the absence of floating-point hardware, the precision of very small numbers is dependent upon the quality of each system's floating-point library. Some SPITBOL implementations, notably SPITBOL-386, may only have numbers as small as 0.23E–307, after which they abruptly switch to 0.)

*A **code.spt** shortcut: typing "OUTPUT = " for each test line quickly becomes tiresome. If you type an "=" right after the "?", **code.spt** will evaluate the expression and display the results.*

*Remember, this only works in **code.spt**; it's not a general feature of the SPITBOL*

Here are some lines to try with **code.spt**:

```
?        OUTPUT  =  5.23
5.23
?        OUTPUT  =  1.8E+3
1800.
?=  1.8E–3
0.0018
?=  –3.0E+400
Error  #231,  Syntax  error:  Invalid  numeric  item
?=  5.3E–400
0.
?=  –5.4.
Error  #231,  Syntax  error:  Invalid  numeric  item
?=  +9.5E29
0.95E+30
```

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 4,194,304 characters (4 megacharacters). If that's not long enough for you, Chapter 13 describes a command line option (–m) that tells SPITBOL to allow longer strings.

String length is limited to 9,000 characters in SPITBOL-8088, and cannot be increased.

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. You'll find its role in SPITBOL 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 single quotation (') 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 quotation marks, with no intervening characters. Here are some samples to try with **code.spt**:

```
?          OUTPUT = 'THIS IS A STRING LITERAL'
THIS IS A STRING LITERAL
?= "So is this one"
So is this one
?= ""

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

(Generally it's easier to use the single quote to mark literal strings, because it is not a shifted character—and that makes for faster typing.)

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. But SPITBOL provides many more, some of which are unique to the SNOBOL language family. SPITBOL also allows you to define your own operators. We'll examine just a few basic operators below.

Unary vs. binary

SPITBOL 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*:

```
–6.92
```

Or, it can be a *binary* operator with two operands:

```
4.25 – 1.33
```

In the first case, the minus sign negates the number. The second example subtracts 1.33 from 4.25. The minus sign's meaning depends on the *context* in which it appears. SPITBOL has very simple rules for determining if an operator is binary or unary:

1. Unary operators are placed immediately to the left of their operand. No blank or tab character may appear between operator and operand.
2. 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 SPITBOL. Most other languages make these *white-space* characters optional. Omitting the right-hand blank after a binary operator will produce a unary operator. While the statement may have proper syntax, it will very likely produce results you didn't expect.

Fortunately, blanks and binary operators quickly become a way of SPITBOL life, and after some initial forgetfulness there are few problems.

Binary operators

A complete list of SPITBOL's operators appears in Chapter 15, "Operators." Here we'll concern ourselves with those operators that are found in most other programming languages.

Assignment =

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, whose value then appears on the standard output file, typically your display screen).

As an extension to the SNOBOL4 language, SPITBOL allows multiple assignments within a statement. Assignments are performed from right to left:

```
OUTPUT = RESULT = 'No errors'
```

Assignments may also be embedded within other expressions:

```
OUTPUT = (RESULT = 10 * 6) / 15
```

stores 60 in RESULT and 4 in OUTPUT.

Arithmetic ^, *, /, +, −

These characters provide the arithmetic operations—exponentiation, multiplication, division, addition, and subtraction respectively. Each is assigned a priority, so SPITBOL knows which to perform first if m

You may use parentheses to change the order of operations. The operands may be integers or real numbers, or a mixture of both. If both operands are integers, the result will be an integer. If either operand is real, the result will be real. Division of an integer by another integer will produce a truncated integer result; any fractional part is discarded. Try the following:

```
?         OUTPUT = 3 - 6 + 2
-1
?= 2 * (10 + 4)
28
?= 7 / 4
1
?= 7. / 4
1.75
```

Note that 7 / 4 is the division of two integers, so the quotient is 1, an integer (no rounding occurs in integer division). With 7. / 4 there is a real number (7.) so the operation is performed with real values—and you get a real result, 1.75.

Here are some more operations to try:

```
?= 3 ^ 5                                        (exponentiation)
243
?= 3 ! 5
243
```

```
?= 3.4 ** 5
454.35424
?= 3.4 ^ 2.9
34.7767393
?= 10 / 2 * 5          (multiplication, then division)
1
?= (10 / 2) * 5
25
?= 1.0 / 0
Error #262, Division caused real overflow
?= 1 / 0
Error #14, Division caused integer overflow
```

As you see, there are several synonyms for exponentiation: the caret (^), the exclamation point (!), and the doubled asterisk (**). We prefer the caret, so that's what we will use from here on. (Back in the old days when the first versions of SNOBOL were developed, keypunch machines didn't have a caret. We kept the ! and ** so that old programs will run easily).

When the same operator occurs more than once in an expression, which one should be performed first? The governing principle is called *associativity*, and operators are classified as being either *left-associative* or *right-associative*. 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:

```
?= 24 / 4 / 2
3
?= 24 / (4 / 2)
12
?= 2 ^ 2 ^ 3
256
?= (2 ^ 2) ^ 3
64
```

Here's the first bit of SPITBOL 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 SPITBOL language; the system tries to convert the operand to a suitable data type. Consider these statements:

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

SPITBOL detects the addition of an integer and a string, and tries to convert the string to a numeric value; it will ignore leading and trailing spaces and tabs. In both cases, SPITBOL was able to, so the *integers* 14 and 54 are added together. If the characters in the string are not acceptable integers or real numbers, SPITBOL will produce an error message.

SPITBOL will convert the null string to integer 0. Observe:

```
?= 14 + ' '
14
?= 14 + ''
```

```
14
?= 'A' + 1
Error #1, Addition left operand is not numeric
?= 14 + ' -54 '
-40
?= 14 + ' -  54 '
Error #2, Addition right operand is not numeric
```

The last statement produced an error because the blank was in the middle of the string. When converting strings to numerics, only leading or trailing blanks are ignored.

Concatenation 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 is 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.spt**:

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

The string resulting from concatenation can not be longer than the maximum allowable string size. If the limit is 4,194,304, you cannot concatenate two 2,500,000 character strings, because the 5,000,000-character result would be too large.

Concatenating two 5,000 character strings would exceed SPITBOL-8088's 9,000-character maximum string length.

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

```
?= 'Fourscore and ' (1863 - 1776 - 4 * 20) ' years'
Fourscore and 7 years
?= 19 (2. / 3.)
190.666666667
```

In the first case, the concatenation has three operands: the string 'Fourscore and ', the expression (1863 - 1776 - 4 * 20), and the string ' years'. SPITBOL evaluates the expression, converts the result to the string '7', and produces the result, 'Fourscore and 7 years'.

In the second example, the integer and real operands are converted to the strings '19' and '0.666666667', to produce the result string '190.666666667'. This is not exactly good math, but it is correct concatenation.

The last example also highlights a problem with SPITBOL. If you accidentally omit an operator, SPITBOL will think you intended to perform concatenation. In the example above, suppose we omitted a minus sign and had really meant to say:

```
?= 19 - (2. / 3.)
18.3333333
```

It is always possible for concatenation to automatically convert a number to a string. But there is one important exception when SPITBOL *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 we tried:

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

the result is the *integer* 3. It looks the same when printed, but internally strings and integers are stored differently. You'll find you'll use this aspect of null string concatenations extensively in your SPITBOL programming.

Before we proceed, let's think about the null string one more time, and the earlier statement identifying it 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 when beginning to add a series of numbers, the null string can serve as the starting place for concatenating a series of strings.

Unary operators

There aren't many interesting unary operators at this point in your tour of SPITBOL. 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.

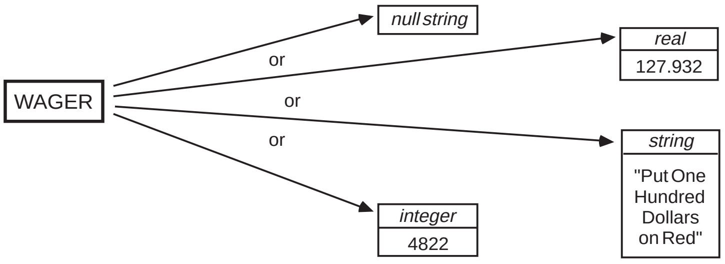
Arithmetic +, –

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, SPITBOL 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.spt**:

```
?= -(3 * 5)
-15
?= +'654321.123456'
654321.123
?= +''
0
```

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 SPITBOL is more flexible. A variable's contents may change repeatedly during program execution. In the figure below, variable WAGER might contain an integer, then a character string, then a real number, then the null string; in fact, *any* SPITBOL data type.



This can happen in SPITBOL because individual data are stored in a small packet of memory that contains both the data value, and a marker that remembers the data's type. Variables do not contain data values themselves, merely pointers to where the data packets reside in memory. However in practice, we usually ignore the underlying pointer, and speak instead of a variable's "contents," or say that "variable WAGER contains 4822."

Variable names

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

1. The name must begin with an upper- or lower-case letter.
2. If it is more than one character long, the remaining characters may be any combination of letters, numbers, or the characters period (.) and underscore (_).
3. The name must fit on a single program line. The maximum program line length is 1,024 characters. Longer names, up to the maximum allowable string length, may be constructed using the method described on page .

Here are some correct SPITBOL names:

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

Normally, SPITBOL performs *case-folding* on names. This means that lower-case alphabetic characters are changed to upper-case when they appear in names, so that Buffer, buFFer, and BUFFER are all equivalent. Naturally, case-folding of data does not occur within a string literal. (Case-folding of names can be disabled by using the –f command line option.)

In some languages, the initial value of a new variable is undefined. SPITBOL 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 re-executed.

Using variables

You store something in a variable by assigning a value to it, usually by placing it on the left side of the assignment operator. You can retrieve its contents simply by using it wherever its value is needed. Using a variable's value is nondestructive; the data packet it points to remains unchanged. Try creating some variables using **code.spt**:

```
?         ABC = 'EGG'
?= ABC
EGG
?         D = 'SHELL'
?= abc d                                      (Same as ABC D)
EGGSHELL
?= NONESUCH       (New variable is null)

?= ABC NULL D
EGGSHELL
?         N1 = 43.9
?         D = 17
?         OUTPUT = N1 + D
60.9
?         output = ABC D
EGG17
```

OUTPUT is a variable with special properties; when a value is stored in it, it is also displayed on your screen. There is a corresponding variable named INPUT, which reads data from your keyboard. It has no permanent value. Whenever SPITBOL 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
```

Generally, SPITBOL variables are global in scope—any variable may be referenced anywhere in the program. In chapter 8, we'll show how variables can be made private to a program section under certain circumstances.

Chapter Summary

Data types

INTEGER	Range: −2147483648 to +2147483647 (for 8088 **spitbols.exe**: −32768 to +32767)
REAL	Range: ±0.18E+309 to ±0.49E−322 and 0.0.
STRING	"chars" or 'chars'

Unary operators

+	Plus
−	Minus

Binary operators

^	Exponentiation
*	Multiplication
/	Division
+	Addition
−	Subtraction
blank	Concatenation
=	Assignment
tab	Concatenation
**	Exponentiation
!	Exponentiation

Variables

- No type declaration: variable's type may change during execution
- Name's first character must be a letter; remaining characters may be letters, numbers, period (.) or underscore (_)
- Maximum name length limited by program statement line length, 1,024 characters.
- Global in scope

Cautions

Single quote (') should not be confused with the grave accent mark (‘) which appears under the tilde on most keyboards. The grave accent may not be used as a string delimiter.

For real numbers, SPITBOL provides full 64-bit accuracy on quantities as small as ±0.22E−309. From there to ±0.44E−322, there is a gradual loss of precision as the number is denormalized. Smaller quantities are converted to 0.0.

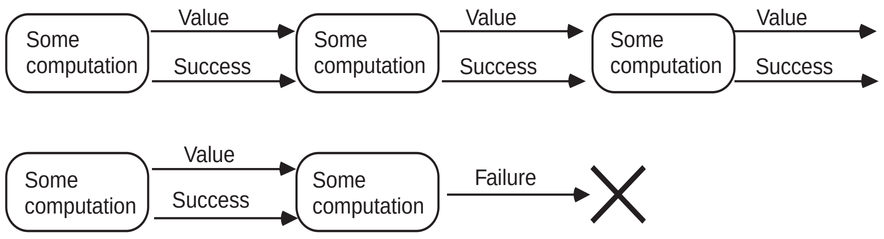
Chapter 4

Control Flow and Functions

Success and Failure

Success and failure are as important in SPITBOL as they are in life. Success and failure are unmistakable signals; something either worked, or it didn't. One reason that a short SPITBOL program can do more than a long program in another language is because SPITBOL recognizes that there's a fundamental difference between a data value and a signal.

The elements of a statement provide values and signals as computation proceeds. SPITBOL accumulates both, and stops executing a particular statement when it finds it cannot succeed. A mechanism is provided to alter program flow based upon these signals.



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 *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 system 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.

If you enter an End-of-File from the keyboard, we can easily demonstrate this type of failure. You do this in different ways on different systems: control-Z or function key F6 under MS-DOS, Windows and OS/2, control-D on most Unix systems. Typographically, we'll use the notation <EOF> in this manual. When you see it, simply enter the appropriate control character.

As you've noticed, the **code.spt** program reports the success or failure of each statement. So far, all examples have succeeded. Now try this one:

```
?= INPUT
<EOF>                   (Enter control-D or control-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 SPITBOL statements are constructed.

A SPITBOL Statement

In general, a SPITBOL statement looks like this:

```
Label    Statement body
:Goto
```

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

Label field

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

Labels provide names for statements. They are analogous to BASIC's line numbers, but SPITBOL's labels are optional. If present, a label must begin in the first character position of a statement, and must start with a letter or number. Additional characters may be anything except blank or tab. Like variable names, lower-case letters are equivalent to upper-case when case-folding is used (the default).

Goto field

Transfer of control is made possible by the *Goto*. It interrupts the normal sequential execution of statements by telling SPITBOL 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 (space or tab) is required before the colon. Label is the name given 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 to the referenced statement 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
:S(COPY)
```

Now let's write a short program which copies keyboard input to the screen, and reports the total number of lines. First stop **code.spt** by typing <EOF>.

Use your text editor to create a file called **linesum.spt**. Here's what you should enter into it:

```
         N = 0
COPY     OUTPUT = INPUT
:F(DONE)
         N = N + 1
         :(COPY)
DONE     OUTPUT = “PROGRAM COPIED ‘  N  ‘ LINES.’
END
```

Once you've typed that in, save the file and exit from your text editor. At the system command prompt, compile and run the program with:

```
spitbol linesum
```

If you typed the above precisely, then moments later, you will see a message like this:

```
DONE     OUTPUT = “PROGRAM COPIED ‘  N  ‘ LINES.’
                  !
linesum.spt(4,14) : Error 232 — Syntax error: Unmatched string quote
```

Well, you've made a mistake. (Actually, you didn't. You've been following our directions. But SPITBOL thinks you made a mistake, even if we know better.)

The error message gives the name of the program file, followed by the line number and character position of the error, and then an explanation of the error. The exclamation point is positioned at the place where SPITBOL first encountered a problem.

Note that the error is in line 4, and reload your text editor with **linesum.spt**. Look at line 4:

```
DONE    OUTPUT = “PROGRAM COPIED ‘  N  ‘ LINES.’
```

All the strings are framed by single quote marks, except right there at the start. So replace the double-quote with a single, save the file, and try compiling and executing it. If all is in order (no error messages), type some of the following test lines, and watch the result.

```
TYPE IN A TEST LINE
TYPE IN A TEST LINE
AND ANOTHER
AND ANOTHER
<EOF>
PROGRAM COPIED 2 LINES.
```

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 SPITBOL 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.

In this example, INPUT reads from the keyboard and OUTPUT writes to the screen. More generally, INPUT reads from your system's *standard input* file, while OUTPUT writes to the *standard output* file. The standard I/O files are associated with the keyboard and screen by default. However, by using the command line redirection characters < and >, INPUT and OUTPUT can read and write any system file. Try this example, using the test file **faustus** provided with SPITBOL:

```
spitbol linesum <faustus >null                    (or /dev/null under Unix)
PROGRAM COPIED 20 LINES.
```

Now INPUT reads lines from **faustus**, and discards the copied data by writing to the null device. The number of lines copied is still reported. Redirecting input and output is discussed more fully on page 165.

To resume use of **code.spt**, which you'll soon need, load it from the system command prompt with:

```
spitbol code
```

Built-in Functions

A *function* is analogous to an operator; it operates upon data to produce a result. The data objects are called the *arguments* of the function. The result produced—the *function of the arguments*—has 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(ARG1, ARG2, ..., ARGn)
```

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, SPITBOL will supply null strings in their place. Some functions, such as one that produces the current date, have no arguments at all.

SPITBOL provides a large number of predefined functions, and allows you to define your own. The large repertoire of built-in functions makes SPITBOL programming easier. Most functions are concerned with pattern matching, input/output, and advanced features of the language.

You needn't memorize all the functions, or feel overwhelmed by them. They are catalogued alphabetically in Chapter 19, "SPITBOL Functions." 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.spt**.

Conditional functions

These functions fail or succeed depending on the value of the arguments. They are sometimes called *predicate functions* because the success of an expression using them is *predicated* upon their success. If they succeed, they produce the null string as their value.

IDENT(S,T)

Succeed if 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)

Succeed if S and T are different. DIFFER is the opposite of IDENT. When used with one argument, function DIFFER(S) succeeds if S is *not* the null string.

EQ(X,Y)

Succeed if the numeric values X and Y are equal. X and Y must be integer or real numbers, or strings which can be converted to them. Because the null string is provided for omitted arguments, and null strings are converted to 0 or 0.0 as needed, the statement:

```
EQ(X)
```

is equivalent to

```
          EQ(X, 0)
```

and succeeds if X is 0 or 0.0.

NE(X,Y) Succeed if numerics X and Y are not equal.

GE(X,Y) Succeed if numeric X is greater than or equal to Y.

GT(X,Y) Succeed if numeric X is greater than Y.

LE(X,Y) Succeed if numeric X is less than or equal to Y.

LT(X,Y) Succeed if numeric X is less than Y.

INTEGER(X) Succeed if X is an integer or a string which can be converted to an integer. It fails if X is a real number or any string that is not a simple integer.

LGT(S,T) Succeed if string S is lexically greater than string T using a character-by-character comparison.

Leading blanks may be used in front of an argument for readability. In this case, blanks are not significant, nor do they specify concatenation. Here are some exercises for **code.spt**:

```
?       N = 3
?       EQ(N, 3)
Success
?       IDENT(N, 3)
Success
?       EQ(3.0, 3)
Success
?       IDENT(3.0, 3)
(real and integer)
Failure
?       EQ(N, 4)
Failure
?       NE(N, 4)
Success
?       INTEGER(N)
Success
?       INTEGER(47.3)
Failure
?       INTEGER('47')
Success
?       IDENT('ABC', 'abc')
Failure
?       DIFFER(3, '3')
(integer and string)
Success
?       IDENT('a' 'b' 'c', 'abc')
```

```
Success
?          LGT('ABC', 'ABD')
Failure
```

When any of these functions succeed, they produce a null string value. As 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 in line. However, if N is less than 10, 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 the value of 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 and natural 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, and fails otherwise.

Chapter 7, "Additional Operators and Datatypes," explains a SPITBOL extension to create expressions which succeed if *any* component succeeds. See "Alternative Evaluation" at the end of that chapter.

Numeric functions

Although the SNOBOL4 languages are renowned for their string-handling abilities, SPITBOL also includes many numeric functions.

These functions always succeed. They have numeric argument(s), and produce a numeric value instead of the null string. All trigonometric functions expect arguments in radians, and logarithmic functions use natural logarithms with base *e*: 2.718281…

ATAN(X)

ATAN(X) produces the arctangent of X. The result is in radians.

CHOP(X)

CHOP(X) discards the fractional part of real number X.

COS(X)

COS(X) produces the cosine of X; X is in radians.

EXP(X)

EXP(X) raises the natural logarithm base e to the power X.

LN(X)

LN(X) produces the natural logarithm of X.

REMDR(X,Y)

REMDR(X,Y) produces the remainder (modulus) of X divided by Y.

SIN(X)

SIN(X) produces the sine of X; X is in radians.

TAN(X)

TAN(X) produces the tangent of X; X is in radians.

You might want to experiment with these with **code.spt**:

```
?         PI = 3.14159265358979
?= CHOP(PI)
3.
?= ANGLE = PI / 6
0.523598776
?= SIN(ANGLE)
0.5
?= COS(ANGLE)
0.866025404
?         TANGENT = TAN(ANGLE)
?= TANGENT
0.577350269
?= ATAN(TANGENT)
0.523598776
?= EXP(1)
2.71828183
?= LN(10)
2.30258509
?= REMDR(10, 3)
1
?= REMDR(PI, 2.2)
0.941592654
?= REMDR(PI, CHOP(PI))
0.141592654
```

String functions

These functions also always succeed; all but SIZE produce a string result.

DATE()

DATE() produces current date and time as a string, such as

```
'09/15/95 16:19:48'
```

DUPL(S,N)

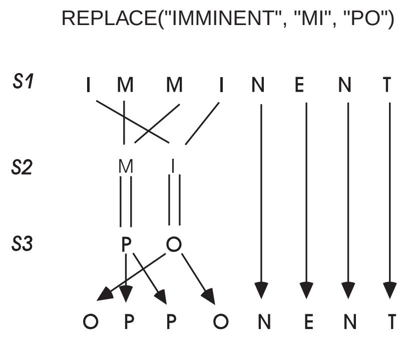
DUPL(S,N) duplicates string S, N times.

REVERSE(S)

REVERSE(S) produces string S in reverse order of characters.

REPLACE (S1,S2,S3)

REPLACE(S1,S2,S3) produces a modified copy of 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. In the following example, all occurrences of the letter "M" are replaced by "P", and "I" is replaced by "O".



Letters in S1 that do not appear in S2 are passed through unchanged. S2 and S3 must be the same length. See page 236 for an example of how this function may also be used for *transposing* characters within a string.

REPLACE never fails, even if no replacements are made.

SIZE(S)

SIZE(S) produces the number of characters in string S. It produces 0 for the null string.

TRIM(S)

TRIM(S) produces string S with trailing blanks and tabs removed.

Exercises for **code.spt**:

```
?= 'THE DATE AND TIME ARE: ' DATE()
THE DATE AND TIME ARE: 09/15/95 16:19:48
?= DUPL('ABC', 14)
ABCABCABCABCABCABCABCABCABCABCABCABCABCABC
?= REVERSE('OSCAR')
RACSO
?= REVERSE(DUPL('CAT',3))
TACTACTAC
?= REVERSE(REVERSE(DATE()))
09/15/95 16:22:36
?= SIZE('ZIPPY')
5
?= SIZE('')
0
?= TRIM('TRAILING BLANKS      ') 'GONE'
TRAILING BLANKSGONE
?= REPLACE('dromedary','ed','ED')
DromEDary
?= REPLACE('seven','ns','l')
Error #171, Null or unequally long 2nd, 3rd args to REPLACE
?= REPLACE('seven','ns','ll')
level
?= REPLACE('iridology','yglodri','nocipus')
suspicion
```

Chapter Summary

Success and failure

- Success has value result associated with it
- Failure has no associated value

Statement labels

- Must begin in first character position of statement
- Label's first character must be letter or number
- Remaining characters may be any characters except blank and tab

Control transfer

:(label)	Unconditional transfer to label
:S(label)	Transfer if statement succeeds
:F(label)	Transfer if statement fails
:S(label1)F(label2)	To label1 if success, to label2 if failure

Conditional functions

	Succeed if:
DIFFER(S,T)	S and T are different
EQ(X,Y)	Numbers X and Y are equal
GE(X,Y)	Number X is greater than or equal to Y
GT(X,Y)	Number X is greater than Y
IDENT(S,T)	S and T are identical
INTEGER(X)	X is an integer or integer in string form
LE(X,Y)	Number X is less than or equal to Y
LGT(S,T)	String S lexically greater than string T
LT(X,Y)	Number X is less than Y
NE(X,Y)	Numbers X and Y are not equal

Numeric functions

- Accept and produce numeric argument
- Always succeed
- Logarithmic functions use natural logarithms; trigonometric functions are in radians
- All can accept reals or integers
- All produce reals, except for REMDR, which produces an integer if both arguments are integer.

ATAN(X)	produces the arctangent of X.
CHOP(X)	discards the fractional part of real number X.
COS(X)	produces the cosine of X; X is in radians.
EXP(X)	raises the natural logarithm *e* to the power X.
LN(X)	produces the natural logarithm of X.

REMDR(X,Y)	produces the remainder (modulus) of X divided by Y.
SIN(X)	produces the sine of X; X is in radians.
TAN(X)	produces the tangent of X; X is in radians.

String functions

DATE()	Produces current date and time as a string
DUPL(S,N)	Duplicates string S, N times
REPLACE(S1,S2,S3)	Replace S2's characters in S1 by corresponding characters from S3
REVERSE(S)	Produces the reverse of string S
SIZE(S)	Number of characters in string S
TRIM(S)	Produces string S with trailing blanks removed

Chapter 5
Input/Output and Keywords

Input/Output

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

The number of files with which SPITBOL can communicate at once is operating-system dependent. For example with MS-DOS, the number is limited by the FILES=n command in file **config.sys**.

Each file is identified by a *channel*. A channel can be a name or a number. What you pick has no special significance; channels are what SPITBOL uses internally to distinguish various input and output paths. You also use the channel for special operations, such as rewinding a file, or telling SPITBOL that you are finished with a file.

Before you can perform any I/O, you must *associate* a variable with a channel and a direction. When a statement tries to use the variable's value, a line is read from the associated file or device. When a value is assigned to the variable, a line is written to the associated file or device.

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

SPITBOL's input and output are powerful and flexible; they can also be confusing. You can get quite bewildered if you don't keep in mind that there's a difference between the *variables* INPUT and OUTPUT, and the *functions* INPUT() and OUTPUT().

When you start up SPITBOL, the INPUT and OUTPUT variables are assigned so that INPUT reads from the keyboard, and OUTPUT writes to the screen. SPITBOL also has a special variable for the keyboard-screen combination, TERMINAL.

Thus in the default case, whenever you say something like:

```
                LINE = INPUT
                OUTPUT = SIZE(LINE) ‘ ‘ LINE
```

it works just the same as if you had written

```
                LINE = TERMINAL
                TERMINAL = SIZE(LINE) ‘ ‘ LINE
```

In both cases, you are telling SPITBOL to read a line from the keyboard, and then write its length, followed by a space and the line, to the screen.

However, TERMINAL *always* refers to the same things — the keyboard for input and the screen for output. The variables named INPUT and OUTPUT can refer to the keyboard and screen, but they can also refer to files or devices if you use command-line redirection.

To illustrate the possibilities, you should try the following examples. If you're running the **code.spt** program, exit it by typing <EOF> (control-D or control-Z, depending on your system) and load your text editor. Create and save a new file which we'll call **in-out.spt**.

```
* program in-out.spt
READ    LINE = INPUT
        :F(END)
        OUTPUT = SIZE(LINE) ‘ ‘ LINE                    :(READ)
END
```

Note that the line with the "*" in the first column is a comment, in this case, the name of the program. It's ignored by SPITBOL.

Default input

From the command prompt, compile and execute this program with

```
spitbol in-out
```

Since the INPUT and OUTPUT associations have not been changed, **in-out.spt** will look for its input from the default place, the keyboard, and send output to the screen.

Type in a few lines and see what happens:

```
The easy snowball jumped over the quick lazy saxophonist.
57 The easy snowball jumped over the quick lazy saxophonist.
Pack my box with five dozen liquor jugs.
40 Pack my box with five dozen liquor jugs.
<EOF>
```

In the statement labeled READ, SPITBOL fetched what it could from INPUT — the keyboard. Input from the keyboard stops when you press the Return or Enter key, and whatever you typed was assigned to LINE.

Then SPITBOL took LINE, determined its size, and concatenated that number with a space and LINE, and assigned all that to OUTPUT — the screen. The Goto at the end told SPITBOL to go back to READ to fetch more data.

When you entered the <EOF>, that told SPITBOL that you were done supplying data. So accessing the INPUT variable failed, and control passed to the END statement, which terminated **in-out.spt**.

Redirection

The operating system allow you to *redirect* standard input and standard output. You can take advantage of these facilities to alter the source and destination for program input and output.

Make sure you have the file **faustus** in your current subdirectory. We've included it just for testing input and output. The text is the famous speech from Act 5, Scene 2 of Christopher Marlowe's play, and it has several signal virtues: It is pretty poetry, it is a convenient length, and it is out of copyright, since it was written in 1589.

Now, we'll use redirection to make INPUT read from a disk file. On the command line enter

```
spitbol in-out <faustus
```

We'll see this on the screen:

```
63 Was this the face that launched a thousand ships,
64 And burnt the topless towers of Ilium?
42 Sweet Helen, make me immortal with a kiss.
…
39 And none but thou shalt be my paramour.
```

By using the input redirection operator <, we have told the system to supply lines from **faustus** instead of the keyboard when our program accesses the INPUT variable. Whether from the keyboard or a file, the INPUT variable is said to read from the "standard input" file.

OUTPUT variable

Naturally, you can perform analogous tasks with program output—send it to a disk file or a device instead of the screen.

The redirection operator for output is >. It looks like this:

```
spitbol in-out >helen
```

If you run this, whatever you type at the keyboard will be sent to a disk file named **helen**. (Again, you terminate keyboard input, and thus the program, by entering <EOF>). Once you've terminated the program, you can examine **helen** with your text editor or by using the TYPE or Unix cat command.

To use redirection with files for both input and output, try this:

```
spitbol in-out <faustus >helen
```

This reads from the **faustus** file, and writes to the file **helen**. If there is no file **helen**, the system creates one; if it does exist, it will be overwritten.

Naturally, you can use redirection to send OUTPUT to a device, rather than a file. Using MS-DOS, you could send output to the printer with:

```
spitbol in-out <faustus >prn:
```

Now we're going to use **in-out.spt** to explore the INPUT and OUTPUT functions.

The INPUT and OUTPUT Functions

All of the previous examples were constructed in terms of the variables INPUT, OUTPUT, and TERMINAL. Furthermore, the file being accessed was specified on the command line.

Often, you'll want to use other variable names to perform I/O, or handle more than two files at once, or be able to specify the names of files or devices from within your program. The INPUT and OUTPUT function calls allow you to do just that. These are *functions*, and are not to be confused with the INPUT and OUTPUT variables, even though the names are the same.

SPITBOL knows you are referring to functions here because the word INPUT or OUTPUT is followed by a parenthesized list of arguments.

INPUT function

We'll start working with the INPUT function. The formal syntax of the INPUT function looks like this:

```
INPUT("Variable", Channel, "Filename[options]")
```

where Variable is the name you want associated with each line or record of input. We'll stick to lines here—each time Variable is accessed in a statement, everything up to the next carriage return will be assigned to Variable.

Channel can be any legal variable name, as well as a number. That is, 1 and "mx37" are acceptable as channel names. If you're not using a number the name must be provided in quotes. In other versions of SNOBOL4, Channel must be a small integer. So we'll use that method here, and it's a good policy to follow, so that your programs will be portable.

Filename is the name of a file or device. Any processing options must be enclosed in square brackets. These options are primarily used for handling binary files, as opposed to text files, or files with unusually long records. Options are also used to specify that a file will be accessed for both input and output (update mode). Since we're working with just text files here, we'll recall that options are, for us, optional.

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

Now, revise program **in-out.spt** to use the INPUT function to read from **faustus**. Make it read like this:

```
* program in-out.spt
        INPUT('POETRY', 1, 'faustus')
:F(ERR)
READ    LINE = POETRY
        :F(END)
        OUTPUT = SIZE(LINE) ' ' LINE
:(READ)
```

```
ERR     TERMINAL = "Couldn't open file 'faustus'"
END
```

But before we run it, let's anticipate what will happen. We have used the INPUT function with its three arguments. The first one, 'POETRY', specifies the variable that the data from the file will appear in. The next argument, 1, is the channel. The last argument, 'faustus', is the name of the file we want to read from.

In the core of the program, material read in from POETRY is assigned to LINE. When that fails, control goes to END. When it doesn't fail, the OUTPUT displays the lines we're used to seeing. Compile and run the revised **in-out.spt**, and watch the output appear on the screen.

Now, let's make one minor change. Revise the INPUT function in **in-out.spt** to read like this:

```
        INPUT(.POETRY, 1, 'faustus')
:F(ERR)
```

In this minor revision, we used the unary name operator (.) to name the variable used by the INPUT function. Using .POETRY works exactly the same here as 'POETRY', and is slightly faster (this is covered in more detail in Chapter 7, "Additional Operators and Datatypes").

Notice that the third argument to the INPUT function is the string 'faustus'. What would happen if instead we had written:

```
        INPUT(.POETRY, 1, faustus)                    :F(ERR)
```

In this example, faustus would be seen as the name of a variable whose *contents* should be the file name. However, in this program, variable faustus will contain the null string, not a file name. Here's an example of how this could be written using a variable:

```
        FILE = 'faustus'
        INPUT(.POETRY, 1, FILE)                       :F(ERR)
```

OUTPUT function

The OUTPUT function is used to associate a variable with a file or device that will be written to. OUTPUT's usage is similar to the INPUT function, and has the same syntax. It fails if the file cannot be created or opened, perhaps because of a write-protected or full disk, or because it is in use by another program. Let's try it first by writing to a file that we specify in the program. Revise **in-out.spt** to look like this:

```
* program in-out.spt
        INPUT(.POETRY, 1, 'faustus')
:F(ERR1)
        OUTPUT(.MIGHTY, 2, 'marlowe')
:F(ERR2)
READ    LINE = POETRY
        :F(END)
        MIGHTY = SIZE(LINE) ' ' LINE
:(READ)

ERR1    TERMINAL = "Couldn't open file 'faustus'"     :(END)
```

```
ERR2    TERMINAL = “Couldn’t open file ‘marlowe’”       :(END)
END
```

We know what the INPUT function is doing here. OUTPUT will take whatever is assigned to the variable MIGHTY on channel 2, and send it to the file **marlowe**.

SPITBOL-8088's default maximum record length is 512 characters.

You now know the fundamentals of SPITBOL's input and output; what you've covered may well include everything you'll ever need to do with SPITBOL input and output. There's more, of course, but that's for specialized purposes. What you've done will handle 99 percent of all the text input and output you'll need SPITBOL to do.

When you're reading and writing in the line mode, as we have been here, the maximum length of a single line (everything up to the end-of-line character(s)) is 1,024 characters. Extra characters are discarded—the line is truncated. The end-of-line character (carriage return and/or linefeed) is not included in what you get. Longer lines and other end-of-line characters can be handled by specifying options following the filename.

I/O options

We mentioned earlier that the file name argument can be followed by processing options enclosed within square brackets. Options allow you to change the file's characteristics, for example, the maximum line length. That might be desirable if you were reading text from a file prepared with a word processor, where each paragraph might appear as a single long line.

Suppose some paragraphs might contain more than 1,024 characters, and that each appears as a single line of input. The –L option would allow SPITBOL to accept longer lines of input, say 4,000 characters. We do so in the INPUT function:

```
INPUT(.FILE, 4, ‘Textfile[–L4000]’)
```

Now when we use variable FILE to read data, SPITBOL will read lines of up to 4,000 characters in length. Notice that the bracketed options string is appended directly to the file name. The option character can be upper or lower case. If the file name were being provided by a string contained in a variable, we would concatenate the option string to the file name string:

```
INFILE = ‘Textfile’
INPUT(.FILE, 4, INFILE ‘[-L4000]’)
```

A complete list of processing options is provided with the description of the INPUT function beginning on page 226.

Specifying files

You've seen how file names can be specified by command-line redirection, and directly within the INPUT and OUTPUT functions. There are other ways as well, offering increased flexibility.

If the file name argument is omitted, SPITBOL examines the command line for a file specified with the same channel number. Modify **in-out.spt** again as shown on the following page.

```
* program in-out.spt
          INPUT(.POETRY, 1)
          :F(ERR1)
          OUTPUT(.MIGHTY, 2)
          :F(ERR2)
READ      LINE = POETRY
          :F(END)
          MIGHTY = SIZE(LINE) ' ' LINE
:(READ)

ERR1      TERMINAL = "Couldn't open input file"          :(END)
ERR2      TERMINAL = "Couldn't open output file"
:(END)
END
```

and start the program like this:

```
spitbol –1=faustus –2=marlowe in-out
```

The INPUT and OUTPUT functions are missing their third argument, the file name, so SPITBOLlooks on the command line for a matching channel number, and uses the file name provided exactly as if it had been supplied in the function.

Chapter 13, "Running SPITBOL" explains still another way of doing this, using system environment variables, but this should provide plenty of flexibility for now.

Completing File Processing

ENDFILE function

When your program terminates, SPITBOL automatically writes out any partially-filled file buffers and closes all files. You can also do this yourself from the program by using the ENDFILE function.

ENDFILE takes one argument, and it is the channel associated with the file you want to close. For example, we could change **in-out.spt** to branch to label DONE when processing is comple:

```
* program in-out.spt
          INPUT(.POETRY, 1, 'faustus')
:F(ERR1)
          OUTPUT(.MIGHTY, 2, 'marlowe')
:F(ERR2)
READ      LINE = POETRY
          :F(DONE)
          MIGHTY = SIZE(LINE) ' ' LINE
:(READ)

ERR1      TERMINAL = "Couldn't open input file"          :(END)
ERR2      TERMINAL = "Couldn't open output file"
:(END)
```

```
DONE    ENDFILE(1)
        ENDFILE(2)
END
```

Most programs don't need to do this. But if you complete processing on an output file, and are then entering a time-consuming phase of your program, closing the output file will insure that no data is lost if there is a power outage or system failure.

Also, channel numbers cannot be reused unless the channel is closed. For example, we might read two files in our **in-out** program:

```
* program in-out.spt
        I = 1
        INPUT(.POETRY, 1, 'faustus')
:F(ERR1)
        OUTPUT(.MIGHTY, 2, 'marlowe')
:F(ERR2)
READ    LINE = POETRY
        :F(EOF)
        MIGHTY = SIZE(LINE) ' ' LINE
:(READ)

EOF     I = EQ(I, 1) I + 1
        :F(END)
        ENDFILE(1)
        INPUT(.POETRY, 1, 'edwardii')
:S(READ)
        TERMINAL = "Couldn't open file 'edwardii'")      :(END)

ERR1    TERMINAL = "Couldn't open file 'faustus'"        :(END)
ERR2    TERMINAL = "Couldn't open file 'marlowe'"        :(END)
END
```

DETACH function

Finally, we'll mention that it is possible to remove the I/O association between a variable and a channel, without closing or disturbing the channel or file in any way. DETACH takes one argument, and it is the name of a variable previously used in an INPUT or OUTPUT function. For example:

```
DETACH(.POETRY)
DETACH("MIGHTY")
```

After detaching a variable, referencing it or assigning to it has no affect on the file it was previously associated with.

Attaching to a channel

It's also possible to attach (that is, create a new association) more than one variable to an existing, open channel:

```
INPUT(.POETRY, 1, 'faustus')
INPUT(.VERSE, 1)
```

Both variables refer to the same file. There is only one current file position, and it belongs to the channel number. Reading from either variable changes the position in the file equivalently.

Keywords

Input and Output allow your program to communicate with the outside world. Your program may also communicate with the SPITBOL system itself. *Keywords* allow you to modify SPITBOL'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 variables. They either provide values or have values assigned to them. Numeric keywords are restricted to integer values.

&TRIM

Remove trailing blanks and tabs

The function TRIM(S) removes trailing blanks and tabs from the argument S. If S is an input associated variable, like TRIM(INPUT), lines read from the file wll be trimmed of trailing blanks and tabs. In this case, a faster and simpler method is to assign a non-zero integer to the keyword &TRIM (the default is 0).

When &TRIM = 0, any trailing blanks and tab characters are preserved on input lines. When &TRIM = 1 (or any other positive integer), SPITBOL removes trailing blanks and tabs. 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 SPITBOL programs.

&MAXLNGTH

Maximum string length

This keyword controls the maximum permissible string length. Its initial value is 4,194,304 (9,000 for SPITBOL-0800). You can set it to any number greater than 1,023, although there's usually no need to reduce the value of &MAXLNGTH from its initial value.

Besides limiting string size, keyword &MAXLNGTH also limits the size of other SPITBOL objects. Although they haven't been introduced yet, SPITBOL provides aggregate objects such as arrays and tables. The amount of memory used by any individual object is restricted to &MAXLNGTH bytes. So even if your string lengths are moderate, you may need to increase &MAXLNGTH if you need to use large arrays.

&MAXLNGTH is also the largest value to which other keywords (except &STLIMIT) can be set.

The largest value to which &MAXLNGTH may be set depends upon the amount of memory available, and upon the –m command line option described in Chapter 13, "Running SPITBOL." See "Caution" at the end of this chapter.

You can inspect the default value of this keyword by starting up **code.spt** and entering:

```
?        OUTPUT = &MAXLNGTH
4194304
```

The statement to change the value of &MAXLNGTH looks like this:

```
        &MAXLNGTH = 50000
```

&DUMP

Termination dump of variables

This keyword is useful for debugging programs because it tells SPITBOL to display the values of your variables when your program terminates. The information is displayed on the screen when your program terminates. It can also be sent to a file by using the –o=file command-line option described in Chapter 13, "Running SPITBOL."

When &DUMP = 0, the default, then these messages are suppressed.

If &DUMP = 1, the screen will show the last values for your variables, and these will be displayed in alphabetical order (a "partial dump"). Following that will be a listing of keyword values. Only variables with non-null values are displayed.

If &DUMP = 2, you'll get all of the preceding dump, plus the non-null contents of arrays, tables, and program-defined data types (a "full dump").

Finally, &DUMP = 3 adds statement labels and all null-valued variables and elements.

You might want to try various values of &DUMP in **in-out.spt** and run it just to see what happens. In such a simple program you won't see any difference between keyword values 1 and 2.

Here's the form of the statement:

```
        &DUMP = 1
```

A useful programming technique is to set &DUMP non-zero at the start of your program, and then set it to zero just before transferring or flowing into the END statement. That way, if the program stops with an unexpected error, the non-zero &DUMP keyword will display all program variables.

&ALPHABET

Complete character set

This keyword contains a 256-character string, the system's character set in ascending sequence. It is called a *protected* keyword because it cannot be modified by your program. In the next chapter, we'll use pattern-matching techniques to extract segments of this alphabet to obtain special characters.

You can see what's there with a statement like this:

```
        TERMINAL = &ALPHABET
```

&LCASE

All lowercase letters

This keyword contains the 26 lowercase alphabetic characters, in ascending sequence. It is equivalent to the string

```
        "abcdefghijklmnopqrstuvwxyz"
```

and is provided for convenience only.

&UCASE

All uppercase letters

This keyword contains the 26 uppercase alphabetic characters, and is equivalent to the string

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Programs Without Pattern Matching

You now have the ingredients to create some simple programs. However, if this were all of the SPITBOL 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—it will only take a few minutes to create and test each of these programs.

fileinfo.spt

Produce information about a file

This program counts the number of characters and lines in a file.

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

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

In such a small program, it's reasonable to use LINES and CHARS without initializing them. The first use of the statement:

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

converts LINES from the null string to an integer 0. We used the expression +LINES in the last statement to produce an integer 0 (instead of the null string), just in case the input file is empty.

The statement

```
        CHARS = CHARS + SIZE(INPUT)
```

demonstrates that when an input-associated variable appears as the argument to a function, an input operation occurs each time the function is called.

If you ran this with familiar **faustus** for input, as with:

```
spitbol fileinfo <faustus
```

you would see a result like this:

```
800 characters, 20 lines read.
```

triplet.spt

Simple text formatting

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 SPITBOL.

```
* Trim input, prepare to 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 zero 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. (We assume all input lines contain 80 or fewer characters.)

The last two statements break the file lines into triplets. Variable N cycles through values 0, 1, 2, 0, …. 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 statement terminates without performing the assignment.

To run this with a disk file for INPUT, remember to redirect standard input on the command line.

palin.spt

Palindromes

This program accepts an input line, and checks if it is a palindrome (a statement that reads the same forward and backward).

```
        &TRIM = 1
        TERMINAL = 'Enter test lines, terminate with EOF'

* Read input line, convert lower case to upper.
LOOP    S = REPLACE(TERMINAL, &LOWERS, &UPPERS)
:F(END)

* Check for palindrome:
        TERMINAL = IDENT(S, REVERSE(S)) 'Palindrome!' :S(LOOP)
        TERMINAL = 'No, try again.'
:(LOOP)
END
```

This program uses the REPLACE function to convert input lines to upper-case before testing. The statement:

```
LOOP    S = REPLACE(TERMINAL, &LOWERS, &UPPERS)
:F(END)
```

begins by evaluating the arguments for the REPLACE function. Retrieving the value of TERMINAL causes a line to be read, and trimmed of any trailing blanks. The remaining two arguments tell REPLACE to find all the lower-case characters in the line, and replace them with the corresponding upper-case characters. The result is returned as the value of the function, and assigned to variable S. As REPLACE never fails (even if no replacements are made), the failure Goto will only be used if variable TERMINAL received an End-of-File.

The testing for a palindrome and reporting success have been combined into one statement:

```
        TERMINAL = IDENT(S, REVERSE(S)) 'Palindrome!' :S(LOOP)
```

Function REVERSE creates a mirror-image of the test line, and function IDENT checks if this reversed line is the same as the original. If it is, IDENT's null value is concatenated with the literal 'Palindrome!', and assigned to TERMINAL. The Goto field detects success, and transfers to LOOP to read another line. If IDENT fails, execution continues with the next statement, and the user is told to try again.

The program accepts a very limited class of palindromes: their punctuation and word spacing must be the same in both directions. It succeeds for Napoleon's lament: "Able was I ere I saw Elba" but fails for others, such as: "A man, a plan, a canal, Panama!" We'll improve the program later, after we learn about pattern-matching.

temp.spt

Temperature conversion

This program asks the user for a low and high Fahrenheit temperature, and an increment. The values are specified as integers. It produces a list of temperatures in Fahrenheit and Celsius.

```
        &TRIM = 1
AGAIN   TERMINAL = 'Enter low temperature (F), or <EOF> to end:'
        LOW = TERMINAL
        :F(END)
        TERMINAL = 'Enter high temperature:'
        HIGH = TERMINAL
        :F(END)
        TERMINAL = 'Enter temperature step or Return for 1:'
        STEP = TERMINAL
        :F(END)

* If step omitted, default to 1:
* When the second argument is omitted, IDENT(STEP) asks whether
* STEP matches the null string. If not, we go to the next state-
* ment. If STEP is the null string, then the function succeeds,
* thereby producing the null string, which is concatenated with the
* following 1. The result is to assign 1 to STEP if STEP is null.
          STEP = IDENT(STEP) 1

* Check for valid input:
          (INTEGER(LOW) INTEGER(HIGH) INTEGER(STEP)
+          LT(LOW, HIGH) GT(STEP,0))
                        :S(GO)
          TERMINAL = 'Must be integers, low<high, step>0'
:(AGAIN)

* Produce results:
GO        TERMINAL = LOW ' ' (LOW - 32) * 5.0 / 9.0
          LOW = LT(LOW, HIGH) LOW + STEP      :S(GO)F(AGAIN)
END
```

The plus sign in column one marks a continuation statement—the full statement was too long to place on one line, so it was split into two parts. SPITBOL removes the plus sign when it combines the two lines.

The last program step is a loop test. The low temperature will be increased by the step value until it is greater than the high temperature.

Chapter Summary

Input/Output variables

INPUT	Read from standard input file, fails at End-of-File
OUTPUT	Write to standard output file
TERMINAL	Write to computer's screen, read from keyboard.

Input/Output functions

DETACH('variable')	Remove variable's I/O association
DETACH(.variable)	Remove variable's I/O association
ENDFILE(channel)	Close file associated with channel

INPUT('variable', channel, 'file[options]')
INPUT(.variable, channel, 'file[options]')
OUTPUT('variable', channel, 'file[options]')
OUTPUT(.variable, channel, 'file[options]')

Keywords

&ALPHABET	String of all 256 possible character values
&DUMP	Nonzero for termination display of variables
&LCASE	String of 26 lower-case letters
&MAXLNGTH	Maximum string length, initially 4,194,304 (9,000 in SPITBOL-8088)
&TRIM	Nonzero to remove trailing blanks or tabs on input
&UCASE	String of 26 upper-case letters

Caution

SPITBOL uses a fast garbage collector to reclaim unused memory. The collection algorithm* needs to distinguish between small integers and memory addresses. This effectively restricts the maximum size of a SPITBOL object (string, array, table header, code or expression block, integer keyword) to be less than &MAXLNGTH bytes. &MAXLNGTH in turn is limited to the value set with the –m command line option described in Chapter 13, "Running SPITBOL."

Another consequence of this algorithm is that the value specified for the maximum object size must be *numerically less than* the starting memory address of SPITBOL's work space. If it is not, SPITBOL ignores (and is not able to use) any memory between the low end of the work space and this value. For most users, the default value of 4 megabytes should pose little problem.

*For a description of the garbage collection algorithm, as well as the internal organization of SPITBOL, see reference 8 of the bibliography.

Chapter 6

Pattern Matching

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:

1. The subject contains several color names. The pattern is the string 'BLUE'. Does the subject string contain the word 'BLUE'?

2. The subject contains a nucleic acid (DNA) sequence. The pattern searches for a subsequence that also appears in mirror-image form.

3. The subject contains a paragraph of text. The pattern describes the spacing rules to be applied after punctuation. Does the subject string conform to the punctuation rules?

4. 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.

5. 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. SPITBOL 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, SPITBOL patterns can specify *context*; they may be qualified by what precedes or follows them, or by their position in the subject.

Knowns and unknowns

Patterns are composed of *known* and *unknown* components. Together, they specify a set of character strings to be recognized.

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.

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 SPITBOL always uses the first complete statement element as the subject. In the statement

```
X Y Z
```

the content of X is the subject, and the concatenation of variables Yand Z is the pattern. Whereas

```
(X Y) Z
```

indicates the subject is the concatenation of the strings in X and Y, while the pattern is in variable Z.

In SNOBOL4, a blank can signify a pattern match as well as concatenation, depending upon position. This sometimes leads to unintentional errors. So as an extension to the SNOBOL4 language, SPITBOL allows the use of the question mark to signify a pattern match. This has two advantages.

First, it makes programs more readable, and for that reason we will use it for the rest of the tutorial. Thus, the last two statements would be written as:

```
X ? Y Z
(X Y) ? Z
```

Second, it permits you to encapsulate one or more pattern matches into an expression. Consider the statement:

```
A = (X ? Y) (Q ? P)
```

The expression (X ? Y) applies the pattern in Y to the subject string in X. If the pattern match succeeds, the matching substring becomes the value of the first parenthesized expression. Similarly, pattern P is applied to Q. If both pattern matches succeed, the matching substrings are concatenated and assigned to variable A. If either fails, the statement fails, and assignment does not occur.

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 elements to be concatenated, they should be enclosed in parentheses.

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 variables, each containing a subpattern, the expression

```
P1 P2
```

is also a pattern. The subject must contain whatever P1 matches, immediately followed by whatever P2 matches. P2 is *subsequent* to 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 above 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
```

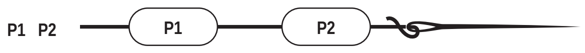
matches whatever P1 matches, *or* whatever P2 matches. SPITBOL tries the various alternatives from left to right.

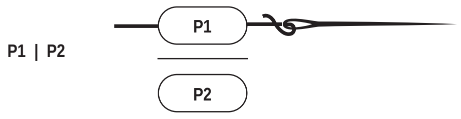
In *The SNOBOL4 Programming Language*, the authors develop the concept of "bead" diagrams. Here we'll only touch on it briefly, but you may find it useful as a mechanical way to sort out subsequents and alternates.

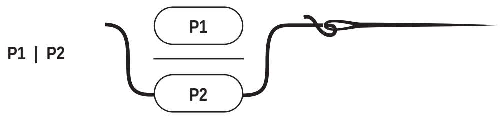
In this view, pattern matching is the process of attempting to pass a needle and thread through a collection of beads—the individual pattern components. Pattern subsequents are drawn side-by-side, left-to-right. Pattern alternates are stacked vertically, in columns, with a horizontal line between each alternative.

The needle passes through a pattern component if that component matches the next set of characters in the subject. A pattern match succeeds if the needle can reach the right side of the diagram by passing through each column of alternatives.

In the simplest cases, P1 P2 and P1 | P2, the diagrams look like this:







If the needle cannot pass through a component, it will try the next alternative in that column. So if P1 does not match, it will try P2, as in the third example above.

If none of the alternatives in a column match, the needle is pulled back to the previous column (if any), and other alternatives are tried there. Alternatives are tried from top-to-bottom in a column, corresponding to the left-to-right manner they are written in a program.

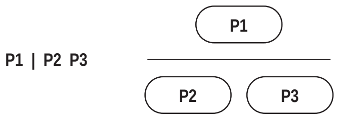
Patterns may contain both concatenation and alternation. Normally, concatenation is performed before alternation, so the pattern

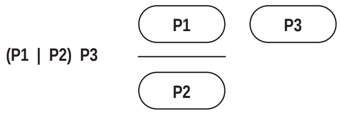
```
P1 | P2 P3
```

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. The diagrams for both cases are shown on the next page.



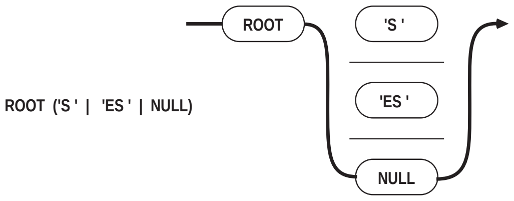


When a pattern successfully matches a portion of the subject, the matching subject characters are *bound* to it. The next pattern in the statement must match beginning with the very next subject character. If a subsequent fails to match, SPITBOL backtracks, unbinding patterns until another alternative can be tried. A pattern match fails when SPITBOL 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

```
PAT = 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 bead diagram looks like this, where NULL is a convenient equivalent for ''.

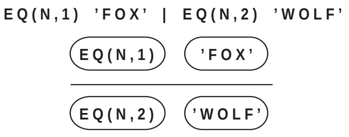


The conditional functions of Chapter 4, "Control Flow and Functions," may appear in patterns. If they fail when evaluated, the current alternative fails. If they succeed, they match the null string, and so do not bind any subject characters.

These functions 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'
```

The bead diagram for the previous pattern looks like this:

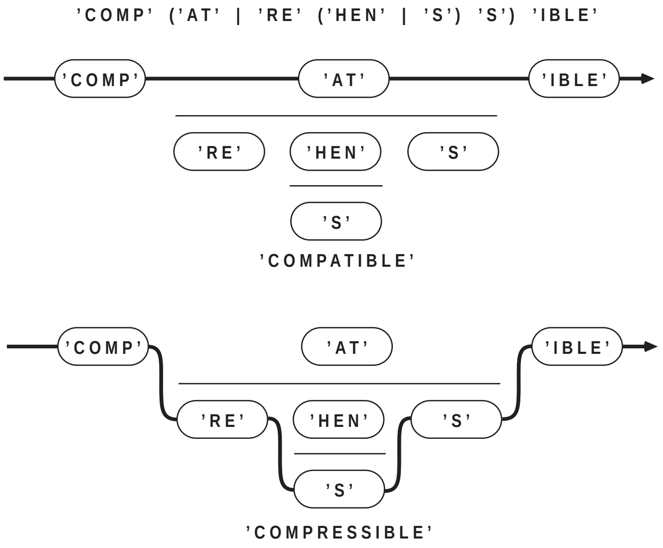


Pattern factoring

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'
```

You may get a better feel for factored patterns by viewing them as bead diagrams. The following example shows how the above pattern would be threaded for the subjects 'COMPATIBLE' and 'COMPRESSIBLE':



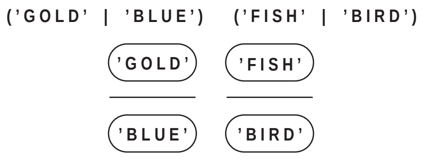
Simple Pattern Matches

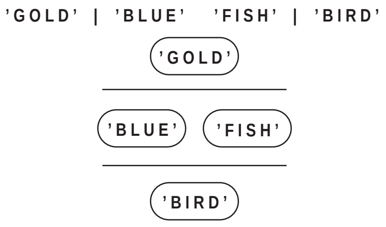
Here are examples of pattern matches using a string literal or a variable for the subject. The patterns consist entirely of known elements. Use the **code.spt** program to experiment with them:

```
?        '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, 'BLUE' and 'FISH' would be concatenated as subsequents, and the pattern would match 'GOLD', 'BLUEFISH', or 'BIRD'. Both combinations are shown in the bead diagrams below.





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 created and 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. It's as if the bead diagram were recorded in memory. COLOR now has the *pattern* data type. The preceding example can now be written as:

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

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.

Conditional assignment

The binary operator *conditional assignment* assigns the matching subject substring to a variable. The operator is called conditional, because assignment occurs *only* if the 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 which matched the subpatterns COLOR and CRITTER were assigned to variables SHADE and ANIMAL respectively. The statement

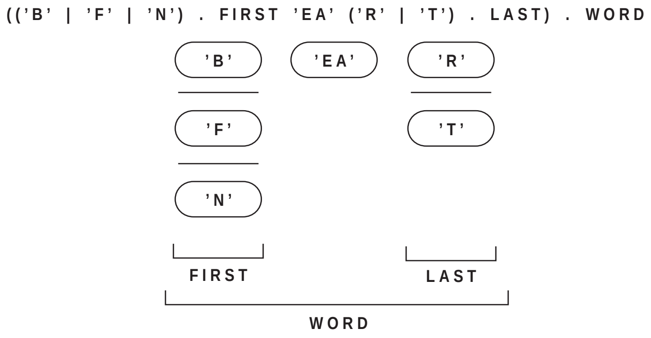
```
           BOTH = COLOR CRITTER
```

had to be re-executed 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 re-executed.

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 a conditional assignment. Try:

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

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.

Primitive patterns

There are seven primitive patterns built into the SPITBOL system. The two used most frequently will be discussed here. Chapter 9, "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.

word1.spt

Here’s a simple test program that demonstrates how knowns and unknowns combine to extract some simple information from our **faustus** data file.

```
* word1.spt
          PAT = “ the ” ARB . OUTPUT (“ of ” | “ a ”)
LOOP      LINE = INPUT                                  :F(END)
          LINE ? PAT                                    :(LOOP)
END
```

The pattern looks for the word “the” (notice the blanks before and after to make sure it’s a word) followed by either the word “of” or “a”. The characters between the words are bound by the ARB pattern, and assigned to the OUTPUT variable.

If you run the program with the command line:

```
spitbol word1.spt <faustus
```

you should see the following output:

```
face  that  launched
topless  towers
beauty
monarch
```

Why can’t the two-line heart of the program be collapsed into a single line like this?

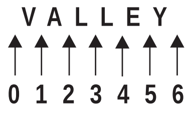
```
LOOP      INPUT  ?  PAT                                        :S(LOOP)
```

The reason is that there are two causes of failure in this statement. Failure could occur because the desired words aren’t found in the input line, or because the proram has encountered the end of the input file. In one case we want to keep reading additional lines looking for our pattern; in the other case we want to end the program.

Combining an input statement with a pattern match makes it impossible to distinguish the cause of failure.

Cursor position

During a pattern match, the *cursor* is SPITBOL’s pointer into the subject string. It is integer valued, and points *between* two subject characters. It may also may be positioned before the first subject character, or after the final subject character. Its value may never exceed the size of the subject string. Here’s an example of the numbering for the substring string ‘VALLEY’:



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 SPITBOL 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.

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

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 LEN with keyword &ALPHABET as the subject provides a simple way to obtain a string of unprintable characters. For example, the ASCII control characters occupy positions 0 through 31 in the 256-character ASCII set. To obtain a 32-character string containing these control codes, use:

```
?          &ALPHABET ? LEN(32) . CONTROLS
Success
```

When you're just working with a few characters, though, the easiest way to obtain a character from an ASCII numeric value is to use the built-in CHAR function. It produces the character which corresponds to the given number. That is, these to statements are equivalent:

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

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

```
?          CHR = 'A'
?          &ALPHABET ? @N CHR
?          OUTPUT = N
65
```

POS, RPOS

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 SPITBOL tries other pattern alternatives, perhaps extending a previous substring matched by ARB, or beginning the match further along in the subject.

Continuing with **code.spt**:

```
?          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
```

RTAB, TAB

RTAB(integer), TAB(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 . ST
```

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.

word2.spt

Pattern functions like LEN, POS, and TAB are useful when dealing with data arranged in columns. The file **treesort.in** in the **demos** directory contains some columnar data. Here are two lines from the file:

```
1896  MARCONI G :        RADIO
1609  GALILEO :         TELESCOPE
```

Consider a program that reads the file and breaks each line into its component fields:

```
* word2.spt
         PAT = POS(0) LEN(4) . WHEN
+                TAB(6) ARB . WHO " :"
+                TAB(24) REM . WHAT
LOOP     LINE = INPUT                                :F(END)
         LINE ? PAT                         :F(LOOP)
         OUTPUT = WHO " invented the " WHAT " in " WHEN
:(LOOP)
END
```

Running the program with redirected input gives output lines like this:

```
spitbol word2 <demos\treesort.in

MARCONI G invented the RADIO in 1896
GALILEO invented the TELESCOPE in 1609
```

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, NOTANY

ANY(string), NOTANY(string) — Match one character

Each function produces a pattern which matches exactly one character from 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 is immaterial.

BREAK
SPAN

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

These are multi-character versions of NOTANY and ANY. Each requires a non-null string 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 SPITBOL functions. Try some examples using **code.spt**:

```
?         '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))
```

word3.spt

Let's recast the previous **word2.spt** program to use the fact that there are two or more blanks between each field, and that ':' is also part of the field separator.

```
* word3.spt
          PAT = POS(0) BREAK(' ') . WHEN (' ' SPAN(' '))
+                 ARB . WHO (' ' SPAN(' :'))
+                 REM . WHAT
LOOP      LINE = INPUT                                :F(END)
          LINE ? PAT                          :F(LOOP)
          OUTPUT = WHO " invented the " WHAT " in " WHEN
:(LOOP)
END
```

Because SPAN(' ') will match one or more blank characters, (' ' SPAN(' ')) will match *two* or more blank characters.

Notice that we could not use the line:

```
+                 BREAK(' ') . WHO (' ' SPAN(' :'))
```

in the pattern to collect the inventor's name, because the names contain embedded blanks. That is, given a name like:

```
BELL A G
```

BREAK(' ') would match 'BELL', but then fail because 'BELL' is not immediately followed by a blank and a colon, or two or more blanks.

The BREAKX function described next solves this little problem.

BREAKX

BREAKX(string) — Extended BREAK function

SPITBOL offers an extended version of BREAK called BREAKX. If necessary, BREAKX will "look past" the place where it stopped to see if a longer match is possible. It will do this if some subsequent pattern element fails to match. The pattern matcher checks to see if extending BREAKX might allow the subsequent pattern element match. If so, the operation succeeds. If not, other pattern alternatives (if any) prior to BREAKX are attempted.

Suppose you want everything before the first 'E' in a subject string, as with:

```
?        'INTEGERS' ? BREAK('E') . OUTPUT
INT
```

BREAK works fine for that. But suppose we want whatever comes before the first occurrence of a two-letter pattern, 'ER'? That's where BREAKX is handy.

```
?        'INTEGERS' ? BREAKX('E') . OUTPUT 'ER'
INTEG
```

BREAKX stopped at the first E in INTEGERS, and tried to match the next pattern element, the two letters 'ER'. But the next subject characters were 'EG', a mismatch, so BREAKX was instructed to try again. BREAKX extended itself to the next E, where 'ER' in the subject matches 'ER' in the pattern.

The above example illustrates that BREAK(S) will never return a string containing any characters in S, while BREAKX(S) might, if a subsequent pattern requires it.

BREAKX(S) provides a more selective, and more efficient version of the ARB pattern. We could have used the construction:

```
?        'INTEGERS' ? ARB . OUTPUT 'ER'
INTEG
```

but ARB pokes along one character at time, matching 'I', 'IN', 'INT', and 'INTE', before finding the desired match, 'INTEG'. In contrast, BREAKX gets the right answer after only two attempts: 'INT' and 'INTEG'. The increased efficiency is even more pronounced with a long subject.

word4.spt

Consider how we might apply this to the previous word3.spt program:

```
* word4.spt
         PAT = POS(0) BREAK(' ') . WHEN (' ' SPAN(' '))
+                 BREAKX(' ') . WHO (' ' SPAN(' :'))
+                 REM . WHAT
LOOP     LINE = INPUT                                  :F(END)
         LINE ? PAT                            :F(LOOP)
         OUTPUT = WHO " invented the " WHAT " in " WHEN
:(LOOP)
END
```

Using BREAKX, we were able to replace the less efficient ARB pattern and skip over the blanks within the inventor's name.

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

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. Let's try a few examples with **code.spt**:

```
?        T = 'MUCH ADO ABOUT NOTHING'
?        T ? 'ADO' = 'FUSS'
Success
?= T
MUCH FUSS ABOUT NOTHING
?        T ? 'NOTHING' =
Success
?= T
MUCH FUSS ABOUT
?        'MASH' ? 'M' = 'B'
Error #212, Syntax error: Value used where name is required
```

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'
?= 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.spt** if the label and Goto are both on the same line. Continuing with the previous example:

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

We can combine replacement with the ANY function to improve upon our earlier palindrome checking program. We would like to remove all punctuation and blanks from the input line. We add one line after the statement labeled LOOP:

```
LOOP    S = REPLACE(INPUT, &LCASE, &UCASE)           :F(END)
PUNCT   S ? ANY(".,;:'?!– ") =
        :S(PUNCT)
…
```

Now we can now use input lines such as: A man, a plan, a canal, Panama!, or Poor Dan is in a droop.

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:

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

Finally, we note that SPITBOL allows you to place the subject, pattern, and replacement within parentheses and used as an expression. The value of the expression is the entire subject string *after* replacement occurs, or failure:

```
?= (RAINBOW ? CYCLE = REST ITEM ',')
YELLOW,GREEN,BLUE,VIOLET,RED,ORANGE,
?         (RAINBOW ? CYCLE = REST ITEM ',') ? BREAK(',') . OUT-
PUT
GREEN
```

Sample Programs

We've introduced a lot of concepts in this chapter; it's time to see how they fit together into programs. These programs should be created as text files (straight ASCII) with your text editor. It's easiest if you use the '.spt' extension when naming files with SPITBOL source code.

words.spt

Word Counting

The first program counts the number of words in the input file. Program lines with an asterisk in the first column are comment lines—their contents are ignored by SPITBOL.

```
*          Simple word counting program, words.spt.
*
*          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.
*
*         Input is read from the standard input file
*
          &TRIM = 1
          NUMERALS = '0123456789'
          WORD = "'–" NUMERALS &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.

To run the program and read data from file **faustus**, use file redirection on the command line:

```
spitbol  words  <faustus
```

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 typing 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.

cross.spt

Word Crossing

This program asks for two words, and displays all intersecting letters between them. A similar program is in Griswold [10].

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 SPITBOL, there are more comments than program statements:

```
* cross.spt – 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
```

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.spt** 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, SPITBOL 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 is 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 Summary

Pattern match statements

```
label    SUBJECT ? PATTERN                 :S(label1)F(label2)
label    SUBJECT ? PATTERN = REPLACEMENT   :S(label1)F(label2)
```

Data types	PATTERN	A structure containing pattern operators, primitive patterns, or pattern functions
Pattern operations	PAT1 PAT2	Subsequent, matches PAT1 followed by PAT2
	PAT1 | PAT2	Alternate, matches PAT1 or PAT2
	PATTERN . NAME	Conditional assignment of matched substring to variable
	@NAME	Copy cursor position to variable
Primitive patterns	ARB	Arbitrary number of characters
	REM	Remainder of subject
Pattern functions	LEN(I)	Match string of specified length
	POS(I)	Match null string at specified cursor position
	RPOS(I)	Match null string at specified cursor position counting from subject end
	RTAB(I)	Match to specified cursor position counting from end
	TAB(I)	Match to specified cursor position
	ANY(S)	Match one character if it appears in S
	BREAK(S)	Match run of characters up to a character in S
	BREAKX(S)	Like BREAK(S), but can extend past first match
	NOTANY(S)	Match one character if it is not in S
	SPAN(S)	Match one or more characters if they appear in S
Keywords	&ANCHOR	Nonzero requires match to include first subject character

Chapter 7
Additional Operators and Datatypes

In this chapter we will explore some additional SPITBOL operators and data types. Some concepts are absent from other languages, but far from being esoteric, they fit quite naturally into SPITBOL, and add to its conciseness and power of expression. In the following examples, we will continue to use the **code.spt** program to illustrate each idea.

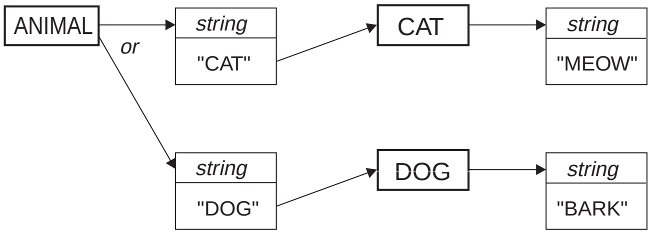
Indirect Reference

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 from the object program. This is not the case in SPITBOL; you can create new variables during execution, and reference existing ones from names specified in character strings.

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

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

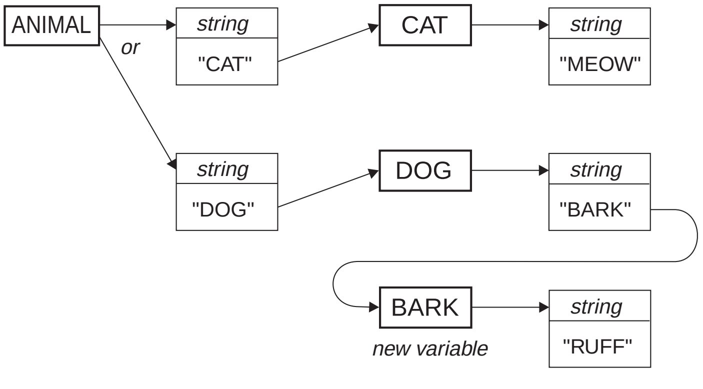
These statements make their indirect reference through the string contained in variable ANIMAL. ANIMAL's contents are treated as a *pointer* to the final destination. That is, using ANIMAL by itself retrieves 'DOG', while $ANIMAL refers to the *variable* DOG.



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'
?=  BARK
RUFF
```

which can be visualized like this:



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

```
?=  $ANIMAL  '–'  $$ANIMAL
BARK–RUFF
?=  $RUFF
Error  #239,  Indirection  operand  is  not  a  name
```

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.

Associative programming

Indirection provides a means of *programming by association*. Suppose we want to write a program allowing the user to enter a state name and get the state's capital in response. We've provided a data file called **capitals.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, 'capitals.dat')                 :F(ERROR)

*         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(ER-
ROR)

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

ERROR     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 then 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 having the state's name, and assign the capital city string to it. For example, the file line 'COLORADO,DENVER' would create variable COLORADO, with contents 'DENVER'. When End-of-File is read from the data file, the program transfers to the statement labeled QUERY.

Having established a data base, completing the program 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? State names and capitals could be stored in some explicit data structure, such as 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 SPITBOL, we did it all with one statement: OUTPUT = $INPUT. Associative programming can often replace a conventional linear search.

Variable names

Earlier we said that variable names were composed of letters, digits, and the characters period and underscore, and that they could be up to 1,024 characters long. These restrictions apply *only* to variables which appear literally in program text. Variable names created or referenced with the indirection operator may be composed of *any* non-null string of characters, and may be as long as any other string. If our previous program 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’
ARIZONA = ‘PHOENIX’
…
NEW HAMPSHIRE = ‘CONCORD’
…
```

The dump reveals a variable named NEW HAMPSHIRE, which contains a *blank* within its name. Clearly, it could never appear in a statement, such as:

```
          NEW HAMPSHIRE = ‘CONCORD’
```

since SPITBOL sees this as a pattern match statement: with NEW as the subject, and HAMPSHIRE as the pattern. To reference this variable, we must use:

```
          $’NEW HAMPSHIRE’ = ‘CONCORD’
```

Try **code.spt** with some unconventional variable names:

```
?         $’"’ = ‘DOUBLE QUOTE’
?         $’$#@!*‘ = 53.1
?         NM = DUPL(‘AB CD’, 1000)
?         $NM = ‘LONG’
?= $’$#@!*‘ $’"’ $NM
53.1DOUBLEQUOTELONG
?= SIZE(NM)
5000
```

Indirect Gotos

Indirection is not restricted to the main body of a statement. It may also be used in the Goto field to transfer control to a label specified by a variable. Suppose variable OP contained 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 indirection to obtain the final transfer label name.

Indirection 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 multi-way control branch based on an item of data. Of course, the computed label name must be defined in the program. SPITBOL provides an error message if your program transfers to an undefined label.

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

Indirection cautions

Creating variables with the unary $ operator carries some danger with it as well. Because SPITBOL's variables are global in scope, allowing data read from a file to create variable names could have unexpected consequences. Suppose our **capitals.dat** file mistakenly contained words like REM or OUTPUT? Our assignment statement would interfere with program patterns and variables with the same names.

One way to avoid this problem is to prefix names with a character that could not appear at the beginning of a name entered directly into the program, such as the reverse slash (\). Then the assignment statement in the program might read:

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

and the access statement would appear as:

```
QUERY  OUTPUT = $('\' INPUT)
       :S(QUERY)
```

There are better ways of solving this problem. Later in this chapter we'll discuss another data type called a table, which allows us to make the kinds of associations used in **capitals.dat** much more elegantly, and within a prescribed and protected domain, with no danger of name collisions.

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
```

Unfortunately, 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. However, reconstructing a pattern whenever it is used is inefficient, so a one-time definition of NPAT is preferable.

The *unevaluated expression* or *deferred evaulation* 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. N's current value is ignored at this time. Later, when NPAT is used in a pattern match, the deferred evaluation operator fetches the then current value of N.

Deferred evaluation may appear as the argument of the pattern functions ANY, BREAK, BREAKX, LEN, NOTANY, POS, RPOS, RTAB, SPAN, or TAB. Here's an example using **code.spt**:

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

Note that I and S were undefined when PAT was first constructed. Later, we will apply this technique to construct recursive patterns.

Deferred evaluation may also be applied to a pattern's alternate or subsequent clause or to the entire pattern. Because deferred expressions are valid arguments only to the pattern functions mentioned above, you'll have to move the operator "out a level" to use it with other kinds of functions. For example, the first statement is incorrect, and will result in an execution error:

```
?         PAT = EQ(*I, 4) 'ABC' | 'DEF'
Error #101, EQ first argument is not numeric
Failure
?         PAT = *EQ(I, 4) 'ABC' | 'DEF'
Success
```

By having the * operator apply to the function EQ(I, 4), the problem with it appearing as a function argument was circumvented.

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.spt** 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
?
```

Immediate assignment and deferred evaluation

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 deferred 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 deferred evaluation operator, so that its *new* value is retrieved. For instance:

```
?        FPAT = SPAN('0123456789') $ N LEN(*N) . FIELD
?        '12ABCDEFGHIJKLMNOPQ' ? FPAT
Success
?= 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 SPITBOL's pattern matching can be. Problem: Examine a subject for an arbitrary three-character substring which appears twice in a row, or is immediately followed by its mirror image. Solution:

```
?          TWOPAT = LEN(3) $ X *(X | REVERSE(X)) . OUTPUT
?          'ABCDECDEFGH' ? TWOPAT
CDE
Success
?          'ABCDEEDCBA' ? TWOPAT
EDC
Success
```

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

Arrays

Array concepts

Arrays in SPITBOL 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* SPITBOL data type, independent of the types in other array elements.

Arrays may have any number of dimensions. 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.

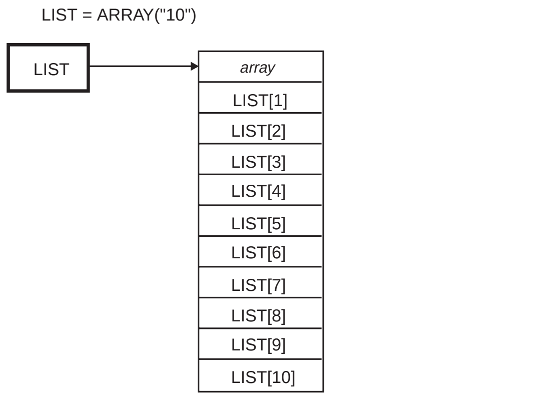
In keeping with SPITBOL's pliability, an array is defined *during* program execution, rather than at compilation time. Its size and shape is specified by a string. Arrays may be deleted or re-created with a different definition at any time.

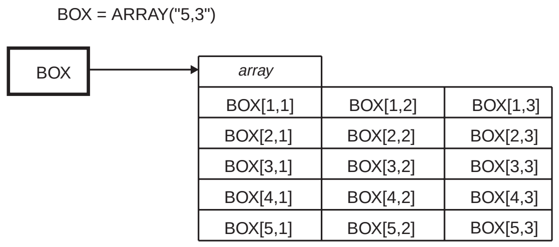
Array creation

Arrays are created by the SPITBOL 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 sample statements for use with **code.spt**. They create one- and two-dimensional arrays named LIST and BOX respectively:

```
?          LIST = ARRAY('10')
?          BOX = ARRAY('5,3')
```

LIST points to a vector of 10 elements. BOX points to a grid, 5 rows high and 3 columns wide, containing 15 elements. The ARRAY function initializes all array elements to the null string.





In the case of a vector, you'll often see the ARRAY function call written with an integer argument instead of a string:

```
LIST  =  ARRAY(10)
```

SPITBOL quietly converts the integer 10 to the string "10" as required by the ARRAY function.

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.spt**:

```
?          LIST<3>  =  'MAPLE'
?          BOX[3,2]  =  3.54
?          LIST[11]  =  4
Failure
?=  LIST[3]  LIST[4]  BOX<3,2>
MAPLE3.54
```

As you can see, angular and square brackets have been used interchangeably.

The reference to LIST[11] failed because the largest subscript allowed for that array is 10. 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
?= B[3]
MAPLE
?           B<3> = 'WILLOW'
?= LIST<3>
WILLOW
```

Assigning the pointer in LIST to B made both variables point to the same array. Since there is but one actual array, array references made using LIST or B are equivalent. Use the COPY function described in Chapter 19, "SPITBOL Functions," when you need a duplicate (and separate) 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
PRINT       I = I + 1
            OUTPUT = SCORE[I]
:S(PRINT)
             ...
```

Array initialization

Arrays may be created with an initial value other than the null string. The ARRAY function accepts a second argument which specifies this initial value. For example, 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
```

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 argument for the ARRAY function has the general form:

```
            'L1:H1,L2:H2,...,Ln:Hn'
```

L and H are integers specifying the lower and upper bounds respectively 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')
?= A[–1]
PIPER
?= A[3]
Failure
```

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

Tables

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 SPITBOL data type*. Strings, real numbers, even patterns may be used as *subscripts*. Tables combine the idea of associative programming with the data grouping of arrays.

Tables are created by the SPITBOL 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.spt**:

```
?           T = TABLE()
?           T['ROSE'] = 'RED'
?           T[0.093] = 6
?= T[0.093] T['THE'] T['ROSE']
6RED
?           FLOWER = 'ROSE'
?           T[FLOWER] = T[FLOWER] ',THORNS'
?= T[FLOWER]
RED,THORNS
```

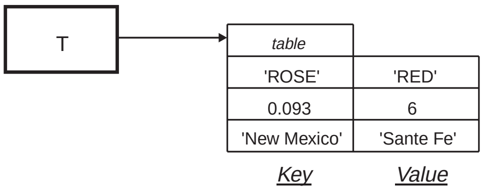
Here, a string and a real number have been used as table subscripts. The concept of an "out-of-bounds" subscript does not exist with tables. The reference to T['THE'], a non-existent table entry, merely returned the null string. T['THE'] was not added to the table.

Entries are added to a table by using the subscripted table value as the object of an assignment. That can be a direct assignment, or the conditional or immediate assignment that occurs in a pattern match.

Tables can be viewed as a two-dimensional structure, with N rows and two columns. N is the number of elements added to the table. The first column contains subscripts, also known as *keys*. The second column is the value. After adding another entry such as

```
?           T['New Mexico'] = "Santa Fe"
```

our three-element table would look like this:



Subscript keys in the first column are all unique. You cannot create a another entry for T['ROSE'] for example—assignments to T['ROSE'] will always modify the existing entry. However, different subscripts can have the same value entries. No ambiguity arises from duplicate values.

Table initialization

As with arrays, a table can be created with initial values for its elements. This value will be returned if you reference a table item before anything has been stored there. Normally, SPITBOL supplies the null string as the initial value.

You may also specify an initial size when you create a table. Table data is accessed slightly faster if the number of table entires remains less than this initial size. (The number you specify is not a limit on the size of the table—it just sets aside an initial amount of memory for the table.)

The easiest way to set up a table is:

```
        T = TABLE()
```

while the full form of a table declaration is:

```
        T = TABLE(Arg1, Arg2, Arg3)
```

Arg1 is the estimated size. Arg2 is just there for compatibility with other versions of SNOBOL; it is ignored by SPITBOL. Arg3 is the initial value for entries in the table; if there is no Arg3, they all start as the null string.

So, to create a table that you expect to have about 100 entries, with an initial value of 'STUFF', you would declare:

```
        T = TABLE(100, , 'STUFF')
```

Returning to our state capitals program, we could eliminate the possibility of conflict with program variables by using a table. The state names will be used as subscripts, and the capitals will be the data values stored in the table. We can even use a default table value to provide a simple error message if a table lookup is made with a state name not in the table. At the beginning of the program, we would declare a table to hold our data:

```
        CAPITALS = TABLE(50, , 'Not found, try again.')
```

As data is read from the file, it is entered in the table:

```
        CAPITALS[STATE] = CAPITAL
:(READF)
```

Finally, the lookup phase would be coded as:

```
QUERY   OUTPUT = CAPITALS[INPUT]
:S(QUERY)
```

(For those who like to write the shortest program possible, we can use the immediate assignment operator discussed earlier to combine the two program lines

```
        LINE ? BREAK(',') . STATE LEN(1) REM . CAPITAL :F(ER-
ROR)
        CAPITALS[STATE] = CAPITAL
:(READF)
```

into one statement:

```
                LINE ? BREAK(‘,’) $ STATE LEN(1) REM . *CAPITALS[STATE]
+
:S(READF)F(ERROR)
```

The + in the first column is a continuation marker that’s needed because the statement spread across two physical lines. The state name will be immediately assigned to STATE. Its usage as a subscript to the CAPITALS table occurs only after the pattern match is complete. Note that a table entry may be the object of assignment during pattern matching.)

Tables are one of the most exquisite features of SPITBOL, because they permit one data item to be associated with another, and there are no limitations on data types. Other tables and arrays can even be used as subscripts or data items.

Conversion between tables and arrays

If a table is loaded with data items read from a file, an immediate question arises. 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 we were using an array, we could step an integer subscript through the array to obtain the data. Applying integer subscripts to a table merely references non-existent entries.

SPITBOL provides a simple solution—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 which were used to create the table. The second column contains the values that were stored with the corresponding table subscript. The number N is the number of table entries with *non-null* 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 SPITBOL CONVERT function, which accepts a table argument and the word ‘ARRAY’, and returns a pointer to the new array. Continuing with the earlier example:

```
?           A = CONVERT(T, ‘ARRAY’)
Success
?= A[1,1] ‘:’ A[1,2]
New Mexico:Santa Fe
?= A[2,1] ‘:’ A[2,2]
ROSE:RED,THORNS
```

Although table entries are stored using a hashing technique, the order in which they are placed in the table is not random. Table entries will appear in the array ordered by time-of-entry into the table.

As you would expect with SPITBOL, 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
?= W[‘ROSE’]
RED,THORNS
```

Dumping arrays and tables

Earlier we mentioned that the &DUMP keyword could be used to display the contents of variables when a program terminates. If you set the &DUMP keyword to the value 2, the dump display will also include the contents of all arrays and tables created by your program. Try it now in **code.spt**:

```
?        &DUMP = 2
?<EOF>
```

worduse.spt

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, worduse.spt
*
*        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.
*
*        The data file is read from standard input,

         &TRIM = 1
         &ANCHOR = 1

*        Define the characters which form a 'word'
         WORD = "'–" '0123456789' &LCASE

*        Pattern to isolate each word and 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 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 use REPLACE to 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] returns the null string. Integer 1 is added to the null string, and the result, 1, is stored in WCOUNT[ITEM], thereby creating the entry in that table. 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 encountered, control transfers to label DONE, and now we would like to 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, displaying 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.

To run the program with file **faustus** as the input file, use:

```
spitbol worduse <faustus
```

The indirection operator should come to mind here because it provides associations similar to a table's. For example, after each word is placed in ITEM, we could create and increment an indirectly-referenced variable with the statement:

```
         $ITEM = $ITEM + 1
```

If ITEM contained the string 'the', the above statement would behave like this:

```
         $'the' = $'the' + 1
```

Counting all the words would proceed in a manner similar to the table method. But how do we learn the names of all the variables we created to display the results? We would need to record all the names, either in an array, or perhaps in a long concatenated string. Tables provide a more elegant and economical solution.

Another advantage of tables is that they are easily sorted. We could have the result words displayed in ascending order merely by replacing the statement DONE with:

```
DONE     A = SORT(WCOUNT)
```

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 indirection 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]'
?=  $V

Success
```

The indirection 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. SPITBOL creates a variable called A[4,2], which 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 SPITBOL's point of view.

To make this work properly, 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 indirection operator:

```
?         V  =  .A[4,2]
?=  $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,  'capitals.dat')
INPUT(.INFILE,  1,  'capitals.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. Of the following two statements, only the second one is correct:

```
INPUT('A[4,2]',  1,  'capitals.dat')              (incorrect)
INPUT(.A[4,2],  1,  'capitals.dat')
```

Note that alternate use of the indirection and name operators "cancel" one another, so

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

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

Alternative Evaluation

SPITBOL introduced a new control structure to the SNOBOL4 language that greatly simplifies programming. It is the *alternative* (or *selective*) evaluation of a series of expressions.

A parenthesized list may appear anywhere an element is used to provide a value. The elements of the list are evaluated from left to right, until an evaluation succeeds. The value of the successful list element is returned as the value of the parenthesized list. No further list elements are evaluated when an element succeeds. If all elements fail, the entire list signals failure. The general form of this type of list is:

```
(expr1, expr2, …, exprn)
```

At least two expressions, separated by commas, must be present.

SPITBOL tries to evaluate the first expression, expr1. If it is successful, its value is used as the value of the list, otherwise expr2 is evaluated, etc. Note that the alternation operator (|) provides alternatives for *patterns*. This construction provides alternatives for *expressions*. Sample statements:

```
MAXIJ = (GE(I,J) I, J)
CARDINAL = (EQ(N,1) 'FIRST', EQ(N,2) 'SECOND', 'OTHER')
SUBJECT ? (EQ(N,1) PAT1, EQ(N,2) PAT2, ABORT) = '***'
```

In the first example, the list element GE(I,J) I is evaluated. If I is greater than J, the GE function succeeds and its null-string value is concatenated with I, and the result, I, is returned as the value of the list. If GE() fails, SPITBOL proceeds to evaluate the next list element, J, which being a simple variable, always succeeds. In that case, J would be returned as the value of the list.

The second example sets CARDINAL to "FIRST", "SECOND", or "OTHER" depending upon the value of N. The last example shows that list elements are not limited to string values. Here N selects one of two patterns for a pattern match, or aborts the match if N is not 1 or 2.

Lists provides a compact method to encode the If…Then…Else notation of other languages. For example, the following sequence (coded in a generic high-level language):

```
IF A > B THEN C = D
  ELSE IF E = 5 THEN CALL PROC(E)
  ELSE CALL OUTPUT('Value Error')
END
```

could be expressed using alternative evaluation as:

```
(GT(A,B) C = D, EQ(E,5) PROC(E), OUTPUT = 'Value Error')
```

Here, the first list expression is GT(A,B) C = D. If A > B, the remainder of the expression, C = D, is performed, and the list processing is complete. Otherwise, the second expression, EQ(E,5) PROC(E) is evaluated, etc.

There are several potential traps to be aware of when using alternative evaluation.

First, SPITBOL is evaluating each expression, and proceeding to the next alternative if the expression fails. This may have unexpected consequences.

Suppose A is less than B, and E is 5 in the previous example. The first expression, GT(A,B), fails. EQ(E,5) succeeds, so function PROC(E) is called, as intended. But suppose PROC(E) returns failure for some reason. SPITBOL sees the entire expression as failing, and proceeds on to the last expression, OUTPUT = 'Value Error'. This may not be what you expected, which was to call PROC(E) and do nothing more.

Another trap occurs when you want to produce either a value or the null string, depending upon a predicate function. Here's an example as it should be written:

```
PADDING = (DIFFER(USE_TABS) CHAR(9), "" )
```

This sets variable PADDING to either the tab character or the null string. Because an empty expression produces the null string, this could also be written as:

```
PADDING = (DIFFER(USE_TABS) CHAR(9), )
```

This is still fine. The problem comes when the last comma is omitted

```
PADDING = (DIFFER(USE_TABS) CHAR(9))
```

in the mistaken belief that merely placing parentheses around an expression creates an alternative evaluation with the null string as an implied last alternative. Both the comma *and* the parentheses are needed to signal alternative evaluation.

Chapter Summary

Unary operators	$	Indirect reference
	.	Name (location) of object
	*	Unevaluated expression
Binary operators	$	Immediate assignment
Alternative evaluation	(E_1, E_2, … E_n)	Evaluate expressions until one succeeds
Data types	ARRAY	N-dimensional, integer subscripts
	TABLE	No fixed size, subscripts may be of any data type
Functions	ARRAY(S,V)	Create array according to prototype string. Array initialized to value V
	CONVERT(I,S)	Convert data type of item to specified type
	SORT(A or T)	Create sorted array from array or table
	TABLE()	Create table of variable size
	TABLE(N,,V)	Create table of variable size with initial values set to V. N is the estimated number of entries.

Chapter 8
Program-Defined Objects

SPITBOL 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 your own new data types, functions, and operators. You can, by creating your own entities, obtain another level of conciseness and power of expression. SPITBOL is truly an amorphous creature.

We will begin with program-defined functions because they allow a program to be partitioned into smaller, more manageable segments. Functions help counter the criticism directed toward the presence of Gotos in SPITBOL. 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 grown to hundreds of lines, with 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.

With time, you will develop an inventory of useful functions that can be included in new programs. To get you started, the SPITBOL release disk contains an assortment of such functions.

Program-Defined Functions

The concept of a function should be clear from all the examples of SPITBOL'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.

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 SPITBOL by using a built-in function reserved for this purpose. Logically enough, it is called the DEFINE function. We call it with a *prototype string* containing the new function's name and arguments. DEFINE makes the new function's name known to SPITBOL, so it can be used subsequently in your program. Blanks are not permitted within a prototype string.

Suppose we want to create a new function called SHIFT, that would 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, which we will call S and N. Any name of your choosing can by used for a dummy argument. 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. SPITBOL'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.

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 is used:

> When the function is used, SPITBOL 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 inside the function body.

Returning 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 SPITBOL 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 the SPITBOL system 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 field then transfers to label RETURN to return back to the caller.

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 in this particular case, 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)
             ...
```

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 DEFINE call would now 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, SPITBOL 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.

Using functions

Once a function has been defined, it may 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* SPITBOL data type, including patterns, may be returned. Earlier, in the pattern match 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 program-defined 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 non-null 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 matchingfunctions.

Organizing functions

Unlike other programming languages, SPITBOL does not know or care what 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 [2, pp. 20–21], which we suggest you 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 string 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 at this time would cause problems.) The function is now defined, and ready to be used.

In general, all of your functions should be prepared in this form:

```
* Fname - Description of use

           DEFINE('Fname(arg1,...,argn)local1,...,localn')
            . . .
*          Any 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. Chapter 14, "SPITBOL Statements," describes a SPITBOL extension called INCLUDE, which allows you to insert these files in your program when it is being compiled. 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$. SPITBOL has the equivalent of CHR$ with its CHAR function. But it would be useful to have an equivalent for the ASC function, and place it in file **asc.inc** for later inclusion when we need that function. The file might look 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
```

Note that the function was written to work correctly regardless of the anchoring mode in use by the calling program.

Call by value, call by name

Function calls in SPITBOL 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 program-defined 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 allow function FILL to modify the contents of array A.

Remember that "?=" is a shortcut for "? output =" within the code.spt

The **code.spt** program was provided to allow interactive experiments with SPITBOL statements. If you create functions using the preceding format, they also can be tested using **code.spt**.

Functions and code.spt

Use your editor to create a disk file containing the SHIFT function on page 105. (Be sure to include the Goto that transfers around the function body.) Call the file **shift.inc**. Now, start the **code.spt** program, and type the following:

```
      ?–INCLUDE  "shift.inc"
      Success
      ?=  SHIFT('COTTON',4)
      ONCOTT
      ?=  SHIFT('OAK',4)
      Failure
```

The first line is a SPITBOL "control statement." It begins with a minus sign in the first column, and directs SPITBOL to take some special action. In this case, –INCLUDE incorporates a group of statements into your program from a file. You can use it to modularize a program under development, and to keep a library of canned functions.

When used with the **code.spt** program, control is transferred to the included statements after they are read and compiled. If function definitions have the proper Goto around the function body, they will "define themselves" after being included.

Recursive functions

The statements which 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 such a simple argument 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.

The classic example of a recursive function is one used to compute the factorial of a number. When called with a number N, we compute N! using the formula N! = N * (N – 1)!. We call ourselves recursively, with successively smaller arguments, until we request 1!. We know the answer to this is simply 1, so we return it without any further recursive calls. The function to do this is provided on the release disk in file **fact.inc**, and looks like this:

```
* FACT(N) – Compute N! using recursion.
*         N must be less than 171.0 to prevent real overflow

          DEFINE('FACT(N)')                              :(FACT_END)

*         If argument is 1 or less, return 1 as result
FACT      FACT = LE(N,1) 1                               :S(RETURN)

*         Otherwise, result is N * (N–1)!
          FACT = N * FACT(N – 1)                         :(RETURN)
FACT_END
```

This function can be "included" in **code.spt** for exploration. Since the value of 13! is 6,227,010,800, you will have to use real values for N>12 to prevent integer overflow:

```
?–INCLUDE "fact.inc"
?= FACT(5)
120
?= FACT(12)
479001600
          OUTPUT = FACT(13)
Error #28, Multiplication caused integer overflow
?= FACT(13.)
0.62270208E+10
```

There is no explicit declaration for recursion; any SPITBOL 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.

roman.inc

Sometimes, recursion can produce dramatically smaller programs. Gimpel [2, pp. 25–26] provides a good example with his recursive function, ROMAN. It will convert an integer in the range 0 to 3999 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 then appended to the result.

The function looks like this (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.

If **roman.inc** is a disk file, you can watch it work by running **code.spt** and typing the following:

```
?–INCLUDE  "roman.inc"
?          &FTRACE  =  100
?=  ROMAN(432)
```

Set &FTRACE to 0 to turn off function tracing, which is described more fully in Chapter 10, "Debugging."

External Functions

Some versions of SPITBOL provide a mechanism for loading and executing functions written in other languages, such as C, Pascal, and assembly language. Such functions are loaded with SPITBOL's built-in LOAD function, which behaves in a manner similar to DEFINE. When no longer needed, external functions may be removed from the system with the built-in UNLOAD function.

See Chapter 19, "SPITBOL Functions," for a description of LOAD and UNLOAD, and Appendix F for a discussion of how to write external functions.

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

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 the field names replacing the arguments. The form of the prototype string is:

```
            'TYPENAME(FIELD1,FIELD2,…,FIELDn)'
```

Blanks are not permitted within a prototype. Try creating a new data type using the **code.spt** program:

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

The DATA function tells SPITBOL to define an object creation function with the new data type's name:

```
            PRODUCT(arg1, arg2, arg3, arg4)
```

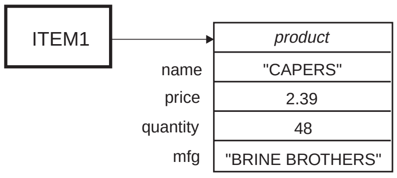
This new function can be called whenever we wish to create a new object with the PRODUCT data type. Its arguments are the initial values to be given to the four fields which comprise a PRODUCT. The function returns a pointer to the new object, which can be stored in a variable, array, or table. Try creating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

```
?           ITEM1 = PRODUCT('CAPERS', 2.39, 48, 'BRINE BROTHERS')
?           ITEM2 = PRODUCT('PICKLES', 1.25, 72, 'PETER PIPER
INC.')
```

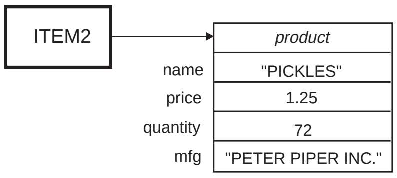
At this point, SPITBOL has created two data structures in memory that are shown on the next page.

Data type use

ITEM1 = PRODUCT("CAPERS", 2.39, 48, "BRINE BROTHERS")



ITEM2 = PRODUCT("PICKLES", 1.25, 72, "PETER PIPER INC.")



The defining call to the DATA function also created several *field reference functions*. In this example, their names would be:

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

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

```
?= MFG(ITEM1)
BRINE BROTHERS
?= PRICE(ITEM1) * QUANTITY(ITEM1)
114.72
```

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.

Tables work very nicely with program-defined data types to provide an index. For example, assume all our products have unique names. We can create a table P, and use the product name as the subscript. The table entry would point to the complete data object for each product:

```
?        P = TABLE()
?        P['SALT'] = PRODUCT('SALT', 0.69, 12, 'BRINE BROTHERS')
?        P[NAME(ITEM2)] = ITEM2
```

After the table is constructed, we can use the product name to retrieve other information about the product:

```
?= MFG(P['SALT'])
BRINE BROTHERS
?= PRICE(P['PICKLES'])
1.25
```

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.spt**:

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

Displaying a data type

Try the following in **code.spt:**

```
?= ITEM1
PRODUCT
```

As you can see, all SPITBOL knows about ITEM1 is its data type. It's up to you to provide a method of displaying the data in a meaningful form. The simplest method is to create a function to display the individual fields. This one will be called PRDSTR (PRoDuct to STRing), but any convenient name could be used.

We'll create it by using the SLOAD function built into **code.spt.** It reads and compiles statements from a file, and uses the special file name "–" to read from the keyboard. Just stay in **code.spt** and enter the following:

```
?          SLOAD('–')
           DEFINE('PRDSTR(X)')
                          :(PRDSTR_END)
PRDSTR PRDSTR = NAME(X) '  ' PRICE(X) '  '
+                   QUANTITY(X) '  ' MFG(X)          :(RETURN)
PRDSTR_END
<EOF>
Success
?= PRDSTR(ITEM1)
CAPERS  2.39  24  BRINE BROTHERS
?= PRDSTR(P['PICKLES'])
PICKLES  1.25  60  PETER PIPER INC.
```

Creating structures

Our inventory example used string, integer, and real values as the field contents. In fact, *any* SPITBOL data type may be stored in a field, including pointers to other program-defined types. Complex structures, such as queues, stacks, trees, and directed graphs may be created. The SPITBOL distribution media contains sample programs for creating and manipulating such structures.

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.spt**, but will sketch out the changes:

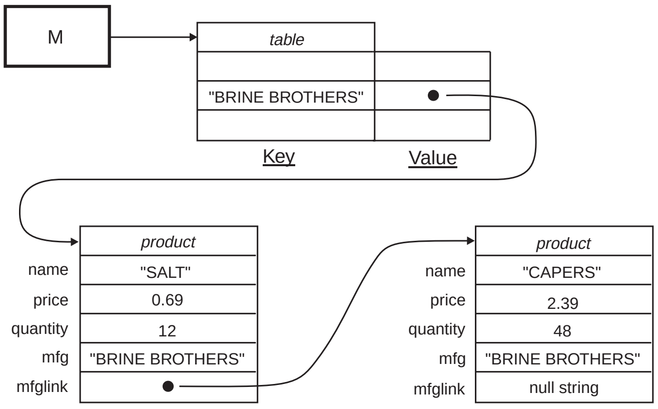
```
DATA('PRODUCT(NAME,PRICE,QUANTITY,MFG,MFGLINK'))
```

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. Once again, 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 an existing product definition uses the same company, MFGLINK will point to that 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. The following diagram shows the resulting data structure.



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

```
        X = M[COMPANY]
LOOP    OUTPUT = DIFFER(X) PRDSTR(X)        :F(DONE)
        X = MFGLINK(X)                                :(LOOP)
DONE
```

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:

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

The data type name can be compared with a known name using function IDENT. Suppose variable X contains a number, and we want to remove any fractional part if it is a real number, and leave it unchanged otherwise. We test for the REAL data type, and if found, call function CHOP:

```
X = IDENT(DATATYPE(X), 'REAL') CHOP(X)
```

This may sound arcane here, but it will come in handy shortly, when we define a FRACTION data type, and then a function to add two fractions. Inside that function, FADD, we will want to make sure we're working with fractions. You'll see IDENT(DATATYPE(X), 'FRACTION').

Program-Defined Operators

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

Operators and functions

The built-in function OPSYN creates synonyms for functions. It also allows you to call a function by using one of SPITBOL's undefined operators.

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

The new name is defined as a synonym of the old name. The old name must be a function already defined—either one that is built into SPITBOL, or a function you have defined in your program.

The third argument is 0, 1, or 2. If it is omitted, SPITBOL assumes that the third argument is 0.

A third argument of 0 means that the first argument is a function name, not an operator.

If the third argument is 1, then the first argument must be one of the unused unary operators (!, %, /, #, =, |).

When the third argument is 2, the first argument must be one of the unused binary operators (&, @, #, %, ~)

Function synonyms

We can make the name LENGTH a synonym for the SIZE function, and PLUS a synonym for + as follows:

```
?          OPSYN('LENGTH', 'SIZE', 0)
?          OPSYN('PLUS', '+', 0)
?= LENGTH('RABBIT')
6
?= PLUS(10, 20)
30
```

Operator synonyms

The tables in Chapter 15, "Operators," list the unused unary and binary operators. Normally, if you try to use such an operator, you'll get an execution error:

```
?OUTPUT = 1 # 1
Error #29, Undefined operator referenced
```

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

```
?          OPSYN('#', 'DIFFER', 2)
?= 1 # 2
Success
?= 1 # 1.0
Success
?= 1 # 1
Failure
```

Note that we can only associate an unused operator with a function or existing operator. The normal SPITBOL operators cannot be redefined:

```
?          OPSYN('+', 'PLUS', 2)
Error #156, OPSYN first arg is not correct operator name
```

This is one of the few places where SPITBOL is more restrictive than other SNOBOL4 dialects. However, by not allowing basic system functions and operators to be redefined, SPITBOL is able to optimize the code it generates.

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

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

Using synonyms

These examples happened to use built-in functions (DIFFER, ANY) for the old name, but a program-defined function could be used as well.

For a simple example, you could make an operator capitalize a string. Terminate **code.spt** by entering <EOF>, then use your editor to create a file called **ucase.inc**. In it, define this function:

```
           DEFINE('UCASE(S)')
           OPSYN('!','UCASE',1)
                          :(UCASE_END)

UCASE      UCASE = REPLACE(S, &LCASE, &UCASE)        :(RETURN)
UCASE_END
```

After you've saved it, restart **code.spt** and include the file you just created. Then you can convert any string to upper case by using the unary ! operator:

```
?–INCLUDE "ucase.inc"
?= !"SPITBOL is very fast."
SPITBOL IS VERY FAST.
```

Synonyms are more than just grammatical curiosities—they can be combined with program-defined data types to extend the language.

Suppose we are working with fractions, and we want to avoid the round-off errors that occur with some computations. When you say:

```
          X = 1 / 3.0
```

the value that is stored for X is not precisely one-third, but a binary approximation. As an exercise, we'll create a new data type that will store the numerator and denominator as separate values.

Exit from **code.spt**, and use your editor to create **fraction.inc**. First we create a new data type, FRACTION:

```
          DATA('FRACTION(N,D)')
```

This stores the exact numerator in N and the exact denominator in D. Since we're storing their exact values, we shouldn't have any rounding errors. But we need to do more with this precise fraction than store it. We would like to be able to add two FRACTIONs. So we define a function, FADD. It should be nice and general. If called with numeric arguments, it should return a simple numeric sum. But if either argument is a fraction, it should return a fraction result.

```
          DEFINE('FADD(X,Y)DX,DY')                      :(FADD_END)

*  Note the datatype of each argument.
FADD      DX = DATATYPE(X)
          DY = DATATYPE(Y)

*  If neither argument is a fraction, just return a simple sum.
          FADD = DIFFER(DX,'FRACTION') DIFFER(DY,'FRACTION')
+                   X + Y                                :S(RETURN)

*  Not so. If either argument is not a fraction, make it into one.
          X = DIFFER(DX,'FRACTION') FRACTION(X,1)
          Y = DIFFER(DY,'FRACTION') FRACTION(Y,1)

*  Now we know they're both fractions. Add them together.
*  Remember that A/B+C/D = (AD+BC)/BD.
          FADD = FRACTION(((N(X) * D(Y)) + (N(Y) * D(X))),
+                                (D(X) * D(Y)))      :(RETURN)
FADD_END
```

Now we have a function that can add two numbers or fractions. To put OPSYN to work, we will define '&', an unused binary operator:

```
          OPSYN('&','FADD',2)
```

Displaying these fractions in a customary form would be simpler if we had a function to do the work. So we'll create FSTR, which takes a fraction and returns it as a string, with the numerator, a slash, and the denominator. If we called FSTR with FRACTION(3,4), we would get the string "3/4" in return. We'll also show off a simple use for SPITBOL's alternative selection feature discussed in Chapter 7, "Additional Operators and Datatypes."

```
        DEFINE('FSTR(X)')                           :(FSTR_END)

* If argument is not a fraction, just return it unchanged.
* Otherwise, return its numerator and denominator.
*
* Test the argument's datatype as the first element of
* an alternative evaluation expression.  If the IDENT
* succeeds, return N(X) and D(X).  If it fails, just
* return X.
*
FSTR    FSTR = (IDENT(DATATYPE(X), 'FRACTION') N(X) '/' D(X), X)
+                                             :(RETURN)
FSTR_END
```

While we're at it, we'll give FSTR a synonym, an appropriate one from the set of unused unary operators:

```
        OPSYN('/', 'FSTR', 1)
```

If you've typed in the preceding, you'll want to experiment with it. Go back to **code.spt** and try this:

```
?–INCLUDE 'fraction.inc'
?       HALF = FRACTION(1,2)
?       THIRD = FRACTION(1,3)
?       SUM = HALF & THIRD
?= /HALF ' + ' /THIRD ' = ' /SUM
1/2 + 1/3 = 5/6
?= /(4 & 5)
9
```

With a little work, you could make FADD return its totals reduced to the lowest common denominator, and you could define functions to subtract, multiply, and divide fractions. You could use other undefined operators to call those functions.

Remember, operator redefinition is not limited to mathematical operations. Operators can be created to maintain a stack, or navigate around a tree. The full generality of functions and program-defined data types are available to the unused operators. Through this technique you can make SPITBOL speak the language of your particular problem.

Chapter Summary

Program-defined function	DEFINE(S)	Create function according to prototype string
	entry	Label with same name as function
	result	Variable with same name as function
	RETURN	Return success from function
	FRETURN	Return failing from function
External function	LOAD(S1,S2)	Load and define external function
	UNLOAD(S)	Unload and undefine external function
Program-defined data type	DATA(S)	Create data type according to prototype string
	DATATYPE(X)	Data type name of X
	name(f1, …, fn)	Object creation function
	f(X)	Field reference function
Synonyms	OPSYN(new,old,N)	Create synonym for function or operator

Chapter 9
Advanced Topics

The material presented so far allows you to write powerful SPITBOL programs. In this chapter, we will examine other interesting and useful features of the language. Often a pattern or function can be greatly simplified by using one of the language features described here.

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, SPITBOL 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.spt** 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
```

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
```

The unevaluated expression operator used with *LIST allows a *forward reference* to a pattern not yet defined. Recursion occurs in the second line, when LIST is defined in terms of ITEM, which was 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.

The patterns should be designed so that some progress has been made in the subject prior to encountering the recursion. In this example, "(" and "," in the LIST pattern serve that purpose.

SPITBOL saves information on a 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. You can also increase the size of SPITBOL's stack by using the –s command line parameter described in Chapter 13, "Running SPITBOL."

A recursive pattern that recurses without consuming subject characters will produce a *recursive plunge* and stack overflow immediately:

```
EXPRESSION = *EXPRESSION | "(" TERM ")"
```

In going to evaluate the first alternative, SPITBOL needs the current value of EXPRESSION, which means evaluating the first alternative, ad infinitum. Switching alternatives like this:

```
EXPRESSION = "(" TERM ")" | *EXPRESSION
```

only helps if the first alternative happens to match. If it doesn't, the second alternative recurses until the stack overflows. Something else needs to appear with the second alternative to consume subject characters prior to recursion. Perhaps something like this is more appropriate:

```
EXPRESSION = TERM | "(" *EXPRESSION ")"
```

Quickscan and Fullscan

Pattern matching can be time-consuming because of the number of possibilities which must be attempted. The original SNOBOL4 system applied heuristics to the process, eliminating match attempts that could not possibly succeed. This was done by carrying out side computations of the remaining subject characters, and the amount required at any point in the pattern.

Whether the extra memory and computations required by the heuristics are worth the savings in pattern matching time has long been debated. What is known is that the introduction of the immediate assignment operator meant that heuristics were no longer invisible to the programmer. Eliminating some match attempts eliminated some immediate assignments, and programs might run differently with the heuristics turned on or off.

The author of SPITBOL has concluded that the heuristics do not result in a significant increase in speed. Furthermore, it often produces malfunctioning patterns when deferred evaluation is used within a pattern. Accordingly, pattern matching is done exhaustively and no heuristics are applied. In particular, deferred expressions are not assumed to match at least one character, and recursive patterns always work properly.

Heuristics were called the "Quickscan mode" of pattern-matching. They were controlled with the &FULLSCAN keyword—zero for Quickscan, or non-zero for "Fullscan mode," where all combinations are tried.

The &FULLSCAN keyword has been retained, but it may only be set to a non-zero value. Attempting to set it to zero will produce an error message.

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 the patterns 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 non-null 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 by other pattern components:

```
?         'AB+(14–2)*C' ? ANY('+–*/') BAL . OUTPUT ANY('+–*/')
(14–2)
Success
```

By combining BAL and the REPLACE function, strings balanced by other character pairs can be found. For example, to find a string in S balanced by square brackets, exchange brackets for parentheses:

```
REPLACE(S, '[]()', '()[]') ? BREAK('(') BAL . RESULT
RESULT = REPLACE(RESULT, '()[]', '[]()')
```

For an example of using the BAL primitive to manipulate algebraic expressions, consult file **prefix.spt** on the SPITBOL distribution disk.

FAIL

Seek other alternatives

The FAIL pattern signals the 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.

Note the difference between ABORT and FAIL. ABORT stops all pattern matching, while FAIL tells the system to back up and try other alternatives or other subject starting positions.

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 immediately followed by a plus sign. In the example below, the 'A's match, we go through the FENCE, and find '+' does not match the next subject character, 'B'. SPITBOL 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, SPITBOL 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. We used to say that there were no serious uses for the SUCCEED pattern, but discussions in September, 1996 on the SNOBOL4 mailing list server have effectively countered that assertion. While the on-line list archives can be consulted for the full discussion (file **list9609.txt**), they can be summarized as follows:

SUCCEED can be used as part of a loop control structure within a pattern when it appears in the form:

```
SUCCEED something_with_possible_ABORT FAIL
```

The pattern matcher will continually move back and forth between SUCCEED and FAIL until something in between causes the match to abort. This example

```
?          P = FENCE(TAB(*(N + 1)) $ OUTPUT @N | ABORT)
?          "abcd" ? POS(0) $ N SUCCEED P FAIL
a
ab
abc
abcd
Failure
```

displays successively larger subject substrings. The FENCE function (discussed in the next section) is necessary to prevent the pattern matcher from seeing the ABORT pattern when it is backing up.

The substrings could be presented to a user-defined function which would determine when to end the loop (by returning the null string to continue or the ABORT pattern to terminate).

```
           DEFINE('FN(T)')                                 :S(FN_END)
FN         FN = GT(SIZE(T), 4) ABORT              :(RETURN)
FN_END
           P = TAB(*(N + 1)) $ T *FN(T) @N
           S ? POS(0) $ N SUCCEED P FAIL
```

Here, the pattern would continue to oscillate until the matched substring contained more than four characters. The ABORT pattern has been moved to within the function, and the FENCE function is not needed because there are no alternatives within P.

Peter-Arno Coppen has devised these rules for the use of SUCCEED:

1. If SUCCEED can be replaced by the null string, it is superfluous.

2. SUCCEED without an ABORT or FENCE, and without a deferred expression will either never terminate or it is superfluous.
3. No sensible use of SUCCEED is possible without the unevaluated evaluation operator (*).

Other Functions

We'd like to briefly point out a few more built-in functions. Chapter 19, "SPITBOL Functions," describes their calling sequences in detail.

APPLY (name, args)	Allows an indirect call to a function through a variable.
CONVERT (arg, type)	Provides explicit conversion from one data type to another. Chapter 17, "Data Types and Conversion," describes the conversions possible.
ENDFILE(S)	Closes a file and detaches all variables associated with it.
FENCE(P)	Given a pattern argument P, FENCE(P) is a same pattern except alternatives within P are only visible when the pattern matcher is moving forward through the pattern. If a subsequent pattern element forces the scanner to back up, alternatives within P are not examined.
HOST(N,args)	Provides access to machine-specific extensions of SPITBOL.
ITEM(array,N)	Allows an indirect reference to an array or table. The use of ITEM is never necessary in SPITBOL, because of the extended syntax for array and table references.
LPAD(S,N) RPAD(S,N)	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.
REWIND(S)	The image conjured up by the word "rewind" made a lot more sense in the days of spinning magnetic tape reels. REWIND positions a file so that the next read or write operation will take place at the beginning of the file.
RSORT(T) SORT(A)	Sorts the elements of an array or table in descending or ascending sequence. The sorted results are returned in a new array.
SET(S,M,N)	Positions a file for random-access I/O.
SETEXIT(S)	Allows interception of execution errors.

SUBSTR

Allows a substring to be extracted from another string. The substring's location and length are specified by integer character positions.

Binary Operator Extensions

These extensions appear in SPITBOL and SNOBOL4+, but not in standard SNOBOL4.

Assignment
=

Multiple use of the equal sign (=) operator within a statement is allowed. The operator is right associative, meaning that multiple instances of the equal sign within an expression are performed right to left. It returns the value of its right hand side. Sample statements:

```
ARRAY<I = I + 1, J = J + 1> = “OSCAR”
A = B = C + D
Q = 1 + (R = 1 + (S = T + 1))
```

Pattern match
?

The binary question mark operator (?) has been used throughout this tutorial to make pattern matching explicit. However, its definition is more general, as matching, and matching with replacement, can be used like any other SNOBOL4 expression. Thus, the expression

```
(SUBJECT ? PATTERN)
```

performs pattern matching and returns the substring matched, or failure as its value. The expression

```
(SUBJECT ? PATTERN = OBJECT)
```

returns the entire subject *after* replacement occurs, or failure. In both cases, a string is returned, so the returned object may be the subject of another pattern match, *but not the subject of replacement*, since that requires a variable as the subject.

If the pattern match fails, the failure signal is treated like any other function failure. Parentheses are not required if the order of operations is unambiguous. Sample statements:

```
P = Q (A ? B = C) R
A ? B . V1 ? C . V2
N = (A ? B, C ? D = E, F ? G =, I ? (J ? K = (L ? M)))
```

Note the implicit null object used in the list element F ? G =. The question mark operator may be combined with multiple assignment and alternative evaluation to create truly indecipherable statements.

Other Unary Operators

Interrogation
?

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, returning the null string. In other words, any value component of X is replaced by the null string.

Remember that predicate functions return the null string if successful, and that we use them to interpose tests in a statement. The interrogation operator now allows us to treat *any* expression in the same manner. For example, this statement adds 1 to N if the pattern match succeeds:

```
          N = ?(S ? P) N + 1
```

Notice that this statement uses the question mark in two different ways. The binary question mark signifies a pattern match, while the unary (first) question mark converts a successful match to the null string.

Programming in this style can eliminate many trivial "jump arounds" and the resulting need to invent labels. For example, without this operator, the statement above would be written:

```
          S ? P
:F(NO)
          N = N + 1
NO        …
```

Negation
~

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.

Like the interrogation operator, the negation operator can eliminate some program labels. Recall that the INPUT function fails if the file specified does not exist. Then this statement tries to open a file and produces an error message if it doesn't exist:

```
          TERMINAL = ~INPUT(.IN,1,'NoneSuch') 'File not found'
:S(ERR)
```

If INPUT(…) fails, then ~INPUT(…) succeeds, allowing the error message to be assigned to TERMINAL. Note that the negation means we have to use a success Goto to test for failure of the INPUT function.

Shafto [7, p. 29] provides an interesting example of evaluating arbitrary Boolean expressions in the pattern field. The null string represents Boolean TRUE, and failure is Boolean FALSE. Functions returning these values can be combined using concatenation for AND, alternation (|) for OR, and negation (~) for NOT. A null string subject is used—it's just a place-holder—as all the action takes place in the pattern field. For example:

```
          '' ? *EQ(N,3) *GT(X,Y) | *~(IDENT(V) LT(Y,Z))
:S(TRUE)F(FALSE)
```

The deferred evaluation operator (*, page 86) is needed here to prevent failures during pattern construction from terminating the match early.

Run-Time Compilation

The two functions described below are among the most esoteric features, not just of SPITBOL, but of any programming language. You won't use them very often, but when you do, you'll be able to save hundreds or thousands of lines of code. While your program is executing, *the entire SPITBOL compiler* is just a function call away.

A SPITBOL program is nothing more than a string of characters. When we first compile our program, SPITBOL reads a long string from a file, and converts it to internal object code. Program execution occurs when the object code is subsequently executed. The functions EVAL and CODE let you supply the compiler with character strings from within the program itself.

EVAL function

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

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

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

2. 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
?= 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.

3. If the argument is a string (other than a simple number), EVAL tries to compile it as a SPITBOL expression. By *expression* we mean something which might appear in the subject, pattern, or replacement field of a statement. Only an expression is permitted—not an entire SPITBOL statement:

```
?= 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 with the **code.spt** program:

```
?LOOP   OUTPUT = EVAL(INPUT)                :S(LOOP)
4 * (5 - 2) / 2
6
2 ^ 14 + N
16609
<EOF>
Success
?
```

The program reads a line of input, compiles it, evaluates it, and then displays the result. Of course, each expression you enter must be well-formed according to SPITBOL's syntax rules. In particular, this means there must be blanks around the binary operators.

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

The expressions used with EVAL may return *any* SPITBOL 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)
```

When we write a program, our patterns freeze, once and for all, the order and kinds of data we will be able to recognize. Of course, we have some flexibility, since input data can be used as the argument of a pattern function, but the overall pattern structure—SPAN following BREAK alternating with LEN, etc.—is fixed when the program is written. For most problems, this is satisfactory, but suppose the *very description itself* of what we want to recognize is not known until the program is executed? This problem is not as farfetched as it seems, as the following two examples illustrate.

1. A data base is being compiled for a medical study. A patient will be categorized by an "information string," in which each character position contains a specific "bit" of knowledge. For example, male or female will be encoded as 'M' or 'F', ages 10 to 14 years as 'B', etc., until a composite string has been constructed, such as 'FBCGBIEMDKEAM'.

   Now a program can be designed to allow an analyst to scan for certain cases. In any particular character position, we can look for a certain value (e.g., 'F'), set of values (ANY('DAL')), or "don't care" (LEN(1)). The analyst either enters a SPITBOL pattern directly, or provides parameters to be translated into a pattern by the program. The pattern is compiled with EVAL, and applied sequentially to the patient information strings. Alternatively, all patient data could be combined into one long string, perhaps with a patient number and a suitable delimiter between patients:

```
‘!FBCGBIEMDKEAM,001!MCCAJEAGIBALH,002!…’
```

The pattern could then be applied to *all* patients at once, and matching cases captured using conditional assignment.

2. Most formal languages are defined by a rigorous grammar. Can we write a SPITBOL program to read such definitions and generate the appropriate patterns to recognize statements written in the language? If such a program were possible, we could recognize new languages merely by changing the input file; the SPITBOL program itself would remain unchanged.

   EVAL makes such a program very simple to write, and the distribution disk provides it in file **bnf.spt**. It accepts a grammar definition file written in a standard notation (Backus-Naur Form), and converts those definitions to a series of interlinked, recursive SPITBOL patterns. The program then reads a file of sample language statements, applies the patterns, and tells you if each statement is well-formed according to the rules of the grammar. The entire program—including input and output—is 30 statements long.

The BNF program demonstrates that EVAL's power is useful even if the input data does not conform to SPITBOL syntax.

The CODE function

The CODE function enlarges upon EVAL to compile entire SPITBOL statements. CODE accepts a string argument containing one or more statements to be compiled. Multiple statements are separated by placing a semicolon (;) between each. Statements may be labeled, and can include all the usual components—subject, pattern, replacement, and Goto. Even comment and control statements are permitted.

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

There are two ways to execute the new object code.

1. Transfer to a label which is defined in the new code. For example:

```
*       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, SPITBOL would terminate when execution "fell out of the bottom" of the new code block.

2. 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 ...
```

Direct Gotos may be used with success and failure branches in the Goto field.

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* SPITBOL programs, and makes the division between "code" and "data" far less distinct than in other high-level languages.

The CODE function made the **code.spt** program possible, and produced more powerful scripts for the **eliza** program.

NRETURN

In our discussion of program-defined functions, we learned that a function returns to its caller by transferring to one of the reserved labels RETURN or FRETURN. If the transfer is to RETURN, the function returns a value for use by the caller.

There is a third way that a function may return, and that is by transferring to the reserved label NRETURN. This action says that the function is returning the *name* (i.e., address) of a variable, to which the caller may assign a value. If function STORE were defined as:

```
STORE   STORE = .DUMMY
:(NRETURN)
```

then STORE() could appear in the following contexts:

```
        STORE() = 43
        SUBJECT ? PATTERN . STORE()
```

In either case, the variable DUMMY would receive the result of the assignments.

As the previous example always returns the address of variable DUMMY, there isn't any reason not to forego STORE() entirely, and to just use DUMMY in its place. NRETURN becomes useful when the function returns a different name with each call, or when the function is being called for other side effects it may perform. We'll illustrate both in the remainder of this chapter.

Maintaining a stack

A *stack* is an expandable list of data items. It operates on a last-in, first-out basis, and is useful for storing intermediate results. There are many ways to implement stacks in SPITBOL, including arrays, tables, and program- defined datatypes. We'll use a table with numeric subscripts, because it has no predeclared size limit like an array. We'll also use it as a vehicle to demonstrate a practical application of NRETURN.

Our stack will be stored in a table named STK. STK[0] will contain an index to the last entry in the table. We'll create a PUSH function that allows us to enter items into the stack:

```
*          PUSH(X) – Push item X onto the stack.
*
*          Returns the address of the top stack element.
*
           DEFINE('PUSH(X)')
           STK = TABLE()                                  :(PUSH_END)

*          Increment the index to the first free element.
*          On the very first call, STK[0] is null.
PUSH       STK[0] = STK[0] + 1

*          We'll return the address of this new stack element
           PUSH = .STK[STK[0]]

*          If a value was supplied in the call, store it there.
           $PUSH = X                                :(NRETURN)
PUSH_END
```

NRETURN allows PUSH() to be used in one of two ways. It can be called with an explicit argument, like this:

```
           PUSH(43)
```

or used as the object of assignment, like this:

```
           PUSH() = 43
           'ABCDE' ? LEN(2) . PUSH() 'D' LEN(1) . PUSH()
```

One refinement here is to include the unevaluated expression operator in the last pattern:

```
           'ABCDE' ? LEN(2) . *PUSH() 'D' LEN(1) . *PUSH()
```

Without it, PUSH() is called when the pattern is *first constructed*. In the modified example, the calls to PUSH() are *deferred* until assignment takes place, and new stack entries are allocated *only* if the pattern match succeeds.

To remove items from the stack, we will construct a very conventional POP() function. It doesn't need to use NRETURN:

```
*          POP() – Remove an item from the table STK
*
*          STK is declared in the PUSH() function.
```

```
                DEFINE('POP()')                                    :(POP_END)
*               Return the value currently on top of the stack
POP             POP = STK[STK[0]]
                STK[0] = STK[0] - 1                                :(RETURN)
```

Next we'll use several of the advanced features of this chapter to create a simple parser.

Parsing and Translation

Parsing programs, like **bnf.spt** on the distribution diskette, demonstrate how tangled patterns can recognize complex linguistic constructions. But recognition is only one part of parsing and translation—one needs to *save* the components matched, and *invoke* "action routines" to perform the translation. A stack is useful here because we don't know in advance how large the subject is, or how many components will be matched, or even the order in which they'll be matched.

Arithmetic parsing

To demonstrate how NRETURN, pattern recursion, and EVAL() can simplify programming, we'll examine a short program that parses and evaluates simple arithmetic expressions. These techniques illustrate how information gathered during a pattern match can be systematically stored and acted upon. In this case, we'll directly evaluate the numbers in the expression—with a little more work we could begin to generate code as a compiler would. Interested readers are referred to Chapter 18 of James Gimpel's *Algorithms in SNOBOL4*, for additional information.

Our program will read a line of input, attempt to parse it, and then call action routines to carry out the specified arithmetic. The program begins by defining the grammar of the arithmetic expressions we want to recognize. First are the definitions of integer and real numbers. Note that sub-patterns having the null string as a final alternative effectively make that sub-pattern optional.

```
                integer = span("0123456789")
                exponent = any("eEdD") (any("+-") | "") integer
                real = integer "." (integer | "") (exponent | "") | integer ex-
        ponent
                constant = real | integer
```

Next we provide the patterns to match simple arithmetic expressions. Expressions are built-up from simpler terms and factors. A factor is a numeric constant or parenthesized expression, optionally preceded by unary plus or minus. A term is either two factors joined by the binary multiplication or division operator, or a simple factor. Finally, an expression is either a term plus or minus another term, or just a term. Note the recursive definition: the exp pattern occurs at the highest and lowest levels.

```
                primary = constant . *push() | “(” *exp “)”
                factor = any(“+–”) . *push() *factor . *unary() | *primary
                term = *factor any(‘*/’) . *push() *factor . *binary() | *factor
                exp = *term any(‘+–’) . *push() *exp . *binary() | *term
```

If we were only interested in applying a pattern match to determine the validity of the subject string, the calls to *push() would not be necessary. However, we want to save the matching components on a stack for later evaluation.

When exp is applied to a subject string, “jockeying for position” takes place between the various sub-patterns. When all are properly aligned, conditional assignment takes place, triggering calls to push(), unary(), and binary(). Push() places the various operands and operators on the stack, while unary() and binary() remove them from the stack, and perform the desired operation. The result of each operation will be left on the stack.

The push() and pop() functions were defined in the previous section, and won't be repeated here. The unary() and binary() functions return the name of a dummy variable—they don't care about the string assigned to them. It's just a ploy to use conditional assignment to get them called at precisely the right moment in the pattern match—when the needed operands are on the stack. They use SPITBOL's eval() function to actually perform the arithmetic. An alternate technique is to branch to specific labels for each of the operators found on the stack.

```
            define(“unary()arg,op”)                         :(unary_end)
unary       arg = pop()
            op = pop()
            push() = eval(op arg)
            unary = .dummy                                  :(nreturn)
unary_end
            define(“binary()op,left,right”)           :(binary_end)
binary      right = pop()
            op = pop()
            left = pop()
            push() = eval(left “ ” op “ ” right)
            binary = .dummy                                 :(nreturn)
binary_end
```

Having defined the functions and the patterns, all that remains is to write a short test loop that reads a line of input and applies the pattern. Note that the use of pos(0) and rpos(0) guarantees that the entire subject string will be used for the match.

```
            &trim = 1
loop        line = input                              :f(end)
            line ? pos(0) exp rpos(0)                       :f(error)
            output = pop()                                  :(loop)
error       output = “Bad input, try again”           :(loop)
end
```

Using NRETURN and deferred evaluation in this manner can only be called an obscure programming technique. However, it does work, and should give you a glimpse of SPITBOL's power and flexibility.

Chapter Summary

Pattern functions	ARBNO(P)	Arbitrary number of occurrences of argument pattern
	FENCE(P)	Bypass alternatives in P when backing up
Unary operators	~	Negation, inverts success/failure
	?	Interrogation, evaluates expression for success/failure result only
Primitive patterns	ABORT	Entire pattern match fails
	BAL	String balanced with respect to parentheses
	FAIL	Subpattern fails, alternatives will be sought
	FENCE	Initially succeeds, but fails during backtracking
	SUCCEED	Matches null string and succeeds
Built-in functions	APPLY	Indirect reference call to function
	CODE	Compile SPITBOL program statements
	CONVERT	Convert data types
	ENDFILE	Close file
	EVAL	Compile and evaluate expression
	ITEM	Indirect reference to array or table element
	LPAD	Pad on left to fixed width
	REWIND	Rewind file
	RPAD	Pad on right to fixed width
	RSORT	Sort array or table, descending order
	SET	Position file
	SETERROR	Trap execution errors
	SORT	Sort array or table, ascending order
	SUBSTR	Extract substring by character positions
Labels	NRETURN	Return by name from function

Chapter 10
Debugging

Debugging and Tracing

You are probably well aware of the diversity of potential errors that can occur in 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 SPITBOL program is not fundamentally different than debugging programs written in other languages. However, SPITBOL'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.

Compilation errors

Compilation errors are the simplest to find. When SPITBOL compiles a program and encounters errors, it produces messages which give the location of the error, an error number, and a brief explanation of the error. These messages normally go to the screen.

Note that there are some text editors that can be configured to compile from within the editor (i.e., Brief, Vedit, Multi-Edit, etc.). When errors occur, the editor will automatically read an error file, find error messages, and position the cursor at each offending line while displaying the appropriate error message. If you have such an editor, specify a SPITBOL command line that writes the errors to the error file that the editor will read:

```
spitbol –e –o=errfile progname
```

When SPITBOL is run directly from the command line, error messages appear on the screen and the listing file, if any. The offending line is displayed, then a pointer to the error, followed by another line with information about the error. If there is more than one error in a line, only the first error is identified.

Consider this offending line:

```
        TERMINAL = CNT+ 1 ‘ items counted."
```

There are two problems. The blank before the binary plus operator is missing after the variable CNT. And the string “ items counted” is not properly framed by matched quotes.

When we compile a program with this statement, an error message like this will appear:

```
program.spt(8,12) : Error 223 — Syntax error: Invalid use of operator.
```

The message gives the name of the source-code file, followed by the line number and character position where the error occured, and then an explanation of the error.

Note that it missed the second error in this statement; we'll ignore it too for the moment. We'll fix the first one, so that we have:

```
        TERMINAL = CNT + 1 ‘ items counted."
```

When we compile and run this, we'll get this error message:

```
program.spt(8,20) : Error 232 — Syntax error: Unmatched string
quote.
```

After we fix the second error by making both string delimiters either ‘ or “, this statement will compile without error.

Execution errors

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

With the first type of execution error, SPITBOL will send a message to the screen, much as it does with compilation errors.

Consider this short program:

```
        &TRIM = 1
        &ANCHOR = 1
        TERMINAL = ‘Program is starting.’      :(NONESUCH)
END
```

The obvious error is that the Goto, NONESUCH, doesn't exist. When we compile and run this, we get this message:

```
program.spt(3) : Error 038 — Goto undefined label
```

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 where the incorrect data was created. Placing output statements in your program, or using the trace techniques described later in this chapter, will usually find such errors.

Here are some common errors to look for first:

1. Setting keywords &ANCHOR and &TRIM improperly. We may have written a program with anchored pattern matching in mind, but let an unanchored match slip in inadvertently. In unanchored mode,

   ```
   'BELL' ? 'E'
   ```

   will succeed, whereas it will fail in anchored mode.

   Forgetting to set &TRIM to 1 causes any trailing blanks to remain on input lines, which can interfere with pattern matching.

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

   ```
   PUTPUT = LINE1
   ```

   creates a new variable and assigns LINE1 to it.

   This kind of spelling error is relatively easy to find. By setting the keyword &DUMP to either 1, 2 or 3, you will get a list of variables after you compile and run your program.

   If &DUMP = 1, the dump will list only variables with non-null values, while &DUMP = 2 will show the non-null elements of tables, arrays, and user-defined datatypes. &DUMP = 3 will include null-valued variables and elements, and statement labels. You can examine the list for an unexpected name.

3. 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.

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

   ```
   X = Y –Z
   ```

   concatenates Y with the element –Z. If Y = 3 and Z = 1, then the above would assign the string "3–1" to X, when you probably expected X to equal the integer 2.

5. Confusion occurring when a variable contains a number in string form. When used as an argument to most built-in 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).

6. 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)
```

7. Unexpected statement failure, with no provision for detecting it in the Goto field. For example, we declare an input file early in the program:

```
          INPUT(.IN, 1, 'nonesuch')
```

   Some lines later, you attempt to read from that file with a statement like this:

```
          LINE = IN                       :F(EOF)
```

   Surprisingly, the statement never fails, and LINE is always set to the null string. Here the error was really up in the INPUT statement, which failed because there wasn't any file called **nonesuch**.

   The solution is to always test for failure when using a function that might fail:

```
          INPUT(.IN,1,'nonesuch')                     :F(ERR)
          …
ERR       TERMINAL = 'The input file could not be found':(END)
```

   You can also include a –NOFAIL control statement at the beginning of your program (see Chapter 14, "SPITBOL Statements"). SPITBOL will alert you when a statement lacking a conditional Goto fails.

8. 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)
```

9. Using operators such as alternation (|) and conditional assignment (.) for purposes other than pattern construction. Using them in the subject field will produce a 'Pattern match left operand is not a string' 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'
```

Such a construction could best be handled with the alternative evaluation construction described in Chapter 7, "Additional Operators and Datatypes":

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

10. 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) = '*'
```

which is not very efficient, or the more desirable:

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

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

```
        NTH_CHR = POS(*(N - 1)) LEN(1) . CHR
```

will place the Nth subject character in variable CHR. 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).

12. Failing to jump around a function body, after executing the definition. Here's a short function definition in good form:

```
        DEFINE('CYC(S,N)')
        CYC_PAT = LEN(*N) . FRONT REM . REAR      :(CYC_END)
CYC     S ? CYC_PAT = REAR FRONT
        CYC = S
        :(RETURN)
CYC_END
```

That will work fine, because after making the necessary definitions, control is passed to the label CYC_END with a Goto. If that Goto were omitted, then control would go straight on to the line with the CYC label which would then transfer to RETURN. Since we're attempting to return from a function that has never been called, you'll get this error message:

```
        program.spt(4) : Error 242 — Function return from level zero
```

The message is a little cryptic if you're not thinking along those lines. When you get such a message, it probably means that your program fell into a function, instead of calling it properly.

13. Omitting quotes around a string literal. This INPUT statement does not open **datafile.txt**:

```
        INPUT(.IN, 1, datafile.txt)
:F(ERR)
```

Instead, it is passing the INPUT function the *variable* datafile.txt, which probably contains the null string as a file name.

Simple debugging

These simple methods should find a majority of your bugs:

1. Set keyword &DUMP to 1, 2, or 3 for a partial or full dump to the screen after your program runs. The dump can be sent instead to a file by using the –o=filename command-line option. Dumps can also be produced at any time during execution by calling the built-in DUMP function. DUMP(*n*) corresponds to &DUMP = *n*.

   One useful trick is to set &DUMP = 2 at the beginning of your program. Then just before the END statement, transfer to a statement that sets &DUMP = 0. That way, if your program terminates unexpectedly, you'll get a full dump. But if your program terminates normally, you won't be getting a dump that you don't need.

2. Use keyword &STLIMIT to end execution after a fixed number of statements are executed. The default value of &STLIMIT is 2,147,483,647. Setting it to a low value, such as 100 or 200, with &DUMP set to 1 or 2, can tell you if you're stuck in a loop somewhere.

   After SPITBOL executes the number of statements specified in &STLIMIT, this appears on the screen:

```
            program.spt(123) : Error 244 — Statement count exceeds
value
                                                    of STLIMIT keyword
```

3. Use the Goto :F(ERROR) to detect unexpected failures and data errors. Do not define the label ERROR—SPITBOL will stop if an attempt is made to transfer to label ERROR.

4. Assign values to TERMINAL to monitor data values. Use immediate assignment and cursor assignment (to TERMINAL) to observe the operation of a pattern match.

5. Set keyword &PROFILE to 1 to obtain an execution profile of how much time is spent in each statement. Are the numbers reasonable? Is the program hung anywhere?

More subtle errors can be pinpointed using SPITBOL's trace facility, described next.

Execution Tracing

Tracing the flow of control and data in a program is usually the best way to solve difficult problems. SPITBOL 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.

Function tracing

Keyword &FTRACE can be set nonzero to produce a trace message each time a program-defined function is called or when it 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.

To use this feature, you place a statement like this near the beginning of your program:

```
          &FTRACE  =  1000
```

Trace messages are written to standard output, (normally, the screen, but it can be redirected to a disk file on the command line). A typical trace messages looks like this:

```
****12****** SHIFT(‘SKYBLUE’,3)
****37******  RETURN SHIFT = ‘BLUESKY’
```

The first number in asterisks is the statement number where SHIFT was called. The next one is the statement number where SHIFT returned.

To interpret these numbers, it will be necessary to produce a program listing by using the –l option on the command line, which creates **program.lst** from **program.spt**. The statement numbers provided in the listing correspond to the statement numbers in trace messages. (It is important to remember that there is usually a difference between line numbers and statement numbers.)

When functions are nested—called from within other functions—the trace messages provide a visual indication of the nesting depth:

```
****37****** SHIFT(‘skyblue’,3)
****30****** I UCASE(‘bluesky’)
****33****** II COUNT(‘blue’)
****22****** II RETURN COUNT = 4
****25****** I RETURN UCASE = ‘BLUESKY’
****38****** RETURN SHIFT = ‘BLUESKY’
```

Here, the successive capital I’s indicate the function call depth. The depth is also available to your program through the &FNCLEVEL keyword. It is zero when your program begins execution, and is incremented by one when a function is called, and decremented by one when a function returns.

Selective tracing

Keyword &TRACE will also produce trace messages when it is set non-zero. SPITBOL will perform as many traces as are specified, so that

```
        &TRACE  =  10
```

would allow 10 traces.

However, the TRACE function must be called to specify what is to be traced. Tracing can be selectively ended by using the STOPTR function. In its simplest form, the TRACE function call looks like this:

```
        TRACE(name,  type)
```

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 particular elements 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. If omitted, a 'VALUE' trace is assumed. The full set of possibilities for type is:

'A' or 'ACCESS'	Produces a trace every time the named item is referenced. If you say X = N, that does not change the value of N, but it is a reference—an "access" to N.
'V' or 'VALUE'	Traces the value of a variable whenever it is the object of an assignment. Assignment statements, as well as conditional and immediate assignments within pattern matching will all produce trace messages.
'K' or '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, and &STCOUNT may be traced. If &STLIMIT is negative, &STCOUNT may not be traced.
'L' or 'LABEL'	Produce a trace when a Goto transfer to statement name occurs. Flowing sequentially into the labeled statement or calling a function that begins with the label oes not produce a trace.
'C' or 'CALL'	Produce a trace whenever the function is called.
'R' or 'RETURN'	Produce a trace whenever the function returns.
'F' or 'FUNCTION'	Produce a trace whenever the named function is called or when it returns. This combines the preceding two types.

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)
```

You might want to experiment with tracing in a short program:

```
        &TRACE = 1000
        TRACE('N', 'VALUE')
        N = 1
LOOP    TERMINAL = CHAR(64 + N)
        N = LT(N,10) N + 1
:S(LOOP)
END
```

If standard output is not re-directed, you'll see this:

```
****3******* N = 1
A
****5******* N = 2
B
****5******* N = 3
C
 …
****5******* N = 10
J
```

where you have the statement number and the value of N from the TRACE function, each followed by the characters generated by your program.

Program trace functions

Normally, each trace action displays a descriptive message, with statement number and the affected variable or function, such as:

```
****371***** SENTENCE = 'Nancy ran to town.'
```

Instead, we can instruct SPITBOL to call our own program-defined function. In it, we can check the variable (or other conditions), and only produce a message if an exceptional situation exists. To do so, we need to use the full form of the TRACE function:

```
        TRACE(name, type, tag_string, function)
```

We define the function to carry out the trace action 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', 'Variable COUNT', .TRFUN)
        DEFINE('TRFUN(NAME,TAG)TEMP')       :(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 to TRACE, it too is passed to your trace function. Its use is entirely optional—use it to pass additional identifying information to your trace function, or omit it.

Our trace function, TRFUN, will be called every time a new value is assigned to COUNT. It's a simple matter then to verify that all is well, and just return, or to issue an alert if a problem is found.

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

```
TRFUN   TEMP = &LASTNO
        GE($NAME,0)
                     :S(RETURN)
        TERMINAL = TAG " negative in statement " TEMP    :(END)
TRFUN_END
```

The first statement of the function captures the number of the *last statement executed*—the statement where the assignment occurred. We then check the variable being traced, $NAME (which will be COUNT in this case), and return if it is satisfactory. If it is negative, we print an error message and stop the program. The error message will pinpoint the statement where COUNT was set negative. You'll need a program listing to find the actual statement corresponding to that statement number.

Although this trace function could have accessed the variable COUNT directly, we chose to access the variable and an identification string indirectly through the arguments NAME and TAG. This allows us to use the same function with several variables that we might want to monitor for the same exception condition:

```
        TRACE(.COUNT, 'VALUE', 'Variable COUNT', .TRFUN)
        TRACE(.N, 'VALUE', 'Variable N', .TRFUN)
```

Notice that our trace function used a new keyword, &LASTNO. There are several such keywords that allow us to use trace functions effectively:

&LASTNO	The statement number of the *previous* SPITBOL statement executed.
&STCOUNT	The total number of statements executed. This keyword is incremented by one as each statement begins execution. However, if &STLIMIT has been set negative for unlimited execution, &STCOUNT is not incremented.
&ERRTEXT	Error message text of the last execution error.
&ERRTYPE	Error message number of the last execution error. A complete list of error message numbers and their associated text is provided in Appendix D.
&ERRLIMIT	Number of nonfatal execution errors allowed before SPITBOL will terminate.

The first four keywords are continuously updated by SPITBOL as a program is executed.

When a program-defined 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 a variable slows down references to it alone, but there is no general execution penalty by having tracing enabled, as there is in other versions of SNOBOL4.

Tracing keyword &STCOUNT will call your trace function before *every* program statement is executed. To do so, use the call:

```
TRACE('STCOUNT', 'KEYWORD', , .MYFUN)
```

If you set keyword &ERRLIMIT non-zero, then you may trace keyword &ERRTYPE to trap nonfatal execution errors. When an error occurs, SPITBOL stores the error number in &ERRTYPE, and that store operation is traceable. A better method is to use the SETEXIT function described below.

Finally, we would like to note that some programmers use the function trace mechanism as a standard programming technique. That is, they are not debugging a program, but instead deliberately plan on having SPITBOL invoke a designated function when some variable is accessed or modified, or a particular function called. For example, tracing an output-associated variable might invoke a line-counting function that inserts a page heading after every 60 lines of output.

This can produce pretty obscure programs, but is interesting because it reveals how SPITBOL's generality can be exploited by the clever programmer.

SETEXIT function

SPITBOL provides an additional debugging mechanism, one not found in standard SNOBOL4. The SETEXIT() function allows interception of execution errors, or compilation errors when using with the CODE or EVAL functions. When an error occurs, the system transfers to a label of your choosing, provided that the value of &ERRLIMIT is non-zero.

Your program can transfer to the special labels ABORT and CONTINUE after processing the error. ABORT causes error processing to resume as though no error intercept had been set. CONTINUE causes execution to resume by branching to the failure exit of the statement in error.

The SETEXIT function can also be used to trap a user attempt to stop program execution by typing the system interrupt key (typically control-C, DEL, break, etc.). Programs may need to perform last-minute clean-up and file closings before terminating. Without such a mechanism, a user interrupt would stop the program immediately, without flushing file buffers, etc.

To accomplish this, a user interrupt will generate error number 320. If &ERRLIMIT is non-zero, this "pseudo-error" will invoke the SETEXIT function declared by the program. The special label SCONTINUE can be used by the error-trapping function to resume execution precisely at the point where the main program was interrupted. Unlike CONTINUE, SCONTINUE does not cause the interrupted statement to fail.

See Chapter 19, "SPITBOL Functions" for additional information on using SETEXIT.

Chapter Summary

Built-in functions	DUMP(I)	Produce partial dump if I is 1, and full dump if I is 2.
	SETEXIT(label)	Intercept execution error
	STOPTR(n,t)	Stop trace
	TRACE(n,t,s,f)	Trace variable, keyword, function, or label
Keywords	&ERRLIMIT	Number of execution errors permitted
	&ERRTEXT	Text of most recent error
	&FTRACE	Enable function tracing
	&LASTNO	Line number of previous statement executed
	&STLIMIT	Number of statements allowed to execute
	&TRACE	Enable TRACE function tracing
Traceable keywords	&ERRTYPE	Error number of most recent error
	&STCOUNT	Number of statements executed
Reserved labels	ABORT	Normal error processing after SETEXIT
	CONTINUE	Resume with statement failure after SETEXIT
	SCONTINUE	Resume interrupted statement after SETEXIT

Chapter 11
Concluding Remarks

What to do next

While this tutorial has covered a lot of ground, it omitted some fine points and features of the language. We suggest you work with the material presented—it's more than adequate for most programming needs.

Chapter 13, "Running SPITBOL," in the reference section explains all of the options available when running SPITBOL. It would be good idea to read it now.

Once you're proficient, try reading the reference section from beginning to end. You'll find many other language features described there.

As a course of study of advanced SPITBOL programming techniques, we heartily endorse James Gimpel's *Algorithms in SNOBOL4*. We thought so much of this book that when John Wiley & Co. let it go out of print, we acquired the rights and reprinted it ourselves. It's 500 pages of ingenious code and algorithms that will make you see the SNOBOL family in a new light. In addition to over 140 sample programs, a formal theory of pattern-matching is developed.

SPITBOL's future

For much of this tutorial we've been concerned with the detailed mechanics of pattern-matching—the functions, primitive patterns, and datatypes involved when applying a pattern to a character string. SPITBOL provides so many primitive functions and operations that it's easy to get lost in the forest. Let's step back and consider SPITBOL's larger significance.

It would be a mistake to think of SPITBOL 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[7], SPITBOL can be made to emulate LISP, and go well beyond it, using pattern-matching, backtracking, and associative programming. Others have noted that it is fairly simple to write a LISP interpreter in SPITBOL, but the converse doesn't hold. Researchers would do well to employ SPITBOL's power and flexibility in the study and development of artificial intelligence.

SPITBOL'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 SPITBOL patterns. For example, we know of one user who maps two-dimensional images into character strings for recognition.

In the past, use of SPITBOL 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.

Historically, the SNOBOL family, of which SPITBOL is a member, has appealed to two divergent classes of computer users. One group is computer professionals, especially those doing systems work, who use SPITBOL for prototyping and "quick and dirty" jobs.

The other group is the academic community, where people who aren't professional programmers have work that has to be done on a computer, primarily text and data analysis. Compared to programs which do the same things in other languages, SPITBOL programs are much shorter, which makes them both easier to write and easier to understand.

SPITBOL is the linguistic researcher's preferred tool, and the systems engineer's secret trick when something needs to be done in a hurry. But the SNOBOL family has somehow missed the vast middle ground of computer users.

A language with so much ease, power, and flexibility should be useful to many others. Can SPITBOL bring new insights to your problems? Are there general applications for SPITBOL's abilities? What do you do now that you can do better and faster with SPITBOL? And what can you discover with SPITBOL's power at your fingertips?

PART III
Reference Manual

Chapter 12

Reference Introduction

This reference section describes the SPITBOL system. It will tell you how to create and run SPITBOL programs. It catalogs all the standard language features, as well as extensions for the Catspaw versions of SPITBOL which run on 32-bit Unix™ , MS-DOS and Windows computers. We have tried to make all features standard across all systems. Where differences exist, they are noted in the text. Consult the tutorial section for illustrative uses of various functions and operators.

Catspaw SPITBOL is a full implementation of the powerful SNOBOL4 programming language. It has all the features of other SNOBOL4 systems, as well as many useful extensions.

Compatibility with other SPITBOL systems is achieved by basing this Catspaw product on the macro implementation used on such systems as the CDC 6600 and DEC VAX. All SPITBOL string- and pattern-matching facilities are now available in a desktop environment.

The earliest versions of SPITBOL compiled directly to executable machine language. However after 1976, a new version of SPITBOL achieved greater portability by compiling to an intermediate language of indirectly threaded code. A highly optimized run-time system then interprets the thread of code and data objects.

Because the language requires run-time type checking and conversion, surprisingly little efficiency is sacrificed by using an interpreter. On average, the interpreted version of SPITBOL (*Macro SPITBOL*) is only 20% slower than the fully-compiled systems (*SPITBOL 370*).

Language Background

SPITBOL has an odd name, one that some people might have trouble mentioning in polite company. There's a long story behind that name, which is supposed to be an acronym for "**SP**eedy **I**mplemen**T**ation of sno**BOL**." SNOBOL, in turn, is an acronym for "**S**tri**N**g-**O**riented sym**BO**lic **L**anguage," and it is with SNOBOL that the SPITBOL story starts.

SNOBOL

In 1962, several researchers at Bell Telephone Laboratories were applying computers to problems such as factoring multivariate polynomials and symbolic integration. Available tools were the Symbolic Communication Language (SCL), an internal Bell Labs product for processing symbolic expressions, and COMIT, designed for natural-language analysis. Both proved inadequate, and frustration led the researchers to design 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 considerable discussion, in part to poke fun at the then current trend toward tortuous acronyms in computer names. No one at Bell Labs was overly enthusiastic about the name SNOBOL, but since the language was a small, internal development, no one expected it to face public scrutiny.

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 outside the Labs on such machines as the IBM 7094, CDC 3600, SDS 940, 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. The other significant difference is that SNOBOL4 had so many new features that it was no longer just a string manipulation language—it had become a general-purpose programming language.

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. By 1970 it was available on nine different mainframe systems. Currently, it is available on most large- and medium-scale computers, and the

IBM PC family. (Contact Catspaw, Inc. for the names of SNOBOL4 suppliers.)

SPITBOL

SPITBOL is essentially a dialect of SNOBOL4. It has the same syntax and an almost identical repertoire of functions, which perform in the same ways. If you know one, you know the other. SPITBOL isn't quite as flexible as SNOBOL4 (which, for instance, allows for redefinition of all operators), but gains in internal efficiency, so that the same program typically runs six or eight times faster under SPITBOL than under SNOBOL4.

SPITBOL was devised in 1970 for the IBM 360 by Robert B. K. Dewar and Ken Belcher, then at the Illinois Institute of Technology. Because of the great deal of custom work involved, only two implementations were ever completed, these being the IBM 360/370 and the Univac 1100. Dewar maintains that he was unaware of what a "spitball" was until after his implementation, along with the name, was already in circulation. (Readers unfamiliar with the American game of baseball need to know that a "spitball" is the illegal wet treatment of a baseball, and so connotes an unfair, or underhanded action. The original SPITBOL 360 broke all rules of reasonable assembly-language programming in its quest for speed.)

In the summer of 1974, Dewar went to England to implement SPITBOL for the ICL 1900, a British mainframe. Out of time at the end of the summer, he realized that designing a new SPITBOL system for each type of computer was too time-consuming. Returning the following year, he discarded the previous year's work, and with A. P. McCann of the University of Leeds, rewrote SPITBOL in a generic assembly language he created, Minimal. That change in SPITBOL was announced in 1976; this method of implementing the language is known as Macro SPITBOL, and has been used in all implementations since, including MaxSPITBOL.

Besides SNOBOL4 and SPITBOL, the SNOBOL family of computer languages includes FASBOL, SITBOL, SNOBAT, and SNOBOL4B, as well as preprocessors like SNOCONE and Rebus. Icon, a relatively new language which is gaining in popularity, is in many respects a descendant of SNOBOL and bears many similarities, especially the concepts of "success" and "failure" for operations. That's understandable, since one of the driving forces behind Icon is Ralph Griswold, who was also one of the creators of SNOBOL.

SPITBOL is in wide use on mainframe and minicomputer systems. For desktop computers, Dewar has implemented SPITBOL for the IBM PC family. Catspaw has developed SNOBOL4 for the IBM PC family (SNOBOL4+), and SPITBOL for the Apple Macintosh™ (MaxSPITBOL), for Intel 80386 architectures, for Motorola 680x0-based Unix systems, and for RISC systems such as Sun Microsystem's SPARC, IBM's RS/6000, and MIPS R-3000.

Chapter 13
Running SPITBOL

TThis chapter provides information on running SPITBOL. It describes:

- the command line used to invoke SPITBOL
- the variety of options available to alter SPITBOL's behavior
- input and output from standard files
- use of environment variables to provide additional information to SPITBOL
- information about save files and load modules

Command Line

The general form of the command line to execute SPITBOL is given below. Items within [ ] are optional, and may be omitted.

```
spitbol  [options]  ifile[.spt  or  .spx]  [args]
```

where

ifile A list of one or more files from which a program is read. The files are read sequentially until an END statement is found. If no **ifile**s are present, the compiler will present a short summary of the options available.

If **ifile** cannot be opened, and the name provided did not have a file name extension, SPITBOL tries to open it with extensions **.spt** and **.spx** corresponding to possible source and save file names.

After all **ifile**s are read in order, the system will read from standard input. An isolated "–" (hyphen) may be placed in the **ifile** list to force standard input to be read at that point instead of a file. When end-of-file (control-D or control-Z) is signaled, reading resumes with the next **ifile**. The source pro-

	gram consists of the sequential concatenation of all files read up to the END statement.
	If a program **ifile** contains data following the END statement to be read by the program, use the –r option described on the next page to force INPUT to continue reading the **ifile** instead of switching to standard input.
options	Options affect the compilation and runtime behavior of SPITBOL, and can provide file names for I/O. They are preceded by a minus sign, and multiple options may be grouped together. Options *must* precede the **ifile** list, if any.
args	Arguments on the command line after the **ifile** that contained the program's END statement are ignored by SPITBOL. HOST(3) returns the index of the first unused argument, and HOST(2, i) retrieves argument i. These can be easily accessed using the **args.inc** file described in Appendix A, page 259.

Command Line Options

Options appear before any input file names:

```
spitbol options ifiles args
```

A brief summary of all options is produced by typing:

```
spitbol
```

Options requiring a numeric argument may have a "k" or "m" appended to the number to indicate units of "kilo" and "mega" respectively. That is, "8k" can be written in place of 8192, or "8m" instead of 8388608. All numbers are decimal, and may not have embedded punctuation.

Compilation and execution

–b	suppress SPITBOL's two-line screen sign-on message (also recorded in any save file or load module created)
–f	don't fold lower-case names to upper case
–k	run program even with compilation error (like –ERRORS control statement)
–n	suppress execution (like –NOEXEC control statement)

Input/output

Options taking a file name may be separated from the name by a colon, equal sign, or blank. A single hyphen "–" may be used anywhere in place of a file name to represent standard input or output.

–e don't send error messages or trace output to the screen; send to standard output instead, which may then be redirected:

```
spitbol –e ifiles >trace.dat
```

–o=ofile

any program listing, statistics, error messages or dumps are written to this file. If no extension is present, SPITBOL appends **.lst**. Data assigned to variable OUTPUT is not affected by this option, and continues to go to standard output.

To write to standard output and simultaneous redirect standard output, use a hyphen for the file name:

```
spitbol –o=– ifiles >outfile.dat
```

–r	INPUT variable should begin reading data at the line following the END statement in the last source file compiled. If no lines follow the END statement, INPUT signals end-of-file immediately. Normally SPITBOL ignores anything beyond the END statement.
–*n*=file	associate file with I/O channel number *n*. INPUT/OUTPUT functions with the same channel number (second argument) may omit the file name (third argument). SPITBOL will use the file specified on the command line for this channel. File processing options may appear in square brackets after the file name (see the description of the INPUT function in Chapter 19, “SPITBOL Functions,” for processing options). For example,

```
–23=infile.dat[–r10]
```

associates **infile.dat** with I/O channel 23. The file can be read in binary mode, using 10-character records, after executing:

```
INPUT(.IN, 23)
```

Listing and statistics

If a listing option (–a, –l, –p, –z) calls for a listing, but no **ofile** is provided, the listing is written to the first **ifile** with its extension changed to **.lst**.

–a	like –lcx. Produce listing and statistics
–c	generate compilation statistics
–g*n*	number of lines per page for listings (default –g60)
–h	suppress SPITBOL version identification string and date/time that normally appear at the start of a listing
–l	produce normal program listing
–p	produce listing with wide titles for printer
–t*n*	page width in characters for listings (default –t120)
–x	generate execution statistics
–z	produce listing with form feeds

Memory control

–d*n*	size (bytes) of maximum allocated dynamic area (the *heap*) (default –d64m, or 64 megabytes). This option is not available in SPITBOL-8088, where the maximum dynamic area size is fixed at 56 kilobytes.
–i*n*	number of bytes by which the dynamic work space (heap) is enlarged each time more memory is required (default –i128k). This is also the minimum starting size for the heap. All program object code and runtime data is held in the heap. Under SPITBOL-8088, the default heap starting size is 56k, and no heap expansion is possible.
–m*n*	maximum size (bytes) of any created object (default –m4m, or 4 megabytes for all systems except SPITBOL-8088, where the default maximum object size is 9,000 bytes). This is the size of the longest individual string allowed, or the largest individual array. Keyword &MAXLNGTH is assigned this value when execution begins. Because of efficiency decisions made by SPITBOL's designers, the number specified for the maximum object size must be *numerically less than* the starting memory address of SPITBOL's work space. If it is not, SPITBOL ignores (and is not able to use) any memory between the low end of the work space and this value. This is not a practical problem under systems with virtual memory because any unused region will likely be swapped out. MS-DOS and OS/2 SPITBOL-386 is linked such that the heap is offset to a virtual address at 4 megabytes.
–s*n*	maximum size (bytes) of stack space (default –s32k, except in SPITBOL-8088, where it is 2,830 bytes)

Parameter string

–u string	string retrievable by program with HOST(0). String must be quoted if it contains any blanks or other command-line delimiters. A better way to process arguments is through the **args.inc** include file described in Appendix A, page 259.

Save files and load modules

–w	create a stand-alone load module after compilation. When operating under Unix, the load module is named **ifile.out**. Under SPITBOL-386, the load module is named **ifile.exe**. The first **ifile** name is used.
–y	create **ifile.spx** as a save file after compilation. The first **ifile** name is used.

Either option can be combined with –n to suppress execution after the file is written.

Help

–?	displays a brief summary of the available options.

Defaults

By default, program listings and statistics are generated only when requested by options. All variable names are folded to upper-case during compilation and execution. Other options are system dependent:

For all systems except SPITBOL-8088:

```
–m4m  –s32k  –i128k  –d64m
```

For SPITBOL-8088:

```
–m9000  –s2830  –i56k  -d56k
```

Examples

Compile and execute program **abc.spt**. INPUT from standard input and OUTPUT to standard output:

```
spitbol  abc
```

As above, but produce compilation and execution statistics:

```
spitbol  –cx  abc
```

Compile **abc.spt**, producing save file **abc.spx**, with flag set to suppress SPITBOL sign-on message when the save file is run. Run **abc.spx**:

```
spitbol  –ybn  abc
spitbol  abc.spx
```

Note that it was necessary to specify the **.spx** extension, since given the simple name **abc**, the system would read **abc.*spt*** by default.

Compile program split across files **prog1.spt** and **prog2.spt**, produce full listing to file **prog.lst**, and suppress execution:

```
spitbol  –pn  –o=prog.lst  prog1  prog2
```

Compile and run **abc.spt**, and make string “FILE1, FILE2" available to the running program via the HOST(0) function:

```
spitbol  –u  “FILE1,  FILE2"  abc
```

Standard I/O and Redirection

Standard input is the file that programs read by default. Standard input, file descriptor 0, may be accessed in two ways. In the absence of the –r option, the INPUT variable will read from standard input when execution begins. Alternately, standard input may be attached by calling the INPUT *function* with “[–f0]” as the third argument. This is described more fully in the description of the INPUT function.

Standard output, file descriptor 1, is associated with variable OUTPUT when program execution commences. Standard output is produced only by assignments to OUTPUT and by error messages. Standard output may also be accessed by invoking the OUTPUT *function* with “[–f1]” as the third argument.

Normally, standard input reads from the keyboard, and standard output writes to the screen. Standard input or output may be redirected by conventional methods. Input may be read from a file or pipe, and output written to a file or pipe. For example, to execute **prog1.spt**, and have INPUT automatically associated with **file1** instead of the keyboard, use

```
spitbol  prog1  <file1
```

In the next example, INPUT reads the results of the MS-DOS or OS/2 dir command, and OUTPUT is written to **FILE2**:

```
dir  |  spitbol  prog1  >FILE2
```

Standard error, file descriptor 2, is always associated with variable TERMINAL, and provides read and write access to the keyboard and screen, regardless of redirection which may have occurred on the command line.

Environment Variables

MS-DOS, OS/2, Windows and Unix all provide environment strings or "shell variables" to a program. SPITBOL uses these strings in three ways:

- To specify sub-directories that will be searched for include files and external functions
- To associate file names with SPITBOL I/O channels
- To provide other information to your program via function HOST(4)

The examples that follow alternate between MS-DOS (or OS/2), Berkeley Unix and System V Unix syntax. With the appropriate syntax, the examples work on all systems.

Directories

Search directories for include files and external functions

The environment variable SNOLIB may be set to a list of sub-directory paths to be searched for files given in an –INCLUDE control statement or in the LOAD function. The name SNOLIB must appear in upper-case letters. Under MS-DOS, Windows and OS/2, paths are separated by semicolon:

```
set  SNOLIB=c:\spitbol\library;d:\spitbol\includes
```

Note that there is no space on either side of the equal sign. For the Berkeley Unix C-shell, paths are enclosed in quotes and parentheses and separated by spaces:

```
setenv  SNOLIB  "(/spitbol/library  /spitbol/includes)"
```

For the System V shell, separate paths with colon, and export the SNOLIB variable:

```
SNOLIB=/spitbol/library:/spitbol/includes
export  SNOLIB
```

SPITBOL looks first in the current sub-directory for the file. If not found there, the SNOLIB sub-directories are prepended one-by-one to the file name until the file is found, or all attempts fail.

I/O files

Associating Files with I/O Channels

Environment variables can associate file names with SPITBOL I/O channels. If your program's INPUT or OUTPUT function omits the file name, the environment is searched for the string form of the I/O channel (second argument). If found, the name supplied is used in the function call. File processing options may appear in square brackets after the file name (see the description of the INPUT function in Chapter 19, "SPITBOL Functions," for processing options). For example, under MS-DOS or OS/2,

```
set 23=infile.dat[–L500]
```

associates **infile.dat** with I/O channel 23. The file will be opened in line mode, with a 500-character record length, upon execution of the statement

```
INPUT(.datasrc, 23)
```

Unlike its command line equivalent, this technique works just as well for the non-numeric channels permitted by SPITBOL. For example, under the Berkeley C-shell,

```
setenv XYZ poetry.txt
```

Within a SPITBOL program, the file **poetry.txt** would be opened by

```
INPUT(.IN, 'XYZ')
```

If the same I/O channel appears both as an environment variable and on the command line, the command line takes precedence. Either of these methods are useful ways to avoid hard-coding file names into programs.

HOST access

Accessing Environment Strings from your Program

The HOST function provides a way for programs to see if a particular string is in the environment. Given the System V Unix shell commands:

```
DIRECTION=NorthEast
export DIRECTION
```

The program statement

```
HEADING = HOST(4, "DIRECTION")
```

will set HEADING to "NorthEast". The HOST function fails if the desired string is not found in the environment. String names must have the same case as they have in the environment (always upper-case for MS-DOS, Windows and OS/2).

Save Files and Load Modules

SPITBOL provides two methods of saving a compiled program.

Save files

A "save file" records the program object code and data in SPITBOL's heap as well as other information necessary to resume the program. Save files are compressed when written, resulting in relatively compact files. Compression also obscures any data strings and variable names in the heap, making the save file unintelligible to the casual browser.

Save files can be created immediately after program compilation and prior to execution by specifying the –y command-line option. The save file is written to a file with the same name as the first source file, but with extension **.spx**.

Save files may also be written after the program has begun execution, perhaps after a lengthy initialization is completed. The program executes function EXIT(–3,filename), which writes a save file with the given name, or **a.spx** if the file name is omitted. After writing the save file, SPITBOL terminates execution. (Use EXIT(–4,filename) to continue execution after writing the save file.)

A save file is loaded and resumed merely by supplying its name in place of a source file when starting SPITBOL:

```
spitbol  a.spx
```

The **.spx** extension can be omitted if there will be no confusion with a file named **a.spt**.

Save files are tied to a specific version of SPITBOL. Subsequent versions of SPITBOL may render existing save files obsolete, and they will have to be regenerated to run with the new version.

Load modules

A "load module" is an executable file containing the *entire* SPITBOL system in addition to your program's object code and data. Because they contain all of SPITBOL, load modules tend to be relatively large compared to save files.

Load modules can be created just prior to program execution by specifying the –w command-line option. They are created with the same name as the first source file, but with extension **.out** for Unix, extension **.exe** for MS-DOS, Windows and OS/2.

Load modules may also be written during program execution by calling EXIT(3,filename), which creates a load module with the given name, or **a.out** or **a.exe** if the file name is omitted. After writing the load module, SPITBOL terminates execution. (Use EXIT(4,filename) to continue execution.)

A load module is executed by typing its name in the shell or command processor, just like any other program. Because they are independent of the SPITBOL program, new versions of SPITBOL do not affect them.

Resumption

When a save file is loaded back into SPITBOL or when a load module is subsequently executed, all I/O files except those associated with INPUT, OUTPUT, and TERMINAL are closed. If created with the –w or –y command-line option, execution begins with the first program statement. If created with the EXIT function, execution resumes by having the EXIT function return a null string value.

If the –b option was present when the save file or load module was created, then execution resumes silently—without SPITBOL's normal sign-on message.

External Functions

External functions are loaded dynamically at execution time by SPITBOL's LOAD function. The implications of this must be clearly understood by programmers who use external functions and wish to create save files or load modules.

External functions are not saved within a save file or load module. When the save file or load module is subsequently loaded and resumed, the LOAD function call(s) must be executed at that time. Therefore, the external function(s)s must be present on that system so that they can be dynamically loaded. You will have to distribute them along with your save file or load module.

External functions that are dynamic libraries must be accompanied by any other dynamic libraries that they load in turn.

Distribution—General

The SPITBOL program file contains material copyrighted by Catspaw, Inc., and may be used only as specified in the license agreement contained in the original package. However, we recognize that users will develop programs that they wish to distribute to others in executable form. This is complicated by the fact that SPITBOL programs can compile new code at runtime, thereby requiring that the compiler be available in some form to third parties executing your programs.

Distributing Load Modules and Save Files

Catspaw, Inc., grants the licensee permission to make and distribute to others copies of load modules (stand-alone executable files) and save files (object program images), *provided such modules or files do not substantially duplicate the function of SPITBOL as a general purpose compiler for the SPITBOL language.* That is, you cannot distribute a load module or save file that behaves like the SPITBOL compiler—reading and executing SPITBOL source programs.

Distributing for MS-DOS and Windows 3.1

Because SPITBOL-386 requires a DOS Extender to operate in the MS-DOS or Windows 3.1 environment, the DOS Extender must also be included if your distribution target is operating in such an environment. Three files must be distributed:

32rtm.exe	run-time manager containing the Portable Execution loader, a floating point emulator, and Win32 emulation
dpmi32vm.ovl	DPMI (DOS Protected Mode Interface) and virtual memory server, used if DPMI services are not otherwise available. DPMI services are available in Windows 3.1 and OS/2 DOS shell windows.
windpmi.386	Provides uncommitted-memory and some floating point support if running under Windows 3.1.

The copyright owner for thse files is Borland International, and they are being distributed with their permission.

The **32rtm.exe**, **dpmi32vm.ovl**, and **windpmi.386** files must be placed in a directory specified in the user's PATH environment variable.

If running Windows 3.1 (or Windows for Workgroups 3.1), the **system.ini** file in the Windows directory should include the following lines:

```
[386Enh]
device=<path>\windpmi.386
```

Where <path> is the full pathname of the directory where you copied **windpmi.386**.

Chapter 14
SPITBOL Statements

512 characters in SPITBOL-8088

Each line of input to SPITBOL consists of a sequence of ASCII characters, terminated by a carriage return. Program lines may be up to 1,024 characters long—characters beyond the 1,024th are ignored.

Comment and control statements are always one line long. However, a program statement may occupy several lines if necessary. A continuation mark (described below) is placed in the first column of the additional lines.

Comment Statements

An asterisk (*) in character position one denotes a comment statement. All text through the end-of-line is copied to the listing file, but is otherwise ignored by SPITBOL.

Control Statements

Control statements provide instructions to the SPITBOL 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. Unlike other versions of SPITBOL, Catspaw SPITBOL follows the SNOBOL4 model and ignores unrecognized control statements.

Multiple controls may appear on the same line, separated by commas:

```
–FAIL, EXECUTE, LIST
```

(In some SNOBOL4 reference books, you will see control "cards" and comment "cards" mentioned. That's a reflection of the old days of punch cards. They are the same as "control statements" or "comment statements".)

There are two types of control statements—those of general usage, and those concerned with producing a program listing suitable for printing.

General controls

–ERRORS
–NOERRORS

These statements control whether SPITBOL should commence program execution in the presence of compilation errors. –NOERRORS is the default, and suppresses execution if there is a compilation error. –ERRORS will begin execution anyway. The –k command line option does the same as –ERRORS.

–CASE N

If N is not 0, then lower-case names are folded to upper-case (the default). If N is 0, or not specified, then upper- and lower-case names are separate; i.e., Label and LABel are different when N is 0, but the same when N is 1. The default is to perform case-folding.

–COPY "file"
–INCLUDE "file"

Compile source code from the specified file. The file name must be enclosed in single or double quotes. Included files may be nested 9 deep. When the End-of-File is encountered, compilation resumes with the line following the –INCLUDE statement. The include nest depth appears in the listing file between the statement number and the statement proper.

As an example, if you compile a program like this:

```
          CAMPAIGN = 'MUDSLINGING'
-INCLUDE "asc.inc"
LOOP      CAMPAIGN ? LEN(1) . C =                  :F(END)
          TERMINAL = ASC(C)                        :(LOOP)
END
```

It will show up in the listing like this:

```
1         CAMPAIGN = "MUDSLINGING"
          -INCLUDE "asc.inc"
2  1               DEFINE('ASC(S)C')
3  1                ASC_ONE = LEN(1) . C
4  1                ASC_PAT = BREAK(*C) @ASC   :(ASC_END)
5  1      ASC      S ? ASC_ONE                        :F(RE-
TURN)
6  1               &ALPHABET ASC_PAT           :(RETURN)
7  1      ASC_END
8         LOOP      CAMPAIGN ? LEN(1) . C =    :F(END)
9                   TERMINAL = ASC(C)          :(LOOP)
10        END
```

The numbers at the far left are the statement numbers, in one sequence for the entire program—the include files are inserted in place. The next number indicates the depth of the included statements; if **asc.inc** had included another function, then that function's statements would be prefixed by 2, and so forth.

SPITBOL will search other directories for included files based on the SNOLIB environment variable. See "Directories" on page 166

To avoid compilation errors due to duplicate labels, SPITBOL will only include a particular file once. That is, if files a.inc and b.inc are both included in a program, and both a.inc and b.inc attempt to include file c.inc, only one copy of c.inc will be read. In situations where you deliberately wish to include multiple copies of a file, simply append a blank after the file name. SPITBOL remembers only the "trimmed" names of files included so far, so the comparison with the blank-padded name will fail, and the file will be included again.

–COPY is a synonym for –INCLUDE, and is provided for compatibility with SPITBOL/370.

–FAIL
–NOFAIL

When the –FAIL mode is set (the default), a failure in a statement without a Goto is ignored, and execution continues with the next statement in sequence. This can lead to undetected errors, especially when array subscripts are out of bounds, or there are pattern matches which fail when you were sure they would always succeed.

Including a –NOFAIL in your program changes this. If a statement without a Goto fails, you'll get an execution error message.

–FAIL and –NOFAIL can be intermixed in the same program; statements will be compiled in the mode currently in effect.

–EXECUTE
–NOEXECUTE

The usual course is to compile a program and then run it. But when you're writing a program, you often want to compile it without running it, so that you can quickly check for compilation and syntax errors. Using the –NOEXECUTE control does just that—compiles your program, but does not run it. When you're ready to run the program, you change it to –EXECUTE, or just eliminate that line.

The command-line option –n also compiles the program but suppresses execution.

–IN###

Sets the maximum line length when compiling the source program to "###". This is also the record length for reading from standard input during execution. Note that there is no space between –IN and the number. The default value is 1,024 (512 for SPITBOL-8088). Typical usage is to set this control to 72 to ignore sequence numbers present in columns 73 to 80 of a card deck.

–OPTIMIZE
–NOOPTIMIZE

SPITBOL normally does some optimization at compile time. It evaluates expressions consisting entirely of constants, and replaces the expression with the result. Thus if you have the statement:

```
A = B / (3 + 4)
```

then the denominator is replaced by 7 when compiled, rather than summing 3 and 4 each time the statement is executed. More dramatic improvements occur when a pattern can be pre-compiled, such as:

```
SUB ? LEN(3) . X BREAK("ABab") . Y SPAN("AABb") . Z
```

In this case, the entire pattern is constructed once, at compile time, instead of being recreated each time the statement is executed.

A second optimization is that unlabeled empty statements without Gotos are not compiled. Blank lines will not incur an execution penalty.

We know of no reason ever to use –NOOPTIMIZE. It is included for compatibility with SPITBOL/370, which provided additional optimizations that could affect program execution.

Listing controls

–EJECT

Advance to a new page in the listing file. Depending on other command line listing switches, this control will either produce three blank lines, or a form feed character. See Chapter 13, "Running SPITBOL," for more information on these options.

–LINE N "file"

Allows the user to override the source file name and line number that will be associated with the next statement. This provides a method for preprocessors such as Rebus and Snocone to place *their* source lines into the target SPITBOL program for use in error messages. The statement formis:

```
–LINE  line_number  'filename'
```

The file name is optional, but must be quoted if it appears.

The current line number and file name are available to your program in keywords &LINE and &FILE. The line number and file name of the *previous* statement are available in &LASTFILE and &LASTLINE.

–LIST

This statement turns on listing, thereby producing a program listing with statement numbers, like that shown under –INCLUDE.

Another way to produce a listing is with the –l command line option.

The listing goes to a file with the same name as the source-code file, but with the extension **.lst**. A different listing file can be specified by using the –o=name command line option.

–NOLIST

This statement turns off listing (the default).

–PRINT
–NOPRINT

Control statements are normally displayed in any listing being made. Control –NOPRINT turns this feature off; –PRINT turns it on (the default).

–SINGLE
–DOUBLE

Listings are normally single-spaced. These control statements allow you to single- or double-space all or part of a program listing.

–SPACE N

This sends N blank lines to the listing. N defaults to one if absent.

–STITL text

Subtitle line for program listing, which appears below the title line. The list file is advanced to a new page prior to printing any title and subtitle.

–TITLE text

Title line for program listing. The list file is advanced to a new page prior to printing the new title and any subtitle line.

Defaults

Catspaw SPITBOL defaults to:

–CASE 1	Case-folding of names
–PRINT	Display control statements in listing
–NOLIST	No listing
–FAIL	Ignore statement failure without Goto
–EXECUTE	Run after compile
–IN1024	Source program line length
–NOERRORS	Don't execute if compilation error
–OPTIMIZE	Opitimize code generation

Program Statements

If a line is not a control or comment statement, it is considered SPITBOL program text. A SPITBOL 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 three elements may appear in various combinations to create different types of statements:

Evaluate expression

```
SUBJECT
```

The expression comprising the subject is evaluated. It may invoke both built-in and program-defined functions.

Assignment statement

```
SUBJECT = REPLACEMENT
```

This is an assignment statement, in which 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.

Pattern match

```
SUBJECT ? PATTERN
```

This is a pattern-matching statement. The subject and pattern expressions are evaluated, and the specified pattern is applied to the subject string, producing success or failure.

Pattern-match with replacement

```
SUBJECT ? PATTERN = REPLACEMENT
```

This statement proceeds like the pattern-matching statement. 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.

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.

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 array or table element, an unprotected keyword, a field-reference function from a program-defined data type, or a program-defined function that has returned by name by branching to NRETURN. 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 or real number, 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 should be enclosed in parenthesis.

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 pattern may assign various matching components to variables with the binary assignment operators dot and dollar sign (., $).

Pattern-matching is a rich language of its own; a sub-language within SPITBOL. Patterns may be recursive, and have looping properties. It is often possible to condense several program lines into a single pattern that will be incomprehensible to its author several days later. Resist the temptation!

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 (except for &ERRTEXT, which will accept a string). 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 succeeded. 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. Loops can be easily constructed:

```
        N = LT(N, 50) N + 1                         :S(LOOP)
```

will increment N and transfer to label LOOP if N is less than 50.

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 SPITBOL 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, 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:

```
        VAR = CODE(string)                    :<VAR>
```

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(COM-
   PILE_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 SPITBOL statement is assumed to have succeeded.

Continuation Statements

A SPITBOL 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, which wends its way across multiple continuation state”
+   “ments. There is an implicit blank at the beginning of each ”
+   “line, which provides the concatenation operator between ”
+   “segments.”
```

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. Program, control, and comment statements are permitted after the semicolon; however, nothing should follow a comment statement, because all characters in it, (including any semicolons), are ignored.

```
        I = 1;  J = 2;  S ? PAT = ‘HENRI’     :S(YES)
        LINE ? WORDPAT =                      ;* Remove next word
        I = 1;OUT OUTPUT = A :F(END);  I = I + 1 :(OUT)
        A = 42 ;–DOUBLE
```

Because of its poor readability, placing labels in the middle of a line is strongly discouraged.

Notice that a comment statement is permitted after the semicolon. This provides a simple method to add end-of-line comments:

```
PARA    NEXT = GETNEXT()    :F(FRETURN)     ;* Return if
End-of-File
        IDENT(NEXT)         :S(RETURN)      ;* Return on empty
line
        PARA = PARA NEXT     :(PARA)        ;* Splice line
```

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
```

Note that for compatibility with mainframe implementations, Catspaw SPITBOL provides the –r command-line option. When this is in effect, the INPUT variable begins reading data at the line following the END statement in the source program file.

Chapter 15
Operators

Following are lists of all the unary and binary operators in SPITBOL. Unused operators may be attached to program-defined or built-in 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.

Unary Operators

Unary operators all have equal priority which is greater than that of any binary operator. If several appear together, they are performed right-to-left.

Operator	Name	Definition
@	at sign	Assigns cursor position to its operand
~	tilde	Negates failure or success of its operand
?	question mark	Interrogation—returns null if operand succeeds
&	ampersand	Keyword
+	plus	Indicates positive numeric operand
–	minus	Negates numeric operand
*	asterisk	Defers evaluation of expression
$	dollar sign	Indirection
.	period, dot	Returns a name

The following unary operator symbols are undefined and are available for user definition using the OPSYN() function.

Operator	Name	Definition
!	exclamation	Unused
%	percent	Unused
/	slash	Unused
#	pound, sharp	Unused
=	equal	Unused
|	vertical bar	Unused

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.

Binary Operators

Binary operators of highest numeric priority are performed first, unless there are parentheses which say otherwise. Note that multiplication has a higher priority than division. When operators of equal priority are adjacent, associativity specifies which one is performed first.

Operator	Association	Priority	Definition
=	right	0	Assignment
?	left	1	Pattern match
|	right	3	Pattern alternation
space	right	4	Concatenation or match
+	left	6	Addition
−	left	6	Subtraction
/	left	8	Division
*	left	9	Multiplication
^, ! or **	right	11	Exponentiation
$	left	12	Immediate assignment
.	left	12	Conditional assignment

The following binary operations are undefined and are available for user definition using OPSYN().

Operator	Association	Priority	Definition
&	left	2	Unused
@	right	5	Unused
#	left	7	Unused
%	left	10	Unused
~	right	13	Unused

Operator Extensions

SPITBOL provides several extensions to the standard SNOBOL4 language. See Chapter 9, "Advanced Topics," for examples of their usage. The extensions are:

- Multiple use of the assignment (=) operator within a statement, such as A = B = C + 1.
- Embedded pattern matching and replacement: A = (B ? C = D) + 1.
- Alternative evaluation, in which expressions in a parenthesized list are evaluated left to right until one succeeds: A = (LT(I,J) I, GT(I,J) J, "Same").

Chapter 16
Keywords

Special variables called keywords allow a program to communicate with SPITBOL. 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, and &STCOUNT. Tracing occurs each time SPITBOL alters their value. For example, tracing keyword &STCOUNT produces a trace after *every* SPITBOL statement is executed.

Protected Keywords

Among these keywords are several which serve as read-only repositories of fundamental system patterns and values, such as &ARB and &BAL. In other SNOBOL4 systems, the non-keyword form of primitive patterns (ARB, BAL, etc.), can be changed by a program, and later restored to its original value by assigning it the corresponding keyword. Because patterns like ARB and BAL may not be altered in SPITBOL, the keyword form is present only for historic reasons.

&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.
&FAIL	The primitive pattern FAIL.
&FENCE	The primitive pattern FENCE.
&FILE	Source file name of the current statement being executed.
&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.
&LASTFILE	Source file name of the previous statement executed.
&LASTLINE	Integer source file line number of the previous statement executed.
&LASTNO	Integer statement number of the previous statement executed.
&LCASE	String of 26 lower-case alphabetic characters in ascending order: "abcdefghijklmnopqrstuvwxyz"
&LINE	Integer source file line number of the current statement being executed.
&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.
&STNO	Integer statement number of the current statement being executed.
&SUCCEED	The primitive pattern SUCCEED.
&UCASE	String of 26 upper-case alphabetic characters in ascending order: "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

Unprotected Keywords

These keywords may be set to integer values to modify SPITBOL's behavior. The normal method for setting them is a statement like this:

```
        &ANCHOR = 1
```

&ABEND

Abnormal ending. This is normally set to 0. Changing it does not perform any useful function in SPITBOL, but it is included to maintain compatibility with programs written to run under other operating systems.

&ANCHOR

Nonzero for anchored pattern match. Initially 0, unanchored. Pattern matching is much more efficient if performed in anchored mode.

&CASE

Setting it to zero will 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 by SPITBOL to the operating system. In MS-DOS, Windows and OS/2 environments, this is accessed from the command processor via ERRORLEVEL; in Unix, with shell variable $? (sh) or status (csh). The default value of &CODE is 0.

&COMPARE

Select collating sequence for lexical comparisons. It is ignored in the implementations described by this manual, but is present for compatibility with programs written for Catspaw's MaxSPITBOL implementation.

&DUMP

Normally, this is 0. Setting it to 1 will cause a partial dump at program termination, and 2 or 3 will produce a full dump. The dump goes to the file specified with –o on the command line, or to standard output if none.

A partial dump includes the values of all non-constant keywords and all non-null variables. A full dump with value 2 also includes values of all non-null array and table elements, and non-null members of any program-defined datatypes. Value 3 additionally lists null-valued variables and statement labels.

&ERRLIMIT

When zero (the default), an execution error or user interupt results in program termination with an error message displayed. When non-zero and either occurs, it is decremented by one, no message is displayed, and execution proceeds as follows: If an error label has been declared with the SETEXIT function described in Chapter 19, "SPITBOL Functions," control is transferred to that label.

If there is no SETEXIT label, SPITBOL converts the error to statement failure. Note that this can cause unexpected loops if the error occurred in the

Goto portion of the statement, such as transferring to an undefined label, or failure when evaluating an expression in a complex Goto.

Setting &ERRLIMIT non-zero without a corresponding SETEXIT error label is strongly discouraged.

&ERRTEXT

If an execution error occurs, then the error message text corresponding to the error code is stored as a string in &ERRTEXT. It is possible to assign a string to &ERRTEXT which is then used in a subsequent error report if &ERRTYPE is assigned a value out of the range used by SPITBOL for its own purposes.

&ERRTYPE

If an execution error occurs, then the error code is stored as an integer in &ERRTYPE. &ERRTYPE may be assigned a value; in this case, an immediate error is signaled. This is sometimes used for reporting errors detected by a program. Standard error codes used by SPITBOL all fall below 400. Values in &ERRTEXT and &ERRTYPE are useful in SETEXIT error intercept routines.

&FTRACE

Nonzero value causes each call and return of a program-defined function to be listed. Decremented for each trace, it is initially 0.

&FULLSCAN

Set non-zero to enable exhaustive pattern scanning (the default). Setting &FULLSCAN to 0 is an error—the "Quickscan" heuristics of older SNOBOL4 systems are not supported.

&INPUT

Set to 1 for normal input (the default). If set to 0, all input associations (of variables) are temporarily ignored.

This keyword is an anachronism from older SNOBOL4 systems that could run compute-bound programs faster if I/O associations were disabled. Having input enabled does not incur a time penalty in SPITBOL.

&MAXLNGTH

The size of the largest object or string permitted by SPITBOL. The default value for &MAXLNGTH is 4,194,304, except in SPITBOL-8088, where it is 9,000. System restrictions on possible settings are discussed in Chapter 13, "Running SPITBOL".

&OUTPUT

Set to 1 for normal output (the default). If set to 0, all output associations (of variables) are temporarily ignored.

&PROFILE

When set to 0, the default, statement profiling is disabled. When set to 1, statement profiling is enabled. When enabled, SPITBOL keeps track of information about each statement executed.

If profiling is enabled at any time during execution, when the program terminates the accumulated profile goes to the file specified with –o on the command line, or to standard output if none. The profile shows statement numbers, the number of times each statement was executed, the total elapsed time in milliseconds spent in each statement, and the average execution time of each statement, in microseconds.

A sample profile is shown below. A program listing will be required to interpret the statement numbers.

```
Program  Profile
STMT     Number  Of         — Execution  Time —
Number   Executions  Total(MSec)                 per  Excn(MCSec)
1        1           0                           0
2        1           0                           0
3        1           0                           0
4        129         83                          643
6        128         17                          132
7        128         17                          132
8        1001        251                         250
9        1001        201                         200
14       1001        164                         163
15       1001        484                         483
```

Because computer clock accuracy varies from one tenth to one thousandth of a second on different systems, the times will not be reliable unless statements are executed repeatedly.

If &PROFILE is set to 2, the accumulated time spent in a user-defined function is charged to the statement calling the function.

Profiling greatly slows down the execution of a program, so the absolute times shown for each statement are not realistic. However, the relative times between statements are accurate.

&STLIMIT

The maximum number of statements permitted to be executed. It is initially 2,147,483,647, except in the 16-bit integer version of SPITBOL-8088 (**spitbols.exe**), where it is 32,767. There are no restrictions on the integer values for &STLIMIT. To inhibit this check on number of statements executed, assign a negative value to the keyword:

```
        &STLIMIT  =  –1
```

This will result in a very slight improvement in execution speed, since internal counts will not have to be updated and checked. The value of &STCOUNT remains frozen at the value reached when &STLIMIT was set negative and the number of statements executed is omitted from the execution statistics. Tracing of &STCOUNT is no longer possible since its value is not incremented.

&TRACE

Nonzero to permit tracing with the TRACE function. Initially 0, it is decremented for each trace performed.

&TRIM

Nonzero to strip trailing blanks and tab characters from lines read from files. This is faster than using the TRIM() function. Initially set to zero, which preserves any trailing blanks or tabs. Note that in SPITBOL, unlike some other SNOBOL4 implementations, there is no padding with blanks of short input lines to a standard width. Regardless of the setting of &TRIM, no blank trimming occurs when reading from a file opened for binary transfers.

Special Names

The following names have special meaning to SPITBOL. If case-folding is in effect, they may appear with any combination of upper- or lower-case letters.

ABORT

Transfer to this special system label only when within an error routine that has been invoked by SETEXIT(). It causes error processing to resume as though no error intercept through SETEXIT() had been made. When used as a variable, ABORT is also a primitive SPITBOL pattern that terminates a pattern match.

CONTINUE

Transfer to this special system label only when within a SETEXIT() error routine. This causes execution to resume by branching to the failure exit of the statement that produced the error. Beware of problems if the original error was in the Goto field of a statement. The error will be triggered again when the attempt is made to branch to the failure exit. See SCONTINUE.

END

This is a special label which marks the last statement of your 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 the standard input file. Normally, INPUT reads from the keyboard, but this can be changed with redirection on the command line. System file descriptor 0 is used for this input.

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). A name result may be thought of as the address of a variable. A discussion of the use of NRETURN appears at the end of Chapter 9, "Advanced Topics."

OUTPUT

Variable associated with the standard output file. OUTPUT normally goes to the screen, but can be sent elsewhere with redirection or the –o specification on the command line. System file descriptor 1 is used for this output.

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 by assigning it to a variable with the same name as the function before transferring to RETURN.

SCONTINUE

Transfer to this special system label only when within a SETEXIT() error routine, and only after an error 320. Error 320 is generated for a user keyboard interrupt (system dependent: control-C, DEL, break, etc.). Transferring to SCONTINUE causes execution to resume by branching back into the statement at the point of interruption. The statement then succeeds or fails normally.

TERMINAL

Variable associated with output to the screen, or input from the keyboard. System file descriptor 2 is used for this input/output.

Chapter 17
Data Types and Conversion

There are nine fundamental types of data in SPITBOL. In addition, new data types may be defined by the user, each being an aggregate of other existing types. This section describes the nine internal types, and how they may be converted from one to the other.

Most other programming languages require the user to explicitly declare the type of data to be stored in a variable. In the SNOBOL family, any variable may contain any data type, and the variable's type may be freely altered during program execution. SPITBOL remembers what kind of data is in each variable. Remember, data is typed, a variable is not.

Aggregate structures such as arrays, tables, and user-defined data types need not be of homogeneous type. Their constituent elements may all be of different data types.

Data Type Names

Built-in types

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

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. The variable may then be subscripted to access the elements of the array.

Each array element consists of a one word memory pointer to the memory where the actual value is stored. The array requires several additional words for housekeeping information. Arrays may not exceed &MAXLNGTH bytes in total size.

Code

The primitive function CODE() compiles a string containing SPITBOL 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 as the argument of a direct Goto by enclosing it in angle brackets. Execution control transfers to the first statement in the object code block.

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, object code is compiled to call LEN and POS and nothing more. It is only when E is referenced in some subsequent pattern match (or EVAL'ed) that LEN and POS are called with the then current values of K and M. The pattern structure is then built.

In this case, it would be more efficient to use

```
E = LEN(*K) POS(*M)
```

because execution could proceed further, and call LEN and POS to build the pattern structure, even if the particular arguments K and M aren't known.

In this second case the result E is a PATTERN, not an EXPRESSION. The unevaluated expression operator must be at the outermost level to create an object of type EXPRESSION. Expressions may also be produced explicitly with the CONVERT() function (see below).

Integer

In all versions except 16-bit SPITBOL-8088 (**spitbols.exe**), this is a decimal number in the range −2,147,483,648 to +2,147,483,647. No fractional part may appear. Integers are stored in twos-complement form, and occupy two 32-bit memory words (one for housekeeping information, one for the value). In 16-bit SPITBOL-8088, the allowable range for integers is −32,768 to +32,767, occupying two 16-bit memory words.

Name

When the unary name operator (.) is applied to a variable the NAME data type results. This 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.

Unlike SNOBOL4, the unary name operator (.) applied to a simple variable name yields a NAME rather than a STRING. Since this NAME is converted to a STRING when necessary, the difference is not normally noticed. The only point at which the difference is apparent is when a NAME value is used as an argument to the IDENT, DIFFER, or DATATYPE functions or when it is used as a TABLE subscript.

Pattern

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
```

Pattern structures may be arbitrarily large within the limits imposed by available memory. They are not limited by &MAXLNGTH.

Real

A floating-point decimal number in the range ±0.19E−322 to ±0.18E+309, and 0. The fractional portion may contain up to 15 significant digits, although only 9 are displayed by SPITBOL. A real value is stored in 64-bit, double-precision IEEE format, with one additional memory word required for housekeeping information.

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.

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 SPITBOL data type including the null string. Arrays require integer subscripts, but table subscripts such as T<"TALLY"> or T<13.52> are acceptable.

Tables are implemented using a hash table data structure. A contiguous block of memory is used to hold a number of one-word "bucket headers," each of which points to a linked list of elements that hashed to the same value.

Tables default to 11 bucket headers, which is inefficient for tables with a large number of elements. You can and should specify a larger number of buckets in the first argument to the TABLE() function. The total memory allocated to bucket headers must be less than &MAXLNGTH bytes in size.

Program-defin ed data type

New data types may be created with the primitive function DATA. The name specified in the prototype string becomes a new data type in SPITBOL. Any object created with the data type's creation function is given this name as its data type.

```
DATA('COMPLEX(REAL,IMAG)')   ;* Define a new data type COMPLEX
NUM = COMPLEX(2.3, −4.0)      ;* Create a COMPLEX object
OUTPUT = DATATYPE(NUM)        ;* Print the string 'COMPLEX'
```

Data Type Conversion

As far as possible, SPITBOL converts from one datatype to another as required. The following table shows which conversions are possible. Blank entries indicate that the conversions are impossible, A indicates that conversion is always possible, and U indicates conversion is usually possible, depending on the value involved.

Convert From	Convert To								
	S	**I**	**R**	**A**	**T**	**P**	**N**	**E**	**C**
String	A	U	U			A	A	U	U
Integer	A	A	A			A	A	A	
Real	A	U	A			A	A	A	
Array				A	U				
Table				A	A				
Pattern						A			
Name	U	U	U			U	A	U	U
Expression								A	
Code									A

Explicit conversion

A program may use the CONVERT() function to explicitly convert an object to another data type. Its first argument is the object to be converted. The second argument is a string containing the formal name of the desired data type. The formal name must be in upper-case (or lower-case if case-folding). If conversion is possible, the function succeeds and returns the converted object. If conversion is not possible, the function fails. The call looks like this:

```
NEWTYPE = CONVERT(OBJECT, "DESIRED TYPE")
```

Implicit conversion

Implicit conversion occurs automatically when SPITBOL 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.

Conversion Details

Type ➜ type

All possible data type conversions are explained here. Most of these occur automatically as necessary.

String ➜ integer

Leading and trailing blanks are ignored. A leading plus or minus sign is allowed, but must be followed by at least one digit. A string consisting only of a plus or minus character is illegal. Leading zeros are allowed. The resulting value must be in the legal range for integer values. A null or blank string is converted to integer zero.

```
                RESULT = ("–14" + "") / "2"
```

stores integer –7 in RESULT.

String ➜ real

Leading and trailing blanks are ignored. A leading sign, if present, must immediately precede the number. Some of the restrictions which apply when a real number appears as a literal in a SPITBOL program are relaxed here: a decimal point is not required, and there need not be any digits to the left of the decimal point. A signed power of 10 may follow the mantissa. It is designated by one of the letters E, e, D, or d. Values smaller than ±0.19E−322 are converted to 0.

```
?          R1 = ' –1.234' + "12345678" + " 19D2 " + '.5E–2'
?= R1
12347577.
```

String ➜ pattern

A pattern is created which will match the string value.

String ➜ name

The result is the name of a natural variable which has the given string as its name. This is identical to the result of applying the unary dot operator to the variable in question. Thus, CONVERT("ABC","NAME") is equivalent to .ABC. CONVERT()creates a variable ABC and returns a pointer to it.

The null string cannot be converted to a name.

String ➜ expression

The string must represent a legal SPITBOL expression. The compiler is used to convert the string into its equivalent expression, and the result can be used anywhere an expression is permitted.

String ➜ code

The string must represent a legal SPITBOL program, complete with labels, and using semicolons to separate statements. The compiler is used to convert the string into executable code. The resulting code can be executed by transferring to it with a direct Goto or by a normal transfer of label within the code. For example,

```
          S = 'LOOP OUTPUT = A; A = LT(A,10) A + 1
:S(LOOP)F(END)'
          CODE(S)
          :(LOOP)
END
```

The string S is converted to executable code by the CODE function, and control passes to its label with the LOOP Goto.

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".

Integer ➜ real

The integer is converted to a real value without any loss of significance. Example:

```
          FIFTH = 3.42 / 5
```

Integer ➜ pattern

The integer is converted to a string, and the string converted to a pattern. Example:

```
          SUBJECT ? 19 = ''
```

Integer ➜ name

SPITBOL first converts the integer to a STRING and then performs a STRING to NAME conversion. Example:

```
          X = CONVERT(1, 'NAME')
```

A subsequent reference to $X would access a *variable* with name "1".

Integer ➜ expression

The result is an EXPRESSION which when evaluated yields the original integer as its value. For example:

```
          X = CONVERT(1, 'EXPRESSION')
          TERMINAL = EVAL(X)
```

produces integer 1.

Real ➜ string

The real number is converted to its standard representation. If an exponent is produced, it will always be signed. In all cases, the string will have a decimal point. Example:

```
          A = 12.1; B = 123456789.012
          OUTPUT = A ', ' B
12.1, 0.123456789E+9
          OUTPUT = A ' ' –B
12.1 –0.123456789E+9
```

Real ➜ integer

The real number must be in the range allowed for integer values. Conversion occurs by truncating the fractional part. For example,

```
R = 3.14159
SUBJECT ? LEN(R) . FRONT
```

uses LEN(3) as a pattern since LEN requires an integer argument.

Note that the real function CHOP() can be used to remove the fractional part of a real number without converting it to an integer, and without any range restrictions.

Real ➜ pattern

The real is converted to a string, and the string converted to a pattern. Example:

```
SUBJECT ? (12.4 / 2.0)
```

searches for the substring "6.2" in the subject.

The conversion can be made explicitly with:

```
P = CONVERT(12.4 / 2.0, 'PATTERN')
SUBJECT ? P
```

Real ➜ name

First convert to STRING and then convert the string to a name.

Real ➜ expression

The result is an expression which when evaluated yields the original real value. After executing

```
X = CONVERT(1.23, 'EXPRESSION')
```

X has the data type EXPRESSION, and EVAL(X) produces the real number 1.23.

Array ➜ table

The array must be two-dimensional with a second dimension of 2, or an error occurs (that is, it must be a two-column array). For each value of the first subscript, J, a table entry is created, which uses [J,1] as the key and [J,2] as the value. The table built has a number of hash headers equal to the first dimension.

If a two-dimensional array A had these values:

```
A[1,1] = 'Dog'
A[1,2] = 'Bark'
A[2,1] = 'Cat'
A[2,2] = 'Meow'
```

and this were executed:

```
T = CONVERT(A, 'ARRAY')
```

the resulting table T would have these values:

```
T['Dog'] = 'Bark'
T['Cat'] = 'Meow'
```

Table ➜ array

This occurs implicitly when a table is the first argument to function RSORT or SORT, or explicitly with CONVERT(T, 'ARRAY'). The table is converted to a two-dimensional array.

The table must have at least one non-null element. If the table is empty, CONVERT, RSORT and SORT will fail.

The array generated is two-dimensional; the first dimension equals the number of non-null elements in the table, and the second dimension is two. For each entry, the [J,1] element is the key and the [J,2] element is the value.

When a table is explicitly converted to an array with CONVERT(), entries are placed in the resultant array in the order they were created in the table. That is, older table elements come before newer ones.

Name → string

The formal data type name "NAME" is produced.

```
?          A = TABLE()
?          A[2] = 'Something'
?          N = .A[2]
?= N
NAME
```

Name → integer

A NAME can be converted to an INTEGER only if it is the name of a natural variable whose character name has the form of an integer constant. Example:

```
?          X = CONVERT(1, 'NAME')
?= 2 + X
3
```

Name → real

A NAME can be converted to a REAL only if it is the name of a natural variable whose character name has the form of a real constant. The result is the corresponding real value.

Name → pattern

A NAME can be converted to a PATTERN only if it is the name of a natural (simple) variable. The result is a string pattern which matches this name.

```
           P = .ABC
           S ? P
```

is the same as

```
           S ? 'ABC'
```

Name → expression

A NAME can be converted to an EXPRESSION only if it is the name of a natural variable whose character name has the form of a valid expression. The result is the corresponding expression value. Example:

```
?          E = CONVERT('1 + 2', 'NAME')
?          $E = 45
?= EVAL(E)
3
?= $'1 + 2'
45
```

(And a variable named '1 + 2' was created, with value 45.)

Name → code

A NAME can be converted to CODE only if it is the name of a natural variable whose character name has the form of a valid code sequence. The result is the corresponding code sequence value.

Chapter 18

Patterns and Pattern Matching

The SPITBOL pattern matcher is called the *scanner*. The cursor is the scanner's pointer into the subject string; it points *between* subject characters (see picture on page 65). It is initially zero when positioned to the left of the first subject character, and is incremented as the scanner moves to the right in the subject.

This chapter discusses SPITBOL's pattern-matching primitives, and provides an introduction to the pattern-matching algorithm used.

SPITBOL also provides pattern *functions*, including ANY(), ARBNO(), BREAK(), BREAKX(), LEN(), NOTANY(), POS(), RPOS(), RTAB(), SPAN(), and TAB(). They are discussed in Chapter 19, "SPITBOL Functions."

Primitive Patterns

Built-in patterns

These variables contain the primitive patterns of the same name. Unlike SNOBOL4, these variables cannot be altered.

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 non-null 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 the pattern, but fails if the scanner has to back up through it, seeking alternatives. FENCE as the first component of a pattern effectively anchors the match, regardless of the setting of the &ANCHOR keyword.

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.

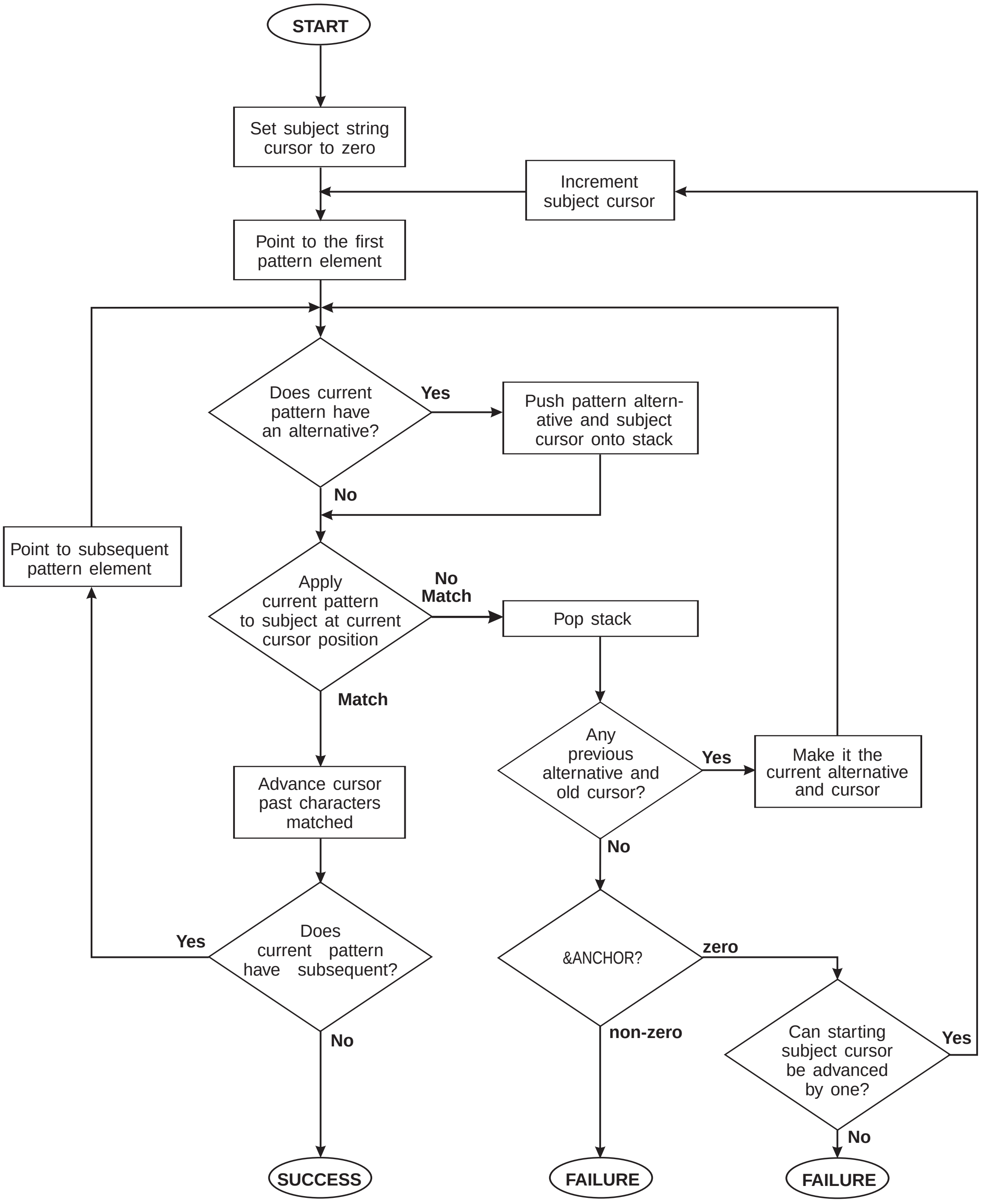
Pattern Matching

Although pattern elements can be combined as subsequents and alternates to produce complex patterns, the basic matching process rules are relatively simple.

Pattern-match algorithm

A pushdown stack is used to remember backtracking possibilities. A SPITBOL pattern match proceeds as described below and shown on the next page:

1. Set the subject string cursor to zero.
2. Point to the first pattern element.
3. If the current pattern has an alternative, push the alternative and current cursor position onto the stack.
4. Apply the current pattern to the subject string at the current cursor position. If it succeeds, advance the cursor past the characters matched and go to step 5. Go to step 6 if it fails.
5. (Pattern element matched) If the current pattern has a subsequent, point to it and go to step 3. If there is no subsequent, the entire pattern match has succeeded.
6. (Pattern element did not match) Pop the stack to get a previous alternative and old cursor position. If there is one, make it the current pattern and cursor and go to step 3. If the stack is empty and keyword &ANCHOR is nonzero, the entire pattern match has failed. If &ANCHOR is zero, advance the starting cursor position by one and go to step 2 if more subject characters remain.


START
Set subject string cursor to zero
Increment subject cursor
Point to the first pattern element
Does current pattern have an alternative?
Yes
Push pattern alternative and subject cursor onto stack
No
Point to subsequent pattern element
Apply current pattern to subject at current cursor position
No Match
Pop stack
Match
Any previous alternative and old cursor?
Yes
Make it the current alternative and cursor
Advance cursor past characters matched
No
Does current pattern have subsequent?
Yes
&ANCHOR?
zero
non-zero
Can starting subject cursor be advanced by one?
Yes
No
No
SUCCESS
FAILURE
FAILURE

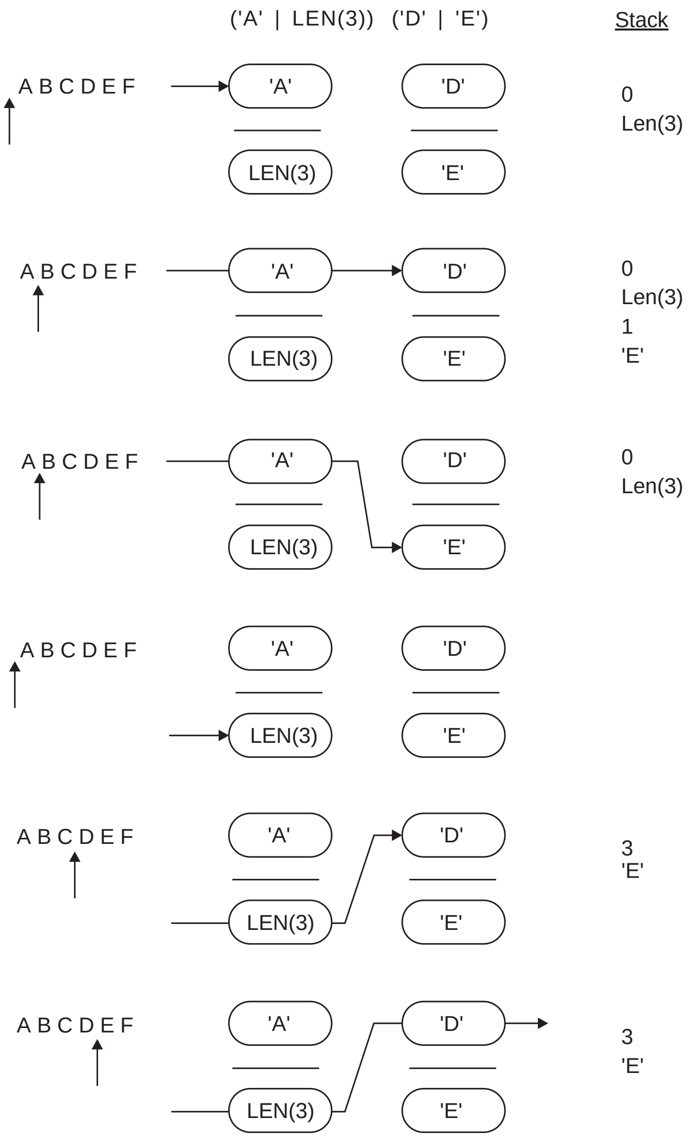
SPITBOL Pattern-Match Algorithm

Pattern-match example

The algorithm attempts to thread a path through one alternative in each subsequent group. We can get a feel for it by applying it to the statement:

```
'ABCDEF' ? ('A' | LEN(3)) ('D' | 'E')
```

We'll use a graphical representation of the pattern-match process known as a "bead diagram." Pattern alternatives are stacked vertically, and pattern subsequents appear as adjacent columns of alternatives. The scanner tries to thread a needle through one "bead" in each column.



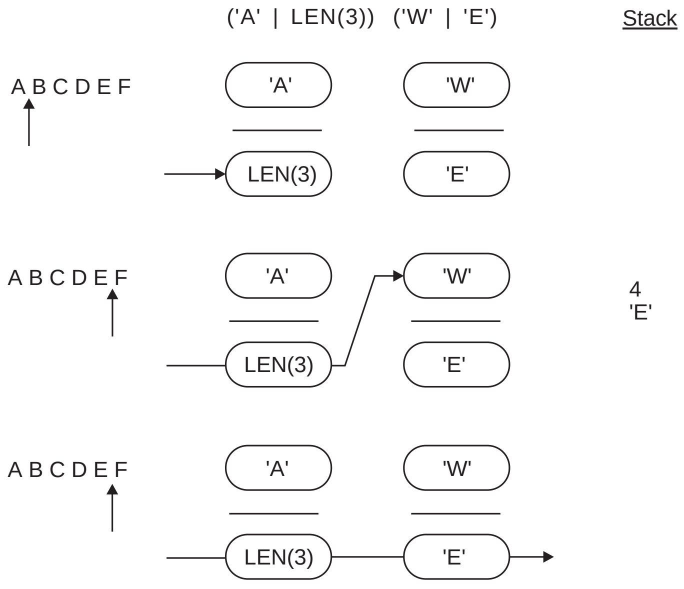
The cursor is positioned to the left of the first subject character. The current pattern element is set to 'A'. We stack the current cursor (0) and alternative, LEN(3). 'A' is matched to the subject, and succeeds. The cursor is advanced to 1 and we check if 'A' has a subsequent. The subsequent is

(‘D’ | ‘E’), so we point to its first element, ‘D’, and stack its alternative, ‘E’. ‘D’ does not match ‘B’, so we pop the stack, and try ‘E’ against ‘B’. Again it does not match, and popping the stack again takes us back to LEN(3) and cursor position 0. LEN(3) matches ‘ABC’. The first subsequent, ‘D’ matches, and there are no more subsequents, so the entire match succeeds.

If the statement had been:

```
'ABCDEF' ? ('A' | LEN(3)) ('W' | 'E')
```

the final step of the previous diagram would fail when ‘W’ and ‘E’ failed to match subject character ‘D’. In the unanchored pattern matching mode, the starting cursor would be advanced to 1, and the whole process started over again. The final portion of the bead diagram would now look like this:



It would succeed this time, with LEN(3) matching ‘BCD’ and ‘E’ matching itself. Given a long subject, you can see how time consuming unanchored matching can become.

The algorithm gives no clue how patterns such as ARB operate. Consider the statement:

```
'ABCDEF' ? 'A' (ARB | 'D') 'E'
```

ARB first matches the null string, and ARB‘s subsequent, ‘E’, fails when applied to ‘B’ of the subject. The algorithm implies the next step should try ARB‘s alternate, ‘D’, which also fails. Since there are no more alternatives, the entire match should fail. But in practice it doesn’t—ARB matches ‘BCD’ as expected.

The answer to this puzzle is that patterns such as ARB and BAL have implicit alternatives which are tried before your explicit ones. ARB behaves as if it were:

```
(LEN(0) | LEN(1) | LEN(2) | LEN(3) | ... )
```

and fails only when further extensions would make it larger than the subject. Only then are other pattern alternatives tried.

Chapter 19

SPITBOL Functions

Built-in SPITBOL Functions

Conventions

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.

arg	A generic argument of any SPITBOL data type.
array	An array.
i	An integer number.
n	A number, either integer or real.
name	The name of a variable, function or label, such as .VAR or 'VAR'. When case-folding is in effect, 'VAR' and 'var' are equivalent as names.
r	A real number.
s	Any SPITBOL string.
table	A table.
channel	I/O channel. An integer or string.

If an argument is omitted in a function call, SPITBOL supplies the null string instead.

19

Note

Functions that may not be available in other SPITBOL or SNOBOL4 systems, or which operate in a substantially different manner on those systems, have a double-outlined box around their names.

When discussing these differences, "standard SPITBOL" means common implementations of Macro SPITBOL, as devised by Dewar and McCann. "Standard SNOBOL4" refers to Bell Labs Macro SNOBOL4, as defined in *The SNOBOL4 Programming Language* by Griswold, Poage, and

Polonsky. “SNOBOL4+” is Catspaw’s MS-DOS implementation of SNOBOL4, version 2.1 or greater. “MaxSPITBOL” is Catspaw’s implmentation of SPITBOL for the Apple Macintosh™ .

Function summary

Built-in functions are grouped here by functionality. Consult the detailed description in the remainder of the chapter for more information.

Arrays and tables

ARRAY(s, arg)	create array, prototype s, initial value arg
ITEM(array, i1, i2, …, in)	reference array element
ITEM(table, arg)	reference table element
PROTOTYPE(array)	produce creating prototype
RSORT(array, i)	sort array descending, column i
RSORT(table, i)	sort table descending on key or value
SORT(array, i)	sort array ascending, column i
SORT(table, i)	sort table ascending on key or value
TABLE(i, x, arg)	create table, i hash headers, initialize to arg

Compilation

EVAL(s)	compile and evaluate expression string
CODE(s)	compile program statements

Function control

APPLY(name, arg1, argn)	call function name with args
ARG(name, i)	retrieve ith prototype arg for named function
DEFINE(s, name)	define new function, prototype s, at label name
LOAD(s1, s2)	load external function
LOCAL(name, 1)	retrieve ith prototype local for named function
OPSYN(s1, s2, i)	make s1 synonym for s2
UNLOAD(name)	remove function definition

Input/output

BACKSPACE(name)	backspace file one record
DETACH(name)	detach variable from channel
EJECT(channel)	issue page eject on channel
ENDFILE(channel)	close and release channel
INPUT(name, chanl, s)	attach file to variable via channel for input
OUTPUT(name, chan, s)	attach file to variable via channel for output
REWIND(channel)	rewind file attached to channel
SET(channel, i, i)	set file position

Memory

CLEAR(s)	reinitialize all variables
COLLECT(i)	garbage collect memory
DUMP(i)	display variable values

Miscellaneous

CHAR(i)	convert integer ordinal to character string
CONVERT(arg, s)	convert arg to data type s
DATATYPE(arg)	produce data type of arg
DATE()	produce date and time
EVAL(expression)	evaluate expression
SIZE(s)	produce length of string
TIME()	return execution time

Numeric

ATAN(n)	arctangent in radians
CHOP(r)	discard fractional part of real
COS(n)	cosine of angle in radians
EXP(n)	natural logarithm *e* to the nth power
LN(n)	natural logarithm of n
REMDR(n1, n2)	remainder after division
SIN(n)	sine of angle in radians
SQRT(n)	square root of n
TAN(n)	tangent of angle in radians

Numeric comparison

EQ(n1, n2)	numbers equal
GE(n1, n2)	n1 greater than or equal n2
GT(n1,n2)	n1 greater than n2
INTEGER(arg)	argument integer or integer string
LE(n1,n2)	n1 less than or equal n2
LT(n1, n2)	n1 less than n2
NE(n1,n2)	numbers not equal

Object comparison

DIFFER(arg1, arg2)	objects different
IDENT(arg1, arg2)	objects identical

Object creation

COPY(arg)	create copy of argument
DUPL(pattern, i)	replicated pattern

Pattern match

ANY(s)	match one character in s
ARBNO(pattern)	match zero or more occurrences of pattern
BREAK(s)	match up to any character in s
BREAKX(s)	like BREAK, but expand on rematch
FENCE(pattern)	pass through pattern on rematch
LEN(i)	match i characters
NOTANY(s)	match one character not in s
POS(i)	verify match cursor at i
RPOS(i)	verify match cursor at i from end
RTAB(i)	match to i characters from end of subject
SPAN(s)	match run of characters from s
TAB(i)	match to cursor position i

Program control

EXIT(i)	create load module or Save file
EXIT(s)	launch another program
HOST(i, arg1, …, argn)	machine-specific functions
SETEXIT(name)	trap errors
STOPTR(name, type)	stop tracing named object
TRACE(name,s)	trace object according to type

Program-defined data type

DATA(s)	define new data type
FIELD(s, i)	ith field name of data type s
DATATYPE(arg)	data type of arg

String comparison

LEQ(s1, s2)	s1 lexically equal to s2
LGE(s1, s2)	s1 lexically greater or equal to s2
LGT(s1, s2)	s1 lexically greater than s2
LLE(s1, s2)	s1 lexically less or equal to s2
LLT(s1, s2)	s1 lexically less than s2
LNE(s1, s2)	s1 lexically different from s2

String synthesis

DUPL(s, i)	replicate s i times
LPAD(s1, i, s2)	pad string on left
REPLACE(s1, s2, s3)	replace characters in s1 according to s2, s3
REVERSE(s)	reverse a string
RPAD(s1, i, s2)	pad string on right
SUBSTR(s, i1, i2)	extract substring from s
TRIM(s)	remove trailing blanks and tabs

Function Descriptions

ANY

Match one character from a set

ANY(s)

Matches exactly one character from the set of characters specified by the argument string. A null argument is not permitted.

APPLY

Indirect call to a function

APPLY(name, arg1, arg2, …, argn)

Call function name with the specified arguments. Since name 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.

The list of arguments may be null in SPITBOL, whereas SNOBOL4 requires the number of arguments to match the function definition. In SPITBOL, extra arguments are ignored, and missing arguments are supplied as null strings.

ARBNO

Match repeated pattern

ARBNO(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. For example, surrounding ARBNO by POS(0) and RPOS(0) forces it to match the entire subject string. In this respect, it is very much like the ARB primitive pattern.

ARG

Get dummy argument name from function definition

```
ARG(name, i)
```

Returns a string which is the ith argument from the formal definition of program-defined function name. ARG fails if i is greater than the number of arguments in name'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(name, i).

ARRAY

Create an array

```
ARRAY(s, arg)
ARRAY(i, arg)
```

In the first form of the call, s is a prototype string which specifies the dimensions of the array created, and the optional arg is the value used to initialize all array elements. The form of the prototype string is:

```
"L1:H1,L2:H2,…,Ln:Hn"
```

where each 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.

The second form of the call is used to create one-dimensional arrays (vectors) with i elements, initialized to the optional arg. Access to vector elements is considerably faster than access to array elements.

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.

```
YEARS = ARRAY('1901:2000')
 …
OUTPUT = YEARS[1989]
```

A common error when defining a multi-dimensional array is to use integers instead of a string for the prototype, that is:

```
X = ARRAY(3, 4)
```

instead of

```
X = ARRAY("3,4")
```

The first statement defines a 3-element, one-dimensional array, with elements initialized to integer 4. The second defines a rectangular array, 3 rows by 4 columns, with no initializing value.

When arguments are supplied through variables, note the difference between commas within the first argument (separating dimensions), and the comma that separates the the two arguments to the ARRAY function:

```
DIMS = "10,10,10"
INITVAL = -1
CUBE = ARRAY(DIMS, INITVAL)
```

ATAN

Arctangent

ATAN(n)

ATAN returns the angle whose tangent is n. The result is in radians.

ATAN is not present in standard SNOBOL4 nor in standard SPITBOL.

BACKSPACE

Backspace file one record

BACKSPACE(channel)

The argument is an I/O channel previously used as the second argument to the INPUT or OUTPUT function. The file associated with the channel is positioned backwards one record. If the file is a binary (raw-mode) file, the file position is decremented by the record size specified when the file was opened (–rn option). If the file is a text (line-mode) file, it is scanned backwards and positioned at the beginning of the previous record. The method used is as follows:

> *If an end-of-line character is found immediately preceding the current file position, it is ignored. The file is then scanned in reverse until another end-of-line character is found. The file is then positioned just forward of the end-of-line character.*

BACKSPACE stops at beginning-of-file. BACKSPACE fails if the channel is not a disk file. If it succeeds, the null string is returned.

BACKSPACE is present in SNOBOL4, but is absent in standard SPITBOL.

BREAK

Match characters not in set

BREAK(s)

Matches zero or more characters provided they are not in the set of characters specified by the argument string. That is, it matches up to, but not including, a character from the argument string. For example:

```
"Now is the time" ? BREAK("ei") . X
```

matches "Now " and assigns it to variable X.

BREAKX

Extended BREAK function

BREAKX(s)

BREAK(s) matches up to any character in s, and will not extend past that character if forced to rematch by subsequent pattern failures. However, BREAKX *will* extend over characters in s on rematch.

Consider the following subject and patterns:

```
SUB = 'EXCEPTIONS ARE AS TRUE AS RULES'
P1  = BREAK('A') . OUTPUT 'AS'
P2  = BREAKX('A') . OUTPUT 'AS'
```

Pattern P1 will fail when applied to SUB, because BREAK will not extend past the first 'A', at the beginning of 'ARE'. However, P2 will succeed, with BREAKX matching the string 'EXCEPTIONS ARE '.

BREAKX can be thought of as a smarter and faster version of the ARB primitive pattern. ARB will extend its match length one character at a time, when required by the mismatch of a "downstream" pattern component. In contrast, BREAKX will extend itself in multi-character segments, by skipping to the next character in its argument string.

BREAKX is part of Catspaw SNOBOL4+ and standard SPITBOL, but it is not present in standard SNOBOL4.

CHAR

Convert character code to string

CHAR(i)

The CHAR function returns a one-character string with the character whose character code is i, which must be an integer between 0 and 255, inclusive.

```
Return  =  CHAR(13)
Tab     =  CHAR(9)
```

This works very similarly to the CHR$ function in most versions of BASIC.

CHAR is a standard SPITBOL feature and it is in SNOBOL4+, although it is not in standard SNOBOL4.

CHOP

Discard fractional portion of real

CHOP(r)

The real valued argument has its fractional portion discarded. It is chopped (truncated) toward zero. The function returns a real result.

CHOP is in SNOBOL4+, but not standard SNOBOL4 or SPITBOL.

CLEAR

Clear variables

CLEAR(s)

If s is omitted, then the null string is assigned to all user variables in the system.

When s is present, it should be a list of variable names separated by commas. These variables will not be affected by the CLEAR function. For example:

```
CLEAR('ABC,CDE,EFG')
```

would cause the values of all variables except ABC, CDE, and EFG to be set to the null string.

The CLEAR function differs from SNOBOL4 by not clearing protected variables, such as ARB. Also, SNOBOL4 does not allow argument s.

CODE

Compile a string

CODE(s)

This converts s to the data type CODE, as described in Chapter 17, "Data Types and Conversion." The string s must contain valid SPITBOL program statements complete with labels and using semicolons to separate statements. The call fails if syntactical errors are found. In this case, it is possible to inspect &ERRTEXT to find the error message, e.g.:

```
CEDAR   =  CODE(CD)                          :S(OK)
OUTPUT  =  &ERRTEXT
               :(BADCOMPILE)
```

The SETEXIT() function can also be used to intercept these errors.

The CODE function returns a pointer to the object code compiled from statements in the argument string. This pointer can be assigned to a variable, and the code executed with a direct Goto (:<CEDAR>).

Statements may be of any length up to MAXLNGTH; the 258-character limit when compiling from a file does not apply. Case-folding of names is controlled by the keyword &CASE.

All versions of SPITBOL allow control statements and comments (separated by semicolons, of course) in the string compiled by CODE, whereas SNOBOL4 does not. One good use for this is to have –LIST produce a listing of the compiled code.

COLLECT

Regenerate storage

COLLECT(i)

The COLLECT function forces a garbage collection which retrieves memory no longer in use and returns it to the block of available storage. The integer argument represents a minimum number of words to be made available (in Catspaw SPITBOL, each word is 32 bits or 4 bytes). If this amount of storage cannot be obtained, the COLLECT function fails. When successful, it returns the number of free words in SPITBOL's memory.

Generally, this is *not* the maximum free memory available to a program, because SPITBOL will request additional memory from the operating system when its working storage is full.

Although the implementation of COLLECT is the same as in SNOBOL4, the values returned will be different because of different internal representations. The organization of SPITBOL is such that forcing garbage collections before they are necessary will always increase execution time.

CONVERT

Convert to specified data type

CONVERT(arg, s)

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', 'REAL', 'TABLE', etc. Data type names may be lower case if case-folding is active. Chapter 17, "Data Types and Conversion," lists allowable conversions.

SPITBOL and SNOBOL4+ differ from standard SNOBOL4 in that s can take an additional data type, 'NUMERIC', which converts the argument to integer or real, as appropriate.

COPY

Make copy of argument

COPY(arg)

Returns a distinct copy of arg. The argument may be an array, table, 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 does not affect element B. By contrast, the assignment B = A makes A and B alternate names for the same array.

SPITBOL allows tables to be copied; SNOBOL4 does not.

COS

Cosine of an angle

COS(n)

Returns the cosine of n, which is presumed to be in radians.

This function is not present in standard SPITBOL or SNOBOL4.

DATA

Create new data type

DATA(s)

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(FIELD1,FIELD2,…,FIELDn)"
```

The DATA function implicitly defines a creation function and n new field reference functions:

NEWTYPE(F1, F2, …, Fn)	Object creation function
FIELD1(x)	Reference to field variable 1
…	
FIELDn(x)	Reference to field variable n

where x is an object created with the NEWTYPE function. The names NEWTYPE and FIELD1 … FIELDn must not be the same as a built-in function, or an error 248 will result ("Attempted redefinition of system function"). Here's an example of proper usage:

```
DATA('COMPLEX(REAL,IMAG)')
```

defines a new data type called COMPLEX, containing two fields: REAL and IMAG. Once the data type is defined, the statement:

```
X = COMPLEX(3.2, 2.0)
```

creates a COMPLEX item with an initial real part of 3.2, and imaginary part of 2.0. A pointer to the new item is assigned to variable X. The individual fields are referenced by statements such as these:

```
Z = ((REAL(X) ^ 2) + (IMAG(X) ^ 2)) ^ 0.5
REAL(X) = 1.0;      IMAG(X) = IMAG(X) / 2
```

The fields may be of any data type, including pointers to other program-defined data items.

If &DUMP = 2, then all user-defined items in the program will have their non-null contents displayed in the dump. If &DUMP = 3, null-valued fields are displayed as well.

DATATYPE

Get data type of argument

DATATYPE(arg)

Returns a string specifying the data type of the argument. Some typical arguments and their data types are:

23.4	"REAL"
12	"INTEGER"
'ABCD'	"STRING"
POS(2) 'C' LEN(3)	"PATTERN"
.Q	"NAME"
*PAT	"EXPRESSION"

If the argument is a program-defined data type, the name from the creating DATA() function is returned, such as "COMPLEX".

DATE()

Get current date and time

DATE()

In Catspaw SPITBOL, DATE() returns a 17-character string of the form:

```
"11/19/96 13:11:32"
```

representing month, day, year, hour, minute, and second.

The form of string returned by this function depends on the hardware and operating system, although it is similar in all versions of SPITBOL and SNOBOL4.

Some systems use hyphens instead of slashes between date elements. For compatibility, programs should decompose dates by positional pattern-matching, such as:

```
DATE() ? LEN(2) . MTH LEN(1) LEN(2) . DAY LEN(1) LEN(2) . YEAR
```

DEFINE

Create program-defined function

DEFINE(s)
DEFINE(s, name)

This function creates a new, program-defined function. S is a prototype string specifying the function's name, arguments, and local variables, if any. Name 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

"FNAME(ARG1,ARG2,…,ARGn)LOCAL1,LOCAL2,…,LOCALn"

where FNAME is the name of the function, and ARGi are names of formal arguments to the function. Blanks are not permitted in the prototype. The values of variables specified in the list of locals 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

Remove I/O association

DETACH(name)

Removes any input or output channel associated with the variable name. The underlying file is not affected in any way. Note that name is the *address* of the variable (e.g. .X or 'X'), not the variable itself.

DIFFER

Check if arguments are different

DIFFER(arg1, arg2)

Succeeds and returns the null string if and only if arg1 and arg2 are different. Strings, integers, and reals are different if they have unequal values. Other data types differ only if they point to different objects. If arg2 is omitted, DIFFER succeeds if arg1 is not null.

DIFFER and IDENT are the only functions where the different handling of the name operator (unary dot) may present problems when moving programs between SNOBOL4 and SPITBOL.

In SPITBOL,

```
DIFFER(.ABC, 'ABC')
```

succeeds because SPITBOL handles .ABC as a name and 'ABC' as a string. The same function would fail in SNOBOL4 because it treats .ABC and 'ABC' identically.

The above example will work properly if treated as a lexical comparison:

```
LNE(.ABC, 'ABC')
```

DUMP

Dump variables

DUMP(i)

This function sends a dump of current values to file specified with –o on the command line, or to standard output if none.

If i is set to 1, the dump includes the values of all non-constant keywords and all non-null natural variables.

If i is set to 2, the dump is expanded to include non-null array and table elements, and non-null field values of program-defined data types.

When i is 3, the dump includes null-valued variables, elements and fields, as well as statement labels.

If i is set to 0, is not present, or is any value other than 1, 2 or 3, then no dump occurs. This allows use of a switch value which can be turned on and off globally during program development.

In SNOBOL4, only non-null variables can be dumped, and any value other than 0 will cause a dump. Standard SPITBOL does not provide DUMP(3).

DUPL

Duplicate string or pattern.

DUPL(s, i)
DUPL(pattern, i)

Returns the argument string s repeated i times. The function returns the null string if i is zero, and fails if i is negative.

SNOBOL4 does not allow a pattern as a first argument.

EJECT

Eject to a new page

EJECT(channel)

This function sends an ASCII form-feed character (decimal 12) to the I/O channel specified, or to the listing file if channel is omitted.

ENDFILE

Close a file or window

ENDFILE(channel)

The argument is a channel which has previously appeared as the second argument to an INPUT or OUTPUT function call. The file attached to this channel is closed, associated storage is released, and all variables associated with the file are detached. Thus, ENDFILE should be used only when no further use is to be made of the file. If the file is to be reread or rewritten, use REWIND.

Upon program termination, SPITBOL will automatically perform an ENDFILE function on all open channels.

The channel argument passed to ENDFILE varies with different implementations of SPITBOL and SNOBOL4. For maximum portability, use small integers for the channel in any INPUT or OUTPUT function call.

EQ

Equality test for numbers

```
EQ(n1, n2)
```

This function succeeds and returns the null string if the two numeric arguments are equal. N1 and n2 must evaluate to integer or real values. The function fails if n1 is not equal to n2.

EVAL

Compile and evaluate expression

```
EVAL(s)
EVAL(n)
EVAL(expression)
```

If the argument is a string, it should contain a valid SPITBOL 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, n, it is returned unchanged. If the argument is an unevaluated expression, it is evaluated, and the result returned.

EXIT

Exit to another application or create a Save file or stand-alone application

```
EXIT(s)
EXIT(n,s)
```

EXIT(s) will terminate program execution and request that the operating system perform the system command s.

Windows 95, Windows NT, Unix and OS/2 systems readily provide this ability to transfer from SPITBOL to another program. MS-DOS requires that both remain in memory simultaneously.

If the system is able to run the given command, SPITBOL terminates execution. EXIT(s) fails otherwise. Use HOST(1, s) to perform a system command without terminating SPITBOL execution.

EXIT(n,s) creates a file that captures the current state of program execution for later resumption. After writing the file, SPITBOL terminates execution. Two file formats are possible: “save” files and load modules.

A save file is a small, compressed file containing just SPITBOL’s impure data segment and object code of the user’s program. To resume execution, the save file must be loaded by either SPITBOL proper or a special SPITBOL runtime program. Save files are keyed to the particular version of SPITBOL that created them, and must be regenerated to run with each new version of SPITBOL.

A load module file contains program object code and *all* of the SPITBOL system, including compiler and runtime. The file is self-contained, and may be directly executed by the operating system.

Argument n is –3 to create a save file, and +3 to create a load module. Other values are not permitted. Argument s is the name of the file created. If omitted, the name defaults to **a.spx** for save files, and to **a.out** or **a.exe** for load module files. We encourage users to use the **.spx** extension when supplying their own save file name.

Save files can also be created prior to program execution by using the command line option –y. Those implementations that support the production of load modules can do likewise with the –w command line option.

When a save file or load module is resumed, all files except INPUT, OUTPUT, and TERMINAL are closed. The HOST function will retrieve command line arguments present at resumption, not those given when the file was created.

All versions of Catspaw SPITBOL can create save files and load modules. Some versions of standard SPITBOL can create load modules and execute commands. Standard SNOBOL4 provides neither. SNOBOL4+ handles similar operations with functions EXECUTE and SAVE.

EXP

Compute e^n

EXP(n)

Returns the base of the natural logarithms, *e* raised to the power n. N must evaluate to a valid integer or real number.

This function is in SNOBOL4+, but not in standard SPITBOL or SNOBOL4.

FENCE

Generate fenced pattern

FENCE(pattern)

The argument must be a pattern. The function returns a pattern value which is the same as the given pattern, except that alternatives within the pattern are only seen by the scanner when it is moving forward. Pattern backup will always pass through FENCE(). Note that backup through FENCE() does not cause the match to abort, as does the &FENCE pattern; it is that alternatives within the fenced pattern are not examined when the scanner is backing up.

For example, this pattern will match a string of text up to the next comma, or where there is no comma, to the end of the string. The text is put in STR, and the match will succeed only if STR is non-null.

```
P = FENCE(BREAK(',') | REM) $ STR *DIFFER(STR)
```

Without the FENCE, failure of the DIFFER(STR) would cause the scanner to try the REM alternative regardless of whether or not a comma is found.

This function is not present in SNOBOL4.

FIELD

Get field name of defined data type

FIELD(s, i)

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, or if i is less than 1.

GE

Greater than or equal test for numbers

GE(n1, n2)

This function succeeds and returns the null string if the two numeric arguments satisfy the relationship n1≥n2. N1 and n2 must evaluate to integer or real values. The function fails if n1 is less than n2.

GT

Greater than test for numbers

GT(n1,n2)

This function succeeds and returns the null string if the two numeric arguments satisfy the relationship n1>n2. N1 and n2 must evaluate to integer or real values. The function fails if n1 is less than or equal to n2.

HOST

Obtain host computer information and machine-specific features

HOST()
HOST(i, arg1, arg2, arg3, ..., argn)

This function is in standard SPITBOL, but not in SNOBOL4. It is used to obtain access to machine-specific features.

By definition, programs using most HOST functions will not be portable to other machine environments without alteration.

With the exception of one case where HOST is called with no arguments, the normal HOST call takes an integer sub-function number as its first argument, to identify the particular machine-specific operation to be carried out.

Different sub-functions will require zero or more additional string or integer arguments. Appendix E details their usage.

There are two HOST functions that are common to all Catspaw SPITBOL implementations, and they are listed here for convenience.

HOST() returns a "host-information" string describing the system on which SPITBOL is executing. The general form of the string is:

"computer type:operating system name:site name version serial"

Not all elements may be present in all implementations. HOST first looks for a file named **/usr/lib/spithost**, and if found returns its contents as the function value. This mechanism allows you to provide any string you desire to HOST. If **/usr/lib/spithost** does not exist, HOST constructs a string using whatever information it can obtain from the operating system. In this case, a typical result returned by HOST() is:

```
"80386:MS-DOS 3.30:Macro SPITBOL 3.7(2.45 I/O) #10001"
```

HOST(0) returns command-line information provided by the **–u** option. If **program.spt** were invoked this way,

```
spitbol -u "file1,file2" program
```

then the string "file1,file2" would be returned by HOST(0). This provides a convenient way to pass run-dependent information to a program. If –u was not specified, it returns the concatenation of all command line arguments.

&PARM produces the command line string in SNOBOL4+; there is no equivalent in standard SNOBOL4. HOST(1) returns the entire command line in PC-SPITBOL.

IDENT

Check if arguments are identical

```
IDENT(arg1, arg2)
```

Succeeds and returns the null string if and only if arg1 and arg2 are identical. Strings, integers, and reals are identical if they have the same values. Other data types are identical only if they point to the same object. If arg2 is omitted, IDENT succeeds if arg1 is the null string.

As with DIFFER, SPITBOL's treatment of the NAME datatype is different from SNOBOL4's, so that

```
IDENT(.ABC, 'ABC')
```

fails in SPITBOL but succeeds in SNOBOL4. To make your program portable, use:

```
LEQ(.ABC, 'ABC')
```

when comparing names and strings, since it performs identically in both dialects.

INPUT

Open file or pipe for input

```
INPUT(name, channel, s)
OUTPUT(name, channel, s)
```

Rather than give a separate description of the OUTPUT function which would be almost exactly parallel with that for INPUT, a joint description is given here.

These functions create an association between a variable and a file or pipe for input or output. The file is opened or the pipe created at the time of the INPUT or OUTPUT call. If successful, data may then be read from or written to the file or pipe, otherwise the function fails. The creation of pipes from within a SPITBOL program is only available on systems which support multi-tasking (AIX, OS/2, Unix, Windows NT, etc.).

name The name of the variable (given either in quotes or with a preceding period) to be input or output associated. Following a call to INPUT, references to this variable will cause the associated file, if any, to be read and the next record returned as the value of the variable. Following a call to OUTPUT, assignments to this variable will cause the assigned value to be written as the next record to the output file. For example:

```
INPUT('IN', …
LINE = IN
```

or

```
OUTPUT(.OUT, …
OUT = LINE
```

channel an integer or string that has no significance to the SPITBOL system other than that it represents a unique binding between a SPITBOL I/O stream and a file. This value can be used in subsequent calls to BACKSPACE, EJECT, ENDFILE, REWIND, and SET. For example:

```
INPUT('IN', 3, …
```

or

```
INPUT('IN', 'INCHANNEL', …
```

If an integer channel number is used, the user may omit the third argument, s, and provide it instead on the command line using an option of the form –channel=s. That is:

```
>spitbol –3=infile.dat prog.spt
 INPUT('IN', 3)                        (within prog.spt)
```

is equivalent to

```
>spitbol prog.spt
 INPUT('IN', 3, 'infile.dat')          (within prog.spt)
```

If the third argument, s, is omitted, the file name can be provided by setting a system environment variable with the same name as the channel to the desired file name:

```
>set INCHANNEL=infile.dat
>spitbol prog.spt
 INPUT('IN', 'INCHANNEL')              (within prog.spt)
```

s specifies the file name and I/O processing options. The file name, if it is not in the current subdirectory, can be proceeded by a full or partial path description. Processing options if present, must be enclosed in square brackets. S can be either a path to a file or a command string to be used as the other end of a pipe. If s is omitted, and no channel association appears on the command line or as an environment variable, then the standard system input and output channels are assumed.

For systems supporting the creation of pipes, command strings are denoted by a leading "!" (exclamation point). The character following the "!" represents the command string delimiter, which is used to separate the command string from

SPITBOL I/O processing options. (A square bracket is not sufficient as will be shown.)

Examples of I/O association *without* I/O processing options:

Read from file **foobar**:

```
INPUT( .in, 0, 'foobar' )                    :F(Cannot_Open)
line = in
```

Read from the **dir** command (via a pipe):

```
INPUT( .user, 'user', "!!dir" )
```

Write to file **summary.lst**:

```
output( .out, 'out', 'summary.lst' )
```

Write to the **wc** command (via a pipe):

```
OUTPUT( .pipe, 5, '!*wc' )
pipe = sentence
```

Open a file whose name is specified by environment variable DLOG:

```
OUTPUT(.out, "DLOG")
```

Notice that the closing delimiter for pipe commands (! and * in these examples) is not needed when there are no SPITBOL file processing options. It is not possible to read and write both ends of a command pipe, because of deadlock considerations.

There are two basic ways of transferring information to and from a file:

1. *Line mode*, where records are delimited by end-of-line characters (a carriage return/newline in MS-DOS, Windows and OS/2 systems, a line feed in Unix). SPITBOL removes them on input and appends them on output. Line mode corresponds to "normal" record-oriented I/O.

   During input, lines can be up to 1,024 characters by default; characters beyond that point are discarded (up to the end of line). Longer lines can be read by specifying a larger number in the –l option described below. On output, the string assigned to the output variable is written as one line. It can be broken into several shorter lines by using the –l option.

2. *Raw mode*, where a predetermined number of characters (set by the –r option) are read on input, and the new-line character(s) have no special significance. On output, the data is written verbatim, and the new-line character(s) are not appended. Raw mode is sometimes called "binary mode."

Line mode is the default for all I/O. Trimming of trailing blanks and tabs occurs only in line mode, and only if keyword &TRIM is non-zero. Raw mode can be obtained by specifying one of the file processing options. These options follow the file name and are enclosed in square brackets. Each option is denoted by a leading "–" (minus sign or hyphen) and is separated from other options by one or more blanks or commas.

–a	Append output to an existing file. If the file doesn't exist, then it is created. If –a or –u (see below) is not specified with the OUTPUT function, then the file is created, thereby replacing any existing file with the same name.
–b*n*	Set internal buffer size to the number *n*. This value is the byte count used for all I/O with files. The is *not* the logical record size or line length, but merely the internal buffer SPITBOL uses to hold data between reads or writes. The default is –b1024. It may not be set larger than &MAXLNGTH. Use –w to disable buffering.
–c	One-character raw mode, it is a shorthand for –r1. This is used by programs which read characters "one-at-a-time" from the keyboard.
–e	End-of-file character detection. MS-DOS, OS/2, and Windows follow the old CP/M convention that the control-Z character (decimal 26) indicates end-of-file in a text file. If this character is encountered when reading a file in line mode, SPITBOL treats it as the end-of-file, and signals failure to the user's program. When the –e switch is present, SPITBOL does not treat control-Z as a special character when reading a file in line mode. It is included with the line of data delivered to the user's program. True end-of-file is detected and signaled only after the last physical byte of the file is read. The –e switch is ignored if the file is opened in raw (binary) mode.
–f*n*	Use *n* as a file descriptor for I/O. SPITBOL assumes that *n* has been opened by the operating system. The presence of a file name and –f*n* *are mutually exclusive. File descriptor 0 is standard input, 1 is standard output, and 2 is the keyboard or screen (sometimes known as* "standard error"). Additionally, under MS-DOS, file descriptor 3 is device AUX and 4 is device PRN.
–i	Make file inheritable. Normally files opened by SPITBOL are private to your program and are not inherited by any child process. When this option is present, the file is opened such that its O/S file handle may be used by a child process.
–l*n*	Line mode: maximum record length is *n* characters. On input, characters beyond this number and up to the end-of-line in a record are ignored. During output, strings longer than this number will be divided into multiple *n*-character records.
–m*n* –n*n*	Specify line-mode end-of-line conventions. End of line can be marked by either a one- or two-character sequence. –m*n* specifies the first end-of-line character, where *n* is the decimal code of the desired character, and must be in the range 0 to 255. During line-mode input this character signals the end

of an input line. During line-mode output, it is appended to each output record.

–n*n* specifies an optional second end-of-line character. *n* is zero to specify no second character; otherwise it must be a decimal number in the range 1 to 255.

Any combination of values for –m and –n can be used on any system, making it simple to produce conversion programs that read and write files using foreign end-of-line conventions. Note that null (0) is an allowed value for –m*n*, permitting the transfer of null-delimited records. The default values of –m*n* and –n*n* are operating system dependent:

Operating System	–m	–n
MS-DOS, OS/2, Windows	13 (return)	10 (newline)
Macintosh	13 (return)	0 (none)
Unix	10 (newline)	0 (none)

The second end-of-line character is always optional. That is, when SPITBOL finds the first end-of-line character in an input record, it examines the next input character. If it is the second end-of-line character, it is discarded. If it isn't, it is remembered as the first character of the *next* record.

Alternate end-of-line characters may not be used with console input, since they are usually determined by the operating system, and hence not under SPITBOL's control.

–q*n* Quiet raw mode. Behaves like –r*n* below, except that input from the keyboard is not echoed to the screen.

–r*n* Raw mode: input record length is *n* characters. Exactly these many characters are read to satisfy an input request, except a smaller number of characters may be read if end-of-file is encountered. The string returned to the program is exactly the data that is read. End-of-line character(s) are returned as part of the string. If the system read call returns 0 bytes, the input request fails. Output record length is exactly the length of the data written; *n* is immaterial for output. It may not be set larger than &MAXLNGTH for input or output.

–s*xx* Share mode. Specifies how a file may be accessed by other programs whilst it is being accessed by SPITBOL. Possible values for *xx* are:

–sdn	deny none: others may open file for reading and writing.
–sdr	deny read: others may open file for writing.
–sdw	deny write: others may open file for reading.
–sdrw	deny read/write: others may not open the file.

If not specified, the INPUT function opens files with "deny write", and the OUTPUT function uses "deny read/write".

Thus, input files can have many readers, but output files are reserved exclusively for the use of the SPITBOL program.

–u Update mode: file will be opened for reading and writing. For example:

```
OUTPUT(.out, 1, ‘customer.dat[–u]’)
INPUT(.in, 1)
```

If the file exists, its contents are preserved. If it doesn't exist, it will be created. The order of these operations can be reversed, provided the –u option is specified with the first one:

```
INPUT(.in, 1, ‘customer.dat[–u]’)
OUTPUT(.out, 1)
```

Note that the second function of the pair does not specify a file name. It just opens the "other direction" of an existing I/O channel. Having done this, records of the file can be read and written using the associated variables. There is only one read/write pointer for the channel, and it can be manipulated with the SET function.

Only one record length can be specified for a given channel. When a file is open for update, the default output record length is the same as the default input record length, which is 1024. If a longer output length is needed, it must be set explicitly with the -l option.

–w Do I/O directly, without buffering. Buffer size (–b) is ignored. Each record is transferred by a single read or write system call with a byte count of the value in –l*n* or –r*n*, depending upon whether the file is operating in line mode or raw mode. Note that although SPITBOL is no longer buffering data, the operating system may do its own buffering. See the –y option below.

–x Make file executable. When creating an output file on file systems that mark files as "executable" (such as Unix), this option causes SPITBOL to create the file with executable status (0777 instead of 0666 under Unix).

–y Do I/O directly, without buffering in the operating system or in SPITBOL. Like the –w option, each record is transferred by a single read or write system call. In addition, if the file is being opened for output, SPITBOL instructs the operating system to disable buffering for this file ("write-through" output mode).

SPITBOL defaults to –b1024 and –l1024 for input, &MAXLNGTH for output; the line mode is the default I/O method.

Examples of I/O associations with options are shown below:

Append to file **foobar**:

```
OUTPUT( .WRITE, 0, ‘foobar[–a]’ )
```

Read with large buffer:

```
input( .in, ‘in’, ‘file.to.read[–b4096]’ )
```

Read one character at a time from terminal:

```
INPUT( .getc, 0, “[–f2 –c]” )
```

Read one character at a time from a pipe:

```
input( .getcls, 99, “!!ls![–c]’’ )
input( .gtclslong, 99, ”!!ls –l![–c]" )
```

Open a file in update mode using 128-byte fixed-length binary records:

```
INPUT(.database, “FILE”, “filename[–r128 –u]”)
OUTPUT(.database, “FILE”)
```

Write 4,000-character blocks to a magnetic tape:

```
output( .tape, ‘mt’, “/dev/rmt0[–b4000 –r1]” )
```

Read a foreign tape with unknown or varying block sizes:

```
input( .tapein, ‘mtin’, ‘/dev/rmt0[–r16000 –w]" )
tapeblock = tapein
blocksize = size( tapeblock )
```

More than one type of transfer may be associated with a channel. You do this by calling INPUT or OUTPUT after the first time with the variable name (which can be different), the channel (which must be the same), and any options. File names are not valid on calls after the first while a given channel is open, and the –b*n* option is processed only on the first call. For example:

```
INPUT( .LINE, 33, “\foo\bar” )
input( .line1, 33, ‘[–c]’ )
input( .line5, 33, ‘[–r5]’ )
```

Error messages are given for syntactically invalid third arguments but in the case where the file corresponding to a syntactically valid file name cannot be found, statement failure occurs and should be tested for in the usual way by a conditional Goto.

The maximum number of files or devices that may be open at one time is operating system dependent.

INPUT/OUTPUT functions work similarly, but are syntactically different in other systems. Standard SNOBOL4 omits the file name entirely, while SNOBOL4+ places it in a fourth argument position, and restricts channels to integers between 1 and 32. SPITBOL/370 does not attempt to open or create a file when the INPUT or OUTPUT function is performed, and thus will never signal failure. SPITBOL/370 defers opening the file until the first actual input or output operation, signalling failure at that time.

INPUT and OUTPUT functions usually need to be adjusted when moving a SNOBOL4 program between systems. Programmers moving programs from SPITBOL/370 systems will need to adjust the point at which failure is detected if a file cannot be opened.

INTEGER

Check if argument is an integer

INTEGER(arg)

Succeeds and returns the null string if arg is an integer, or a string which can be converted to an integer. Real numbers, even if integer-valued, are not considered integers by this function. If the argument is not an integer, the function fails.

SPITBOL allows leading or trailing blanks or tabs on the number if the argument is a string; SNOBOL4 does not.

ITEM

Get array or table element

```
ITEM(array, i1, i2, …, in)
ITEM(table, arg)
```

Returns the specified array or table element. I1, i2, …, in are array subscripts, and arg is a table subscript. ITEM is analogous to the APPLY function. For example, if F(X) is a program-defined function that returns an array name,

```
ITEM(F(X), 20) = 123
```

references the 20th element of that array.

The use of ITEM is never necessary in SPITBOL because of the extended syntax for array references. The above example can be written as:

```
F(X)<20> = 123
```

LE

Less than or equal test for numbers

```
LE(n1,n2)
```

This function succeeds and returns the null string if the two numeric arguments satisfy the relationship n1≤n2. N1 and n2 must evaluate to integer or real values. The function fails if n1 is greater than n2.

LEN

Matches fixed-length string

```
LEN(i)
```

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.

LEQ

Lexical equality test for strings

```
LEQ(s1, s2)
```

This function succeeds and returns the null string if s1 is lexically equal to s2. See function LGT. Note that LEQ(S1,S2) and IDENT(S1,S2) have slightly different behavior. LEQ converts its arguments to data type STRING, and performs a string comparison. IDENT compares its arguments without data type conversion. Thus LEQ('1',1) succeeds, while IDENT('1',1) fails (STRING data type is not identical to INTEGER data type).

This function is not present in standard SNOBOL4, but is in SNOBOL4+ and standard SPITBOL.

LGE

Lexical greater than or equal test for strings

LGE(s1, s2)

This function succeeds and returns the null string if s1 is lexically greater than or equal to s2. See function LGT.

This function is not present in standard SNOBOL4, but is in SNOBOL4+ and standard SPITBOL.

LGT

Lexical greater than test for strings

LGT(s1, s2)

This function succeeds and returns the null string if s1 is lexically greater than s2. 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 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.

This function is present in all versions of SNOBOL4 and SPITBOL.

LLE

Lexical less than or equal test for strings

LLE(s1, s2)

This function succeeds and returns the null string if s1 is lexically less than or equal to s2. See function LGT.

This function is not present in standard SNOBOL4, but is in SNOBOL4+. It is a standard SPITBOL function.

LLT

Lexical less than test for strings

LLT(s1, s2)

This function succeeds and returns the null string if s1 is lexically less than s2. See function LGT.

This function is not present in standard SNOBOL4, but is in SNOBOL4+. It is a standard SPITBOL function.

LN

Natural logarithm

LN(n)

Returns the natural logarithm (base *e*) of the number n. N should evaluate to a positive, non-zero integer or real number or an error message for numeric overflow results.

This function is in SNOBOL4+, but not standard SPITBOL or standard SNOBOL4.

LNE

Lexical not equal test for strings

LNE(s1, s2)

This function succeeds and returns the null string if s1 is lexically different from s2. See function LEQ.

LNE differs from DIFFER in that its arguments are converted to strings, so that both of the following fail:

```
LNE(10, '10')
LNE(.ABC, 'ABC')
```

This function is not present in standard SNOBOL4, but is in SNOBOL4+. It is a standard SPITBOL function.

LOAD

Load external function

LOAD(s1, s2)

This function loads compiled functions coded in another language. String s1 is a prototype of the form:

"FNAME(DATATYPE1,DATATYPE2,…,DATATYPEn)DATATYPEr"

where FNAME is the function name, and DATATYPEi specifies how the supplied arguments will be converted prior to calling the external function.

Recognized values for DATATYPEi are EXTERNAL, FILE, INTEGER, REAL, and STRING. Any other type string means that the argument will not be converted, and is provided to the external function in internal form. DATATYPEr specifies the type of result. It is presented to the external function, but is otherwise ignored by SPITBOL. The external function will always indicate the type of result it is returning.

String s2 is the name of the file containing the desired function. Appendix F describes the writing of assembly-language programs for use with LOAD. LOAD produces an error message if the specified file is a character device, or cannot be opened, or if insufficient memory remains to load the function. This error can be trapped by SETEXIT() if necessary.

LOAD will produce an error message in implementations where it is not supported.

LOCAL

Get local variable name from function definition

LOCAL(name, 1)

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 name's definition. LOCAL is useful when one function is used to trace another. The trace function can access the local variables used with the function being traced with an indirect reference: $LOCAL(name, i).

LPAD

Pad left end of string

LPAD(s1, i, s2)

This function is useful for right-justifying columnar output. It returns s1 padded on its left end until its total size is i characters. The pad character used is the first character of s2 if present, otherwise a blank (ASCII character 32) is used if s2 is absent or null. If i is less than or equal to the length of s1, s1 is returned unchanged.

This is in standard SPITBOL and SNOBOL4+, but not in standard SNOBOL4.

LT

Less than test for numbers

LT(n1, n2)

This function succeeds and returns the null string if the two numeric arguments satisfy the relationship n1<n2. N1 and n2 must evaluate to integer or real values. The function fails if n1 is greater than or equal to n2.

NE

Not equal test for numbers

NE(n1,n2)

This function succeeds and returns the null string if the two numeric arguments are not equal. N1 and n2 must evaluate to integer or real values. The function fails if n1 is equal to n2.

NOTANY

Match one character not in set

NOTANY(s)

Matches exactly one character provided it is not in the set of characters specified by the argument string.

OPSYN

Create operator synonym

OPSYN(s1, s2, i)

The function or operator name s1 becomes a synonym for s2.

The second argument must always be an already-defined function name.

If i is omitted or 0, the first argument must be a function name; it cannot be an operator.

If i is 1, s1 must be an undefined unary operator (! % / # = |). If i is 2, then s1 must be an undefined binary operator(& @ # % ~). See Chapter 15, "Operators," for more information about these operators.

In all three cases, subsequent use of s1 results in calling of the function corresponding to s2 with the appropriate arguments.

See Chapter 8, EProgram-Defined Objects," for examples of OPSYN at work.

In contrast with SNOBOL4, the second argument must always be an already defined function name. SPITBOL also differs from SNOBOL4 by not allowing built-in functions or operators to be redefined.

OUTPUT

Open file or pipe for output

OUTPUT(name, channel, s)

This function has similar arguments and behaves similarly to INPUT, which should be consulted for a detailed description. The difference is that the first argument becomes output associated so that assignments to this variable subsequent to the OUTPUT function call will cause the assigned value to be written as the next record to the output file. Also, if the second and third arguments are omitted, then the variable is by default associated with the standard output file.

POS

Verify scanner position

POS(i)

Succeeds if the pattern matcher'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.

PROTOTYPE

Get prototype which created an array

PROTOTYPE(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 or SORT functions, the string has the form 'N,2', where N is the integer number of rows in the array.

REMDR

Get remainder after division

REMDR(n1, n2)

If n1 and n2 are both integers, REMDR returns the integer remainder resulting from n1 divided by n2, that is, n1 modulus n2. If either n1 or n2 are real values, REMDR calculates n1–CHOP(n1/n2)*n2, where CHOP truncates its real argument to an integer by rounding toward zero. In both cases, the result has the same sign as n1.

In SNOBOL4+ and MaxSPITBOL, REMDR works as it does here. In standard SNOBOL4 and standard SPITBOL, REMDR accepts only integers.

REPLACE

Replace characters in string

REPLACE(s1, s2, s3)

This function returns s1 transformed according to a translation specified by s2 and s3. Each character of s1 found in s2 is replaced by the corresponding character in s3. S2 and s3 must be the same length. If duplicate characters appear in s2, the rightmost one is used to obtain the mapping character from s3. Normally, s2 and s3 are thought of as parameters, and REPLACE performs character substitutions on the variable s1. 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 s1 and s2 are considered parameters, and s3 allowed to vary. If s1 and s2 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'). S1 and s2 can be different lengths—only s2 and s3 must be the same size. If s2 contains characters not in s1, the corresponding characters in s3 are dropped from the result. If s1 contains characters not in s2, they will appear in the result. The function call

```
REPLACE('Yy/Mm/Dd', 'Mm/Dd/Yy xx:xx:xx', DATE())
```

returns the date in the form YY/MM/DD (e.g., 88/11/20). Duplicate characters in s1 are permitted, so:

```
REPLACE('aaabbbccc', 'abc' '(1)')
```

produces '(((111)))'.

REVERSE

Swap string end-for-end

```
REVERSE(s)
```

This function returns a mirror image (end-for-end reversal) of string s.

This function is in SNOBOL4+ and SPITBOL, but it is not included in standard SNOBOL4.

REWIND

Rewind file

```
REWIND(channel)
```

The file associated with the specified channel is rewound. The next read or write will take place at the beginning of the file. Existing variable associations to the channel are unaffected.

RPAD

Pad right end of string

```
RPAD(s1, i, s2)
```

This function is useful for left-justifying columnar output. It returns s1 padded on its right end until its total size is i characters. The pad character used is the first character of s2 if present, otherwise a blank (ASCII character 32) is used if s2 is absent or null. If i is less than or equal to the length of s1, s1 is returned unchanged.

This is in standard SPITBOL and SNOBOL4+, but not in standard SNOBOL4.

RPOS

Verify scanner position from end

```
RPOS(i)
```

Succeeds if the pattern matcher's current cursor position is the specified number of characters from the end of the subject 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.

RSORT

Sort array or table in descending order

```
RSORT(array, i)
RSORT(array, name)
RSORT(table, i)
```

The specified array or table is sorted in descending order, with larger keys appearing before smaller ones in the resultant array. Consult function SORT for more information on how a sort is performed.

Sorting functions are present in most implementations of SPITBOL, and in SNOBOL4+, but not in standard SNOBOL4.

RTAB

Match through position counting from end

```
RTAB(i)
```

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. RTAB(0) is equivalent to the primitive pattern REM, matching to the end of the subject string. RTAB will match the null string. The function fails if the current scanner position is to the right of the target position.

SET

Set file position

SET(channel, i1, i2)

The channel must have been established by a previous INPUT or OUTPUT function. SET repositions the file attached to this channel and returns the new position as the result. The next read or write operation will take place at that position in the file. The general form of the call is:

```
SET(channel, offset, whence)
```

Offset and whence are integers. If whence is 0, the file pointer is set to offset bytes from the beginning of the file.

If whence is 1, the file pointer is set to its current position plus offset. (Offset may be negative to position backward.) SET(channel, 0, 1) simply returns the current file position without changing it.

If whence is 2, the pointer is set to the current size of the file plus offset. Here offset must be zero or negative to position backward from the end of file.

This function is part of standard SPITBOL, but is handled differently in SNOBOL4+, which uses the SEEK and TELL functions to perform the same tasks.

SETEXIT

Set error exit

SETEXIT()
SETEXIT(name)

SETEXIT allows interception of execution errors, including any detected during calls of CODE and EVAL. The argument is a label to which control is passed if a subsequent error occurs, providing that the value of &ERRLIMIT is non-zero. (This is a direct branch to the label, *not* a function call.) The value of &ERRLIMIT is decremented when the error trap occurs. A SETEXIT call with a null argument causes cancellation of error intercepts. A subsequent error will terminate execution as usual with an error message.

The result returned by SETEXIT is the previous intercept setting (i.e., a label name or null if no intercept was set. This can be used to save and restore the SETEXIT conditions recursively. The error routine may inspect the error code and text stored in &ERRTYPE and &ERRTEXT, and take one of the following actions.

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

2. Branch to the special system label CONTINUE. This causes execution to resume by branching to the failure exit of the statement in error.

   Caution: If your error routine was called because of problems in the Goto field, such as an undefined label or evaluation failure, CONTINUE will try to perform that same Goto again. The system will loop this way until &ERRLIMIT becomes zero. Your error routine should test &ERRTYPE for Goto-type errors, numbers 20, 23, 24, and 38, and transfer to ABORT instead of CONTINUE in these cases.

3. Branch to the special system label SCONTINUE. This causes execution to resume at the point of interruption, by branching into the statement. The statement resumes execution and succeeds or fails normally. Generally this should only be attempted after receiving an error 320 (user interrupt). It lets the user's SETEXIT function record a keyboard-generated interrupt for processing later.

4. Continue execution elsewhere by branching to another section of the program. If the error occurred inside a function, execution is still "down a level."

5. If the error occurred inside a function (&FNCLEVEL is non-zero), branch to label RETURN, FRETURN, NRETURN. This avoids possible difficulties with alternative 4.

The occurrence of an error cancels the intercept. An error routine must reissue a SETEXIT call if error interception is to continue.

To display an error and continue processing after an error, use program statements like this:

```
          SETEXIT(.ERRTRACE)
          &ERRLIMIT = 100
           ...
ERRTRACE
          OUTPUT = "Error #" &ERRTYPE ", " &ERRTEXT
          SETEXIT(.ERRTRACE)
          (DIFFER(&ERRTYPE,20) DIFFER(&ERRTYPE,23)
+          DIFFER(&ERRTYPE,24) DIFFER(&ERRTYPE,38))
+                       :S(CONTINUE)F(ABORT)
```

Another example of error processing may be found in the SPITBOL program **code.spt**.

SETEXIT is part of standard SPITBOL, but not SNOBOL4, although SNOBOL4+ provides a SETBREAK function to trap user interrupts.

SIN

Sine of an angle

SIN(n)

Returns the sine of n, an angle expressed in radians.

This function is not part of standard SPITBOL or SNOBOL4.

SIZE

Get length of string

SIZE(s)

The function SIZE returns an integer value which is the number of characters in its argument string. A null string argument returns 0.

SORT

Sort array or table in ascending order

SORT(array, i)
SORT(array, name)
SORT(table,i)

This function will sort one column of an array or table in ascending order, with larger keys appearing after smaller ones in the resultant array. Function RSORT may be used to sort in descending order.

If a table is provided as the first argument, it will be converted automatically to an array. If an array is provided, the sort is performed on a copy of the array; the original array is unchanged. In either case, the sorted array is returned as the result of the function.

When an array is specified, it must be a one- or two-dimensional array. The second argument specifies a column number. If it is omitted, it defaults to the smallest column number for the array. Comparisons are made using elements of this column. When two elements are found to be out of sequence, the entire rows to which they belong are exchanged.

If the first argument is a table, it is first converted to a two-dimensional array, by internally invoking the CONVERT(table, "ARRAY") function. This produces an N row by 2 column array, where the first column contains the table keys, and the second column contains the table values. Only table entries with non-null values are placed in the array. When using a table, the second argument should be 1 to sort by key, or 2 to sort by entry value. If all table entries are null, SORT fails. If the resultant array would be too large for the current value of &MAXLNGTH, SORT produces error number 256.

The array or table entries may contain any SPITBOL data type. The rules used when comparing two entries of different data types are:

string-string	Compared lexically
integer-integer	Compared algebraically
real-real	Compared algebraically
integer-real	Integer converted to real, then real-real comparison

other	For all other data types: If the data type of the two entries is the same, an unsigned comparison is made of their memory addresses. This effectively sorts them by time of creation, because older objects have smaller addresses. If the data types are different, the lexical data type names are used, and effectively sorts them in the order: array, code, expression, integer, keyword, name, pattern, real, string, and table. Program-defined data types will appear within this group in their proper lexical position.

In case a vector (one-dimensional array) is being sorted, the optional second argument may be the name of a field of a programmer-defined data type created by DATA(). In this case, if any of the values in the vector are of this type, the contents of the field corresponding to name are used as the sort key. If the second argument is omitted, the sorting is carried out by using the values the vector contains as keys. For example:

```
DATA("Complex(Real,Imag)")
Points  =  ARRAY(50)
Points[1]  =  Complex(3.4,  3.2)
  ...
Points(50)  =  Complex(0.7,  8.4)
  ...
OrderedPoints  =  SORT(Points,  .Imag)
```

will sort the array Points based upon the value of the Imag field of the program-defined datatype.

The SORT function produces an error message if the first argument is an empty table, or if the array has three or more dimensions, or if the column number is out of range.

The sorting method used is Heapsort (see Horowitz and Sahni, *Fundamentals of Data Structures*) modified so that no interchanging of equal keys occurs. It is efficient in usage of both space and time, the time taken to sort N items being proportional to N*log(N).

SNOBOL4+ and many other implementations of SNOBOL4 and SPITBOL offer built-in sorting functions, but they are not part of standard SNOBOL4.

Like SNOBOL4+, and unlike standard SPITBOL, Catspaw SPITBOL *does not* attempt to convert strings to numbers when comparing strings to reals or integers. To do so can result in an unstable sort that is dependent upon the order of the input data. If your array or table contains integers, reals, and numeric strings, the integers and reals will now sort ahead of all strings. This can be remedied by converting all numeric strings to integers or reals when they are first added to the array or table.

SPAN

Match characters in set

SPAN(s)

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.

SQRT

Square root of a number

SQRT(n)

Returns the square root of n as a real number.

This function is not part of standard SPITBOL or SNOBOL4.

STOPTR

Stop trace

STOPTR(name, type)

Discontinues the type of trace of the named item. Consult the TRACE() function for a list of tracing types available, as well as the differences between SPITBOL and SNOBOL4.

SUBSTR

Extract substring

SUBSTR(s, i1, i2)

Returns a substring extracted from the argument string. Integer i1 specifies the starting character position in s (1 = first character), and i2 is the length of the desired substring. If i2 is omitted or zero, the remaining characters of the argument string are extracted. The statement fails if i1 or i2 specify a string which is not properly contained in s.

It is always faster to use SUBSTR instead of pattern matching to extract characters from a fixed position in a string.

This function is in standard SPITBOL and SNOBOL4+, but not in standard SNOBOL4.

TAB

Match through fixed position

TAB(i)

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.

TABLE

Create a table

TABLE(i,x,arg)

A table is similar to a one-dimensional array, but the subscripts may be any SPITBOL data type. The TABLE function creates an associative table and returns a pointer to it.

All three arguments are optional, and if omitted, the defaults will be used. However, to have just the third argument in effect, it must be prefixed by two commas, as with TABLE(,,'Whatever').

The integer i is the number of hash headers used internally. If it is omitted, 11 is used by default.

If N is the number of entries in the table, and H is the number of hash headers, the average number of probes to find an entry rises from about 1.0 for small N, to N / 2H if N is large compared with H. Since the overhead for hash headers is small compared to the size of a table element, a useful guide is to use as argument i an estimate of the number of entries to be stored in the table. This value is subject to SPITBOL's general restriction on largest object size, so the value of i selected should also be less than &MAXLNGTH/4.

The second argument to TABLE, x, is of no relevance to SPITBOL and is ignored. It is part of the syntax in order to maintain compatibility with SNOBOL4.

The third argument, arg, is a value which is to be returned instead of the default null string when table look-up is performed using a key which has not been entered into the table. For instance:

```
WORDS  =  TABLE(1000,,'No  such  word')
....
TERMINAL  =  WORDS['nonesuch']
```

will produce the string "No such word".

Remember, argument i does not limit the total number of values stored in a table—it only affects the efficiency of table access. Thus, 80,000 elements might be stored in a table created by TABLE(5000) (if the workspace were large enough). A typical table access would require an average of 8 probes.

SPITBOL differs from SNOBOL4 in that referring to a non-existent table entry in SNOBOL4 will create a new entry, while in SPITBOL, an entry is inserted into a table *only* when an explicit assignment is made. Thus table look-up for a currently missing key does not create an entry for that key. SNOBOL4+ provides functions FREEZE and THAW to overcome this difficulty; these functions are not needed in SPITBOL.

TAN

Tangent of an angle

TAN(n)

Returns the tangent of n, an angle in radians. Real overflow results when the argument is a multiple of $\pm\pi/4$.

This is not in standard SPITBOL or SNOBOL4.

TIME

Get execution time

TIME()

Returns the execution time in milliseconds since the start of program execution. Under Unix, this is the amount of time spent computing, and excludes time spent waiting for I/O or keyboard input. Under MS-DOS and OS/2, this is a measure of total elapsed time, and includes all I/O wait time. Resolution is dependent upon the hardware and operating system.

The function appears in all SPITBOL and SNOBOL4 systems, but varies in the time unit used, resolution, and meaning of "execution" time.

TRACE

Trace an entity

TRACE(name1, s1, s2, name2)

The TRACE function, an invaluable debugging aid, starts a trace of the item whose name is given by the first argument. The second argument, s1, specifies the sense of the trace:

Second Argument	Trace Type
'A' or 'ACCESS'	Access
'V' or 'VALUE' or null	Value
'K' or 'KEYWORD'	Keyword
'L' or 'LABEL'	Label
'F' or 'FUNCTION'	Function call and return
'C' or 'CALL'	Function call
'R' or 'RETURN'	Function return

TRACE information is written to standard output, (normally, the screen, but it can be redirected to a disk file on the command line).

The access trace mode produces trace output each time an item is referenced. Attempts to trace a function before a DEFINE statement has been executed will produce an error. If the value of &STLIMIT is negative, &STCOUNT may not be traced. To give a visual impression of depth of nesting, the letter "I" is included in the trace output for each additional level of function call.

The keyword &TRACE must be set to nonzero for tracing to occur.

Name2 is the optional name of a program-defined function that will be called when a trace occurs. It is called with two arguments: the name of the item being traced, and argument s2, which can provide additional information to the trace routine. See Chapter 10, "Debugging," for examples of use.

TRACE has the same syntax in both SNOBOL4 and SPITBOL, but SNOBOL4 does not offer the access trace. SNOBOL4 allows a function trace to be specified *before* a function is defined with DEFINE(). SPITBOL requires that the function be defined before it can appear as the first argument to TRACE() when the second argument is 'FUNCTION', 'CALL', or 'RETURN'.

For more information on tracing, see Chapter 10.

TRIM

Remove trailing blanks and tabs

TRIM(s)

Returns the argument string with trailing blanks and tabs removed. 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')
```

Standard SNOBOL4 contains a TRIM function, but it removes blanks only, not tabs.

UNLOAD

Remove function definition

UNLOAD(name)

This undefines the user-defined function name. If name is an external function, reclaiming the memory occupied by the function is implementation dependent. See Appendix F.

In SPITBOL, only user-defined functions can be UNLOADed. SNOBOL4 allows even built-in functions to be UNLOADed.

Chapter 20
Programming Notes

Space Considerations

Generally, you can ignore the details of SPITBOL's internal construction. Data storage and memory management occur effortlessly and invisibly. However, if you are writing programs that will process large volumes of data, you may wish to understand the storage overhead associated with various data types.

The internal organization of SPITBOL is quite different from that of SNOBOL4. Consequently the relative speed of various operations differs. This section attempts to give some idea of how to obtain high efficiency in SPITBOL programs. Much of the material for this section was provided by Robert B. K. Dewar, the principal designer of Macro SPITBOL.

In the following discussions, a "word" occupies four bytes (32 bits) of RAM memory in all implementations except SPITBOL-8088, where a word is two bytes (16 bits) long.

1. The ANY, NOTANY, BREAK, BREAKX, and SPAN functions use translate-and-test tables for arguments longer than one character. Such a table is 256 words plus one additional word of overhead. For each function call, a bit column is allocated so that a single table suffices for 32 calls (16 calls in SPITBOL-8088). With a constant argument, the table entry is precomputed at compile time, thus avoiding erosion of space by repeated calls in a run-time loop. Single-character arguments incur no space overhead.
2. Integers and reals have an overhead of one word above the space for their values, one word for integers and two words for reals. The long integer version of SPITBOL-8088 (**spitboll.exe**) requires two 16-bit words to store a 32-bit integer.
3. Multidimensional arrays have a space overhead of 8 + 2D words, where D is the number of dimensions. One-dimensional arrays

(vectors) with a low bound of 1 are treated specially and have an overhead of only three words. The actual array storage in an N-element array requires N words for pointers to the data (the actual data values are stored outside the array).

4. The TABLE datatype is implemented using a hash table. This consists of a block of hash headers, each pointing to a linked list of table elements. The block consists of 4 + N words, where N is the first argument used in the creating TABLE() function call (N defaults to 11 if not specified). Each non-null table element requires 4 words in addition to the space for the key and entry values themselves.
5. Program-defined data require 3 + F words, where F is the number of fields. They are quite compact and can be used freely. The data type definition (one for each different data type) requires 5F words.
6. Strings are stored in a contiguous block of words and require 2 words of overhead per string. The last word of the string is filled with zero characters if the string size is not a multiple of four.
7. Each user-created name (label, function name, or variable) produces a variable block that requires 8 words plus space for the text of the name, the characters of which are packed 4 per word. This space is constant irrespective of whether the name has a single use or is used multiply as a label, function, variable, etc. This space is never reclaimed once it has been allocated. It is thus inefficient to use variables to store associative data with the $ operator. Instead use the TABLE data type.
8. The interpretive code produced by the compiler is held in code blocks and is subject to garbage collection when no longer accessible. You can take advantage of this by writing label-free initializing code at the start of the program. Even if the labels are present, it is possible to make initializing code collectible by calling CODE with a string argument in which the labels are redeclared.
9. Considerable amounts of memory are used in repeatedly building patterns. They should either be preassigned to variables outside program loops or alternatively, if written in-line in loops, should be constant so that they may be precomputed at compile time in order to avoid this overhead.
10. Pattern concatenation is accomplished by making a copy of the left pattern operand, then pointing the copy's "subsequent" words to the right operand. This can result in substantial memory usage in the case of complex patterns. Applying the deferred evaluation operator to the left operand will result in a much smaller structure, because all that is copied is a pointer to the original pattern. That is,

```
        Pat3  =  *Pat1  Pat2
```

will be far more efficient of space than

```
        Pat3  =  Pat1  Pat2
```

when Pat1 is large.

11. Setting &TRIM non-zero ensures that memory is not wasted in storing trailing blanks in strings.
12. The COLLECT function can be used to obtain detailed information of memory utilization for various structures.

Speed Considerations

As its name implies, SPITBOL is very fast. However, a general knowledge of the relative efficiency of various SPITBOL constructions can lead to programs that rival those written in conventionally compiled languages. Here then are some guidelines to keep in mind when developing large, production programs.

1. To a greater extent than is the case with SNOBOL4, there is a loss of efficiency in encoding complex structures as strings. Use arrays, tables and program-defined datatypes where possible, since all of these are highly efficient in SPITBOL. The fast associative lookup (hashing) feature of tables make them a particularly recommended feature to be exploited in a wide range of applications.
2. Programmers frequently do not appreciate that execution speeds may be reduced by an order of magnitude if poorly designed patterns fruitlessly scan data in unanchored mode. With the pattern matching primitives of SPITBOL, it is rare that unanchored matching is necessary and since anchored matching is much less expensive, it is worth acquiring the habit of initially setting &ANCHOR non-zero. If unanchored matching is needed for some purpose, take care that it is not unduly wasteful with data for which match failure is common.
3. Immediate pattern assignment with the binary $ operator is faster than conditional pattern assignment (.), and may be used freely.
4. SPITBOL precomputes constant expressions before execution. No efficiency is lost by writing pre-evaluable patterns in-line rather than predefining them. Use of the unary unevaluated expression operator (*) to defer computation is useful in some cases. For example, consider the in-line matches:

   ```
   X ? ANY('PQR') BAL PAT 'X' RPOS(0)
   X ? ANY('PQR') BAL *PAT 'X' RPOS(0)
   ```

   The second form is more efficient, since the compiler can precompute the entire pattern where the variable PAT occurs as a deferred expression.
5. The ANY, NOTANY, BREAK, BREAKX, SPAN, RSORT and SORT functions are fast and highly recommended.

6. ARB and ARBNO are slow and can very often be avoided by using other constructions.
7. Time for datatype conversions may be significant. Where efficiency is important, avoid repeated unnecessary conversions.
8. The SETEXIT error intercepts are fast and may be used for program control as well as for debugging.
9. Tracing or I/O associating a variable substantially slows down references to it but there is no residual access penalty if the trace or I/O associations are removed by STOPTR or DETACH.
10. The unary $ (indirection) operator applied to a string argument in SPITBOL corresponds to a hash search of existing variables. The process of applying $ to a NAME (including the name of a natural variable), is much faster, which is why unary dot (name operator) returns a NAME instead of a string. It is thus better to use names rather than strings in applications such as passing variable names or labels indirectly as in

    ```
    F( .X )
    ```

    rather than

    ```
    F( “X” )
    ```

11. Use of the REPLACE function is optimized when, on repeated calls, the second and third arguments are found to be unchanged, since in this case the previously constructed replace table is re-used. REPLACE is further optimized when the second argument is &ALPHABET, because no replace table need be built at all (the third argument is used as the replacement lookup table directly). Private replacement tables can thus be constructed as follows:

    ```
           TO_LOW  =  REPLACE(&ALPHABET,  &UCASE,  &LCASE)
    LOOP   LINE  =  REPLACE(INPUT,  &ALPHABET,  TO_LOW)  :F(EOF)
    ```

12. Use ANY instead of an explicit list of one-character strings and the alternation operator. That is, use:

    ```
    ANY(“AEIOU”)     rather  than     (“A”  |  “E”  |  “I”  |  “O”  |  “U”)
    ```

13. 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 using them with replacement and replacing more than expected.
14. Keep strings modest in length. Although SPITBOL allows strings to be thousands of characters long, operating upon them is time-consuming. Furthermore, they use large amounts of memory, and force SPITBOL to rearrange memory frequently.

Other notes

1. The pattern match:

    ```
    &ALPHABET  LEN(N)  LEN(1)  $  C
    ```

puts the Nth character of the host machine character set into C. SPITBOL's built-in function CHAR is far more efficient:

```
C = CHAR(N)
```

2. The interrogation operator, unary ?, is useful to annihilate an expression which is evaluated for its side effects rather than for its value. For example:

```
S ? BREAK(*DELIM) $ K *?(TABLE = TABLE + 1)
```

Appendices

Appendix A Distribution Media

This appendix describes files on the SPITBOL distribution media. Additional programs derived from *String and List Processing in SNOBOL4* [13] are listed in Appendix B.

The three most important files are: **spitbol**, **code.spt**, and **read.me**.

read.me

Last-minute changes

This file contains last minute information about SPITBOL that became available after this manual was printed.

spitbol

SPITBOL compiler and interpreter

This is the main SPITBOL program file. Chapter 13, "Running SPITBOL," explains how to run this program. Under MS-DOS and OS/2, the file name will be **spitbol.exe**.

code.spt

Experiment with SPITBOL statements

This program allows you to enter individual statements for compilation and immediate execution.

Begin each statement by placing a blank or tab in column one to bypass the label field. Type the statement and press Return or Enter. The program will compile and execute your statement, and report its success or failure. Multiple statements and small loops may be entered by using semicolon between statements. If a statement contains a Goto field, append a semicolon to keep your Goto distinct from the one that **code.spt** appends to your input line:

```
? M = 1;LOOP TERMINAL = M; M = LT(M, 20) M + 1 :S(LOOP);
1
2
...
Success
```

You can transfer to labels S and F from within a statement to report success or failure.

As a shortcut to avoid having to type "TERMINAL =" to display an expression, an equal sign in column one will evaluate the remainder of the line:

```
?=SIN(3) + 4
4.44112001
```

You can also execute system commands by placing an exclamation point in column one:

```
?!dir *.spt
```

Case-folding is in effect, and &TRIM = 1. Statements are cumulative—a variable defined in one statement is available later. Terminate the program by typing end in column 1, or entering <EOF> (control-D or control-Z for Unix or MS-DOS, OS/2) on an empty line.

Tutorial files

These files are used in the tutorial section of the manual.

asc.inc

Obtain integer character code of a character

This simple function is explained in the description of program-defined functions in Chapter 8, "Program-Defined Objects."

capitals.dat

List of states and capitals

A sample data file containing the fifty states and their capital cities. It is used in the associative programming example in Chapter 7, "Additional Operators and Data Types."

fact.inc

Produce factorials by recursion

This file is included by **code.spt** in Chapter 8, "Program-Defined Objects," to demonstrate recursion.

faustus

Sample text data file

This is a small file of text which you can use as sample input to some of the word counting and concordance programs. We've chosen a speech from Act 5, Scene 2 of Christopher Marlowe's play.

host.inc

Synonyms for HOST function

This include file contains function definitions for the HOST functions listed in Appendix E.

palin.spt

Check for palindrome

A simple program to check for palindromic input. Try the old saying:

```
ABLE WAS I ERE I SAW ELBA
```

roman.inc

Produce Roman numerals

This function converts an integer to its representation in Roman numeral form. It illustrates the use of recursion in string processing, and is described further in Chapter 8, "Program-Defined Objects."

Demonstration files

This assortment of SPITBOL programs and functions is provided in the **demos** sub-directory. The files can be examined or printed to obtain additional information on their use.

atn.spt
atn.in

Compiler for an Augmented Transition Network

This compiler was written by Shafto [7], and is provided by permission. It compiles a network description of English sentence structure into SPITBOL code. Sentences are then the "source input" to the network, which tries to parse them.

File **atn.in** contains a sample network description and test sentences.

eliza.spt

The ELIZA program written in SPITBOL

The program ELIZA is one of the most famous early works in the field of artificial intelligence. Originally written in FORTRAN in 1966, it was converted to SNOBOL4 by R. T. Duquet in 1969. Other related files are:

eliza.txt	Article describing the ELIZA system.
eliza.scr	A sample ELIZA script.

gotos.spt

Record Gotos executed by program

This program uses the trace feature of SPITBOL to keep a history of the Gotos recently executed by your program. It is useful for post-mortem analysis of the execution path taken.

kalah.spt

Plays the African board game Mancala

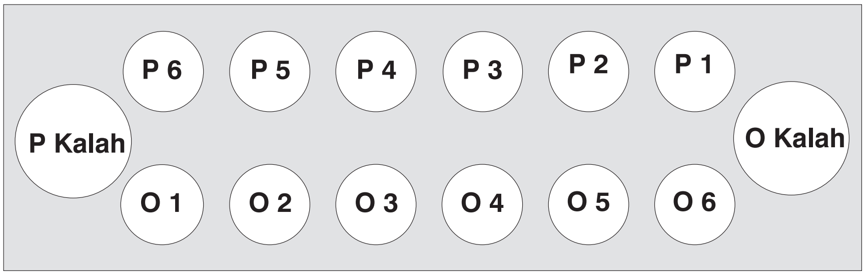
This is a large artificial intelligence program written by Shafto [7]. It was translated from a similar program written in the LISP language. Because it was meant to parallel the original LISP program, it relies more upon functions and data structures than string processing. However, reading the program file is rewarding, as it contains a wealth interesting SPITBOL programming techniques.

It is one of the many variations of Mancala games. Mancala games are popular in Africa and India. It is a very old game; boards have been found in Ancient Egyptian ruins. Some of the names of different versions are: Mankala'h, Pallanguli, Wari, Awari, and Ba-Awa.

The board consists of two rows of six depressions, called *pits* or *pots*. A larger pit at each end holds captured pieces.

The board is shown below: integers are pot numbers, "P" is the program, "O" is the opponent (user).

The move path is counter-clockwise. For the program: P1⊗P6⊗P Kalah ⊗O1⊗O6⊗P1, and for the opponent, O1⊗O6⊗O Kalah⊗P1⊗P6⊗O1.



Initially, P1-P6 and O1-O6 are filled with the desired number of stones. A move is made by taking all the stones from a numbered pot on your side, and sowing them one-by-one into succeeding pots along your path. If your last stone went into your Kalah, you get another turn. If the last stone went into a numbered pot *on your side*, which was empty, you take that stone, and any stones in your opponent's opposite pot, and place them in your Kalah. The game ends when one side has a majority of the stones in its Kalah. If it is your turn and all of your pots are empty (you have no play), the other side's stones are placed in the other side's Kalah, the game ends, and the side with the most stones wins.

The program will prompt for the initial number of stones in each pot, and the search depth that will be used for the internal tree search of moves. Try numbers of 4 and 2 respectively to get started.

keyword.spt

Find keywords in file

This program reads a list of words from file **keywords** and inserts them into a table. It then reads text from the file **keytext**, and reports the number of occurrences of the keywords in the text.

sentenc.spt

Parse simple sentences

Contributed by Michael Feldman, this program demonstrates how simple English grammars can be constructed by developing patterns incrementally from simpler ones. The program expects you to type in a sentence using its restricted vocabulary, then tells you if it is well formed according to the built-in grammar. You'll have to display the program to see the vocabulary. However, you can start out by trying the following:

```
Zippy eats the yellow banana slowly.
The aggressive monkey reads the large book, however, Dick is a boy.
```

treesort.spt

Constructing and displaying a sorted binary tree

This program was provided by Robert Dewar, author of SPITBOL, and demonstrates several programming techniques. It reads data from file **treesort.in** and enters it into two sorted trees. After all data is read, the trees are displayed using a recursive technique that displays the left sub-tree, the intermediate node, then the right sub-tree. Setting keyword &DUMP = 2 gives a dump from which you can construct a diagram of the tree structure.

Miscellaneous files

These files are provided to solve unusual programming problems. Some are specific to the SPITBOL-386 version of SPITBOL. Documentation for each is provided at the beginning of each file.

args.inc

Prepare command line arguments in array

This file uses HOST(2) and HOST(3) to copy the user's command-line arguments into an array named ARGV. The number of user arguments is provided in ARGC. The name of the user's program is provided in ARGV[0].

For example, if the program **test.spt** includes **args.inc**, then the command line:

```
spitbol test.spt aaa bbb ccc
```

will set ARGC to 3, and ARGV as follows:

```
ARGV[0] = "test.spt" ARGV[1] = "aaa"
ARGV[2] = "bbb"      ARGV[3] = "ccc"
```

The advantage of this method is that it works equally well with load modules. For example, if the programmer generated load module **test.exe**, then the command line:

```
test aaa bbb ccc
```

would produce the same result shown above, except ARGV[0] would contain "test.exe".

intcall.inc

Call MS-DOS and BIOS interrupt services

HOST function 212 provides a way to call MS-DOS and BIOS interrupt routines. This include file provides a higher level interface to HOST(212).

logic.inc
logic.slf

Perform bit-wise logical operations, unsigned arithmetic and radix conversion

This include file and external function provide bit-wise logical operations such as And and Or upon numbers or strings. It also provides radix conversion between numbers and strings of digits in an arbitrary base.

The include file defines 19 functions as shown below. The logical and arithmetic functions may be called with arguments that are integers, strings, or a mixture of each. When a string argument is used, the indicated operation is performed on a character-by-character basis. That is, to add one to *each* character in the string S, use

```
UPLUS(S, 1)
```

Similarly, each characters in strings S1 and S2 can be added on a pair-wise basis by using

```
UPLUS(S1, S2)
```

The source code for the LOGIC function may be found in file **logic.asm**in the **externals\asm** subdirectory.

NOT(X)	return logical NOT of X
AND(X1,X2)	return X1 AND X2
OR(X1,X2)	return X1 OR X2
XOR(X1,X2)	return X1 XOR X2
NAND(X1,X2)	return X1 NAND X2
NOR(X1,X2)	return X1 NOR X2
UPLUS(X1,X2)	return X1 + X2 (overflow ignored)
UMINUS(X1,X2)	return X1 – X2 (overflow ignored)
UMUL(X1,X2)	return X1 * X2 (unsigned)
UDIV(X1,X2)	return X1 / X2 (unsigned)
SHL(X1,X2)	return X1 shifted left X2 bits
SHR(X1,X2)	return X1 shifted right X2 bits (unsigned)
SAR(X1,X2)	return X1 shifted right X2 bits (sign extension)
ROL(X1,X2)	return X1 rotated left X2 bits
ROR(X1,X2)	return X1 rotated right ARG3 bits
HI(X)	hex digit string in X converted to integer
IH(X)	integer X converted to hex digit string
DIB(X,B)	base B digit string in X converted to integer
IDB(X,B)	base B integer in X converted to digit string

pathname.inc
pathname.slf

Obtain file name from SPITBOL channel number

This include file and external function uses the channel number (the second argument to the INPUT/OUTPUT functions) to obtain the name of the file opened for this channel.

pchost.inc

PC-SPITBOL-compatible synonyms for HOST function.

This include file may be substituted for **host.inc** if you need to resolve incompatibilities in HOST function numbering in PC (8088) SPITBOL.

snobol4.inc

Emulate some SNOBOL4+ extensions

Catspaw's SNOBOL4+ system provided a number of functions that are not available in SPITBOL. This include file emulates some of those functions, and provides error messages for others. Its use can assist the transition from SNOBOL4+ to SPITBOL.

Appendix B Programs from String and List Processing

The following files are provided with SPITBOL in the sub-directory **slp**. They contain SNOBOL4 programs and functions derived from the book *String and List Processing in SNOBOL4* [13].

These programs appear with permission of their author, Ralph Griswold. We've reformatted them, altered some, and provided comments.

Some functions will be useful by direct inclusion in other programs. Others are artifices to illustrate techniques made possible by the SNOBOL4 language. Many of the arithmetic functions are incomplete; some operators are provided, others are left as an exercise for the reader.

The description at the beginning of each file should be consulted for more information. The files are listed here in approximately the same order they appear in the book.

bnf.spt	Generates a recognizer for BNF grammars
grammar.def	Test grammar definition for **bnf.spt**
grammar.in	Test input file for **bnf.spt**
property.spt	Programs to examine and print property strings
tictac.spt	Plays the game Tick-Tack-Toe
center.inc	Function to center a string
rotate.inc	Function to rotate a string
delete.inc	Function to delete characters from a string
compress.inc	Function to compress characters from a string
trunc.inc	Functions to truncate and extend arrays

trim.inc	Trim arbitrary characters from the end of a string
fact.inc	Recursive and iterative factorial functions
fibon.inc	Recursive and iterative Fibonacci functions
acker.inc	Ackermann function and call depth histogram
prefix.inc	Convert expression between prefix and infix form
random.inc	Random number and random character generators
ngram.inc	Functions to produce n-gram strings
sort.inc	Shell sort written in SNOBOL4
concord.spt	Word citation in text
stack.inc	Functions to implement stack operations
queue.inc	Functions to implement list operations
bintree.inc	Functions to implement binary trees
tree.inc	Functions to implement more general trees
ration.inc	Operations on rational numbers
lrgint.inc	Operations on large integers
poly.inc	Operations on polynomials
collate.inc	Function to blend two strings
decollat.inc	Function to unblend a string
cryptsub.inc	Cryptography with substitution ciphers
crypttran.inc	Cryptography with transposition ciphers
cryptply.inc	Cryptography with polyliteral ciphers
decrypt.inc	Deciphering tools

Appendix C
Summary of Differences

The main difference between SPITBOL and standard SNOBOL4 are summarized in this appendix. SPITBOL features absent from standard SNOBOL4 but present in Catspaw's SNOBOL4+ for MS-DOS systems are noted. Much of this material was provided by Robert B.K. Dewar.

Features Not Implemented

1. The capability of redefining standard system functions and predefined operators. This restriction permits compile time pre-evaluation of a wider range of expressions and patterns than would otherwise be possible. It also affects the OPSYN function.
2. The VALUE function.
3. The keywords &STFCOUNT and &FULLSCAN. The heuristics associated with the Quickscan mode of pattern matching are complex and for many programs do not result in a significant increase in speed. Accordingly only Fullscan matching is provided and no heuristics are applied. In particular deferred expressions are not assumed to match at least one character.
4. The variable PUNCH has no predefined association to a punch stream. If PUNCH is to be referenced, a statement like OUTPUT(.PUNCH,3) should be included in the program. This will write data assigned to PUNCH to the standard output file.

Features Implemented Differently

Some of the differences here may require program changes.

1. In SPITBOL the value of &ANCHOR is obtained only at the start of the match. In SNOBOL4, changing the value during a match can lead to unexpected results.
2. The pattern valued variables ABORT, ARB, FAIL, REM and SUCCEED are write-protected so that attempts to assign to them will fail.
3. The same stack is used for pattern matching and for function calls. Thus the diagnostic issued for an infinite pattern recursion is the standard stack overflow message.
4. Recovery from most execution errors is possible. See SETEXIT() in Chapter 19, "SPITBOL Functions."
5. Input/Output. In particular, FORTRAN I/O is not provided. Dynamic association to files is possible through the third argument and statement failure is possible if a file cannot be found as in

   ```
   INPUT( .IN, 3, 'INFIL' )                        :F(NOFILE)
   ```

   SNOBOL4+ provides a similar capability.
6. The TABLE function is implemented so that table elements can be rapidly accessed by the efficient technique of hashing. In order to set a suitable size for the hash table it is important to choose a reasonable value for the argument of TABLE. Using an inappropriate value will not cause program failure but may slow down access to elements or waste memory. Hashing is more efficient if the value chosen is a prime number.

   Like SNOBOL4+ and SPITBOL/370, when a table is converted to an array, the oldest elements appear first.
7. SPITBOL allows some datatype conversions not allowed in standard SNOBOL4. For example a real value may be used in patterns and is converted to an appropriate datatype if at all possible. SNOBOL4+ provides these additional conversions.
8. The unary **.** (name) operator applied to a natural variable yields a NAME datatype, rather than a STRING. Since this NAME is converted to a STRING when necessary, the difference is not normally noticed. The only point at which the difference will be apparent is when a NAME value is used as an argument to the IDENT, DIFFER or DATATYPE functions or when it is used as a TABLE subscript. The NAME may be explicitly converted to a STRING using the CONVERT function. SNOBOL4+ and SPITBOL can be distinguished with this statement:

   ```
   IDENT(DATATYPE(.X), "NAME")        :S(SPITBOL)F(SNOBOL4)
   ```

9. SPITBOL permits leading and trailing blanks on numeric strings which are converted to integer or real numbers. SNOBOL4+ provides the same capability.

10. Several of the built-in functions are slightly different. They are identified by a double-ruled box around their names in Chapter 19, "SPITBOL Functions."

11. Constant sub-expressions and patterns are pre-evaluated at compile time. This may occasionally result in execution errors (e.g. integer overflow) being reported during compilation. Significant speed increases may be obtained by ensuring that in-line patterns are constant so that they may be pre-evaluated. Patterns built from pattern-valued variables (e.g. ARB) and pattern functions with constant arguments (e.g., ANY(*ARG), RTAB(0)) are themselves constant.

12. A compact and fast garbage collector is used that needs to distinguish between small integers and memory addresses. This effectively restricts the maximum size of any SPITBOL object (string, array, table, code or expression block, integer keyword) to be less than a value referred to as MXLEN. This is in practice not a restriction for most users. Where it might prove restrictive, the –m*n* command line option allows you to alter its value.

    Since the value of MXLEN is used to initialize &MAXLNGTH, outputting &MAXLNGTH during execution gives its exact value. You may subsequently assign a smaller value to this keyword but values exceeding that of MXLEN may not be assigned to it.

13. A value of zero for &TRIM does not necessarily imply that trailing blanks will be added to records in which they are not originally present. SNOBOL4+ provides a similar capability by setting &TRIM to –1. Trimming is always disabled when reading from a binary file.

14. In standard SNOBOL4, line-mode record lengths default to 80 characters for input and output. By default, SPITBOL provides a 1,024 character record length for input, and an infinite record length for output. Both may be changed by use of the –l record length option as described in the INPUT function in Chapter 19, "SPITBOL Functions."

15. Like standard SNOBOL4, Catspaw SPITBOL ignores unrecognized control statements. Standard SPITBOL will report an unrecognized control statement as an error.

16. SNOBOL4+ provides FREEZE and THAW functions to prevent the creation of new table entries merely upon table access. These functions are unnecessary in SPITBOL, because it only creates table entries as the result of explicit assignments.

17. SNOBOL4+'s extended pattern matching functions LEN(–1), MARB, and absolute TAB are not available in SPITBOL.

Additional Features

The following SPITBOL features are not found in standard SNOBOL4.

1. The functions BREAKX, EJECT, EXIT, FENCE, HOST, LEQ, LGE, LLE, LLT, LNE, LPAD, REVERSE, RPAD, RSORT, SETEXIT, SORT, SUBSTR. The sorting functions, the extended break pattern BREAKX, and the outputting formatting functions LPAD and RPAD are especially useful. See Chapter 19, "SPITBOL Functions," for details of all these functions. Most of these functions are provided in SNOBOL4+.
2. The numeric functions ATAN, CHOP, COS, EXP, LN, SIN, SQRT, TAN, and X**Y for real X and Y. Some of these functions are in SNOBOL4+.
3. The keywords &ERRTEXT, &FILE, &LASTFILE, &LASTLINE, and &LINE. (Also in SNOBOL4+.)
4. The symbolic dump optionally includes elements of arrays, tables and programmer-defined datatypes, and null-valued items.
5. An access trace mode is provided in addition to the other modes.
6. A selection or alternative feature is provided to evaluate successive expressions from a list of expressions. (Also in SNOBOL4+.)
7. The CONVERT functions allows conversion to "NUMERIC". Conversion will be to integer or real according to the form of the data. (Also in SNOBOL4+.)
8. The assignment symbol = is treated as an ordinary binary operator and the binary operator ? is given a defined meaning as a pattern matching operator. (Also in SNOBOL4+.)
9. The name TERMINAL is available with pre-association for input and output to the keyboard and the screen. Note that:

   ```
   T        TERMINAL  =  EVAL(TERMINAL)        :S(T)
   ```

   acts as a desk calculator. TERMINAL is also available in later versions of SNOBOL4+.
10. All lower-case letters appearing in a name are by default folded to upper case. For example, terminal and Terminal are both treated like TERMINAL. If compatibility with standard SNOBOL4 is desired, disable case-folding via the –f command line option or the –CASE control statement. (Also in SNOBOL4+.)
11. The assignment &STLIMIT = –1 inhibits all the checks on numbers of statements executed. (Also in SNOBOL4+.)
12. SPITBOL contains a statement profiler that may be used to assist in program optimization. The keyword &PROFILE controls its use.

Syntax Differences

This section describes differences in syntax between SPITBOL and standard SNOBOL4. Such differences should not generally affect existing SNOBOL4 programs.

1. Reference to elements of arrays or tables which are themselves elements of arrays or tables is possible without using the ITEM function. Thus the following are equivalent:

   ```
   A<J><K>  =  B<J><K>
   ITEM(A<J>,K)  =  ITEM(B<J>,K)
   ```

   A similar language extension is provided in SNOBOL4+.

2. The compiler permits real constants to be followed by a FORTRAN-style exponent E±NN or D±NN. (Also in SNOBOL4+.)

3. A selection or alternative construction may be written anywhere that a value is permitted. It consists of a series of expressions separated by commas and enclosed in parentheses:

   ```
   ( e1, e2, e3, …, en )
   ```

   The semantics are to evaluate the expressions from left to right until one succeeds and then use its value. Failure is signaled if all evaluations fail. This feature trivially provides an "or" function for predicates and also has many other uses as shown by the following examples:

   ```
   A  =  ( EQ(B,3), GT(B,20) ) B + 1
   NEXT  =  ( INPUT, '%EOF' )
   MAXAB  =  ( GT(A,B) A, B )
   ```

   The alternative structure provides an If-Then-Else capability, and as such is a useful programming feature. Note incidentally that the semantics of ordinary parentheses is a correct degenerate case of an alternative structure with one alternative. This selection construction is provided in SNOBOL4+, but not in standard SNOBOL4.

4. The array brackets [ ] may be used instead of < > if desired. Thus X[I,J] and X<I,J> are equivalent. (Also in SNOBOL4+.)

5. By treating = as a right associative operator of lowest priority, multiple assignments within a single statement may be coded. The value returned by an assignment is that of its right hand side. After executing

   ```
   A[J  =  J + 1]  =  INPUT
   ```

   J is the index of the element of the array into which data has been read. (Also in SNOBOL4+.)

6. The question mark symbol (?) is defined to be an explicit binary pattern-matching operator. It is left associative and has priority lower

than that of all operators except assignment (=). It returns as its value the substring matched from its left argument (a string) by its right argument (a pattern). Thus

```
‘ABCD’ ? LEN(3) $ OUTPUT ? LEN(1) REM $ OUTPUT
```

causes printing of ABC followed by BC. See “Binary operator extensions” in Chapter 9, “Advanced Topics.” (Also in SNOBOL4+.)

SPITBOL for SNOBOL4+ Users

SNOBOL4+ and SPITBOL offer a high degree of compatibility in their implementations of the SNOBOL4 programming language. However, as with any two systems that have evolved independently, differences arise. This section summarizes the things you should be aware of in order to move programs from SNOBOL4+ to SPITBOL.

Many SPITBOL features (such as BREAKX) have been included in version 2 of SNOBOL4+ under the generic name “PlusOps,” simplifying the transition to SPITBOL. However, there are some non-standard language features (and a few standard ones) in SNOBOL4+ that are not present in SPITBOL.

Identifying the system

By using only features common to both versions, it is simple to write programs that will run under either system. Even system-specific features can be used if your program identifies the system it is running under. The first step is to distinguish SNOBO4+ from SPITBOL.

```
DIFFER(.NAME,’NAME’)
          :S(SPITBOL)F(SNOBOL4)
```

will succeed under SPITBOL and fail under SNOBOL4+ (see paragraph 8, page 264 for the reason). If SPITBOL, the particular platform can be obtained from the HOST() call (no arguments).

Command line differences

1. SNOBOL4 is usually invoked from the operating system command line with a line in this form:

   ```
   SNOBOL4 filename [options]
   ```

   SPITBOL follows Unix conventions, where options are listed first:

   ```
   SPITBOL [options] filename
   ```

2. SNOBOL4 allows the characters / or – to specify options; UNIX versions of SPITBOL only permit – because / is part of a path name.
3. Both systems allow input/output files to be specified on the command line as –n:filename, where n is the decimal channel number used in the INPUT or OUTPUT function. When used in this fashion, the file name is omitted from the INPUT or OUTPUT function:

   ```
   INPUT(.Varname, n)
   ```

4. Most other command-line options have different meanings. SPITBOL's options are described in Chapter 13, "Running SPITBOL."

Features absent

1. Setting compiler options via environment variables: the SNOCMD environment variable is not supported.
2. Setting compiler options via control statements: the –OPTION control statement is not supported.
3. Extended pattern matching. SNOBOL4+'s experimental pattern matching features such as LEN(*negative argument*), MARB, ATAB, and ARTAB are not provided.

Functions absent

These SNOBOL4+ functions are absent from SPITBOL. The include file **SNOBOL4.inc** provides emulation for many of them.

ASC	obtain decimal code of character, emulated
ENVIRONMENT	look up environment variable, emulated
EXECUTE	execute shell command, emulated
FREEZE, THAW	not needed: SPITBOL only expands a table when a new value is stored, emulated by empty functions
PATHNAME	obtain channel's file name, emulated in SPITBOL-386 only via external functon in file **pathname.slf**
REMOVE	remove characters from string, emulated
SEEK	set file position, emulated
SETBREAK	trap user interrupt, emulated
STATEMENTS	obtain statement count, partially emulated
TELL	obtain file position, emulated
TRUNCATE	trancate portion of file, not emulated
VALUE(NAME)	value of field or variable, emulated

Input/output

1. In SPITBOL, the file name is provided as the third argument to the INPUT/OUTPUT functions, and processing options are appended to the file name (see INPUT, page 224). In SNOBOL4+, the third argument is used to specify options, and the fourth argument is the file name.
2. The syntax for file processing options is completely different.
3. Unit numbers 5, 6, and 7 have no special meaning in SPITBOL. File processing option –F*n* provide access to standard I/O files.
4. I/O options L and O (to fine-tune end-of-line character conventions) are provided in SPITBOL via the –M# and –N# options to the INPUT and OUTPUT functions. See those functions in chapter 19, "SPITBOL Functions."
5. I/O options T (tab expansion/compression) and Z (control-Z control) are not present in SPITBOL.

Keywords

1. &FULLSCAN does not appear in SPITBOL. Pattern matching is performed in fullscan mode automatically.
2. &PARM is not present in SPITBOL. However, function AMP_PARM() in file **SNOBOL4.inc** returns the same information.
3. &STFCOUNT does not appear in SPITBOL.

Miscellaneous

1. SPITBOL's DATE function returns the date and time in the form "DD/MM/YY HH:MM:SS". SNOBOL4+'s DATE function returns "DD–MM–YY HH:MM:SS.CC".
2. SNOBOL4+'s external functions are incompatible with SPITBOL's.

Appendix D
Error Messages

Special errors

Error messages are generally self-explanatory, and SPITBOL will display the source file name and line number where the error occurred. During compilation, the character number within the line is also displayed. However, several messages require additional explanation:

204 Memory overflow
Insufficient extended memory to load program

The program workspace was exhausted. The workspace size is limited by the physical or virtual memory provided to SPITBOL by the operating system, and by the –d command line option. Since this option defaults to 64 megabytes, it is not likely to be the limiting factor. Unix users should consult their system administrator about obtaining a larger memory partition.

Windows 95, Windows NT, and OS/2 provide virtual memory to applications, and SPITBOL will use it automatically. Memory overflow may occur if insufficient swap file space is present on the hard disk.

205 String length exceeds value of MAXLNGTH keyword

Keyword &MAXLNGTH determines the longest string allowed. The default value for this keyword is 9,000 under 8088 SPITBOL, and 4 megabytes under all other Catspaw SPITBOLs. If you need to use longer strings, increase keyword &MAXLNGTH via the –m command line option. You may also have to adjust your workspace size upward.

068 Array size exceeds maximum permitted

260 Conversion array size exceeds maximum permitted

Both errors refer to the same problem. Keyword &MAXLNGTH determines the largest block of memory that can be allocated internally by SPITBOL. The program tried to create an array that was larger than &MAXLNGTH bytes. Error 068 arises when calling the ARRAY function. Error

260 comes from converting a table to an array, either explicitly with the CONVERT function, or implicitly, by sorting a table. Consult Chapter 20, "Programming Notes," for information on how to calculate the amount of memory required for an array, and adjust &MAXLNGTH accordingly via the –m command line option.

213 Syntax error: Statement is too complicated.

SPITBOL was unable to allocate a memory block large enough to contain the object code produced when compiling a very complicated statement. Simplify the statement, or increase the largest object size via command line option –m.

246 Stack overflow

This usually indicates an error in a recursive function or pattern, with a resulting "stack plunge." If a stack larger than the default 32 kilobytes is needed, increase it with command line option –s.

299 Internal logic error: Unexpected PPM branch

This rather cryptic error should never appear. Please contact Catspaw, Inc. and supply enough documentation to allow us to duplicate the error.

329 Requested &MAXLNGTH too large

There was not enough memory to accept the size requested by the –m command line option.

Stack memory unavailable

There was not enough memory to provide the execution stack specified by the –s command line option.

Workspace memory unavailable

SPITBOL begins execution with an initial workspace size specified by the –i command line option. This defaults to 128 kilobytes, and is also the size by which the workspace is expanded when memory is 85% full. This error means there was insufficient memory to allocate the initial amount.

Error numbers

The following list details all error messages which can be generated either at compile time or execution time. The number is provided in &ERRTYPE, and the text in &ERRTEXT.

1	Addition left operand is not numeric
2	Addition right operand is not numeric
3	Addition caused integer overflow
4	Affirmation operand is not numeric
5	Alternation right operand is not pattern
6	Alternation left operand is not pattern
7	Compilation error encountered during execution
8	Concatenation left operand is not a string or pattern
9	Concatenation right operand is not a string or pattern
10	Negation operand is not numeric
11	Negation caused integer overflow
12	Division left operand is not numeric
13	Division right operand is not numeric
14	Division caused integer overflow
15	Exponentiation right operand is not numeric
16	Exponentiation left operand is not numeric
17	Exponentiation caused integer overflow
18	Exponentiation result is undefined
20	Goto evaluation failure
21	Function called by name returned a value
22	Undefined function called
23	Goto operand is not a natural variable
24	Goto operand in direct goto is not code
25	Immediate assignment left operand is not pattern
26	Multiplication left operand is not numeric
27	Multiplication right operand is not numeric
28	Multiplication caused integer overflow
29	Undefined operator referenced
30	Pattern assignment left operand is not pattern
31	Pattern replacement right operand is not a string
32	Subtraction left operand is not numeric
33	Subtraction right operand is not numeric
34	Subtraction caused integer overflow
35	Unexpected failure in -NOFAIL mode
36	Goto ABORT with no preceding error
37	Goto CONTINUE with no preceding error
38	Goto undefined label
39	External function argument is not a string
40	External function argument is not integer
41	FIELD function argument is wrong datatype
42	Attempt to change value of protected variable
43	ANY evaluated argument is not a string
44	BREAK evaluated argument is not a string
45	BREAKX evaluated argument is not a string
46	Expression does not evaluate to pattern

47	LEN evaluated argument is not integer
48	LEN evaluated argument is negative or too large
49	NOTANY evaluated argument is not a string
50	POS evaluated argument is not integer
51	POS evaluated argument is negative or too large
52	RPOS evaluated argument is not integer
53	RPOS evaluated argument is negative or too large
54	RTAB evaluated argument is not integer
55	RTAB evaluated argument is negative or too large
56	SPAN evaluated argument is not a string
57	TAB evaluated argument is not integer
58	TAB evaluated argument is negative or too large
59	ANY argument is not a string or expression
60	APPLY first arg is not natural variable name
61	ARBNO argument is not pattern
62	ARG second argument is not integer
63	ARG first argument is not program function name
64	ARRAY first argument is not integer or string
65	ARRAY first argument lower bound is not integer
66	ARRAY first argument upper bound is not integer
67	ARRAY dimension is zero,negative or out of range
68	ARRAY size exceeds maximum permitted
69	BREAK argument is not a string or expression
70	BREAKX argument is not a string or expression
71	CLEAR argument is not a string
72	CLEAR argument has null variable name
73	COLLECT argument is not integer
74	CONVERT second argument is not a string
75	DATA argument is not a string
76	DATA argument is null
77	DATA argument is missing a left paren
78	DATA argument has null datatype name
79	DATA argument is missing a right paren
80	DATA argument has null field name
81	DEFINE first argument is not a string
82	DEFINE first argument is null
83	DEFINE first argument is missing a left paren
84	DEFINE first argument has null function name
85	Null arg name or missing) in DEFINE first arg.
86	DEFINE function entry point is not defined label
87	DETACH argument is not appropriate name
88	DUMP argument is not integer
89	DUMP argument is negative or too large
90	DUPL second argument is not integer
91	DUPL first argument is not a string or pattern
92	EJECT argument is not a suitable name
93	EJECT file does not exist
94	EJECT file does not permit page eject
95	EJECT caused non-recoverable output error

96	ENDFILE argument is not a suitable name
97	ENDFILE argument is null
98	ENDFILE file does not exist
99	ENDFILE file does not permit endfile
100	ENDFILE caused non-recoverable output error
101	EQ first argument is not numeric
102	EQ second argument is not numeric
103	EVAL argument is not expression
104	EXIT first argument is not suitable integer or string
105	EXIT action not available in this implementation
106	EXIT action caused irrecoverable error
107	FIELD second argument is not integer
108	FIELD first argument is not datatype name
109	GE first argument is not numeric
110	GE second argument is not numeric
111	GT first argument is not numeric
112	GT second argument is not numeric
113	INPUT third argument is not a string
114	Inappropriate second argument for INPUT
115	Inappropriate first argument for INPUT
116	Inappropriate file specification for INPUT
117	INPUT file cannot be read
118	LE first argument is not numeric
119	LE second argument is not numeric
120	LEN argument is not integer or expression
121	LEN argument is negative or too large
122	LEQ first argument is not a string
123	LEQ second argument is not a string
124	LGE first argument is not a string
125	LGE second argument is not a string
126	LGT first argument is not a string
127	LGT second argument is not a string
128	LLE first argument is not a string
129	LLE second argument is not a string
130	LLT first argument is not a string
131	LLT second argument is not a string
132	LNE first argument is not a string
133	LNE second argument is not a string
134	LOCAL second argument is not integer
135	LOCAL first arg is not a program function name
136	LOAD second argument is not a string
137	LOAD first argument is not a string
138	LOAD first argument is null
139	LOAD first argument is missing a left paren
140	LOAD first argument has null function name
141	LOAD first argument is missing a right paren
142	LOAD function does not exist
143	LOAD function caused input error during load
144	LPAD third argument not a string

145	LPAD second argument is not integer
146	LPAD first argument is not a string
147	LT first argument is not numeric
148	LT second argument is not numeric
149	NE first argument is not numeric
150	NE second argument is not numeric
151	NOTANY argument is not a string or expression
152	OPSYN third argument is not integer
153	OPSYN third argument is negative or too large
154	OPSYN second arg is not natural variable name
155	OPSYN first arg is not natural variable name
156	OPSYN first arg is not correct operator name
157	OUTPUT third argument is not a string
158	Inappropriate second argument for OUTPUT
159	Inappropriate first argument for OUTPUT
160	Inappropriate file specification for OUTPUT
161	OUTPUT file cannot be written to
162	POS argument is not integer or expression
163	POS argument is negative or too large
164	PROTOTYPE argument is not valid object
165	REMDR second argument is not numeric
166	REMDR first argument is not integer
167	REMDR caused integer overflow
168	REPLACE third argument is not a string
169	REPLACE second argument is not a string
170	REPLACE first argument is not a string
171	Null or unequally long 2nd, 3rd args to REPLACE
172	REWIND argument is not a suitable name
173	REWIND argument is null
174	REWIND file does not exist
175	REWIND file does not permit rewind
176	REWIND caused non-recoverable error
177	REVERSE argument is not a string
178	RPAD third argument is not a string
179	RPAD second argument is not integer
180	RPAD first argument is not a string
181	RTAB argument is not integer or expression
182	RTAB argument is negative or too large
183	TAB argument is not integer or expression
184	TAB argument is negative or too large
185	RPOS argument is not integer or expression
186	RPOS argument is negative or too large
187	SETEXIT argument is not label name or null
188	SPAN argument is not a string or expression
189	SIZE argument is not a string
190	STOPTR first argument is not appropriate name
191	STOPTR second argument is not trace type
192	SUBSTR third argument is not integer
193	SUBSTR second argument is not integer

194	SUBSTR first argument is not a string
195	TABLE argument is not integer
196	TABLE argument is out of range
197	TRACE fourth arg is not function name or null
198	TRACE first argument is not appropriate name
199	TRACE second argument is not trace type
200	TRIM argument is not a string
201	UNLOAD argument is not natural variable name
202	Input from file caused non-recoverable error
203	Input file record has incorrect format
204	Memory overflow
205	String length exceeds value of MAXLNGTH keyword
206	Output caused file overflow
207	Output caused non-recoverable error
208	Keyword value assigned is not integer
209	Keyword in assignment is protected
210	Keyword value assigned is negative or too large
211	Value assigned to keyword ERRTEXT not a string
212	Syntax error: Value used where name is required
213	Syntax error: Statement is too complicated.
214	Bad label or misplaced continuation line
215	Syntax error: Undefined or erroneous entry label
216	Syntax error: Missing END line
217	Syntax error: Duplicate label
218	Syntax error: Duplicated goto field
219	Syntax error: Empty goto field
220	Syntax error: Missing operator
221	Syntax error: Missing operand
222	Syntax error: Invalid use of left bracket
223	Syntax error: Invalid use of comma
224	Syntax error: Unbalanced right parenthesis
225	Syntax error: Unbalanced right bracket
226	Syntax error: Missing right paren
227	Syntax error: Right paren missing from goto
228	Syntax error: Right bracket missing from goto
229	Syntax error: Missing right array bracket
230	Syntax error: Illegal character
231	Syntax error: Invalid numeric item
232	Syntax error: Unmatched string quote
233	Syntax error: Invalid use of operator
234	Syntax error: Goto field incorrect
235	Subscripted operand is not table or array
236	Array referenced with wrong number of subscripts
237	Table referenced with more than one subscript
238	Array subscript is not integer
239	Indirection operand is not name
240	Pattern match right operand is not pattern
241	Pattern match left operand is not a string
242	Function return from level zero

243	Function result in NRETURN is not name
244	Statement count exceeds value of STLIMIT keyword
246	Stack overflow
247	Invalid control statement
248	Attempted redefinition of system function
249	Expression evaluated by name returned value
250	Insufficient memory to complete dump
251	Keyword operand is not name of defined keyword
252	Error on printing to interactive channel
254	Erroneous argument for HOST
255	Error during execution of HOST
256	SORT/RSORT 1st arg not suitable ARRAY or TABLE
257	Erroneous 2nd arg in SORT/RSORT of vector
258	SORT/RSORT 2nd arg out of range or non-integer
259	FENCE argument is not pattern
260	Conversion array size exceeds maximum permitted
261	Addition caused real overflow
262	Division caused real overflow
263	Multiplication caused real overflow
264	Subtraction caused real overflow
265	External function argument is not real
266	Exponentiation caused real overflow
268	Inconsistent value assigned to keyword PROFILE
270	BACKSPACE argument is not a suitable name
271	BACKSPACE file does not exist
272	BACKSPACE file does not permit backspace
273	BACKSPACE caused non-recoverable error
281	Char argument not integer
282	Char argument not in range
284	Excessively nested INCLUDE files
285	INCLUDE file cannot be opened
286	Function call to undefined entry label
287	Value assigned to keyword MAXLNGTH is too small
288	EXIT second argument is not a string
291	SET first argument is not a suitable name
292	SET first argument is null
293	Inappropriate second argument to SET
294	Inappropriate third argument to SET
295	SET file does not exist
296	SET file does not permit setting file pointer
297	SET caused non-recoverable I/O error
298	External function argument is not file
299	Internal logic error: Unexpected PPM branch
301	ATAN argument not numeric
302	CHOP argument not numeric
303	COS argument not numeric
304	EXP argument not numeric
305	EXP produced real overflow
306	LN argument not numeric

307	LN produced real overflow
308	SIN argument not numeric
309	TAN argument not numeric
310	TAN produced real overflow
311	Exponentiation of negative base to non-integral power
312	REMDR caused real overflow
313	SQRT argument not numeric
314	SQRT argument negative
315	LN argument negative
320	User interrupt
321	Goto SCONTINUE with no preceding error
326	Calling external function – bad argument type
327	Calling external function – not found

(Save files and execution modules do not include any Dynamic Link Libraries loaded by a program. They must be explicitly reloaded when execution resumes. Error 327 will appear if an external function is called without it having been reloaded.)

328	LOAD function – insufficient memory
329	Requested MAXLNGTH too large

Appendix E
The HOST Function

This appendix describes SPITBOL's HOST function. HOST provides actions specific to the computer system on which your program is running. The integer first argument to HOST() describes the particular action desired.

Introduction

Unfortunately, HOST functions were never standardized in the different versions of Macro SPITBOL. Only one HOST action is standard across all versions of Macro SPITBOL:

- Obtain a string that identifies the host computer and its operating system

A second HOST function is standard in all Catspaw versions of SPITBOL, but different in 8086 PC-SPITBOL:

- Obtain any command-line or parameter string entered by the user prior to execution (HOST(0) on Macintosh, Unix, and SPITBOL-386, HOST(1) in PC SPITBOL).

We have provided include file **pchost.inc** in an attempt to resolve the incompatibilities with PC SPITBOL. If you have a program that must run on both PC SPITBOL and one of Catspaw's SPITBOLs, copy this file into your program. It provides a new function, PCHOST, that can be used instead of HOST. It maps PC-SPITBOL HOST functions 1 through 4 to their equivalent Catspaw SPITBOL functions, if any.

Any other use of the HOST function is likely to introduce machine-specific dependencies into your program.

Having to remember the specific numbers of the various HOST functions and sub-functions is tedious and error prone. Therefore, we've provided the include file **host.inc** to encapsulate HOST calls into user-defined functions with more mnemonic names. To use these in your programming, simply add the control statement:

```
–INCLUDE  "host.inc"
```

In the descriptions that follow, we give the common-name form from the **host.inc** include file first, followed by the corresponding HOST function.

Argument Conversion

Unlike other built-in SPITBOL functions, the only automatic type conversion performed by the HOST function is from string to integer where required. *In all other cases, arguments must have the appropriate types.* If necessary, use CONVERT() within the argument list to perform an explicit conversion. For example:

```
NUM  =  3.0
   ...
HOST(2,  CONVERT(NUM,  "INTEGER"))
```

Universal HOSTs

These two sub-functions appear in all versions of Macro SPITBOL.

SYSTEM
HOST()

Get machine type and operating system

SYSTEM()
HOST()

Called with no arguments, HOST returns a "host-information" string describing the system on which SPITBOL is executing. The general form of the string in all SPITBOL implementations is:

```
"computer type:operating system name:site name version serial #"
```

For SPITBOL-386, a typical string returned by this function is:

```
"80386:MS-DOS 3.30:Macro SPITBOL 3.7(2.45 I/O) #10001"
```

PARM
HOST(0)

Obtain parameter string text

PARM()
HOST(0)

Returns the string specified with the –u command line option. To supply multiple words with this parameter, quote them: for example, –u "abc def". If

–u was not specified, it returns the concatenation of all the user's command line arguments with one blank character between each. That is, it returns all arguments *after* the user's source program name. The maximum length of the returned string is 512 characters.

This sub-function provides a convenient way to pass run-dependent information to a program. If SPITBOL is invoked with

```
spitbol –u "this is an arg" myprog.spt more stuff
```

HOST(0) returns the string "this is an arg". If invoked with

```
spitbol myprog.spt more stuff
```

HOST(0) returns "more stuff".

The dual behavior is necessary to accommodate existing Unix SPITBOL programs.

The –u option is not available to load modules created by SPITBOL. Most programmers will find that HOST sub-functions 2 and 3 (below) and the include file **args.inc** provide a simpler and more consistant way of processing user arguments.

Miscellaneous HOSTs

These HOST functions appear in all versions of Catspaw SPITBOL.

EXECUTE
HOST(1)

Execute a command string

```
EXECUTE(s1, s2)
HOST(1, s1, s2)
```

Executes the command string s1 and returns the integer result code returned by the sub-process.The string s2 is ignored, and may be omitted.

On Unix and OS/2 systems, there are no special problems with this function. However under MS-DOS, the **command.com** processor used to execute the string s1 discards any error code returned by the sub-process, returning zero instead. If access to the sub-process error code is required, SPITBOL-386 provides another method of use. S2 is the complete pathname of a COM or EXE file to execute (including any .COM or .EXE name extension. S1 is provided to the program as its "command-line string." The integer result code returned by the sub-process is returned as the function value.

By using the MS-DOS command processor, the first method (s2 null or absent) provides access to the full range of MS-DOS command line capabilities, including redirection, use of batch files and searching sub-directories specified with the MS-DOS PATH command. In doing so, any sub-process result code is lost. The second method does not permit redirection or batch files, but does return the sub-process result code.

Assume the program file **LINK.EXE** is located in sub-directory **\MS**, and that **\MS** has been included in the MS-DOS PATH command. Using the first method, we would say:

```
RESULT = EXECUTE('LINK TEST.OBJ;')
```

RESULT would be set to integer zero, even if the **LINK** program detected errors, because the DOS command processor is *between* SPITBOL-386 and **LINK**. Using the second method, we could say:

```
RESULT = EXECUTE('TEST.OBJ;', '\MS\LINK.EXE')
```

Now RESULT will be set to zero if the **LINK** program succeeds, and to a non-zero error code otherwise. Note though that we had to specify the linker's complete path name, and that the program name was omitted from the beginning of the first argument.

The choice between using these two forms is dependent on the need for the result code, the availability of memory for the command processor, and whether the target program must be started by the command processor (some MS-DOS utilities, such as **CHKDSK** are so restricted). Generally, the first form is preferred.

Use EXIT(s) to terminate SPITBOL and run another application.

GETARG
HOST(2)

Obtain argument from command line

```
GETARG(i)
HOST(2, i)
```

HOST(2, i) returns argument i (zero-based) from the command line. The function fails if i is out of range or not an integer. This facility is useful for programs that are to be executed as load modules; when executed, they can access all command line arguments.

FIRSTARG
HOST(3)

Obtain index of first unused command line argument

```
FIRSTARG()
HOST(3)
```

HOST(3) returns the index of the first command line argument not examined or processed by SPITBOL. For example, given the command line

```
spitbol prog.spt aaa bbb ccc
```

this function returns integer 2, corresponding to argument aaa. This can be combined with HOST(2) to fetch the arguments. The include file **args.inc** uses HOST(2) and HOST(3) to create the C-like array ARGV of command line arguments. ARGV[0] contains the program name. ARGC is the number of arugments.

In the example above, if **prog.spt** contained the statement

```
-INCLUDE "args.inc"
```

then following the include statement ARGC would be set to 3, and the values of ARGV would be:

```
ARGV[0] = "prog.spt"ARGV[1] = "aaa"
ARGV[2] = "bbb"     ARGV[3] = "ccc"
```

SHELLVAR
HOST(4, s)

Search environment for shell variable

```
SHELLVAR(s)
HOST(4, s)
```

HOST(4, s) returns the value of environment variable s. If the value is longer than 512 bytes, it is truncated. The function fails if the string is not found.

For MS-DOS and OS/2 systems, any alphabetic characters in the string s should be upper-case, because that is how names are saved in the environment block.

SPITBOL-386 HOSTs

The remainder of this appendix describes HOST functions that only appear in the MS-DOS and OS/2 80386 versions of Catspaw SPITBOL. They provide direct access to the screen, keyboard, memory and BIOS, as well as a limited capability to produce notes and music. With the exception of the functions numbered 200 and above, the HOST numbers and arguments are the same as PC-SPITBOL.

SOUND
HOST(200)

Sound a tone on the speaker

```
SOUND(i1, 12)
HOST(200, i1, i2)
```

This sounds a tone on the speaker with frequency i1 Hertz for duration i2 milliseconds. This is achieved under MS-DOS by reprogramming the system's timer chip, and therefore requires IBM hardware compatibility. Under OS/2, it uses a standard system service.

PLAY
HOST(201)

Play tune on speaker

```
PLAY(s)
HOST(201, s)
```

This call plays a tune according to the specifications in the string s. The format of this string is compatible with the format used in IBM BASIC, except that MF and MB are not supported, and X is not supported (or needed since the argument can be computed by concatenation). A seven octave range is supported. The following is a brief summary of the allowed sub-commands, see BASIC manual for more details:

A,B,C,D,E,F,G	sound corresponding note
–	make previous note flat
# or +	make previous note sharp
Ln	set note length, n = 1 (long) to 64 (short). Note length is 1/n (e.g., n = 8 is an eighth note). Default = 4.
n	(after note), set length for this note only (n as in Ln)
. (period)	(after note), length of this note is multiplied by 3/2.
On	set octave, n = 0 (lowest) to 6 (highest). Each octave includes the notes from C to B. Octave 3 begins with middle C. The default octave is 4.
〉	go up one octave
〈	go down one octave
Nn	play note n (0 = pause, 1 = lowest C, 84 = highest B)
Pn	Pause for length n

Tn	Set tempo, the number of quarter notes per minute. n = 32 (slow) to 255 (fast). Default = 120.
ML	Legato mode (note plays full time)
MN	Normal mode (note plays 7/8th of time, default)
MS	Staccato mode (note plays 3/4th of time)

Because OS/2 is a multi-tasking system, time periods may not be as accurate as operation under MS-DOS.

CURSOR
HOST(5)

Set cursor type

```
CURSOR(i)
HOST(5, i)
```

This call sets the cursor type as indicated by the integer code i:

i = 0	Normal underline cursor
i = 1	Full block cursor
i = 2	Half block low on line
i = 3	Half block high on line (not supported in Windows NT version of SPITBOL)

The cursor is displayed when keyboard input is active. This function returns the previous cursor type as its return value.

GOTO
HOST(6)

Set screen position

```
GOTO(i1, i2)
HOST(6, i1, i2)
```

This call sets the current screen position to line i1 and column i2, for use by subsequently described calls. Positions are zero-based, with values dependent on the current screen size (for example, 0 to 24 and 0 to 79). See page 299 for functions that return the current screen position.299

READY
HOST(7)

Test keyboard input available

```
READY()
HOST(7)
```

This call tests if keyboard input is available, if so, it succeeds and returns the null string. If no keyboard input is available, the call fails.

READKEY
HOST(8)

Read one character from the keyboard

```
READKEY()
HOST(8)
```

This call returns an integer code for the next key from the keyboard. While waiting for the key to be pressed, the cursor is displayed at the current position (see HOST(6) call). The code returned indicates the key pressed according to the chart on the following page.

The key pressed is not echoed to the screen. If echoing is required, it must be programmed explicitly.

Numeric Key Codes Returned by HOST(8) Function

0	Ctrl @	180	ALT space bar
1-26	Ctrl A-Z	181-192	F1-F12 (unshifted)
27-31	Ctrl [\ ] ^ _	193	Home
32-126	Standard ASCII codes (from main keyboard, not numeric pad)	194	Cursor up
		195	PgUp
		196	Numeric pad minus
127	Ctrl backspace		
128	Shift backspace	197	Cursor left
129	Backspace	199	Cursor right
130	Tab	200	Numeric pad plus
131	Back tab	201	End
132-157	ALT A-Z	202	Cursor down
158	Numeric pad period	203	PgDn
159-168	Numeric pad 0-9	204	Ins
169	Numeric pad plus (shifted)	205	Del
		206-215	Shift F1-F10
170	Numeric pad minus (shifted)	216-225	Ctrl F1-F10
		226-235	ALT F1-F10
171	Numeric pad *	236	Ctrl numeric pad *
172	Enter (unshifted)	237	Ctrl cursor left
173	Shift Enter	238	Ctrl cursor right
174	Ctrl Enter	239	Ctrl End
175	Esc (unshifted)	240	Ctrl PgDn
176	Shift Esc	241	Ctrl Home
177	Ctrl Esc	242-250	ALT 1-9
179	Ctrl space bar	251	ALT 0

Note 1: this keyboard facility is not completely compatible with Prokey and similar products which intercept the system keyboard driver but fail to provide a transparent interface to the shift key status (which is needed for such distinctions as the upper- and lower-case numeric pad + and – keys).

Note 2: this function's result is an integer, *not* a string. To obtain a string, use CHAR(READKEY()).

Note 3: Under OS/2, Shift-Enter and Control-Enter are used to switch among OS/2 tasks, and may not be input to a SPITBOL program.

Note 4: The Shift-, Alt-, and Control-forms of functions keys 11 and 12 are not available for input to SPITBOL.

READFIELD
HOST(9)

Read screen field

READFIELD(i)
HOST(9, i)

This call reads a field of length i, starting at the current screen position. The data keys are echoed to the screen as they are entered and normal DOS or OS/2 editing functions are available. When Enter is pressed, the entered string is returned as the result and the screen position is updated past the entered data.

HOST Screen Calls

SPITBOL-386 maintains an internal screen buffer, separate from the actual hardware display buffer. Most of the screen calls operate on this internal buffer, and to actually update the display, one of the update calls must be used. This allows efficient screen operations, particularly when the color screen is in use. The screen operations module is written to work with monochrome, color, EGA and VGA cards. Other display systems may or may not work exactly correctly. In particular, attribute characters may be incorrect on some compatibles and the screen may flash unnecessarily.

To obtain the greatest possible display speed under MS-DOS, screen functions bypass the ROM BIOS and write directly to video memory when updating the screen.

The first use of any keyboard or screen HOST function automatically clears the screen, sets the screen position to line 0, column 0, and allocates the internal screen buffer. The default foreground and background screen attributes are obtained from column one of the last display line. These attributes can be changed with HOST(10).

If conventional screen/keyboard I/O (using variables INPUT, OUTPUT, or TERMINAL) is interspersed with these HOST functions, a similar initialization occurs before and after the conventional I/O, and upon program termination. However, default foreground and background screen attributese are *not* modified on these subsequent initializations.

SPITBOL queries the video adapter to obtain the number of rows and columns in use when execution of SPITBOL begins. This will work for all standard video adapters and display modes, but is likely to fail for non-standard (e.g., full-page) displays operating in non-standard modes. The number of columns and rows in use can be obtained from HOST(202) and HOST(205). When operating in a windowed OS/2 environmnet, the number of screen rows can be set (within limits) with HOST(211).

CLEARSCN
HOST(10)

Clear screen

CLEARSCN(s)
HOST(10, s)

This call clears both the internal screen buffer and the actual display screen. The current screen position is set to line 0, column 0.

The argument s is optional and provides default screen attributes if present. The first and second characters of s provide the normal and reverse screen attributes respectively. If only one character is provided, SPITBOL constructs the reverse screen attributes automatically by exchanging the foreground and background fields, and removing the Blink bit if necessary. The eight bits of each character consists of several fields:

MS-DOS, Windows 3.0 and OS/2
Full Screen Mode

B	Background			Foreground			
7	6	5	4	3	2	1	0

B = Blink = 128

Windows 3.0 and OS/2 Windowed Mode

Background				Foreground			
7	6	5	4	3	2	1	0

The low-order four bits specify the foreground color that characters are painted. The next three or four bits (depending on mode) specify the background color used to fill the remainder of the character cell. When operating in a windowed environment, there are sixteen possible background values. When operating in a full-screen environment, one of the background bits is used instead to control blinking versus normal characters. Color and monochrome values are shown in the table on the next page.

The argument string can be constructed using the CHAR function and the fact that the multiplying by 16 shifts a value left by four bits. For example, to specify Bright White characters (15) on a Red (4) background, use:

```
HOST(10, CHAR(15 + 4 * 16))
```

This sets the foreground value to 15 and the background value to 4. Because only one character was provided, SPITBOL constructs the reverse attributes (Red on Bright White) automatically. (In full screen mode, this would be Red on White because SPITBOL would change the 15 to 7 to avoid setting the Blink bit). To specify both attribute characters explicitly:

```
HOST(10, CHAR(15 + 4 * 16) CHAR(4 + 7 * 16))
```

Value	Monochrome	Color
0	Black	Black
1	Underline[1]	Blue
2		Green
3		Cyan
4		Red
5		Magenta
6		Brown
7	White	White
8		Gray[2]
9		Light Blue[2]
10		Light Green[2]
11		Light Cyan[2]
12		Light Red[2]
13		Light Magenta[2]
14		Yellow[2]
15	Bright White[2]	Bright White[2]

Video Character Attributes

Note 1: Monochrome Underline value in foreground field only.

Note 2: Values 8 through 15 allowed in background field only in OS/2 and Windows 3.0 windowed environments. In full-screen environments, the high-order bit specifies blinking.

DELETECHR
HOST(11)

Delete character

DELETECHR()
HOST(11)

The character at the current screen position is deleted and remaining characters are moved left, with a blank being inserted at the end of the line. This call affects only the internal screen buffer, not the display.

ERASEEOL
HOST(12)

Erase to end of line

ERASEEOL()
HOST(12)

Characters from the current screen position to the end of the line are replaced by blanks. The current position is unchanged. This call affects only the internal screen buffer, not the display.

INSERTCHR
HOST(13)

Insert character

INSERTCHR()
HOST(13)

A blank is inserted at the current screen position with remaining characters on the line being moved to the right. The current position is unchanged. This call affects only the internal screen buffer, not the display.

READSTR
HOST(14)

Read string from screen buffer

```
READSTR(i)
HOST(14, i)
```

This call reads a string from the screen buffer starting at the current position. The argument i indicates the length which must not extend past the end of the current line. On return the screen position is updated past the string read. Neither the screen buffer nor the display are affected by this call.

READCHR
HOST(15)

Read character from screen buffer

```
READCHR()
HOST(15)
```

This call reads a single character from the current screen buffer position. On return, the screen position is updated past the character read. Neither the screen buffer nor the display are affected by this call.

SCROLLDN
HOST(16)

Scroll screen down

```
SCROLLDN(i1, i2)
HOST(16, i1, i2)
```

The arguments i1 and i2 specify the top and bottom lines of a region of the screen which is to be scrolled down one line, with a new line of blanks being inserted at the top line. For example HOST(16,0,24) scrolls the entire screen down one line (if operating in 25-line mode). This call affects only the internal screen buffer, not the display.

SCROLLUP
HOST(17)

Scroll screen up

```
SCROLLUP(i1, i2)
HOST(17, i1, i2)
```

The arguments i1 and i2 specify the top and bottom lines of a region of the screen which is to be scrolled up one line, with a new line of blanks being inserted at the bottom line. For example HOST(16,0,24) scrolls the entire screen up one line (if operating in 25-line mode). This call affects only the internal screen buffer, not the display.

SETCHRATR
HOST(18)

Set attribute characters

```
SETCHRATR(i1, i2)
HOST(18, i1, i2)
```

This call modifies the video attribute characters stored in the hardware display buffer, starting at the current position. The argument i2 is an integer in the range 0-255 giving the new attribute to be set and argument i1 is the count of character positions to be set to this value. The following is a summary of the attribute characters, for more details, consult your system's technical reference manual:

Monochrome case:

0 = non-display
1 = underline
7 = normal
114 = reverse video

Color case:

16 * (background color) + (foreground color)

0 Black	4 Red
1 Blue	5 Magenta
2 Green	6 Brown
3 Cyan	7 White

For color and monochrome cases, add 128 to get a blinking field, and add 8 to get high intensity (lighter foreground color). Also see the discussion of default attributes provided by HOST(10).

On EGA and VGA systems, color numbers are treated as indices into a set of "palette" registers, providing an additional level of color mapping. These palette registers can be directly manipulated on MS-DOS systems by means of calls to the system ROM BIOS using HOST(210). The interested reader should consult a hardware technical manual for additional information on programming color EGA and VGA displays.

SETLINEATR
HOST(19)

Set line attribute

```
SETLINEATR(i1, i2)
HOST(19, i1, i2)
```

This call sets the given video attribute for all characters on a particular line in the hardware display buffer. The attribute is given in argument i2, and is the same as that given above for SETCHRATR. Argument i1 is the line number. The current screen position is not referenced and not modified.

UPDATELINE
HOST(20)

Update line

UPDATELINE(i)
HOST(20, i)

This call updates the specified display line to match the contents of the internal screen buffer. It is typically used after a call such as insert or delete character, or write string, to cause the change to become visible on the screen.

UPDATESCN
HOST(21)

Update screen

UPDATESCN()
HOST(21)

This call updates the display to match the contents of the internal screen buffer. It is typically used after a scroll call, or a series of calls writing data to the screen buffer to cause the change or changes to become visible on the screen.

WRITESTR
HOST(22)

Write string

WRITESTR(s)
HOST(22, s)

This call writes the specified string to the internal screen buffer, starting at the current screen position. On exit, the screen position is updated past the written string, which must not extend past the end of the current line. This call affects only the internal screen buffer, not the display.

WRITECHR
HOST(23)

Write character

WRITECHR(s)
HOST(23, s)

This call writes a single character (string of length 1) both to the internal screen buffer and to the display at the current screen position. On exit, the current screen column is incremented by one.

Note: HOST(23, ‘a’) differs from HOST(22, ‘a’) in that the write character call does update the display for this character, whereas the modification from the WRITESTR call is not displayed until a subsequent screen update call.

NORMATR
HOST(24)

Get normal attribute

NORMATR()
HOST(24)

This call returns an integer representing the normal display attribute, as read from the bottom screen line, column 0, at initialization time, or subsequently specified by HOST(10). Normally this value will be 7, but use of HOST(24) rather than 7 will allow a program to respect a non-standard display setting (e.g. a non-standard color setting).

REVATR
HOST(25)

Get reversed attribute

REVATR()
HOST(25)

This call returns the reversed attribute corresponding to the normal attribute returned by HOST(24). Normally this would be 114, but use of HOST(25) rather than 114 allows a program to respect a non-standard display setting.

GETTYPE
HOST(26)

Get display type

GETTYPE()
HOST(26)

This call returns 'C' if operating with the color display, and 'M' if operating with the monochrome display.

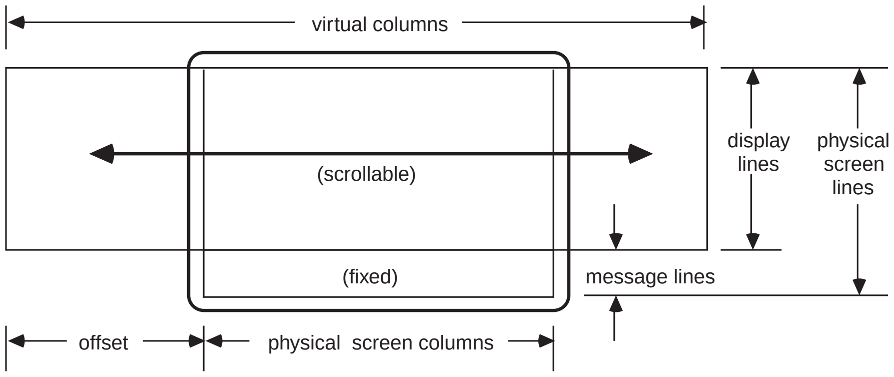
Extended Screen Calls

These HOST functions are only present in MS-DOS and OS/2 SPITBOL-386. They are absent from 8088 PC-SPITBOL.

In the preceding material, we have assumed that the internal screen buffer contains the same number of lines and columns as the physical screen, and that all lines have identical display properties. For example, when using the standard monochrome video adapter containing 25 lines of 80 characters each, the internal screen buffer will also consist of 25, 80-character lines.

In fact, the screen functions can provide the user with additional flexibility when displaying text. SPITBOL will maintain a screen buffer that is wider than the physical screen (up to 255 characters), while letting the programmer pan the physical screen left and right over this *virtual* screen.

The screen may be divided vertically into an upper region of *display lines* and a lower region of *message lines*. Display lines can be wider than the physical screen, and can be scrolled horizontally and vertically. The message lines are limited to the physical width of the screen, and cannot be scrolled in either direction. The diagram below illustrates the model implemented.



In this model, the physical screen provides a viewing window onto a series of display lines which may be panned horizontally beneath the window. The message lines are not affected by this motion. A typical application is a text editor, where edited text may be scrolled in one part of the screen while status information remains constant at the bottom of the screen.

The *offset* specifies the number of characters that the display lines are panned to the left of the physical screen. It is initially zero, and may be as large as (virtual columns – physical columns).

In the default case, the number of virtual columns equals the number of physical columns, the number of display lines equals the number of physical lines, and there are no message lines. Only vertical scrolling is possible, and all lines are homogeneous.

HOST function 209 may be used to create a larger number of virtual columns, and to divide the screen into display lines and message lines. Under such circumstances, the preceding HOST functions require some clarification:

- HOST(6) (GOTO) accepts a virtual column number, not a physical screen column when referencing a display line. Column numbering for message lines remains unchanged.
- HOST(16) and HOST(17) (SCROLLDN and SCROLLUP) affect display lines only. Message lines cannot be scrolled.
- HOST(21) (UPDATESCN) only affects the display lines. Message lines must be updated individually with HOST(20) (UPDATELINE) when modified.

GETCOLS
HOST(202)

Get number of screen columns

```
GETCOLS()
HOST(202)
```

This call returns the number of text columns on the physical screen or in the screen window under OS/2 or Windows 3.

GETETYPE
HOST(203)

Get extended adapter/display type

```
GETETYPE()
HOST(203)
```

This call returns an integer describing the type of video adapter and display present according to the following table:

0	No video adapter present
1	Monochrome display adapter (MDA)
2	Color graphics adapter (CGA)
3	EGA with monochrome display
4	EGA with color display
5	VGA with monochrome display
6	VGA with color display
7	MCGA (PS/2 model 30) with monochrome display
8	MCGA (PS/2 model 30) with color display

GETFSIZE
HOST(204)

Get screen font size

```
GETFSIZE()
HOST(205)
```

This call returns the number of video scan lines in a screen character box. Note that the actual character height will be smaller than this because of blank scan lines used to provide vertical spacing between text lines.

Typical values returned for various adapters and combinations of columns and rows:

8	CGA 80x25; EGA 80x43; VGA, MCGA 80x50
14	MDA, EGA 80x25; VGA, MCGA 80x28
16	VGA, MCGA 80x25

GETLINES
HOST(205)

Get number of screen lines

```
GETLINES()
HOST(205)
```

This call returns the number of text lines on the physical screen or in the screen window under OS/2 or Windows 3.

GETMODE
HOST(206)

Get video operating mode

```
GETMODE()
HOST(206)
```

This call returns the operating mode of the video display adapter. Consult a technical reference manual for a description of video operating modes.

GETPAGE
HOST(207)

Get current display page

```
GETPAGE()
HOST(207)
```

This call returns the number of the video display page that is currently active.

SETOFFSET
HOST(208)

Set horizontal offset

```
SETOFFSET(i)
HOST(208, i)
```

Sets the horizontal offset by which display lines are shifted left beneath the physical screen to i characters. The function limits i to values between 0 and the difference between the virtual width and the physical width. Values outside this range are ignored. The function returns the current offset value.

SETSIZE
HOST(209)

Set size of screen buffer

```
SETSIZE(i1, i2)
HOST(209, i1, i2)
```

Establishes a scrollable display region on the screen of i1 display lines, where each line has a virtual width of i2 characters. If the physical screen consists of HOST(205) physical lines, this function implicitly creates (HOST(205) – i1 message lines. I1 cannot exceed the number of physical lines, and i2 cannot exceed 255.

This function clears the screen and sets the horizontal offset to zero. It *does not* change the hardware display mode of the video adapter board.

SETLINES
HOST(211)

Set number of screen lines (OS/2 SPITBOL-386 only)

```
SETLINES(i)
HOST(211, i)
```

This function is available under OS/2 only, and only when operating in a windowed environment. It changes the vertical extent of the window. The maximum value of i is operating system dependent. The screen is cleared and the cursor set to row 0, line 0.

GETCURCOL
HOST(212)

Get current column position

```
GETCURCOL()
HOST(212)
```

This function returns the column number of the current screen position maintained by SPITBOL.

GETCURROW
HOST(213)

Get current row position

```
GETCURROW()
HOST(213)
```

This function returns the row number of the current screen position maintained by SPITBOL.

Memory Access

The following three functions provide read/write access to specific 80386 memory locations. They are available in the MS-DOS version of SPITBOL-386 only.

SETADR
HOST(27)

Set address for peek/poke (MS-DOS SPITBOL-386 only)

```
SETADR(i1, i2)
HOST(27, i1, i2)
```

This call sets the given segment selector (i1) and offset (i2) values as the current address for peek and poke calls. Note that offset values above 2 gigabyte will require that they be given in signed form, e.g. –2 for 0FFFFFFFEh).

Because SPITBOL-386 operates in protected mode, the offset is a 32-bit value. The segment selector is *not* a physical paragraph address, but rather a 16-bit number that selects various logical segments established by SPITBOL's DOS Extender. The program has read/write access to all segments. Available segment numbers are given below (decimal):

4	SPITBOL-386's program segment prefix (PSP) set up by MS-DOS.
20	Segment containing SPITBOL-386's code, data, and stack, and the workspace containing the user's program and data.
28	Segment pointing to the system's video buffer.
44	Segment pointing to the MS-DOS environment block for SPITBOL-386. The block contains a sequence of strings, each terminated with a zero character.
52	A segment mapping the first megabyte of conventional memory used by MS-DOS.

PEEK
HOST(28)

Read byte at peek/poke address (MS-DOS SPITBOL-386 only)

```
PEEK()
HOST(28)
```

This call returns the byte at the current peek/poke address as an integer in the range 0-255. The offset of the current address is incremented by 1 (with wrapping from 0FFFFFFFFh to 00000000).

POKE
HOST(29)

Store byte at peek/poke address (MS-DOS SPITBOL-386 only)

```
POKE(i)
HOST(29, i)
```

This call sets the byte at the current peek/poke address to the value given, which must be an integer in the range 0-255. The offset of the current address is incremented by 1 (with wrapping from 0FFFFFFFFh to 00000000).

BIOS, MS-DOS Interrupts

The following function provides access to the BIOS and MS-DOS system calls that are accessible through a software interrupt. Only data values may be communicated to and from these functions; there is no facility for transmitting data in memory. This function is not available under the OS/2 or Windows 9x/NT versions of SPITBOL.

INTCALL
HOST(210)

Invoke software interrupt (MS-DOS SPITBOL-386 only)

```
INTCALL(i, eax, ebx, ecx, edx, esi, edi, ebp, ds, es)
HOST(210, i, a)
```

This call invokes software interrupt i with the hardware register values contained in array a. The interrupt number i must be between 0 and 255. The function returns an array of (possibly modified) register values, including the hardware flags and condition codes returned by the interrupt call.

Because this function requires data in a highly structured form and must follow special rules, the user should not call this HOST function directly. Instead, an include file, **intcall.inc**, has been provided to assist the user.

Function INTCALL is called with an interrupt number and up to nine arguments which provide the machine registers shown in the function prototype above. Note that registers other than segment registers are 32-bit extended registers of the 80386 machine architecture. Null or omitted arguments will cause the previous value used for that register to be retained (initial value 0). Non-null arguments are explicitly converted to type integer.

INTCALL returns an array of ten integer values—the result values of the nine incoming registers, and a tenth value containing the machine status flags produced by the interrupt call. The array may be indexed by symbols defined in **intcall.inc**, specifically R_EAX, R_EBX, R_ECX, R_EDX, R_ESI, R_EDI, R_EBP, R_DS, R_ES, and R_FLG.

Interested readers should also consult the include file **logic.inc**, (see also page 259) which provides a number of bit-level logical operations on integers, as well as conversion functions between hexadecimal, octal, and decimal.

The following simple example shows how the MS-DOS version number might be obtained with this function. The MS-DOS interrupt is 21 hexadecimal, or 33 decimal. The "obtain version" sub-function is 3000h; it returns the major and minor version numbers in registers AL, and AH respectively. This example uses functions defined in **logic.inc**: HI converts a hexidecimal digit string to an integer, AND performs logical And, and SHR shifts an integer right by a given number of bit positions.

```
–INCLUDE  “INTCALL.INC”
–INCLUDE  “LOGIC.INC”
          R  =  INTCALL(33,  HI(“3000"))
          MAJOR  =  AND(R<R_EAX>,  HI(”FF"))
          MINOR  =  SHR(AND(R<R_EAX>,  HI(“FF00")),  8)
          TERMINAL  =  ”MS-DOS  Version  “  MAJOR  ”."  MINOR
END
```

Appendix F
External Functions

Introduction

Some implementations of SPITBOL allow the user to load and execute external functions written in other programming languages. Such functions can provide services that are inefficient or impossible to perform in the SNOBOL4 language. For example, specialized system calls for communications or screen graphics might be accessible to an external function written in C or assembly language.

At this time, the MS-DOS and OS/2 versions of SPITBOL-386 as well as MaxSPITBOL provide the ability to load and execute external functions. The remainder of this appendix describes the interface provided by SPITBOL-386. The material provides:

- an overview of external functions
- an exhaustive description of the assembly-language interface, including entry conventions, argument types, internal blocks, function return mechanisms, and the real number interface
- a description of how external functions may be written in the C language, provided a 32-bit C compiler is available

MS-DOS and OS/2 require slightly different implementation techniques. These are noted where appropriate.

The material in this appendix is highly technical in nature, and is not for the casual user.

SPITBOL-386 External Function Overview

Language support

External functions may be created using any assembler or C compiler capable of producing 32-bit, 80386 opcodes and operands (such as Microsoft's *MASM 5.1*, Borland's *Turbo Assembler*, or Meta Ware's *High C 386*). Assembly-language functions require no additional interface code, while some C-language functions need a small interface module to initialize C's memory allocator and I/O system. The distribution disk contains the source code for such an interface for High C. If you use a different 32-bit language or compiler, you will have to adapt the interface code to your particular system.

There are three primary forms that external functions can take:

- COM files produced by Borland's *TLINK* program or the DOS utility program *EXE2BIN*. Because of the inherent nature of COM files, functions created in this manner are limited to 64K bytes (combined code and data).
- EXP files produced by 32-bit development tools, such as Phar Lap Software's *386|LINK* program. These files can be thought of as protected-mode extensions of MS-DOS EXE files. Only the single segment, flat memory model is supported. However, segment size is limited only by available memory. Files packed (compressed) by the PharLap linker to reduce disk storage are also acceptable to SPITBOL. SPITBOL ignores any symbolic information in an EXP file, and just loads the code and data.
- DLL files are Dynamic Link Libraries produced by a wide variety of OS/2 development tools. The only restriction is that the DLL must obey SPITBOL's calling and parameter-passing conventions.

Files containing functions of the first two forms should be renamed to have file name extension "SLF" (**S**PITBOL **L**oad **F**unction) instead of "COM" or "EXP". DLLs can use the extension "DLL" or "SLF".

Due to implementation restrictions, not all file types are available to all versions of SPITBOL-386. This table shows the allowable combinations:

File Type	MS-DOS (Intel-extended)	MS-DOS (PharLap-extended)	Win 9X/NT OS/2 2.0
COM files	Yes	Yes	No
EXP files	No	Yes	No
DLL files	No	No	Yes

The same COM-file function is interchangeable between the Intel- and Pharlap-extended versions of SPITBOL. No re-assembly is necessary.

Argument types

An external function may be passed INTEGER, REAL, STRING, FILE, or EXTERNAL arguments. The data type FILE allows access to SPITBOL's internal state variables and the data buffer associated with an I/O channel. The data type EXTERNAL permits a function to accept and return its own private data blocks whose contents are preserved but otherwise ignored by SPITBOL. The function may return an INTEGER, REAL, STRING or EXTERNAL result. Other data types may be passed to the function or returned by it, but utilizing them requires a thorough understanding of SPITBOL's internal organization and block types.

Real arithmetic

Assembly language external functions have access to the SPITBOL's real number routines. These routines use an internal software emulator or numeric coprocessor as appropriate. The external function need not be aware of the particular hardware configuration. Functions coded in C should use the real arithmetic library furnished with the C compiler.

Memory model

Because SPITBOL-386 runs as a 32-bit, protected-mode program, external functions must be written to execute in 32-bit mode. This means that functions should use the 80386's 32-bit wide extended registers, and should use 32-bit memory addresses. MS-DOS functions will execute using the "small" memory model in their own private 32-bit segment, which may be up to 4 gigabytes long. In contrast, OS/2 functions will share the upper portion of SPITBOL's memory region.

Only one physical segment per function is supported; multi-segment functions are not permitted. While there is only one segment, *two* segment selector numbers are allocated to map it — a read-and-execute-only segment for code, and a read-write segment for data. They both point to the same physical memory, but have different access rights.

Memory for functions is allocated in increments of 4K bytes (the system page size).

MS-DOS: Once loaded, functions are saved and restored to and from SPITBOL save files and load modules just like any other piece of SPITBOL code or data. When run under PharLap-extended SPITBOL, a function may enlarge or reduce the size of its memory segment at any time, subject to the availability of system memory.

MS-DOS: Functions must be linked to begin at virtual offset 0 in their segment. When reloaded from a Save file, the relative position of code and data within a segment is unchanged. However, the segment selector numbers for the function may be different after a reload.

OS/2: Because Dynamic Link Libraries are loaded and managed by OS/2, they cannot be saved in save files or load modules. Any necessary DLLs must be distributed along with the save file or load module. If EXIT() was used, the external functions must be explicitly reloaded when execution resumes. DLLs are relocatable and are not linked for a particular address.

Loading an External Function

External functions are loaded with SPITBOL's LOAD function:

LOAD('fname(dt1,dt2,...,dtn)dtr', 'filename')

Fname is the name by which the function will be referenced in the remainder of the program. Fname is strictly a SPITBOL name; it need not appear anywhere in the assembly language source program.

The data type names dt1 … dtn specify how the supplied arguments will be converted prior to calling the external function. Recognized names are:

INTEGER	Pass argument as an integer.
REAL	Pass argument as a real number.
STRING	Pass argument as a string.
FILE	Pass pointer to file information for an I/O associated variable.
EXTERNAL, any other text, or null string	Argument is not converted. A pointer to the internal memory block containing the data is presented to the function.

Dtr specifies the datatype of the result returned by the function. If dtr is present *and* is INTEGER, REAL, or STRING, then the function *must* return a result of that type. Programmers should omit dtr or use unrecognized names for dtr if the function will return other data types, or if the datatype returned will vary. For example, a function might normally return an integer result, but could return a textual error message under some conditions. It should omit any specification for dtr.

It is the function's responsibility to indicate the result type being returned.

The second argument, "filename", is a string specifying a disk file name. The name itself has no special significance. If the second argument is absent, SPITBOL will attempt to load the function from a file named **fname.slf** (MS-DOS) or **fname.dll**(OS/2), where "fname" is the function name given in the first argument.

MS-DOS: SPITBOL will examine the first few bytes of the file to determine if it is a COM or EXP file, and will load it using the appropriate method. Only one function is allowed per file.

OS/2: The file must be linked as a Dynamic Link Library that exports the public label "fname". One or more external functions may be placed in the same library file, provided each functions name is exported.

Here are some sample calls to the LOAD function. Like SPITBOL's DEFINE function, LOAD is not a language declarative — it must be *executed* to actually load in the external function. Note that blanks are not permitted within the prototype string, and lower-case letters are converted to upper-case.

```
LOAD("TESTEF(STRING,INTEGER)STRING", "TESTEF.SLF")
LOAD("scan(string,file)")
LOAD("custom(,,,)", "custom.exp")
LOAD("J0(REAL)REAL", "BESSEL.DLL")
```

SNOLIB

As described in Chapter 13, "Running SPITBOL," the environment variable SNOLIB can be used to specify alternate search directories for external functions. If SPITBOL is unable to locate the external function file in the default directory, it will search in all directories specified by this environment variable.

OS/2: If filename is omitted, OS/2 will search for **fname.dll** in directories specified by the LIBPATH variable in file **config.sys**. But beware this OS/2 quirk: if LIBPATH does not include the current directory (".\"), then OS/2 *does not* look in your current directory for the file.

Creating an Assembly-Language Function

The external function is coded as a 32-bit assembly language program which is then assembled and linked to create a COM, EXP or DLL file. The details depend upon the tools available:

Microsoft MASM (MS-DOS)

Because Microsoft's *LINK* program is not 32-bit aware, certain coding restrictions are necessary. These are discussed in the following section, "A Microsoft 'Gotcha'." To assemble and link a file, use these commands:

C>masm file,,;	Assemble the function in **file.asm** to create **file.obj**.
C>link file,,;	Link the function to create **file.exe**. This operation will produce one warning message saying there is no stack segment, and should be ignored.
C>exe2bin file.exe file.slf	Create final COM file. The *EXE2BIN* program is provided on your PC/MS-DOS diskette.

Borland Turbo Assembler (MS-DOS)

Borland's *TLINK* program is 32-bit aware, and does not require any of the special coding restrictions needed when using Microsoft's tools. Use these commands to produce an assembly-language function:

C>tasm file	Assemble the function in **file.asm** to create **file.obj**.
C>tlink /3 /t /i file, file.slf	Link the function to create **file.slf**. The switches specify a 32-bit-aware link, produce a COM-style file, and force output of any trailing uninitialized data region.

PharLap 386|ASM (MS-DOS)

PharLap's tools allow you create an EXP-style file without the 64K COM-file limitation:

```
C>386asm  file –nolist       Assemble the function in file.asm to create file.obj

C>386link  file  –exe file.slf  –pack  –maxdata 0
                             Link the function to create packed file file.slf.
```

Microsoft MASM386 (OS/2)

The normal OS/2 2.0 tools can be used to create a Dynamic Link Library. In addition to the external function source file, you must create a "definition" file that tells the linker the name(s) of the functions being exported by the DLL. A typical definition file named **hyperbol.def** is shown here. It exports two external functions, SINH and COSH.

```
; HYPERBOL.DEF  module  definition  file
;
LIBRARY HYPERBOL
DESCRIPTION          'SPITBOL   Hyperbolic  math  functions'
PROTMODE
DATA                 NONSHARED
EXPORTS              SINH
                     COSH
```

The OS/2 commands to assemble and link are:

```
C>masm386  file;             Assemble the function in file.asm to create file.obj

C>link386  file,file.dll,,,file.def
                             Link the function to create DLL file file.dll.
```

Near vs. Far

MS-DOS SPITBOL-386 loads external functions as "far" procedures. That is, the function return pointer and all data pointers are 48-bit values: a 16-bit segment and a 32-bit offset. In contrast, OS/2 loads external functions as "near" procedures — all pointers are 32-bit offsets, and the segment registers are not used at all.

In languages like C, switching pointers between near and far is just a matter of changing keywords or command-line switches. Writing assembly-language code to be transparent to pointer sizes is somewhat more involved.

We take the position that assembly-language functions should only be written once. Therefore we provide you with a set of macros, equates, and structures in file **extrn386.inc** that make it possible to write functions that can be used with either the MS-DOS or OS/2 versions of SPITBOL-386. Because of limitations of the Microsoft assembler, things are not completely transparent, but they do come close.

Which version is assembled is controlled by the included file **system.inc**, which sets the equated symbol os2 to 0 or 1 for MS-DOS or OS/2 respectively. This in turn sets the symbol nearptr to 0 if far pointers are in use, or to 1 for near pointers. The remaining macros, structures, etc., are conditioned on these symbols.

Pointers are described with this dpt structure defined in **extrn386.inc**:

MS-DOS: All far pointers are 6-bytes long:

```
dpt     struc
  o     dd        ?                        ; 32-bit offset of data
  s     dw        ?                        ; 16-bit segment se-
lector
dpt     ends
```

OS/2: All near pointers are 4-bytes long, with no segment selector:

```
dpt     struc
  o     dd        ?                        ; 32-bit offset of data
dpt     ends
```

In the remainder of this appendix, we will use dpt as a generic pointer, regardless of which system it is being assembled for.

Function skeleton

MS-DOS: The source file should contain one external function, coded as a far procedure in a 32-bit segment. An ORG 0 statement guarantees that the program begins at location 0.

OS/2: The source file should contain one or more external function(s), coded as near procedure(s) in a 32-bit segment.

A prologue similar to the following should appear at the beginning of your assembly-language source file:

```
        title     FILE - sample external function "FNAME"
        .386
        name      file
include system.inc                         ;  specifies DOS or
OS/2
include extrn386.inc

        preamble  _TEXT                    ; define CGROUP
        entry     FNAME                    ; declare procedures
        enter     0,0                      ; push ebp,  mov
ebp,esp
```

The body of your function follows this prologue, and will typically end with these five lines:

```
        leave                              ; pop ebp
        ret                                ; far return
FNAME   endp

_TEXT   ends
        end                                ; transfer address
omitted
```

Note that a starting address label was not specified in the operand field of the end statement. For COM-style files MS-DOS SPITBOL always transfers to location 0. Borland's *TLINK* will complain if a label is present because it expects all COM files to begin at 100h.

The code segment (_TEXT) is read-only. If a read-write data segment is needed, change the preamble to:

```
                preamble    _TEXT,_DATA                 ; define CGROUP
```

and the end of the function to:

```
        _TEXT   ends

        _DATA   segment     dword use32 'DATA'
        ;
        ; place read-write data here
        ;
        _DATA   ends
                end                                     ; transfer address
        omitted
```

If a small amount of temporary storage is needed, a better strategy is to allocate it on the stack with the 80386 ENTER instruction.

The include files **system.inc**, **extrn386.inc** and **blocks.inc** are provided on the release disk in sub-directory **external\asm**. They contain numerous equated definitions that can be used by the external function to access SPITBOL's arguments and datatypes.

A Microsoft "Gotcha"

If you are using OS/2, or Borland's *TASM* or PharLap's *386|ASM*, skip this section. For those using Microsoft's assembler (MASM 5.1) under MS-DOS, important coding considerations apply.

While *MASM* is perfectly capable of processing 32-bit, 80386 opcodes, there are pitfalls in using Microsoft's *LINK* program to produce an EXE file. *LINK* is *not* 32-bit aware, and this causes problems for instructions containing 32-bit relative offsets. Consider this code fragment:

```
        TEST1:  MOV         EAX,EBX
                  …
                < more than 128 bytes of code here >
                  …
                JE          TEST1
```

The negative offset needed for the JE instruction exceeds 8-bits, forcing the assembler to choose the long form which permits a 32-bit offset. Since the distance between the JE and the target label is known, *MASM* could simply assemble in the proper hexadecimal offset. Unfortunately, it does not. Instead, it produces a "fixup" record in the object file, and leaves it to the linker to generate the negative offset.

But the *LINK* program thinks it is working with a 16-bit object file. It does the fixup using 16-bit arithmetic, and stores the negative offset in the *low-order* 16-bits of the instruction's offset field. The high-order 16-bits are unaffected (they remain zero). Thus a proper negative offset, such as 0FFFFFF5Ch (–164) appears in the object file as 0000FF5C. The 80386 CPU interprets this offset as +65372, a very different number! The usual result is a page fault.

This problem affects *all* near CALLs to previous addresses, and jumps to previous addresses more than 128 bytes distant. Extreme care must be exercised, because neither the assembler nor the linker issue warnings in these situations. There are two work-arounds: 1) rearrange your code so that all

CALLs are to forward addresses, and all backward jumps are within 128 bytes, or 2) use the jump and call macros provided in file **jumps.inc**. A better solution is to use a different assembler.

While the same fixup error occurs with forward CALLs and forward jumps more than 127 bytes distant, the error is benign because the upper 16-bits will be zero anyway for a positive offset.

(Interestingly, even though both the Borland and PharLap assemblers have 32-bit-aware linkers, each is smart enough to not bother with a fixup record at all, and simply assemble in the necessary 32-bit negative offset.)

Entry Register Conventions

The contents of the CPU registers when the external function is invoked are as follows:

CS	Execute/read-only code segment number for the function.
EIP	0 if function is a COM file. An EXP or DLL file will designate its entry EIP, which may be non-zero.
DS, ES	Read/write data segment number for the function.
FS, GS	SPITBOL's data segment, including the workspace containing the user's program and data.
SS:ESP	The hardware stack containing function arguments and return information. If the function is a COM or DLL file, the stack will always be in SPITBOL's data space (that is, SS equals FS and GS). If the function is an EXP file *and* the source program included a stack segment, then SS:ESP will point to it and SS equals DS and ES.

Because OS/2 DLLs are loaded into SPITBOL's memory space, registers DS, ES, FS, GS, and SS all contain the same value.

Input Arguments

When an external function is called, the calling arguments are pushed onto the stack in left to right order. Some additional information is pushed on top of the arguments, and then the return link. For efficiency, the stack pointer ESP is always maintained on a double-word (4-byte) boundary.

MS-DOS: 6-byte far pointers on the stack will have 2 bytes of padding:

```
dps      struc
         dpt        ?                           ; 48-bit pointer to
data
         dw         ?                           ; padding to 4-byte
boundary
dps      ends
```

OS/2: Near pointers on the stack are 4-bytes long:

```
dps      struc
         dpt        ?                           ; 32-bit pointer to
data
dps      ends
```

Note: Assemblers do not allow structure declarations to be nested, as shown here. The **extrn386.inc** file actual expands the definitions to make them legal. However, for clarity and brevity, we will continue to show them as nested in this appendix.

Arguments on the stack take different forms depending upon their datatype and whether the LOAD function prototype string specified type conversion. The table below shows how arguments appear on the stack for various datatype names in the prototype.

INTEGER	A 32-bit signed integer is pushed on the stack.
REAL	A 64-bit real number is pushed on the stack. The most significant 32-bits are pushed first, and thus appear at the *higher* memory address.
STRING	Two items are pushed on the stack: a 32-bit integer providing the string length, then a pointer to the first character of the string.
FILE	A pointer to a file control block (**blocks.inc**).
EXTERNAL, any other text, or the null string	A pointer to the internal SPITBOL block containing the data. The first word of the block provides the type of the argument. Users interested in working with internal block types should see consult the section “SPITBOL’s Internal Data Blocks” later in this appendix. File **logic.asm** provides an example of an external function that determines whether it is called with integer or string arguments.

External functions that do not declare any arguments in the prototype will receive one unconverted argument (the null string) when called.

Other Stack Information

After placing the arguments on the stack, SPITBOL pushes two other key pieces of information:

- a pointer to a block of miscellaneous information. The block contains the number of arguments, as well as pointers to other SPITBOL data structures. The block is described by structure misc in file **extrn386.inc**:

```
misc     struc
vers     dw          ? ; version number of this structure
env      db          ?   ; 0 = PharLap, 1 = Intel, 2 = OS/2
         db          ?   ; reserved
nargs    dd          ?   ; number of arguments to function
ptyptab  dpt         ?   ; pointer to table of data types
pxnblk   dpt         ?   ; pointer to xnblk describing function
pefblk   dpt         ?   ; pointer to efblk describing function
pflttab  dpt         ?   ; pointer to floating point table
spds     dw          ?   ; SPITBOL's data segment
spcs     dw          ?   ; SPITBOL's code segment
misc     ends
```

- a pointer to a *result area*, where the function will place the data to be returned as the function value. The use of the result area is explained later in this appendix in the section "Returning a Function Result."

Many functions can be written without reference to or knowledge of this structure. However, the information here permits additional flexibility and may be of use to you. Consult files **extrn386.inc** and **blocks.inc** for additional descriptions of the items in this structure.

Calling Example

Assume a function prototype for a function that takes three arguments and returns an integer result:

```
LOAD("MYFUN(INTEGER,REAL,STRING)INTEGER")          :F(NOLOAD)
```

Because the second (filename) argument was omitted, SPITBOL will search for the function in a file named **myfun.slf** (or **myfun.dll** for OS/2).

Once the function is loaded, we will call it with the following statement:

```
N = MYFUN("256", 1000, "A test string")
```

The prototype states that arguments are to be converted to specific types. In this case, the string "256" will be converted to integer 256, and 1000 will be converted to the real number 1000.0. At function entry, the stack will contain the subroutine return link, pointers to the result area and misc structures, and finally the three calling arguments.

MS-DOS: Because far pointers are in use, the stack would look like this:

```
          low memory
                       ;return Information:
02000010h              ;  return offset   <——   SS:[ESP] point here
0000000Ch              ;  return segment + padding
                       ;result area pointer:
02013DE4h              ;  offset of result area
00000014h              ;  segment of result area + padding
                       ;miscellaneous Information pointer:
020145A0h              ;  offset of miscellaneous info area
00000014h              ;  segment of miscellaneous info area + pad-
ding
                       ;arguments:
020231FCh              ;  offset of string (3rd arg)
00000014h              ;  segment of string (3rd arg) + padding
0000000Dh              ;  length of string, 13 characters (3rd arg)
00000000h              ;  1000.0, least significant half (2nd arg)
408F4000h              ;  1000.0, most significant half (2nd arg)
00000100h              ;  integer 256 (1st arg)
          high memory
```

OS/2: Using near pointers, the stack would look like this:

```
          low memory
02000010h              ;  return offset   <——   SS:[ESP] point here
02013DE4h              ;  offset of result area
020145A0h              ;  offset of miscellaneous info area
                       ;arguments:
020231FCh              ;  offset of string (3rd arg)
0000000Dh              ;  length of string, 13 characters (3rd arg)
00000000h              ;  1000.0, least significant half (2nd arg)
408F4000h              ;  1000.0, most significant half (2nd arg)
```

```
    00000100h                  ;   integer 256 (1st arg)
              high memory
```

Most functions will begin by pushing EBP and copying ESP to EBP. This is most conveniently done with the 80386 ENTER n,0 instruction, where n is the number of stack bytes to reserve for storage of temporary data. A function that was expecting these argument types could define a structure to access them as offsets from register EBP. The sample structures shown on the next page includes the saved value of EBP.

We use the conditional-assembly feature to define both the MS-DOS and OS/2 forms of the stack.

```
if   nearptr
stk             struc               ;  OS/2 form of stack
                                    ;items invariant for all functions:
  stk_ebp       dd       ?          ;  save EBP <— SS:[EBP] point
here
  stk_ret       dd       ?          ;  return link
  presult       dd       ?          ;  pointer to result area
  pmisc         dd       ?          ;  pointer to miscellaneous info
area
                                    ;items that must be customized
for
                                    ;arguments of each function:
  parg3         dd       ?          ;  string pointer, third arg
  larg3         dd       ?          ;  string length, third arg
  rarg2         dq       ?          ;  real number, second arg
  iarg1         dd       ?          ;  integer, first arg
stk             ends
else
stk             struc               ;  MS-DOS form of stack
                                    ;items invariant for all functions:
  stk_ebp       dd       ?          ;  save EBP <— SS:[EBP] point
here
  stk_ret       df       ?          ;  return link
                dw       ?          ;  pad word
  presult       df       ?          ;  pointer to result area
                dw       ?          ;  pad word
  pmisc         df       ?          ;  pointer to miscellaneous info
area
                dw       ?          ;  pad word
                                    ;items that must be customized
for
                                    ;arguments of each function:
  parg3         df       ?          ;  string pointer, third arg
                dw       ?          ;  pad word
  larg3         dd       ?          ;  string length, third arg
  rarg2         dq       ?          ;  real number, second arg
  iarg1         dd       ?          ;  integer, first arg
stk             ends
endif
```

With EBP set up this way, it's easy to access the various pieces of information. Here are some examples:

To load the first (integer) argument into register EAX:

```
                mov     eax, [ebp].iarg1                ; 32-bit in-
teger
```

The real number second argument can be loaded into registers EDX:EAX with:

```
                    mov     eax, dword ptr [ebp].rarg2      ; least
  signif.
                    mov     edx, dword ptr [ebp].rarg2+4  ; most
  signif.
```

If you wanted to scan the string provided in the third argument, you might load the appropriate registers with:

```
                    mov     ecx, [ebp].larg3                ; string
  length
                    sload   es, edi, [ebp].parg3    ; string address
```

Here sload is a macro that loads a 48- or 32-bit pointer as appropriate. Under MS-DOS, it generates

```
                    les     edi, [ebp].parg3
```

while under OS/2 it produces:

```
                    mov     edi, [ebp].parg3
```

The number of arguments can be loaded from the miscellaneous information area into register ECX with:

```
                    sload   es, ebx, [ebp].pmisc    ; point to misc
  area
                    mov     ecx, ses:[ebx].nargs    ; load number of
  args
```

Ses is an equate that generates an ES: override for MS-DOS far pointers, and nothing for OS/2 near pointers.

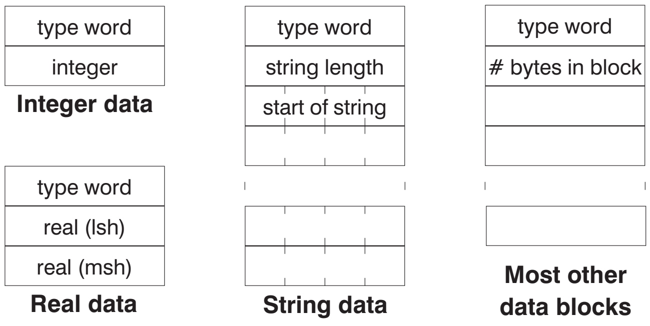
The stk structure will have to be customized for each different function to reflect the actual number and types of arguments used.

By restricting argument types to integers, reals and strings, and using those type names in the function prototype, your function need not be concerned with SPITBOL's internal data blocks. The data is presented on the stack in a language-independent fashion. Omitting the data type in the prototype will require additional work and knowledge to extract the datum from SPITBOL's heap.

SPITBOL's Internal Data Blocks

If your function fully declares data types in the LOAD function prototype and will only produce integer, real, or string results, you can skip this section. But if you need to work with unconverted input data, this section will provide an overview of how SPITBOL's stores data internally.

Unconverted input arguments are presented to the function as pointers to SPITBOL's internal data block. Each block is an aggregate of 32-bit words. The first word identifies the type of data in the block. Subsequent words depend upon the block type, as shown in the figure below.



Integer and real data follow the type word directly. String data is preceded by a word specifying the number of characters in the string. Most other block types are variable in length, and so have a word that specifies the total number of bytes used by the block. The layout of these blocks is described by structures in file **blocks.inc**.

The type word heading each block identifies the block type in a curious way. Rather than being a small integer type code, it is instead a full 32-bit value that points to program code within the SPITBOL system responsible for processing that type of data. Because there is a different section of program code for every data type, the data type pointers are unique. Furthermore, this means that your program must never rely upon the absolute value of this type word; it will change with each new release of SPITBOL.

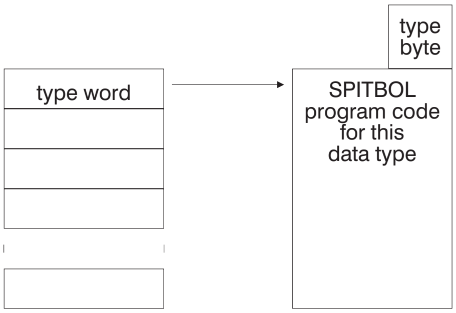
This roundabout system permits the indirect-threaded code organization used internally by SPITBOL. SPITBOL programs are converted to a list of pointers to data blocks and operators. The interpreter merely jumps indirectly through the first word of each object pointed to by the list.

While this organization results in improved execution speed and internal flexibility, it does cause one problem. Given a pointer to an arbitrary block in memory, how does an external function determine its block type? Simple, small integer type codes suitable for use in a compare operation or dispatch table would be more convenient.

Identifying block types

SPITBOL lets you identify block types in two ways. The program code pointed to by the type word is preceded in memory by a one byte, small integer type code. Suppose for example that registers ES:[EBX] point to a SPITBOL data block, and we want to obtain a one-byte type code. The miscellaneous information area contains the segment selector of SPITBOL's code segment. With this information, fetching the type code into AL is straightforward:

```
if nearptr
                                            ; OS/2 - near ptr
                mov     ecx, [ebx]          ; load type word
                mov     al, [ecx–1]         ; load type byte
else
                                            ; MS-DOS - far ptr
                lfs     ecx, [ebp].pmisc ; pointer to misc info
                mov     fs, fs:[ecx].spcs; get code segment
                mov     ecx, es:[ebx]       ; load type word
                mov     al, fs:[ecx–1]      ; load type byte
endif
```



The block-type data structure is shown schematically in the figure above. The code in AL could then be compared with known values. These values are defined symbolically in **blocks.inc** in the form BL_xx, where xx describes the type. For example, to test if the code in AL represents string data, use:

```
                cmp     al, BL_SC           ; test for string constant
```

There is another, and usually simpler way to perform this test. The miscellaneous information area contains a pointer to a table of type words which may be indexed by small integer type codes. Because this table resides within SPITBOL, it is updated with each new release. Each table entry is four bytes long, so the type code must be multiplied by four to access the table. Using the same example above, with a block pointer in ES:[EBX], we can test if the block contains a string as follows (independent of pointer size):

```
                mov    eax, ses:[ebx]          ; load type word
                sload  fs, ecx, [ebp].pmisc    ; ptr to misc info
                sload  fs, ecx, sfs:[ecx].ptyptab ; ptr to type ta-
ble
                cmp    eax, sfs:[ecx+BL_SC*4]; test for string
```

This situation arises so often that the file **extrn386.inc** contains equates for all the common block types, assuming the type table pointer is in SFS:[ECX]. The last line above could be replaced by:

```
                cmp    eax, sc                 ; test for string
```

Accessing the type table is also useful for functions that will return data blocks unconverted. In this case, it is necessary to set the result block's type word explicitly, and the type table gives you the word to store there.

Using internal blocks

When an internal block is passed to a function, the method just described permits a function to identify the block type. Or if your function is returning an unconverted block, you now know how to obtain the proper value to store into the type word. But what about the remaining words in the block?

The file **blocks.inc** contains structure definitions for various common block types. A full discussion of how SPITBOL operates internally is beyond the scope of this manual. String, integer, and real blocks are straightforward, but the curious user armed with **blocks.inc** and a debugger should be able to figure out most of the rest.

MS-DOS: In situations where a block contains a pointer to another block, remember that the target block resides within SPITBOL's heap, and must be referenced through a segment register loaded with SPITBOL's data segment. This segment value is available in the spds field of the miscellaneous information area.

SPITBOL distinguishes between relocatable and non-relocatable words within a block. A relocatable word is a pointer to the first word of another block in the heap. When the target block is moved during storage regeneration, SPITBOL will adjust the pointer automatically. For each block type, SPITBOL knows which words can contain relocatable pointers. For this reason, it is important that all words within a block be properly filled in when returning unconverted data. Note that all words within an external (XNBLK) block are treated as non-relocatable, so do not place pointers to other heap blocks there. Such pointers would become invalid after the first storage regeneration.

Argument data blocks containing standard SPITBOL data types should be treated as read-only by an external function. If you need to have a data block whose contents can be freely modified, create an external data block of a suitable size. SPITBOL will preserve its contents, but otherwise ignore it. External data blocks are described in the next section.

Returning a Function Result

A function returns by placing its result in the designated result area and executing a RET instruction. The result area is in SPITBOL's workspace; its address is provided on the stack in presult. Register EAX is set to indicate function success or failure, and in the case of success, the type of result being returned. Input registers do not need to be preserved.

The following table provides the result type indicators that may be returned in register EAX. Symbol definitions are obtained by including file **extrn386.inc**.

Register EAX	Result Type
FAIL	Function returns failing, no result.
BL_IC	32-bit signed integer.
BL_RC	64-bit real number.
BL_SC	String in the result area. The string length may be between 0 and 512.
BL_FS	String of up to &MAXLNGTH characters via a pointer.
BL_XN	External data block. A block of up to 512 bytes of user-defined data becomes a SPITBOL object.
BL_FN	External data block of up to &MAXLNGTH bytes of user-defined data via a pointer.
BL_NC	No conversion. The result block must be properly set up with SPITBOL's internal block information.

In the case of a failure signal, no additional information need be provided. All other return types must place their data (or a pointer to the data block) into the result area provided to the function in [EBP].presult. Each result type requires that a different data structure be applied to the result area. The structures are defined in file **blocks.inc**, and common ones are provided below for reference. The following information presumes that ES:EDI has been set to the result area by performing:

```
sload    es, edi, [ebp].presult
```

Integer Result, EAX = BL_IC

```
icblk          struc
  ictyp        dd      ?       ; type word (not used)
  icval        dd      ?       ; integer value
icblk          ends
```

The integer result is stored in SES:[EDI].icval.

Real Result, EAX = BL_RC

```
rcblk       struc
  rctyp     dd     ?     ; type word (not used)
  rcval     dq     ?     ; real value (lsh, msh)
rcblk       ends
```

The real result's least significant half is stored in dword ptr SES:[EDI].rcval. The most significant half is stored in dword ptr SES:[EDI].rcval+4.

String Result, EAX = BL_SC

```
scblk       struc
  sctyp     dd     ?     ; type word (not used)
  sclen     dd     ?     ; string length
  scstr     db     ?     ; start of string
scblk       ends
```

The length of the string being returned is stored in SES:[EDI].sclen, and must be between 0 and BUFLEN (defined in **extrn386.inc**, nominally 512) characters. Use a zero length to return a null string result.

The characters of the string are stored in the buffer beginning at SES:[EDI].scstr. If your function needs to return strings longer than BUFLEN, you will have to provide your own buffer within the function. You then return a pointer to the buffer using the method described below.

Pointer to String Result, EAX = BL_FS

```
fsblk       struc
  fstyp     dd     ?     ; type word (not used)
  fslen     dd     ?     ; string length
  fsptr     dpt    ?     ; pointer to string
fsblk       ends
```

The length of the string being returned is stored in SES:[EDI].fslen, and is limited to &MAXLNGTH. Use a zero length to return a null string result.

A pointer to the string is placed in SES:[EDI].fsptr. SPITBOL copies the string into its own workspace.

MS-DOS: The pointer is a far pointer, consisting of a 32-bit offset value followed by a 16-bit segment number.

OS/2: The pointer is a 32-bit near pointer.

Setting a pointer can be handled in a size-independent manner by using the sstore macro. Assume a pointer in SFS:[EBX]:

```
            sstore  ses:[edi].fsptr, ebx, fs   ; save ptr
```

Under MS-DOS, it generates

```
            mov     dword ptr [edi].fsptr, ebx
            mov     word ptr [edi].fsptr+4, fs
```

while under OS/2 it produces:

```
            mov     [edi].fsptr, ebx
```

External Block Result, EAX = BL_XN

```
xnblk      struc
  xntyp    dd     ?     ; type word (not used)
  xnlen    dd     ?     ; size of block in bytes
  xndta    dd     ?     ; start of user's data
xnblk      ends
```

Functions may return an external data block. SPITBOL copies the block into its workspace, and returns it as the value of the function. The data block may then be copied, assigned to variables, and passed to functions, etc., using normal SPITBOL statements. However, SPITBOL does not examine or access any of the user's data within the block. It is not treated as string data, and cannot participate in pattern matching. Interpretation of the data is left exclusively to the external function.

The result area provides room to store up to BUFLEN (defined in **extrn386.inc**, nominally 512) bytes of user data, starting at SES:[EDI].xndta. Your function must set SES:[EDI].xnlen to the number of bytes being returned. External data blocks larger than BUFLEN may be created using the method below.

If you subsequently pass this block back to your function as an argument, you will find that SPITBOL has increased the contents of the xnlen word by eight to account for the size of the xntyp and xnlen words themselves, and rounded the result up to be a multiple of four. Example files **testef5.asm** and **testcef5.c** on the distribution disk illustrate this phenomenon.

Pointer to External Block Result, EAX = BL_FX

```
xfblk      struc
  xftyp    dd     ?     ; type word (not used)
  xflen    dd     ?     ; data length in bytes
  xfptr    dpt    ?     ; pointer to user's data
xfblk      ends
```

The length of data being returned is stored in SES:[EDI].xflen, and is limited to the current value of keyword &MAXLNGTH.

A pointer to the data is placed in SES:[EDI].xfptr. SPITBOL copies the designated data into its own workspace. The length word is adjusted in the destination block as mentioned above.

MS-DOS: The pointer is a far pointer, consisting of a 32-bit offset value followed by a 16-bit segment number.

OS/2: The pointer is a 32-bit near pointer.

Unconverted Result, EAX = BL_NC

This method is normally used only by those with an intimate knowledge of SPITBOL's internal operation. The type word must be set to the correct SPITBOL value obtained from the table provided in the miscellaneous information area in ptyptab. For example, the internal type word corresponding to the vector type BL_VC can be loaded into register EAX with these statements:

```
                sload   fs, ecx, [ebp].pmisc     ; misc info area
                sload   fs, ecx, sfs:[ecx].ptyptab ; ptr to type ta-
ble
                mov     eax, sfs:[ecx+BL_VC*4]
```

Using the data type equates in **extrn386.inc**, the last instruction can be expressed more succinctly as:

```
                mov     eax, vc          ; get vector block type
word
```

All other words appropriate to the data type being returned must be correctly set. Block lengths are expressed in bytes, and must be a multiple of four.

Debugging Functions

Interactive debugging requires a 32-bit, protected mode debugger. You cannot debug external functions with a 16-bit MS-DOS debugger.

MS-DOS, PharLap-extended SPITBOL: SPITBOL-386 invokes the external function by a CALL instruction. This CALL instruction is located at location CS:200000Ch in the PharLap-extended SPITBOL system. Since this instruction is at a fixed address, the programmer may plant a breakpoint and interrupt SPITBOL just prior to the function call. You may then trace the function with your PharLap or MetaWare High C debugger. An example of setting up to debug is provided in the figure below.

```
C:>386DEBUG SPITBOL.EXE TEST.SPT
386|DEBUG: 2.2b — Copyright (C) 1986-89 Phar Lap Software, Inc.
[80386 protected mode]
–BP 200000C
–G
SPITBOL-386  Version 3.7(2.4 I/O) Serial 20001
(c) Copyright 1987-1990 Robert B. K. Dewar and Catspaw, Inc.
Breakpoint at 000C:0200000C Elapsed time = 0.1 seconds
EAX=00000000 EBX=0201F5C0 ECX=02014518 EDX=00000001
ESI=0201FE08 EDI=02013D58 EBP=0201458C ESP=0201EFB0
DS=004C SS=0014 ES=004C FS=0014 GS=0014
CS:EIP=000C:0200000C EFLAGS=00000206 NV UP EI PL NZ NA PE NC
000C:0200000C 2EFF5B10          CALLF   CS:[EBX+10]
```

Setting up to debug an external function

MS-DOS, Intel-extended SPITBOL: The Intel CodeBuilder debugger does not support the use of far pointers and segments registers that are used in external functions. You cannot debug external functions with Intel-extended SPITBOL.

OS/2: There are no fixed addresses that can be used to plant a breakpoint prior to calling the function. The only alternative is to assemble in a breakpoint instruction (INT 3) at the beginning of your function. Then load SPITBOL and your program into the OS/2 debugger and run SPITBOL. Execution will be interrupted when the INT 3 is encountered, after which debugging can proceed normally.

Real Numbers

Real numbers are 64-bit entities stored in standard IEEE format (compatible with the 80x87 coprocessor). They are normalized 64-bit signed magnitude values, containing a sign, binary exponent, and mantissa. A value is represented by the formula:

$$-1^{\text{sign}} * \text{mantissa} * 2^{\text{exponent}}$$

Bit 63	Sign bit for the mantissa. One for negative numbers.
Bits 62-52	Binary exponent biased by 3FFH. Thus the binary exponent +2 is stored as 401H.
Bits 51-0	The mantissa. In normalized form, the most significant mantissa bit is always one, and is not explicitly stored. It is an implied 1 to the left of bit 51. The binary point lies between it and bit 51.

The number –3.25 would be represented as:

$$-1 * 1.625 * 2 = -1^{1} * 1.101 * 2^{1}$$

Hexadecimal: 00000000h, C00A0000h

where:

Sign:	1
Mantissa:	[1]101 (1 + 0.5 + 0.125) (leftmost 1-bit not stored)
Exponent:	1 + BIAS = 400h

Zero is represented by all 64 bits being zero. The special values NAN (Not-A-Number) and ∞ (infinity) have an exponent field of all ones (7FFh). If *all* mantissa bits are zero, the value is ∞, with the sign bit specifying plus or minus ∞. If *any* mantissa bit is a 1, the value is NAN, and the sign bit is ignored.

Operations on real numbers are performed by pushing them onto the stack and calling one of the real arithmetic functions provided by SPITBOL. Reals are pushed left to right, and it is the caller's responsibility to remove arguments from the stack when the function returns.

The SPITBOL real number functions are available through an indirect call made via a dispatch table whose address is provided in pflttab in the block of miscellaneous information. Equated definitions in file **extrn386.inc** make these functions readily available provided the pointer to the table has been loaded into registers DS:[EBX]. Because this is a callback into SPITBOL, DS must be set to SPITBOL's data segment. The equated function names are:

fix	real to integer, i = fix(r)
float	integer to real, r = float(i)
r_add	real addition, r = r_add(r1,r2)
r_sub	real subtraction, r = r_sub(r1,r2)

r_mul	real multiplication, r = r_mul(r1,r2)
r_div	real division, r = r_div(r1,r2)
r_neg	real negation, r = r_neg(r)
r_atn	real arc tangent, r = r_atn(r)
r_chp	real chop, r = r_chp(r)
r_cos	real cosine, r = r_cos(r)
r_etx	real e to the x, r = r_etx(r)
r_lnf	real natural logarithm r = r_lnf(r)
r_sin	real sine, r = r_sin(r)
r_sqr	real square root, r = r_sqr(r)
r_tan	real tangent, r = r_tan(r)

This example divides the real number in location r1 by the number in r2, and stores the result in r3:

```
          spush  ds               ; save function's DS
          push   dword ptr r1+4 ; push first arg msh
          push   dword ptr r1   ; push first arg lsh
          push   dword ptr r2+4 ; push second arg msh
          push   dword ptr r2   ; push second arg lsh
          lds    ebx, [ebp].pmisc ; get address of misc
info
          sload  ds, ebx, [ebx].pflttab ; get adr of vectors
          call   r_div            ; perform division
          add    esp, 2*8         ; remove arguments
          spop   ds               ; restore function's DS
          fstore r3               ; save result
```

spush and spop are macros that generate no code under OS/2, where segment registers are not modified. fstore stores a floating point result in the argument address.

MS-DOS: Floating point results are returned in registers EDX:EAX.

OS/2: Floating point results are returned in ST(0), the top of the co-processor stack (real or emulated by OS/2).

SPITBOL's real number functions are easily accessible from assembly language using this technique. Because of segment addressing problems, external functions coded in other languages, such as C, may find it simpler to forego SPITBOL's functions and to simply use the real number support provided in the compiler's library.

Allocating Memory

MS-DOS, PharLap-extended SPITBOL: Each function is allocated its own memory segment, whose length is a multiple of the 80386's 4,096-byte page size. The size of this segment may be modified by invoking DOS interrupt 21h, function 4Ah. The function has been slightly altered from the DOS standard to accommodate the 80386's 4 gigabyte segments. Register EBX provides the requested segment size in four-kilobyte pages. Register ES should contain the function's data segment. For example, to set the function's data segment to 20 kilobytes, use the following:

```
push   ds
pop    es              ; function's segment to ES
mov    ebx, 5          ; five 4,096-byte pages
mov    ah, 4Ah
int    21h
jc     error
```

This technique modifies the size of the data segment only. Although the code segment is an alias for the same region of memory, it is not modified. At the time of writing, a bug in the PharLap DOS Extender prevents you from also modifying the size of the code segment. In practice, this is not a problem, because your existing code continues to execute in the existing code segment.

MS-DOS, Intel-extended SPITBOL: Additional memory cannot be allocated by an external function when running with this version of SPITBOL.

OS/2: SPITBOL allocates (but does not commit) 256 megabytes of address space for its own use. External functions are free to use OS/2's DosAllocMem and DosAllocSharedMem to allocate memory from the 512 megabyte address space provided by OS/2.

Impact on Save Files

MS-DOS: External functions are captured in save files and load modules created with the EXIT() function. To reduce the size of these files, SPITBOL only includes the memory actually used by the function, not the entire segment. Since external functions are usually quite small, recording 4,096 bytes for each would increase the size of files unnecessarily.

The actual number of bytes being used by the function (as opposed to the full segment size) is recorded in location xnsiz of an xnblk. A pointer to this block is provided in pxnblk of the miscellaneous information area. After changing the size of a function's data segment, you should update the function size in the xnblk. Assume the new size is in register EAX. Use:

```
sload  es, edi, [ebp].pmisc
sload  es, edi, ses:[edi].pxnblk
mov    ses:[edi].xnsiz, eax     ; size in bytes
```

OS/2: External functions are not saved in save files and load modules, so there is no need to inform SPITBOL of a function's private memory allocations.

Sample External Functions

The SPITBOL distribution disk contains a number of sample programs that illustrate the techniques for accessing arguments, as well as returning various types of results. For each test case, there are several files: the assembly language source, the binary version of the function ready for loading into SPITBOL, and a short SPITBOL test program. For example, for test case one, the three files are **testef1.asm**, **testef1.slf**, and **testef1.spt**.

The first six test cases illustrate accessing integer, real, and string arguments. If all are found to be in order, they return results as follows:

testef1.asm	return a string
testef2.asm	return an integer
testef3.asm	return a real number
testef4.asm	return a string via a pointer
testef5.asm	return an external data block
testef6.asm	return external data via a pointer

The last test case, **testef7.asm**, illustrates how a function may change the size of its memory segment, and runs under PharLap-extended SPITBOL only.

In addition to these test cases, the file **logic.asm** provides an example of a complex external function. It provides logical and mathematical operations upon strings and integers, as well as base conversion. The file is worth studying as an example of a complete function. The SPITBOL include file **logic.inc** provides SPITBOL programs with simplified access to the external function.

C-Language External Functions

Writing external functions in C is in many ways simpler than writing them in assembly-language. The register bookkeeping chores of juggling multiple pointers is handled automatically by C, and C's strong typing assures that pointers are associated with the proper data structures. The low-level flexibility of the C language can now be harnessed to SPITBOL.

The distribution disk contains C-language analogs of the files used by assembly-language programmers. This section will not duplicate all of the assembly-language material previously introduced. Rather, it will highlight how these assembly-language operations can be performed in C.

MS-DOS: Only the PharLap-extended version of SPITBOL supports loading the EXP files generated by the MS-DOS C compilers and linkers. The Intel-extended version of SPITBOL can only load assembly-language files.

OS/2: Because OS/2 embraces run-time dynamic loading of library functions (DLLs), interfacing arbitrarily complex C-language functions is simple and straightforward. In fact, functions can be written in *any* OS/2 32-bit development language capable of producing DLLs.

Header files

The header files **system.h**, **extrn386.h** and **blocks.h** are the C equivalents of files **system.inc**, **extrn386.inc** and **blocks.inc** used by assembly-language programmers. **Extrn386.h** provides basic definitions while **blocks.h** provides definitions of SPITBOL's data blocks. **System.h** defines the os2 manifest constant used to select between MS-DOS and OS/2 implementations. These files are in directory **external\c**.

Loading an external function

Because SPITBOL does not care how a particular external function was created, the LOAD prototype, calling sequence, and stack layout are identical for both C and assembly-language external functions. The only difference is that instead of being COM files, C-language functions will be EXP files created by a 32-bit linker (MS-DOS), or Dynamic Link Libraries (OS/2). The SLF or DLL filename extensions are still used.

Creating C functions

MS-DOS: We use the Meta Ware High C 386 compiler to create external functions. Users of other 32-bit C compilers (or other 32-bit languages, such as Pascal), may have to adjust some parts of this sequence. Only one function per linked file is allowed. The entry point is identified by a C function named mainslf(). SPITBOL calls mainslf() when the external function is invoked.

MS-DOS: After compiling all C source files, they are linked with one of four small interface modules provided. The choice depends on whether your external function uses C-library functions, real number arithmetic, or C memory allocation. The files are:

inittiny.obj	The external function uses SPITBOL's stack. Because many C library functions assume the stack and data segments are the same, **inittiny.obj** should be used with extreme care.
initstk.obj	Provides the external function with its own 2K byte stack. This satisfies the C library requirement that the stack and data segments be one and the same.
initheap.obj	Similar to **initstk.obj**, but also initializes a heap for dynamic memory allocation.
initfull.obj	Similar to **initheap.obj**, but also initializes things so that a function may use real numbers and the complete C I/O library.

MS-DOS: Initstk.obj, **initheap.obj** and **initfull.obj** are compatible with the SMALL memory model assumed by the C library and compiler. By using SPITBOL's stack, **inittiny.obj** is simulating the COMPACT memory model. While it is certainly possible to instruct your compiler to generate code for the COMPACT model, High-C's library assumes the SMALL model, and is incompatible with **inittiny.obj**.

MS-DOS: Initstk.obj provides a 2K byte stack and nothing more. **Initheap.obj** provides a stack as well as a dynamic heap and C memory allocation via C functions malloc(), free(), brk() and sbrk(). **Initfull.obj** includes the stack and memory allocation of **initheap.obj**, but also initializes C's I/O system and provides variables for C's real number library. With **initfull.obj**, your function can use such C library functions as fopen() and fprintf().

MS-DOS: All versions of Init receive initial control from SPITBOL when your function is called. After performing any initialization, they jump to your main function, mainslf(). Select the version of Init appropriate for the complexity of your C function.

OS/2: We use the Microsoft CL386 compiler provided with the OS/2 SDK to create external functions. Any other OS/2 tools capable of producing DLLs may be used. More than one external function may appear in a DLL. The entry point of each is just the SPITBOL name of the function, and is communicated to SPITBOL by being EXPORTed in the corresponding module definition file.

Extrnlib.obj

Your function should be linked to the file **extrnlib.obj** provided with SPITBOL. It contains functions useful for returning results. For MS-DOS, it also contains functions for performing the basic string operations on the far strings typically encountered in the argument list. Here's a brief list of the functions present in **extrnlib.obj**; consult the file **extrnlib.c** for additional details:

For returning results:

retint	return integer

retnstrf	return n-char string via a pointer
retnstrn	return n-char near string
retnxdtf	return n-char external data via a pointer
retnxdtn	return n-char near external data
retreal	return real number
retstrf	return C string via a far pointer
retstrn	return C string

Other utility functions (MS-DOS only):

memcmpff	compare n-byte far areas of memory
memcpyff	copy n-byte far areas of memory
memcpyff	copy n-bytes from near area to far
strcpyff	copy far C string to far string
strcpyfn	copy near C string to far string
strcpynf	copy far C string to near string
strlenf	length of far C string
strncmpfn	compare n-char far and near strings
strncpyff	copy n-char far C string to far string
strncpyfn	copy n-char near C string to far string
strncpynf	copy n-char far C string to near string

MS-DOS: Consult your C-language reference manual for additional information on near and far objects.

OS/2: Under OS/2, where all pointers are near, the utility functions above are defined to their C library analogs: memcpy, strcpy, strlen, strncmp, and strncpy respectively.

MS-DOS: Below we show a compilation and link operation for a modest function that will not use floating point, C I/O, or memory allocation. The –maxdata 0 option is required to prevent the EXP file loader from allocating all remaining free memory to the function:

```
hc386  /c  myfun.c
386link  myfun.obj  initstk.obj  extrnlib.obj  –pack  –maxdata  0
                    –e  myfun.slf
```

A more complex function will require linking in the C libraries:

```
386link  myfun.obj  initfull.obj  extrnlib.obj  –pack  –maxdata  0
                    –lib  \bin\hce  –e  myfun.slf
```

OS/2: Here is the compilation and link under OS/2. The definition file is similar to the definition file on page 308:

```
cl386  /c  /Gs  myfun.c
link386  /nod  /noe  /noi  myfun.obj  extrn386.obj,  myfun.dll,  ,
                    libcdll  os2386,  myfun.def
```

Function prototype

The form of your C function prototype depends upon the arguments used in the corresponding SPITBOL prototype. Consider an external function loaded with this statement:

```
LOAD("MYFUN(INTEGER,REAL,STRING,EXTERNAL)")
```

Your function should be declared as a procedure returning a word (long) result. We use a Pascal calling sequence to reflect the left-to-right order that SPITBOL pushes arguments onto the stack. Under MS-DOS, the function name is mainslf; under OS/2 it is the SPITBOL name of the function. This can all be done in a platform-independent fashion by using the entry() macro provided in **extrn386.h**.

After the function arguments, SPITBOL pushes pointers to the miscellaneous information and result areas. Using the typedefs in **extrn386.h**, the function's entry would look like this:

```
#include "system.h"
#include "extrn386.h"

entry(MYFUN)(iarg1, rarg2, larg3, parg3, parg4, pinfo, presult)
word             iarg1;        /* arg1 integer              */
double           rarg2;        /* arg2 real number          */
word             larg3;        /* arg3 length               */
far char         *parg3;       /* pointer to arg3 string    */
far union block  *parg4;       /* pointer to arg4 block     */
far misc         *pinfo;       /* pointer to misc info      */
far union block  *presult;     /* pointer to result area    */
{  ...
```

The far keyword is defined to the null string under OS/2 to produce the 32-bit near pointers that are the norm under OS/2. Note that the pointers to the result area, and to the unconverted fourth argument are presented as pointers to a union of all possible SPITBOL block types. The union is defined at the end of file **blocks.h**. The word typedef corresponds to a 4-byte long integer.

Accessing arguments

The previous assembly-language sections provide detailed information on the stack forms taken by various arguments. Rather than repeat the material here, the interested reader should consult the C-language sample files **testcef*n*.c**. Just remember that under MS-DOS, the strings and internal data blocks reside in SPITBOL's data segment (*not* the function's data segment), and must be accessed via a far pointer consisting of a 32-bit offset and a 16-bit segment selector.

Provided you used the dummy argument name pinfo, the datatype of a SPITBOL block can be checked using simple type names like ic (integer), or sc (string). Definitions for these types are provided in **extrn386.h**.

Returning a result

As with assembly-language functions, results are stored in the result area pointed to by the presult argument. It's simplest to use the ret*xxx*() functions in **extrnlib.c** listed earlier. They store the correct data in the result area, and provide the correct value to be returned to SPITBOL. For example, to return integer 1000 as the result of the function, use:

```
return retint(1000, presult);
```

Other return functions allow returning real numbers, strings, and external data. Consult the documentation in **extrnlib.c** and the examples in **testcef*n*.c**.

Real arithmetic

MS-DOS: In order to use SPITBOL's real arithmetic functions, register DS must point to SPITBOL's data segment. Because this setup is difficult to achieve in C, we advise against attempting to use SPITBOL's real functions. Simply link with **initfull.obj**, and use your compiler's own real number library. Although this results in duplicated code, it makes things much easier for all concerned.

OS/2: Program code uses hard-coded floating-point instructions. These are executed by the numeric co-processor or emulated by OS/2 if the co-processor is absent. More complex operations such as trigonometric functions are obtained from the C library.

C library I/O

MS-DOS: The module **initfull.obj** initializes the High C I/O system the first time your function is called, either during initial execution, or after being reloaded as a Save file. It permits you to use all normal C I/O functions, such as fprintf() and fscanf(). Opening files other than standard input, output and error is your responsibility. All files (including the standard input, output, and error files) are closed after a function is reloaded from a Save file. Opening files when a Save file is reloaded must be done explicitly by your function.

OS/2: Normal C I/O functions such as fprintf() and fscanf() can be used freely. All necessary initialization is performed automatically by OS/2 when the DLL is loaded.

Files remain open between function calls. SPITBOL closes files opened via C library functions in all loaded functions when SPITBOL terminates. It also closes files opened by a particular external function if that function is unloaded.

Appendix G
Configuring SPITBOL

There are no special configuration actions required for Unix or OS/2 versions of SPITBOL. The material in this appendix is concerned solely with configuring MS-DOS SPITBOL-386. Information on using SPITBOL-386 with Microsoft Windows is provided at the end of this appendix.

Configuring MS-DOS SPITBOL-386

Introduction

SPITBOL-386 incorporates a "DOS-Extender." Your 80386/486 computer normally executes in real mode as a very fast 80286 computer. Real mode and virtual 8086 mode are the only modes in which MS-DOS can execute, and DOS remains ignorant of the true 32-bit nature of your system.

The DOS-Extender switches your computer into 32-bit protected mode. In this mode programs like SPITBOL have access to 4 gigabyte memory segments, hardware memory paging, and 32-bit CPU registers. When SPITBOL needs to call MS-DOS, such as for file I/O, the DOS-Extender switches the computer back to real mode and moves the data between the protected region and the conventional memory that DOS is aware of. In this way, MS-DOS thinks it is running on an 80286 system while SPITBOL can run in 32-bit protected mode.

The fact that a program is running in 32-bit protected mode is usually invisible to the user. However, sufficient differences exist among 80386 and 80486 systems that some individual reconfiguration may be necessary.

Versions

There are two versions of SPITBOL on the release disk: PharLap-extended SPITBOL (**spitbol.exe**) and Intel-extended SPITBOL (**spitboli.exe**). Each is described in a separate section of this appendix. Chapter 1, "Installation," describes the differences between the two versions.

Intel-Extended SPITBOL-386

Intel-extended SPITBOL runs in extended memory only; it cannot use expanded memory (EMS). It uses only a small amount of conventional memory below 1 megabyte, so most of that remains available for running other MS-DOS programs from within SPITBOL via the HOST(1) function.

Much of the following material distinguishes between two different environments:

1. Operation under a DPMI host such as a DOS shell in Windows 3.0 or 3.1 Enhanced mode or OS/2 2.0. The DOS Extender lets the DPMI host provide memory management.
2. Operation under native MS-DOS or with a VCPI host, such as QEMM-386. No DPMI host is present, and the Intel DOS Extender provides memory management services to SPITBOL.

Virtual memory

Virtual memory is used if SPITBOL's workspace must be expanded to accommodate your program and data, and insufficient physical RAM remains. The virtual memory is obtained by swapping portions of your program and data from RAM to the hard disk. If running under a DPMI host, the host provides the virtual memory management. Absent a DPMI host, the Intel DOS Extender manages virtual memory itself. In either case, virtual memory is maintained by use of a hard disk *swap file*.

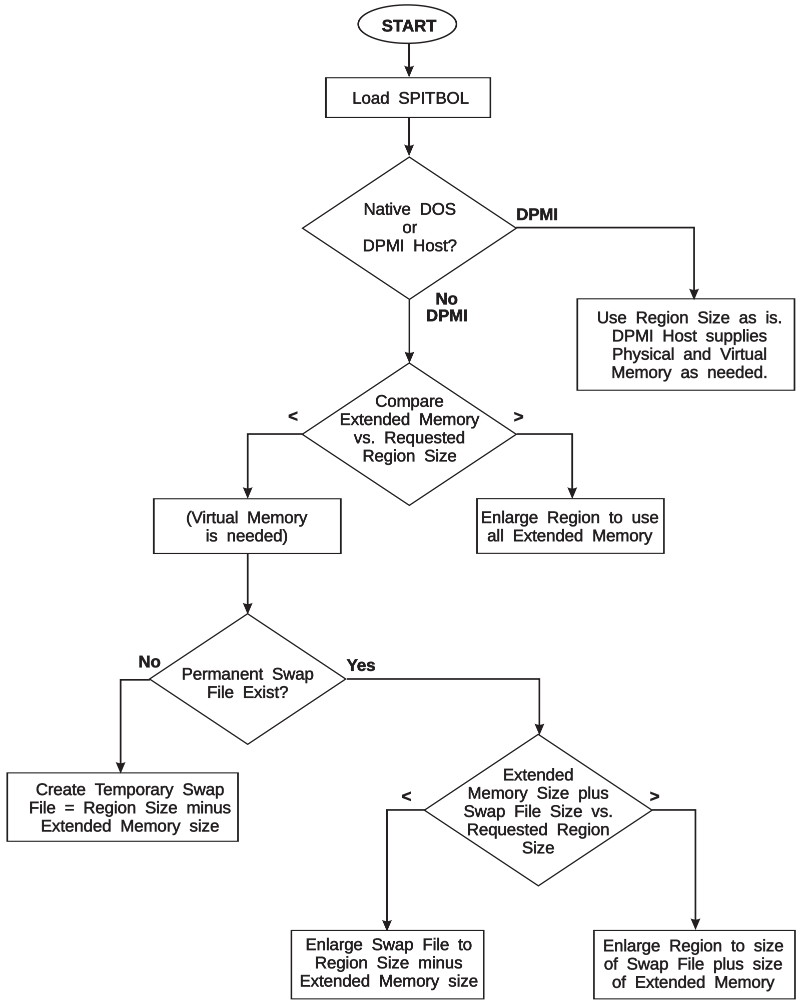
Region size

Intel-extended SPITBOL is configured with a particular *region size*. The region size determines the maximum amount of memory (physical and virtual) that can be used. By default, Intel-extended SPITBOL sets the region size to 1 megabyte. The interpretation of region size is slightly different depending on whether you are running with a DPMI host or not, and is described below.

The **modxconf.exe** program can be used to specify region and swap file sizes. The program is provided by Intel Corp. for use with their DOS Extender. Start it at the DOS command line like this:

```
>modxconf spitboli.exe
```

To understand how to set the options, we will discuss how the region size behaves with and without DPMI hosts. This behavior is shown schematically in the figure on the next page, and described below.


START
Load SPITBOL
Native DOS or DPMI Host?
DPMI
Use Region Size as is. DPMI Host supplies Physical and Virtual Memory as needed.
No DPMI
Compare Extended Memory vs. Requested Region Size
<
>
(Virtual Memory is needed)
Enlarge Region to use all Extended Memory
Permanent Swap File Exist?
No
Yes
Create Temporary Swap File = Region Size minus Extended Memory size
Extended Memory Size plus Swap File Size vs. Requested Region Size
<
>
Enlarge Swap File to Region Size minus Extended Memory size
Enlarge Region to size of Swap File plus size of Extended Memory

Region Size Adjustment for Intel-extended SPITBOL

DPMI host

When running with a DPMI Host (Windows, OS/2 2.0, etc.), virtual memory is maintained by use of a hard disk "swap file." For speed of access, the swap file must occupy contiguous blocks on the disk. Chapter 13 "Optimizing Windows" of MicroSoft's *Windows User's Guide* provides detailed information on setting up a swap file. Briefly, there are two types of swap files: *permanent* and *temporary*.

A permanent swap file remains on your hard disk (occupying space) even when Windows is not running. Its size and location are controlled by the SwapFile program provided with real-mode Windows. A temporary swap file is one that is created each time Windows is started, and deleted when Windows terminates execution. Besides the delay added to each Windows startup, the size of the temporary swap file will be limited by the current organization of the hard disk, in particular, the largest contiguous free area on the disk.

If you work with Windows a lot, and can spare the disk space, you will find a permanent swap file to be the most convenient. Consult the *Windows User's Guide* for additional information on setting up this file.

If you attempt to start SPITBOL and receive the message "Insufficient extended memory during program load" it means that extended memory plus the DPMI-host's swap file could not accommodate SPITBOL's currently configured region size. Either make more space available on the swap device and increase the size of the Windows swap file, reduce SPITBOL's region size using the MODXCONF program, or remove extended memory utilities such as VDISK.

Native DOS

When running without a DPMI host, the Intel DOS Extender will manage its own swap file. By default, the swap file is **xmswap.tmp** in the root directory of drive C, but the name and location may be altered by setting the DOS environment variable SWAP:

```
>SET SWAP=D:\TEMP\MYXMSWP.TMP
```

A permanent, pre-allocated swap file may be created by using the MODXCONF program. Because the region size used by an executing program is comprised of both physical RAM and virtual memory from the swap file, the swap file does not need to be the full size of the region. It only needs to be large enough to accommodate the additional memory desired beyond the physical extended memory of the machine.

When SPITBOL is loaded without a DPMI host, the region size will be automatically adjusted according to the following algorithm:

1. If the available extended memory is larger than the desired region size, the region size is temporarily enlarged to match the physical memory, and execution commences. Your SPITBOL program will then use *all* available extended memory.
2. If the available extended memory is smaller than the requested region size, a swap file and virtual memory will be needed. The DOS Extender's first step is to check for the existence of a swap file via the SWAP environment variable:
    a. If there is no swap file, the DOS Extender will attempt to allocate one. The file must be created large enough so that combined with the physical extended memory on the machine, the region size is available to SPITBOL. If the disk does not have enough space, the message "Insufficient extended memory" is displayed. Otherwise, the file is created and execution begins.
    b. If a permanent swap file already exists, the DOS Extender examines its size:

1. If the swap file plus extended memory is greater than the region size, the region size is enlarged to match.

2. If the swap file plus extended memory is smaller than the region size, additional disk space will be allocated if possible. If space is not available, the "Insufficient extended memory" message is displayed.

Unlike Windows, the MODXCONF utility does not require that the swap file be contiguous. However, a fragmented swap file will degrade performance, and you should consider using a disk de-fragmenting program to rearrange and consolidate disk free space.

modxconf.exe

Intel's MODXCONF program allows you to do two things:

1. Adjust SPITBOL's region size.
2. Create a permanent swap file to be used when running without a DPMI host (native DOS).

Usage is straightforward. The program takes one command line argument, the name of the SPITBOL executable file. Invoke it like this:

```
>MODXCONF SPITBOL.EXE
```

and follow the directions. The "M" option allows you to modify SPITBOL's region size, the "C" option allows to create or modify a permanent non-DPMI swap file.

Note 1: Do *not* attempt to use a RAM disk as a swap file.

Note 2: Avoid swapping and virtual memory as much as possible. It greatly slows down SPITBOL. We suggest setting the region size to a value several hundred K bytes smaller than the *available* extended memory. This should eliminate any use of a swap file when running without a DPMI host. With a DPMI host such as Windows, even smaller values will be required to avoid swapping, because Windows will be using some of your extended memory for its own purposes.

Load modules

Because Intel-extended SPITBOL can generate executable files (via command-line option –w or the EXIT(3) function) for distribution to others, the considerations above will apply to these executables as well. The MODXCONF program can be used on executable files produced by SPITBOL.

Other notes

1. If you receive the message "DOS Stack Overflow," note the following: Certain combinations of device drivers and network cards can cause a cascading of interrupts, using a lot of space on the DOS stack. Coupled with the DOS extender's 32-bit pushes onto the same stack, your system can halt with the following message: "Internal Stack Overflow, System Halted." If you receive this message, insert the following line in your **config.sys** file and reboot (see your DOS manual for more information on the STACKS command): STACKS=9,384.

2. If you use 386MAX V6.0 or QEMM-386, do not use the EMS=0 (386MAX) or NOEMS (QEMM-386) option; this option disables services required by Intel-extended SPITBOL. Otherwise, executing your application can cause unpredictable results. Note that the NOEMS option *is* permitted with EMM386. Also, because extended memory is required to run, do not use QEMM's NOXMS option.

3. If you use MODXCONF to specify a region size larger than extended memory, and do not have a permanent swap file, you will experience a delay when SPITBOLI.EXE starts up and ends. The DOS Extender is creating and then deleting a temporary swap file whose size is:

```
(region size) - (extended memory size)
```

PharLap-Extended SPITBOL-386

Even if PharLap-extended SPITBOL-386 runs without incident on your system, you may still need to adjust the way it uses system memory. For example, the HOST(1) function runs another MS-DOS program from within a SPITBOL program. This requires that enough conventional memory below 640K be available to execute the other program. Normally, PharLap-extended SPITBOL reserves at least 64K bytes for this purpose, but that may be adjusted up or down by the configuration utility.

Altering SPITBOL-386's Configuration

The options described below are made effective by directly modifying the **spitbol.exe** file. They *cannot* be specified at the time a program is run. Therefore, make sure you have a backup copy of **spitbol.exe,** and only modify the copy.

cfig386.exe

CFIG386 is a program produced by PharLap Software, Inc., and distributed with SPITBOL-386 by Catspaw, Inc. under license from PharLap. It is sub-licensed to you as part of the SPITBOL-386 software package; the same License and Warranty terms apply.

Configuration changes are made by running CFIG386 with file **spitbol.exe** as its argument, and specifying the options to be added to the file. CFIG386 can also remove *all* options from **spitbol.exe**, allowing you to start over without resorting to the backup copy.

The program is run by using a DOS command line like this:

```
cfig386 spitbol.exe [configuration switches]
```

If no switches are present, CFIG386 reports the switches currently set in **spitbol.exe.** For example:

```
cfig386 spitbol.exe
```

when used with SPITBOL-386 as distributed produces:

```
Configured switch values:
                        –minreal 4000
                        –maxreal 7000
```

Clearing All Switches

All switches installed in **spitbol.exe** can be removed by using the special switch "–CLEAR". For example:

```
cfig386 spitbol.exe –clear
```

Additional switches can follow "–CLEAR" and will be installed in the executable file.

Switches MINREAL, MAXREAL

Most switches are highly technical in nature, and are provided for non-conforming 80386 systems or early versions of the 80386 chip. The only switches you will probably need to know about are –MINREAL and –MAXREAL, described here.

Memory on 80386 systems is divided into *conventional* memory and *extended* memory. Conventional memory is that memory below 640K, and is the area in which normal MS-DOS programs execute. Extended memory is the region above 1 megabyte, extending as high as 4 gigabytes. SPITBOL's DOS-Extender can combine all available conventional and extended memory into one large, virtual region in which SPITBOL-386 will run.

MS-DOS commands can be executed from within SPITBOL, provided enough conventional memory has been reserved for that purpose. This reserved memory is not available for SPITBOL program or data storage. Therefore, if your SPITBOL program does not execute other programs, far less conventional memory need be reserved, and you may wish to alter the configuration so that SPITBOL can use the memory.

The amount of conventional memory reserved is controlled by switches –MINREAL and –MAXREAL. They take an argument giving the number of 16-byte paragraphs to be reserved. The number is assumed to be decimal, but may be treated as hexidecimal by appending the letter "h".

–MINREAL #
–MAXREAL #

These switches instruct the DOS-Extender to *attempt* to reserve MAXREAL 16-byte paragraphs, but to *always* reserve at least MINREAL 16-byte paragraphs of conventional memory. For example,

```
cfig386 spitbol.exe –clear –minreal 8000 –maxreal 16000
```

removes existing settings, and reserves between 8000 * 16 (128,000) and 256,000 bytes of conventional memory for DOS programs. Notice that a space appears between each switch name and the number.

# can vary from 0 to 65535. If MINREAL amount of memory is not available when SPITBOL starts up, a message is provided and the program will not run. Note that **spitbol.exe** is shipped to reserve between 64,000 and 112,000 bytes of conventional memory. You may wish to set these switches to a much smaller value if your SPITBOL programs does not execute MS-DOS commands. We recommend a minimum value of 320 bytes (–MINREAL 20 –MAXREAL 20). SPITBOL will use these 20 paragraphs to build

a file handle table that allows your program to simultaneously open as many files as are allowed in your system's **config.sys** file (less 4 for standard system files).

Other Switches

The remaining switches should only be needed in extraordinary circumstances.

–MINIBUF #
–MAXIBUF #

These switches control the size of the buffer used to communicate file data between SPITBOL-386 and MS-DOS. The # is between 1 and 64, and is the buffer size in kilobytes. By default, MINIBUF is 1 and MAXIBUF is 64. The system attempts to allocate a buffer of MAXIBUF kilobytes, but will run provided MINIBUF kilobytes can be allocated. Since SPITBOL's I/O system allocates 1K buffers, the default settings are more than adequate. However, if you are short of conventional system memory, you may wish to reduce MAXIBUF to a much smaller value, such as 4K bytes. This does *not* affect or restrict the record length used in I/O, merely the internal efficiency of moving data through memory. To reduce MAXIBUF to 4K, use:

```
cfig386 spitbol.exe –maxibuf 4
```

–EXTLOW #
–EXTHIGH #

SPITBOL-386 will use extended memory at or above the EXTLOW address, and below the EXTHIGH address. By default, EXTLOW is 100000h (one megabyte) and EXTHIGH is FFFFFFFFh (four gigabytes). SPITBOL-386 should detect other programs such as RAM disks that are using extended memory, and avoid interference. However, if you have a non-conforming program using extended memory, it will be necessary to explicitly tell SPITBOL-386 what area of extended memory is safe to use.

–NOBIM

Compaq 80386 systems provide "built-in" memory that SPITBOL-386 will use automatically if not in use by another program. This switch forces SPITBOL-386 to ignore that memory.

–VDISK

Use this switch if you have a RAM disk program that does not follow the VDISK standard for using extended memory.

–B0

Use this switch only if your system has an 80386 chip that is Intel step B0. You may not be able to use the 80287 or 80387 with such an old chip (see NO87 environment variable at the end of Chapter 13, "Running SPITBOL").

–XT

You may need to use this switch if you are running on a PC or XT system with a 386 board installed (e.g., Intel Inboard/PC). SPITBOL-386 should detect this configuration automatically, but may fail if the board has a non-standard BIOS.

–AT

Use this switch if SPITBOL-386 cannot detect that your system has an AT bus architecture.

–MCA

Use this switch if SPITBOL-386 cannot detect that your system has a Micro Channel bus architecture.

–MAXVCPI #

SPITBOL-386 is compatible with expanded memory emulators such as Quarterdeck's QEMM-386 that provide a Virtual Control Program Interface. Normally, SPITBOL-386 can use all of extended memory, leaving none left for the execution of other programs. This switch limits the amount of extended memory that SPITBOL-386 will use to # bytes.

Use with Microsoft Windows

PharLap-extended SPITBOL is compatible with Windows under the following circumstances:

1. Windows is started in standard mode.
2. the Microsoft driver **himem.sys** is installed in your **config.sys** file.

Intel-extended SPITBOL is compatible with Windows if:

1. Windows is started in 386 Enhanced mode.

Installing PIF

Copy files **spitbol.pif** and **spitbol.ico** from the distribution disk to your Windows directory.

If you are running Windows in standard mode, make sure Microsoft's **himem.sys** driver is installed in your **config.sys** file. Then startup Windows:

```
>WIN  /S          (Standard mode)
>WIN  /3          (386 Enhanced mode)
```

Execute Windows' PIF Editor program. Open the **spitbol.pif** file in the Windows directory. Change the Program File Name line to include the disk drive and full path of the directory where you copied the correct version (PharLap or Intel) of file **spitbol.exe**.

The other options may be changed, but should be correct as is. Note that it is important to specify 64 for the XMS Memory Required entry. This forces Windows to permit SPITBOL to access extended memory. The –1 for XMS Memory Limit allows SPITBOL to use as much memory as it needs, up to the system maximum. You may wish to place a specific number here, such as 2000 KB if other programs will need to access extended memory simultaneously. Normally this is not necessary, because SPITBOL is not greedy, and only acquires additional memory when absolutely necessary.

Save the **spitbol.pif** file, and close the PIF Editor.

Group

Select the Program Manager group in which you wish to install SPITBOL, and select New from the File menu, and Program Item from the dialog box. For Description, enter SPITBOL, and for the Command Line enter **spitbol.pif**. Press the Change Icon button, and on the File Name line, enter the complete pathname to the **spitbol.ico** file that you previously copied to your hard disk, for example:

Associating source files

Start Windows' File Manager Program, and navigate the directory structure to find the **test.spt** program previously copied from the hard disk. (Actually, any file with the "spt" extension will do.) Select the file so it appears in reverse video. From the File Menu select Associate… and fill out the form to say that ".SPT" files are associated with **spitbol.pif**.

From now on, when you double-click any .SPT file, SPITBOL will automatically begin execution with that file as input.

Other Notes

If you will be using Windows' DOS Prompt icon to start a DOS session, and then manually starting **spitbol.exe** at the DOS prompt, observe the following. It will be necessary to create or modify the PIF file for the DOS Prompt (**command.pif**) to have the correct entries for the XMS memory lines:

```
XMS Memory Required: 64
XMS Memory Limit: –1
```

If you do not make these changes, the DOS Session will not let SPITBOL have access to extended memory, and an "out of memory" error message from SPITBOL will result.

If you are using Windows enhanced mode, Windows can supply virtual memory to the Intel DOS Extender. Additional information on configuring both Windows' and SPITBOL's memory usage will be found in an earlier section of this Appendix.

Bibliography

Books in print

1. Butler, Christopher., *Computers in Linguistics*. Oxford: Basil Blackwell Ltd., 1985. ISBN 0-631-14267-3, paper, 270 pages.

   Two-thirds of this book is a SNOBOL4 tutorial, aimed at non-programmers, so that the reader "should attain a level of competence which will allow him to write programs for his own purposes."

2. Gimpel, James F., *Algorithms in SNOBOL4*. Salida, CO: Catspaw, Inc., 1986. Originally published 1976 by Wiley Interscience; republished by Catspaw, Inc. ISBN 0-939793-00-8, paper; ISBN 0-939793-01-6, hardcover, 506 pages.

   A cookbook of useful SNOBOL4 algorithms and techniques, highly recommended for all serious SNOBOL4 programmers. Besides illustrating subtle uses of the language, it provides an extensive collection of general programming algorithms.

3. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky, *The SNOBOL4 Programming Language* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall, Inc., 1971. ISBN 0-13-815373-6, paper, 256 pages.

   The complete, official definition of the SNOBOL4 language. It is written as a clear, step-by-step examination of the language.

4. Griswold, Ralph E., and Madge T. Griswold, *A SNOBOL4 Primer*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1973. ISBN 0-13-815381-7, paper, 192 pages.

   An introductory description of SNOBOL4 for individuals with little or no programming experience. It is written for workers in the humanities who want to apply computers to textual problems.

5. Hockey, Susan, *SNOBOL Programming for the Humanities*, Oxford: Clarendon Press, 1985. ISBN 0-19-824676-5, paper, 178 pages.

   An excellent introduction to SNOBOL programming for humanities researchers. The book derived from classes given at Oxford by Hockey. See also *Translations*, below.

6. Maurer, W. Douglas, *The Programmer's Introduction to SNOBOL*. New York: North Holland, 1976. ISBN 0-444-00172-7, paper, 141 pages.

   A concise, yet very complete guide to the SNOBOL4 language and its dialects, such as SPITBOL, SITBOL and FASBOL.

7. Shafto, Michael G., *Artificial Intelligence Programming in SNOBOL4*. Salida, CO: Catspaw, Inc., 1987, 170 pages. Originally published in 1982 by University of Michigan; available only on diskette.

   A report of SNOBOL4 programming techniques applicable to research in artificial intelligence. Includes extensive library of code for emulating LISP in SNOBOL4. In addition to the machine-readable report text, all programs and library code are provided in SPITBOL form.

All of the above materials are available by mail-order from Catspaw, Inc. Contact Catspaw for current pricing and shipping information.

Other references

8. Dewar, Robert B.K., and McCann, A.P. MACRO SPITBOL—a SNOBOL4 Compiler. *Software—Practice and Experience*, 7 (1977) 95-113.

   A description of the implementation techniques and design of MACRO SPITBOL, the system upon which Catspaw SPITBOL is based.

9. Gimpel, James F. A Theory of Discrete Patterns and Their Implementation in SNOBOL4. *Comm. ACM* 16, 2 (February 1973), 91-100.

   A formal treatment of SNOBOL4's pattern matching, including a detailed analysis of the quickscan heuristics.

10. Griswold, Ralph E. *REBUS - A SNOBOL4/Icon Hybrid*. Tucson, AZ: Department of Computer Science, The University of Arizona, TR 84-9, 1984.

    REBUS is a language that combines Icon-like control structures and syntax with SNOBOL4's pattern matching. It is implemented as a free-standing preprocessor that converts REBUS programs to SNOBOL4 or SPITBOL.

11. A *SNOBOL4 Information Bulletin* is published irregularly by the SNOBOL4 Project, Department of Computer Science, The University of Arizona, Tucson, AZ 85721.

12. *A SNOBOL's Chance* is another irregular newsletter, this one published by Catspaw, Inc., P.O. Box 1123, Salida, CO 81201.

Translations

13. Coppen, Peter-Arno, and Ben Salemans, *SNOBOL4 voor Iedereen*. Nijmegen: Stichting LOC, 1988. ISBN 90-5088-012-6, paper, 278 pages.

    A Dutch introductory SNOBOL4 text.

14. Toda, Shinichi, Bin Umino, and Kyo Kageura, *SNOBOL Nyumon: Tekisto Shorino Tameno Puroguramingu.* Maruzen, 1988. ISBN 4-621-03302-6, 241 pages.

A Japanese translation of Hockey's *SNOBOL Programming for the Humanities.*

Out of print

The following books are no longer available from booksellers. However, copies may be found at large academic libraries.

14. Griswold, Ralph E. *String and List Processing in SNOBOL4*. Englewood Cliffs, N.J.: Prentice-Hall, Inc. 1975. ISBN 0-13-853010-6, cloth, 288 pages.

A text on advanced programming topics and techniques using SNOBOL4. Much of the program material is included on the SPITBOL distribution media.

15. Griswold, Ralph E. *The Macro Implementation of SNOBOL4*. San Francisco: W. H. Freeman and Company, 1972, ISBN 0-7167-0447-1, 310 pages.

A detailed description of the internal operation of the standard SNOBOL4 compiler and interpreter; a case study of machine independent software.

16. Shapiro, Stuart C. *Techniques of Artificial Intelligence*. New York: Van Nostrand 1979, ISBN 0-442-80501-2, 166 pages.

This book consists of programming problems and examples of well-written programs in subareas of artificial intelligence. The example programs are written in LISP, SNOBOL4, or MICROPLANNER.

17. Uhr, L. *Pattern Recognition, Learning, and Thought: Computer Programmed Models of Higher Mental Processes*. Englewood Cliffs, N.J.: Prentice-Hall, Inc. 1973, ISBN 0-13-654095-3, 506 pages.

Contains a large number of AI programs written in the EASEy language, which in turn was written in SNOBOL4.

INDEX