LOGOS

User's Guide

REUTERS

LOGOS

User's Guide

REUTERS

ACKNOWLEDGEMENTS
LOGOS is a trademark of Reuter:file Ltd.

Printed in Canada
Publication Code: 1310 9008 E1
Version: 2.0

CONTENTS

BEFORE YOU START

This manual is your guide to learning and using LOGOS. It assumes that you know APL and want to learn LOGOS, whether you are a novice learning the basics, or an experienced user learning the more advanced features of LOGOS.

How to Use this Manual

Each chapter in this manual covers a certain aspect of LOGOS. The Table of Contents lists only chapter and main headings. The beginning of each chapter lists everything in that chapter, including subheadings.

Conventions Used in this Manual

The following conventions are used in this manual.

alias	Commands you type as shown are in lowercase italic font.
save **pathname** +*value*=**text**	Values you must supply in order to execute a command are shown in lowercase bold font.
save **objects**	Values for which you can supply a single item or a list of items are indicated by the plural form of the noun.
password: ***oranges***	Values entered by the user in sample sessions appear in bold face, italic font.
modifiers	New terms in the text are shown in bold face.
Ctrl-S	Keys on your keyboard that you must press simultaneously are separated by a hyphen (-).
F1	Function keys are shown as a capital F suffixed with the number of the function key you must press.
■	Instructions that are buried within a section of text are flagged with a black box.

Related Documents

For more information on LOGOS, see the following manuals:

LOGOS Reference Manual (pub code 1311 9008 E1)

LOGOS Pocket Reference (pub code 1313 9008 E1)

Requirements for Using LOGOS

To use LOGOS you must:

- have an APL terminal connected to a SHARP APL system
- have an account on your SHARP APL system
- have a keyboard equipped with the APL character set
- be enrolled in LOGOS. Your LOGOS steward can enroll you in LOGOS and give you an alias.

A Strategy for Learning LOGOS

LOGOS provides features that support many aspects of the software development process. You do not have to use all of these features to take advantage of LOGOS. Try a gradual approach. You will find that as you use more features, you gain more benefits. For example:

1. Acquaint yourself with LOGOS by reading the first few chapters of this manual.
2. Move one of your own applications into LOGOS, choosing a suitable hierarchical structure for it in the file system.
3. If the application already makes use of some paging mechanism or program file, have LOGOS build that file from the objects now in its own database.
4. Some time later, consider converting your application to use LOGOS paging.

CHAPTER 1: WHAT IS LOGOS?

Introduction to LOGOS

LOGOS is an environment for developing software using the APL language. It introduces structure to APL software development while preserving the flexibility and productivity that APL provides.

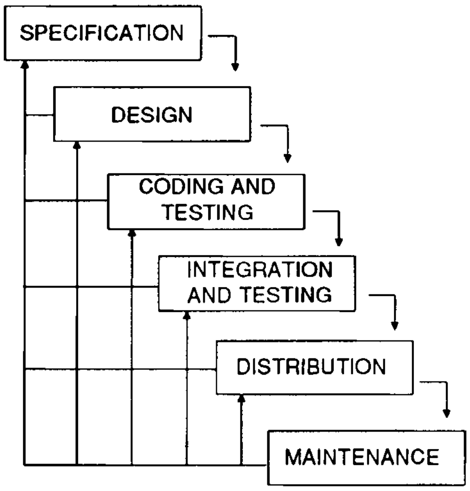
LOGOS can be used for projects of all sizes, from small, one-person projects to large projects handled by programming teams, and includes features that are especially useful to teams. For example, when a system is too large for one person to design and build, LOGOS encourages planning, centralization, sharing, and exploring. This results in:

- consistency among applications
- minimal redundancy
- more easily maintained systems
- improved communications within development teams

Managing the Software Lifecycle

The software lifecycle comprises several phases involving tremendous feedback and regression. LOGOS allows you to control each phase of the lifecycle, and the overall process.

Figure 1.1 The Software Lifecycle



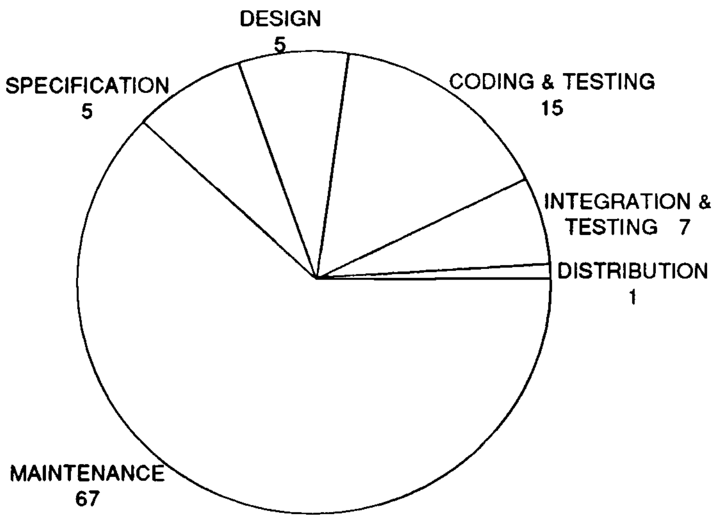
The phases are defined in the following table:

Table 1.1 Phases of the Software Lifecycle

Phase	Description
Specification	The description of what the system is supposed to do.
Design	The description of how the system will operate.
Coding and testing	The creation of programs and testing of local areas of the system.
Integration and testing	The combination of the parts of the system (possibly developed by different individuals) into a whole.
Distribution	The release and installation of the finished product.
Maintenance	The activities related to the continued operation of the system, including the correction of problems and the implementation of new capabilities.

LOGOS also provides the most tools where they are most needed. The coding phase is only about a sixth of the total effort required during the entire lifecycle; maintenance and support represent two thirds of the activities performed on a project. Therefore, LOGOS provides the greatest assistance during these phases.

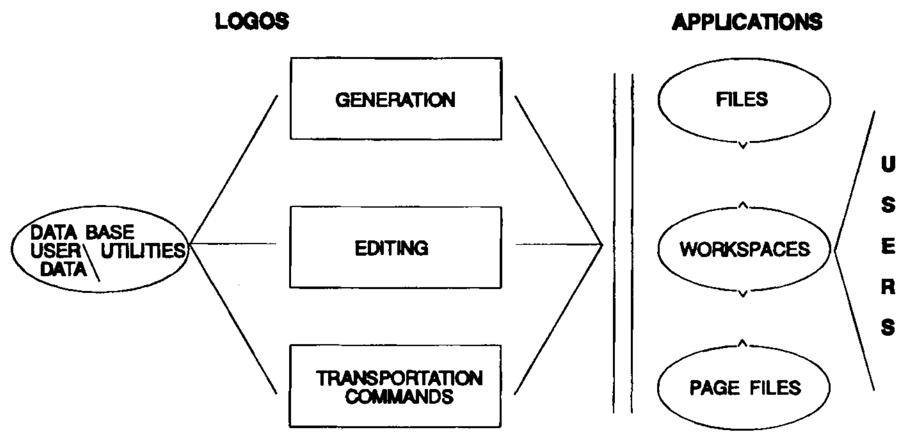
Figure 1.2 Breakdown of Software Lifecycle Activities



Structural Overview of LOGOS

From the end application's perspective, LOGOS looks like any conventional APL system: one or more workspaces cooperating (often invisibly) with files and page files.

Figure 1.3 Structural Overview of LOGOS



During development or maintenance, objects can move between the LOGOS file system and applications by:

- generating or regenerating a file or workspace
- editing objects
- moving objects explicitly using commands designed for this purpose

Since LOGOS knows where each object is being used, changes made to an object can be made to take immediate or deferred effect upon the applications that reference it.

Supporting End Applications

LOGOS supports the end environments that make up an application by providing simple but powerful mechanisms for generating and maintaining them.

However, LOGOS is not directly involved with running an application once it has been generated. From that point on, it looks just like a non-LOGOS application. Even if your application is using LOGOS paging to materialize and remove objects from a workspace, LOGOS and its database of objects are not involved in the process. The work is being done through streamlined paging utilities, which access the page files built by LOGOS. LOGOS itself is out of the picture in the running application.

Since LOGOS doesn't control your application while it is running, developing a system in LOGOS won't increase its overhead during production usage. In fact, the run-time efficiency of your application may increase due to space optimizations made by the LOGOS compiler or the relative efficiency of the LOGOS paging mechanism compared to more naive paging systems.

CHAPTER 2: GETTING STARTED

Loading LOGOS

LOGOS is an entirely self-contained system accessed through one of two functions. The *logos* function is an interactive interface that prompts you for commands and executes them. The *∆logos* function is a non-interactive interface to which you pass commands. (For more information, see the section, *Using LOGOS Commands from APL Functions*, in this chapter.)

You can use LOGOS in your application workspace or in workspace *1 logos*.

Using LOGOS in Workspace *1 logos*

Because of the multi-tasking nature of LOGOS, you can perform most of your work from a central workspace. For your convenience, workspace *1 logos* has been provided with a latent expression that autostarts LOGOS.

To use LOGOS in workspace *1 logos*, type:

```
)load 1 logos
```

LOGOS starts automatically. You see the message:

```
⋆logos⋆ r2.0
∪
```

This message indicates that you are using Release 2.0 of LOGOS. The union symbol (∪) means that LOGOS is ready to accept commands from the terminal.

Using LOGOS in Your Application Workspace

You can also use LOGOS in your application workspace. Complete the following procedures.

1 Load your workspace. For example:

```
)load sourcews
```

You see a message such as:

```
saved 1986-01-05 13:38:52
```

2 Copy LOGOS to your workspace by typing:

```
)copy 1 logos
```

You see a message similar to this:

```
saved 1986-01-16 04:45:27
```

3 Start LOGOS by typing:

```
logos
```

You see the message:

```
*logos* r2.0
∪
```

This tells you that you are using Release 2.0 of LOGOS. The union symbol (∪) means that LOGOS is ready to accept commands from the terminal.

Loading LOGOS without Access

If you are not enrolled in LOGOS and you attempt to load it, you receive the message:

```
*** you are not enrolled in logos ***
please send a message to <lstew> for access
```

Contact your LOGOS steward, who will give you access to LOGOS. Read the section, *Before You Start,* at the beginning of this manual to make sure you have all the other requirements to use LOGOS.

Inquiring about Your Alias

Your **alias** is a short form of your name, no longer than twelve characters, that LOGOS uses to identify you.

To inquire about your alias, type:

```
alias
```

LOGOS displays the alias you are currently using. For example, if your alias is *john,* you see the message:

```
john
```

This alias is your **primary alias** and is associated with your account number. The account number is called the **primary user number** and is the account number that owns the data for your alias, and pays for its storage.

Other Types of Aliases

An alias doesn't always refer to a person. An alias can also refer to a **group**. A group is one alias that includes several LOGOS users, rather than just one. For example, a group called *projdev* might include all individuals working on a common project. Groups make it convenient to grant access to data to everyone on a project simultaneously, rather than to individuals.

As a LOGOS user, you can belong to any number of groups. LOGOS still identifies you with your alias, but also recognizes you as part of any groups of which you are a member.

Entering Simple Commands

The *alias* command is an example of a simple command. You enter other LOGOS commands in exactly the same way you entered the *alias* command. When you see the prompt, you simply type the command and press Enter. Another simple command is *version*. It gives you more information about the version of LOGOS you are running. To use the *version* command, type:

```
version
```

You see the message:

```
logos version 2.0 (07sep90 18:38)
copyright (c) 1990 reuter:file ltd.
```

The *whois* command lets you inquire about other LOGOS users. You supply an alias, and LOGOS tells you the full name of that user. For example:

```
whois john
```

In this case, the alias you supply is the **argument** to the *whois* command. The command returns the full name of the LOGOS user with that alias. For example:

```
simon, john m.                          john
```

Another useful command is ⍎. With it, you supply an APL expression as the argument. This command works just like the APL primitive, and evaluates an APL expression. For example:

```
⍎⎕ts
```

This returns a result such as:

```
1990 1 9 16 44 27 731
```

You can even use the ⍎ command to conduct a short dialogue, by not providing an initial expression to be evaluated. For example:

```
∪ ⍎
⍎⍞ 24×60×60×60
5184000
⍎⍞ ⎕size 1
1 5 5864 93824
⍎⍞ 10000 ⎕resize 1
⍎⍞
```

To end the dialogue, enter one or more spaces and press Enter.

How to Recall a Line

If you make a typing error you want to correct or if you want to modify the last line you entered, you can recall it for editing by typing:

```
)
```

You can edit a line as you would in immediate execution in APL.

You can recall a line and position the cursor under a specific character within the line by supplying a number with). For example:

```
)7
```

This recalls the line and positions the cursor under the seventh character in the line.

Using Modifiers with Commands

You can modify the action of LOGOS commands by using them with parameters called modifiers. Modifiers always begin with a +. Several LOGOS commands can be used with several different modifiers.

For example, the *whois* command has a modifier called *+summary* which causes it to include group membership in its display. To use the modifier, you could type, for example:

```
whois john +summary
```

You see a message such as:

```
simon, john m.                          john
  inventdev probs
```

The *whois* command also has a modifier called *+name,* which searches for matching names rather than aliases.

For example:

```
whois sim +name
```

You see a message such as:

```
simon, john m.                  jsimon
simpson, margaret a.            msimpson
```

Modifiers can take values. For example, you can associate a password with your LOGOS alias using the *alias* command with the *+newpass* modifier. You must supply a value for *+newpass.* For example:

```
alias john +newpass=yellow
```

The value for *+newpass* is *yellow.* It becomes the password associated with the alias *john.*

Getting Help

If you need information, you can get help with the *help* command. Type:

```
help
```

You see the message:

```
For the names of all commands, type: ?
for a brief summary of all commands, type: ??
for a brief summary of a particular command, type: ?command
for a detailed description of a particular command, type: ??command
```

For example, to get a one-line summary of the *alias* command, type:

```
?alias
```

You see the message:

```
a[lias] [Alias [Password]] +newpass[=[PASSWORD]]
```

This tells you that the command:

- can be abbreviated to *a*
- takes two optional arguments representing the alias and password
- takes one modifier specifying the new password

The value for the modifier is optional too. If you don't specify it, LOGOS will ask for it with a protected prompt so that what you type is not shown on the screen.

Getting Out of LOGOS

You can leave LOGOS temporarily and enter APL immediate execution mode, or you can exit LOGOS altogether.

Leaving LOGOS Temporarily

You can temporarily suspend your LOGOS session and enter APL immediate execution mode by signalling an input interrupt in response to the main LOGOS prompt. You can then resume your session by branching.

1 To signal an input interrupt, type:

o **<Backspace>** *u* **<Backspace>** *t*

LOGOS suspends execution and displays the message:

```
input interrupt
type →⎕lc to restart
```

You can execute APL expressions and system commands. For example:

```
)opr please increase my file reservation by 1e6
sent
opr: done. your total reservation is now 2e6
)oprn thanks
sent
```

2 To resume your session, restart LOGOS by typing:

```
    →⎕lc
```

LOGOS displays the command separator character to indicate that LOGOS is ready to accept commands.

Exiting LOGOS

You can leave LOGOS at the end of your session using the *exit* command. Type:

```
exit
```

The *exit* command takes you out of LOGOS.

Loading Another Workspace

If you want to load another workspace when you exit LOGOS, you can include an expression for LOGOS to evaluate as the argument to the command. For example:

```
exit ⎕load '''666 box'''
```

You see a message such as:

```
saved 1989-10-12 18:40:56
```

NOTE: You must supply extra quotes around the workspace name, because LOGOS removes one set itself before passing the expression to APL.

Resetting the State of Your Workspace

When you invoke LOGOS, it reserves some internal storage in your workspace for maintaining information about your LOGOS environmental settings. Ordinarily this information is retained when you exit LOGOS, allowing you to restart the system with the same environment available when you left. This means, however, that there is less space available to other applications you might run in the workspace.

You can reclaim this space using the *exit* command with the *+reset* modifier. Type:

```
exit +reset
```

This also has the effect of untying any files LOGOS tied during your session. You would want to do this, for example, if you were planning to exit LOGOS, expunge the LOGOS function, and save the workspace.

Using LOGOS Commands from APL Functions

∆logos is a non-interactive interface to LOGOS, found in the path *.public.logos.∆logos*. With this interface you can invoke LOGOS commands from APL functions.

To use *∆logos*, invoke it with a character vector argument containing a command line. Type:

∆logos **'command line'**

For example:

∆logos 'list .public'

∆logos is a monadic function which returns an explicit result. The result is a character vector containing the result of the last command executed.

CHAPTER 3: USING THE FILE SYSTEM

Figures

Tables

LOGOS Objects

LOGOS is fundamentally about **objects**: any of the items you store in LOGOS. For example, an object can be a program, a numeric matrix, a package, or a nested array. There are no restrictions placed on the composition of objects within LOGOS.

Object Types

LOGOS classifies each object you store as a **type**. The following table lists the object types.

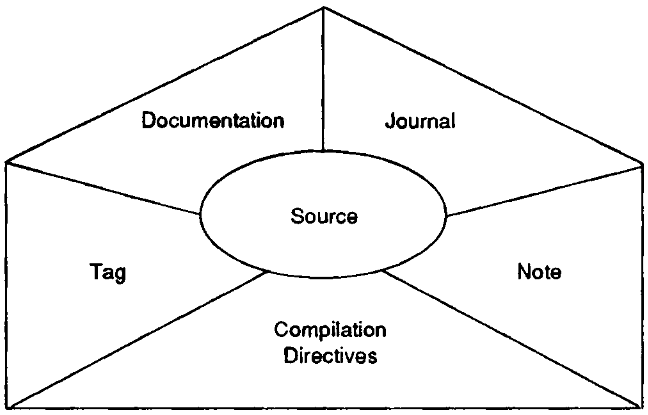
Table 3.1 LOGOS Object Types

Type	Short Form	Description
Cluster	c	An object built from other objects stored in LOGOS. A cluster is built using the *build* command.
Directory	*d*	A node in the LOGOS file system that imposes structure on a hierarchy of paths. A directory has a superior directory (up to the root of the file system), and may have descendant directories or objects. While objects of all other types provide content to the hierarchy, a directory's prime purpose is to provide structure.
Function	*f*	An APL user-defined function. A function may be of any valence, and may or may not return a result.
Link	*l*	A surrogate path in the file system that points to another path elsewhere. A link allows the same object to appear to reside in several different places.
Script	*s*	A program in which both APL expressions and LOGOS commands may be used together. A script can be called and executed like a LOGOS command. Scripts are described in *Chapter 5: Using Scripts*.
Variable	*v*	An APL variable. A variable may be of arbitrary type, rank, and shape, including a nested array or a package.

Object Attributes

Along with the object itself, LOGOS stores several kinds of information about each object. These kinds of information are called **attributes** of the object. They are really character vectors into which you put the appropriate information.

Figure 3.1 Composition of an Object



The following table lists the object attributes.

Table 3.2 LOGOS Object Attributes

Name	Attribute	Description
Compilation Directives	:*c*	Transformations that apply to the source form of an object, to produce a compiled form for production use. For more information, see *Chapter 10: Using the Compiler.*
Documentation	:*d*	A character vector that describes the purpose or intent of the path.
Journal	:*j*	A character vector that records the journal of changes made to the path.
Note	:*n*	A character vector that is displayed whenever a LOGOS command refers to an attribute of a path.
Source	:*s*	The essential object, such as the definition of a function or script, or the value of a variable.
Tag	:*t*	A character vector that is a short text on which categorization and searching can be performed.

LOGOS displays note attributes whenever a command references an attribute of a path. Typically, a note might include:

- general words of caution
- notice of recent changes
- semantic dependencies upon other objects
- a disclaimer

You are responsible for putting the documentation, journal, and tag attributes to use. There is no practical difference among these three attributes, but each is intended for a different purpose, as described in the table above. By convention they can help you organize and keep track of your application in LOGOS.

Sample Attributes

Your documentation attribute might look like this:

```
inserts an arbitrary indent of <n> characters before each carriage return in the argument array
<x>. if <x> is not a vector, it is converted to one by appending a carriage return to the front of
each row.

the program is origin independent.

examples:

      ρ⎕←2 indent 'note',CR,'tag'
  note
  tag
12

       5 indent 1 ⎕fd 'indent'
          ∇ z←n indent x;i;
      [1] ⍝ inserts an indent of . . .
      [2] ⍝ globals: v - CR.
      [3]  x←1↓,CR,x ⋄ . . .
      [4]  i←(1+(ρx)+n×ρi)ρ1 ⋄ . . .
          ∇
```

Your journal attribute might look like this:

```
 date     by   ver  description
 -----    ---  ---  --------------------------------
12jul84   lhg   1   program written
18jul85   dba   2   remove origin-1 dependency
02dec85   lhg   3   correct edge condition where text ends in CR
```

One convention for tags is to use single words separated by commas or blanks to delineate properties of an object. For example, your tag might look like this:

restartable, globals

The word *restartable* might mean that any line of the program is restartable by using →⎕lc, and the word *globals* might mean that the program uses at least one global function or variable.

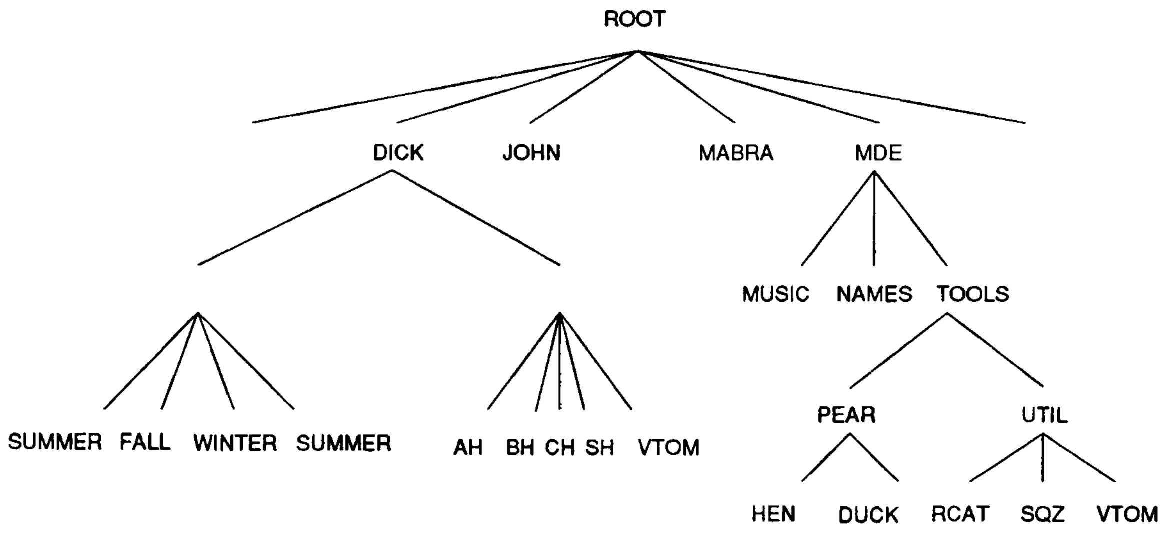
The File System

The LOGOS file system is the central database where objects are stored, and information about them is maintained. Just about any interaction with LOGOS makes some use of the file system, and processes such as the generation of workspaces and files use it extensively.

Immediately below the root of the LOGOS file system are the **alias level directories.** Each LOGOS user has a directory at the alias level, which is composed of a dot followed by your alias (for example, *.john*). The immediate descendants of your alias directory must be other directories, called **file level directories.** These can contain other directories and objects and so on.

Because there is only one root to the hierarchy, it is possible to reach any node from any other, resulting in an environment that leads to a natural sharing of objects among users.

Figure 3.2 A LOGOS Hierarchy



A level in the hierarchy with other levels beneath it is called a **non-terminal node** and is a directory. A level in the hierarchy with no other levels below it is called a **terminal node**, and can be an object or a directory with nothing stored in it.

About Pathnames

If you trace the hierarchy from the root directory to the alias level directory to the file level directory, you get a sequence of directory names. Together these names, separated by dots, are called a **pathname**. You use the pathname to access any object. The pathname tells LOGOS exactly where to find the object.

There are three types of pathnames, described in the following table.

Table 3.3 Types of Pathnames

Type	Example	Description
rooted	*.mde.util.tst*	The pathname begins at the root of the hierarchy and starts with a dot.
relative	*util.tst*	The pathname begins at a directory somewhere below the directory and does not start with a dot.
extended	*util.tst[:d]* *util.tst[f]* *util.tst[8]*	The pathname is succeeded by square brackets containing the extra information you want to access about that object. For example, the documentation attribute of an object, or a particular object type or version.

There is no overall limit to the length of a pathname, nor to the number of segments it may contain. Certain segments of the name do have special properties:

- The first segment always corresponds to an alias, and is therefore always a directory. It can be at most 12 characters long, and may not contain the character ⍙.

- The second segment is also always a directory; it corresponds to a SHARP APL file on the owner alias' account. The file has the same name as this segment of the path, prefixed by the character ⍙. This segment can be at most 10 characters long.

- Intermediate segments must begin with a letter.

- The final segment must also begin with a letter. If it's an object, the name can begin with ⎕ and must represent a valid APL object name. For example, ⎕*io* and *strings* are valid terminal object names, whereas @*xxx* is not.

LOGOS stores the following kinds of information for each path in the system:

Table 3.4 Types of Information Stored with each Path in LOGOS

Type	Description
Object type	One of: cluster, directory, function, link, script, or variable.
Object attributes	Any of: compilation directives, documentation, journal, note, source, or tag.
Retention	The number of versions of the object retained.
Permission	The access privileges a given user has to an object.
References	The workspace and file cross-references that LOGOS maintains automatically, so that you can determine who is using an object of yours. For more information, see *Chapter 11: Maintaining Systems.*
Compiled forms	For more information, see *Chapter 10: Using the Compiler.*

Setting a Working Directory

When you are working in LOGOS, you must set a **working directory**. This is a directory somewhere in the hierarchy in which you want to work.

For example, suppose you want to work in the directory *util*, in the directory *vtom*, in the directory *.john*. You can set your working directory to *.john.vtom.util*. From then on, LOGOS assumes you are working in *util*.

You can specify objects and directories within the working directory without preceding them with a dot. A pathname not preceded by a dot is called a **relative pathname** (as opposed to a **rooted pathname**, which does begin with a dot). When you supply a relative pathname, LOGOS assumes you are working within your working directory.

Displaying Your Current Working Directory

When you first use LOGOS, your default working directory is your alias level directory. To display your working directory, type:

workdir

The command responds by displaying the current working directory. For example:

.john

Changing Your Working Directory

You can change your working directory by supplying the pathname of the new working directory. For example:

workdir util

This changes the working directory from whatever it was (for example, *john*) to *util*. The system returns the new working directory:

.john.util

Resetting Your Default Working Directory

If you have modified your working directory, you can restore it to its default value at any time during your LOGOS session. Type:

workdir +reset

The command returns the default working directory. For example:

.john

Setting Multiple Working Directories

You might want to set a list of directories as your working directory. When you refer to a relative pathname, LOGOS searches the directories in your working directory list, in the order in which you specified them, until it finds a match. This layering effect can be very useful in setting up multiple test and production environments. (See *Chapter 8: Building Applications with LOGOS*.)

To set multiple working directories, specify each directory you want added to the list. For example:

workdir .john.modules .john.utils .john.cmds

The first directory in your list is called your **primary working directory**.

Creating and Saving Objects

The *save* command allows you to save objects from your workspace into the LOGOS hierarchy, and to create new objects in LOGOS. You can also create new objects using the *edit* command. For more information on creating objects using *edit*, see *Chapter 6: Using the Editor*.

To use the *save* command, you supply the name of the object you want to create, or the name of the object you want to copy from your workspace.

- To save a workspace object into your primary working directory, supply the name of the object. For example:

  save chart

- To save more than one object at a time, separate the object names with spaces. For example:

  save chart print report

- To save the object into a directory other than your primary working directory, you must specify the directory name as well. For example:

  save .john.modules.chart

The *save* command displays each pathname saved, the object type, and its version number. For example, if you save the objects *chart, print,* and *report,* the system displays the message:

```
.john.modules.chart[fl]
.john.modules.print[fl]
.john.modules.report[fl]
```

The version number is automatically updated when you save the source of an object.

Saving Object Attributes

To save an attribute, you must supply an extended pathname indicating the attribute you want to save. You must also use the *+value* modifier and provide a value for it. For example:

```
save chart[:n] +value=still not quite right
```

This creates a note for an object called *chart* in your primary working directory.

LOGOS displays the pathname of the attribute created. For example:

```
.john.modules.chart[fl:n]
```

Specifying Values

To use an object in your workspace as the value of an attribute, type:

save **objectname** *+value=* ⍎**expression**

For example, suppose you have a variable in your workspace called *charthow,* which describes how *chart* works. You can use the following variant of the *+value* modifier to save it as the path's documentation:

```
save chart[:d] +value=⍎charthow
.john.modules.chart[fl:d]
```

The construct *+value=*⍎**expression** means that **expression** is to be evaluated in the active workspace, and the result of the evaluation is to form the specified attribute. Note that attributes other than source must be character vectors.

- You can specify character data by typing:

  save **pathname** *+value=***text**

  In this case, **text** is used as the object's value without evaluation.

  Be careful to observe that this is quite different from using the ⍎ command, as in *+value=*(⍎**expression**), which instead causes the result of the evaluation to be interpolated directly into the command line as a character vector. In this example, *+value=*(⍎*1*) would have saved ⎕*io* as the one-element character quantity 1, instead of the numeric scalar quantity 1.

  If you specify source using *+value,* the resultant path is presumed to be a variable by default.

- To save a different object type, (for example, a function or a script) include the object type in brackets following the pathname.

  For example:

  save chart[*f*] *+value=*⍎*newchart*

  Here, *newchart* is a character representation of the function display, such as that returned from ⎕*cr* or *1* ⎕*fd*. Using the *save* command in this manner, you can write your own editor and let *save* store the altered representation back in LOGOS.

Setting Retention

When you change an object in the LOGOS file system and save it again, you can still retain previous versions of the objects. The value of a path's *retention* specifies how many versions are kept.

Retention is inherited from the parent node. If a directory has a particular retention, then any directories or objects created below it will automatically inherit that retention. Similarly, any directories created below these will inherit their parent's retention. In this fashion, you need only establish the desired retention setting at a suitably high point in the hierarchy, and LOGOS will ensure that the value propagates to all objects subsequently created.

When you are enrolled in LOGOS, a retention of 10 is associated with your alias. Any directories or objects you create have this retention by default.

- To set or alter the retention of a path, use the *retain* command and provide the number of the new retention value and the pathname to which it applies. For example:

  retain 15 .john.util.?⋆

LOGOS displays a message informing you of the retention changed. For example:

retention changed for 3 paths

This changes the retention of the objects in the directory *.john.util* to 15. A retention value of *all* causes LOGOS to keep all versions of the objects.

NOTE: To change the retention value for every path below the named directory, specify the *+recursive* modifier.

Saving a Version

To save a version of an object, supply an extended pathname indicating the version you want to save. For example:

save chart[*1*]

LOGOS displays a message indicating the version of the object saved. For example:

.john.modules.chart[*f1*]

Version 1 has now been updated to reflect its new definition. Version 2 is, however, still considered the latest version. It will be referenced if you don't specify a version number in commands such as *edit* and *get*.

Version numbers are not automatically incremented when you save non-source attributes. You can lose a version of the documentation that you might want to keep available. However, you can always force a new version to be saved by including the desired version number with the pathname.

Saving a Version of an Attribute

To save a version of an object's attribute, supply an extended pathname indicating both the version number and the attribute. For example:

save chart[*3*:*d*] *+value=♝charthow,newstuff*

LOGOS displays a message indicating the version of the object attribute saved. For example:

.john.modules.chart[*f3* : *d*]

Conversely, you can avoid creating a new version of a path when saving the source form, simply by supplying the current version number in the *save* command. For example:

save .john.modules.chart[*4*]

Complementary Indexing

One way to refer to the current version of an object is through **complementary indexing.** For example, to refer to the current version, you can specify:

chart[*0*]

To refer to the version one before the latest version, you can specify:

chart[¯*1*]

To refer to the version two before the latest version, you can specify:

chart[¯2]

Creating Directories

Because a directory is a special type of object, you must provide an extended pathname indicating object type when you save one. For example, to save a directory into your working directory, type:

save util[*d*]

To save a directory into a directory other than your working directory, you must also supply a rooted, extended pathname. For example:

.john.modules.util[*d*]

NOTE: There is one exception. If the directory you are creating is a **file level directory** (one level below the alias), it is not necessary to provide an extended pathname. File level directories can contain only directories, so anything you create at this level is assumed to be a directory.

Creating Links

A **link** is an object whose attributes are, in every respect, defined by another object or directory. A link allows the same object or directory to appear in several directories under different pathnames, without creating more than one copy of the entity.

In instances where the original pathname is quite long, or where you use your working directory to help shorten the lengths of your paths, links can save typing. But, more important, links help reduce the number of working directories you regularly use, without complicating the maintenance of the objects.

When to Use a Link

Use a link when you want to reference a name from a particular directory, but the name is actually along a different path in the hierarchy. Rather than maintaining several up-to-date copies of the same object, maintain a single copy and define links to it wherever it's needed.

For example, suppose you want the object *.mde.tools.util.sqz* to be resident in the directory *.john.util.* You have two choices. You can:

- copy the object, creating a new object that is not related to the object in *mde*; or
- create a link to the object in *mde*

By creating a new object, the version in *.john* is separate from the original and must be maintained independently. By establishing a link, you automatically pick up the latest version along the path *.mde.tools.util.sqz,* any time you reference the object *.john.util.sqz.*

Creating a Link

You can link to both objects or directories using the *link* command.

To link to an object, specify the objects you want to link. For example:

```
link .john.util.sqz .mde.tools.util.sqz
```

To link to a directory, specify both the object and the directory you want to link. For example:

```
link .john.util.mdeutil .mde.tools.util
```

Now, you can use any path in *.mde.tools.util* by referring to it as if it were along the path *.john.util.mdeutil.* For example, *.john.util.mdeutil.rcat* is the same object as *.mde.tools.util.rcat.*

Disabling Link Resolution

Link resolution is a property of a command, and most commands resolve links. When a link appears as a pathname or a portion of a pathname, LOGOS replaces the link with its value and then performs the action of the command on the newly-formed name.

To disable link resolution, specify your command, and supply an extended pathname of *l*. For example:

```
save .john.util.sqz[l]
```

The pathname will not be resolved even if it is a link.

There are a few commands that normally do not resolve links; they are *copy*, *delete*, and *list*. Even with these commands, however, a link specified as the non-ultimate segment of a path will be resolved. For example, *list link* won't resolve the link, but *list link.vtom* will.

Listing Directory Contents

You can list the names of everything you store in LOGOS using the *list* command. Simply supply the pathname of the directory you want to examine. For example:

list util

You see a message such as:

ah
dth
htd
sh
vtom

NOTE: LOGOS lists only the directories and objects to which you have some permission. If you are not authorized to use an object or a directory, it will not appear when you use the *list* command. For more information on permission, see the section, *Controlling Access*, in this chapter.

Listing Additional Information

You can display much more information about every pathname by using modifiers with the *list* command. For example, you can display:

- the type of each path
- the full pathnames of each path
- a summary of each path
- headings with the list
- current retention of paths
- information on all versions of each path
- who created the path, when, and the number of versions in LOGOS

For more information on using modifiers with the *list* command, see the description of the *list* command in the *LOGOS Reference Manual*.

Looking at Structure

You can list all the directories and objects below a particular point in the hierarchy using the *+recursive* modifier with the *list* command. The *+recursive* modifier causes *list* to enumerate the contents of any directories it finds below the path you specify. If these directories contain others, they too are enumerated.

- To list all the directories and objects below a particular point in a directory, supply the object you want to list and the *+recursive* modifier. For example:

  ```
  list .mde +recursive +type
  ```

  You see a message such as:

  ```
  d .mde
  d .mde.music
  d .mde.names
  d .mde.tools
  d .mde.tools.pear
  v .mde.tools.pear.hen
  v .mde.tools.pear.partridge
  d .mde.tools.util
  f .mde.tools.util.rcat
  f .mde.tools.util.sqz
  f .mde.tools.util.vtom
  ```

  Notice that *.mde.music* and *.mde.names* are directories, but contain no descendants.

- To restrict the *list +recursive* command to a particular object type, provide an extended pathname indicating the type.

  For example, to list just the directories below a certain node in the hierarchy:

  ```
  list .mde[d] +recursive
  ```

  You see a message such as:

  ```
  .mde
  .mde.music
  .mde.names
  .mde.tools
  .mde.tools.pear
  .mde.tools.util
  ```

NOTE: A general LOGOS utility, called *tree*, can turn this display into a graphic tree structure. For an example of *tree*, see *Chapter 15: Using the Utility Library.*

Accessing Objects in the File System

Once you have created objects and directories, you can access them (for example, list them, display them, edit them) by providing the pathname with the command. For example:

display .cmds.edit

If your list of working directories includes *.cmds,* you can type:

display edit

If you begin a pathname with ↑, LOGOS interprets this as referring to the parent of the current primary working directory. For example, with a primary working directory of *.john.util,* the path *↑.lang.english* is equivalent to *.john.lang.english.* Starting a path with ↑↑ climbs two levels in the hierarchy, and so on.

Accessing Extended Pathnames

To access an object's type, version, or attributes, you must supply an extended pathname indicating the type, version, or attribute you want to access.

- To access an object type, supply an extended pathname indicating the type. For example:

  display util.htd[*f*]

  This selects the function *htd* within the directory *util.*

- To access a particular version of an object, supply an extended pathname indicating the version. For example:

  display util.htd[*3*]

  You can specify version numbers using a complementary form as well, where *util.htd*[*0*] is the most recent version (whatever its number), *util.htd*[¯*1*] is the second-to-last, and so on.

- To select an attribute of an object, supply an extended pathname indicating the attribute. For example:

  display util.htd[:*d*]

  This selects the documentation of *htd.*

- To access a pathname with several qualifications (for example, the documentation attribute of the fifth version of a function), supply an extended pathname indicating those qualifications.

For example, to select the eighth version of the function *util.htd*, type:

util.htd[*f8*]

To select the documentation of version 8 of the function *util.htd*, type:

util.htd[*f8*:*d*]

NOTE: You can specify multiple type qualifications (for example, *util.htd*[*fv*]), but you can specify only one version number and one attribute.

Using Pattern Matching

A simple way to select and manipulate pathnames is using regular expressions. These are metacharacters combined with partial pathnames. (For more information on regular expressions, see *Chapter 12: Software Development Tools*, and *Appendix A: Using Regular Expressions*.)

For example, you can combine the partial pathname with the metacharacters *?* (match any character) and ⋆ (as many times as possible) to form the regular expression *temp?*⋆. You can use it to list those names that begin with *temp*, by typing:

list temp?⋆

To find all descendants of the pathname *.john.util* ending in *s*, type:

list .john.util.?⋆*s*

To find all those descendants of the pathname *.john.util* with two-letter names, type:

list .john.util.??

To find all those descendants of the pathname *.john.util* containing the character Δ anywhere within them, type:

list .john.util.?⋆Δ*?*⋆

NOTE: The ⋆ metacharacter works on whatever character precedes it. For example, to find *vtom* in any directory beginning with *util* and ending with zero or more repetitions of *s*, you can use:

list .john.utils⋆*.vtom*

In particular, this pattern might match the names *.john.util.vtom* and *.john.utils.vtom*, as well as *.john.utilss.vtom*.

HINT: If you name your directories and objects in a systematic way, you can use patterns to select families of names as easily as a single one. For example, if all subfunctions in a particular directory begin with the character Δ, you can select all of them at once using the pattern Δ?★.

Separating Several Patterns

You can use the | metacharacter to separate several patterns that you are looking for. For example, to find all two-letter names or all names beginning with *temp*, type:

?? | temp?★

To find *ah* and *vtom* in *.john.util*, type:

.john.util.ah | vtom

This form provides a convenient way to apply a common prefix, which is not your working directory, to a series of pathnames.

To look for the four paths *pear.hen, pear.sqz, util.hen,* and *util.sqz* within *.mde. tools*, use:

.mde.tools.pear | util.hen | sqz

If you know that only the first and last of these exist, you can use alternation as a very concise way to express the paths.

Patterns and Multiple Working Directories

Patterns and multiple working directories have an interesting common application. With multiple working directories, LOGOS matches only the first occurrence of the pattern in each of the working directories. For example, if you have two working directories set, you might type:

list ?★

LOGOS lists the first occurrence of every object in both directories.

Setting Up a Simple System

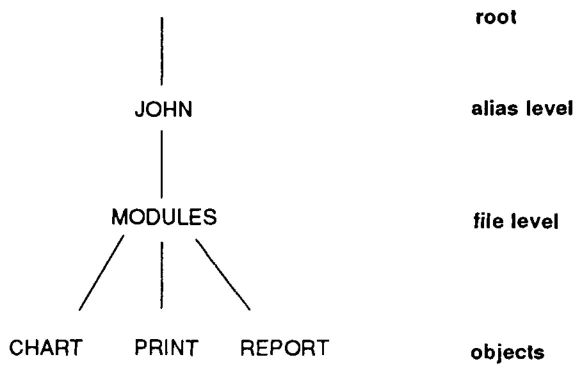
When you begin saving several objects into LOGOS, the organization of your hierarchy becomes an important issue. Should you spread out the objects among a large number of directories, or group many objects into a small number of directories? The ultimate choice depends on the structure of the system and the way you intend to deal with the objects afterward. Data organization is described at length in *Chapter 9*.

Creating Directories and Saving Objects

One way to organize your data is to group all subroutines and global variables subordinate to a main routine in a directory below the routine.

For example, suppose you have a directory that looks like this:

Figure 3.3 A Simple Directory



The names *chart, print,* and *report* are all objects. You can create directories in which to save subroutines and global variables.

1 Create the directories.

For example, if your working directory is *.john.modules,* you could specify:

save chartsub[*d*] *printsub*[*d*] *reportsub*[*d*]

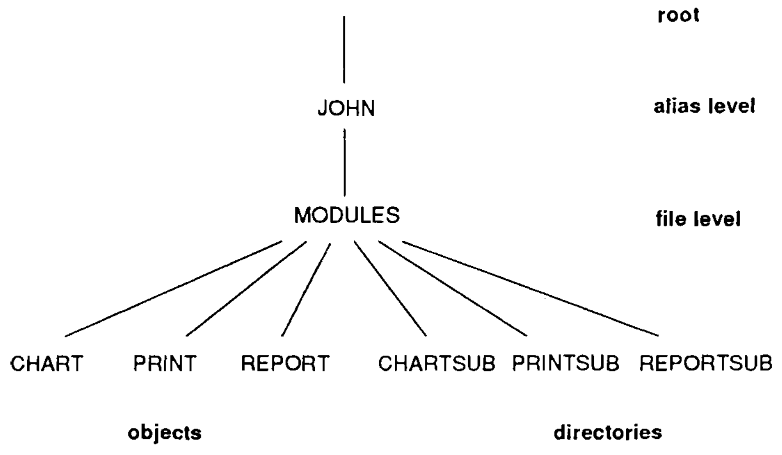
You must specify the object type [*d*] to tell LOGOS that you are saving directories. Otherwise, LOGOS will attempt to find objects by these names in your workspace.

You see the message:

.john.modules.chartsub[*d1*]
.john.modules.printsub[*d1*]
.john.modules.reportsub[*d1*]

The hierarchy looks like this:

Figure 3.4 A Simple Hierarchy with Directories



2 Save the subroutines into the new directories.

For example, to save *chart*'s subroutines in *chartsub*, type:

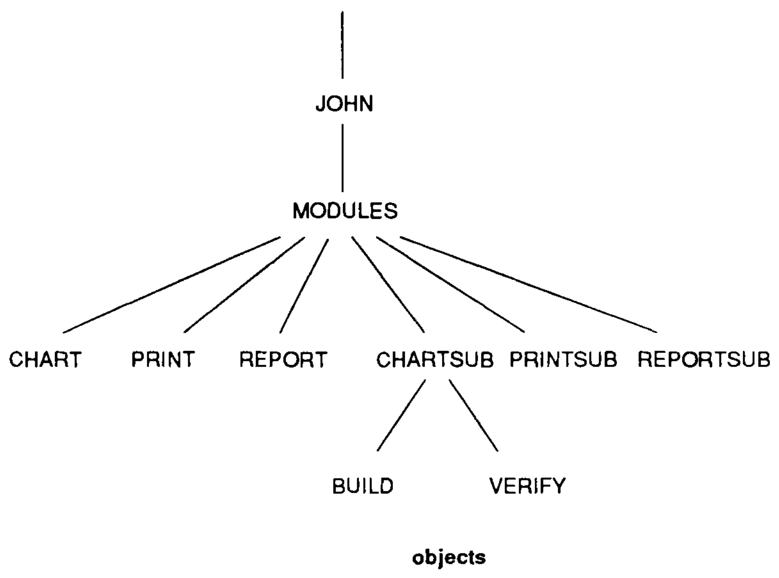
save chartsub.build chartsub.verify

You see the message:

.john.modules.chartsub.build[*f1*]
.john.modules.chartsub.verify[*f1*]

With the addition of *chart*'s subroutines, the hierarchy now looks like this:

Figure 3.5 A Simple Hierarchy with Directories and Objects



HINT: You can often save typing by using the *+workdir* modifier. For example:

save build verify +workdir=chartsub

The *+workdir* modifier sets a temporary working directory relative to your primary working directory for the duration of the command. When the command finishes, your working directory reverts to the value it had before.

Storing Several Objects in One Directory

Where there are functions, there are usually subroutines; and where there are subroutines, there are usually common utilities that are shared by more than one caller. To avoid duplication and to facilitate maintenance, you can store all utilities in the same directory.

To create the directory and store the objects, type:

save **objects** *+workdir=***pathname** *+makedir*

For example:

```
save mtov rcat vtom +workdir=util +makedir
```

You see a message such as:

```
.john.modules.util[d1]
.john.modules.util.mtov[f1]
.john.modules.util.rcat[f1]
.john.modules.util.vtom[f1]
```

The *+makedir* modifier tells the *save* command to create any intermediate directories implied by the pathnames specified. In this case, *.john.modules.util* did not exist before the command was executed, so it was automatically created by LOGOS. Without the *+makedir* modifier, the command would have failed because *.john.modules.util* was not a valid directory.

Copying Paths

Once you have some objects in LOGOS, you'll need to move them about. The commands that allow you to do this are *copy* and *delete*. The *copy* and *delete* commands operate on an entire hierarchy; they affect all nodes at and below the paths you specify. The first part of this section discusses the *copy* command, the remaining part discusses the *delete* command.

Copying a Single Path

The *copy* command duplicates paths or hierarchies in the LOGOS file system, giving you two distinct occurrences of the objects.

To copy one path to another, specify the source object's pathname first and the target directory's pathname second. Type:

copy **source target**

For example, to copy the path *.john.modules.□io* to the directory *.john.defaults*, type:

```
copy .john.modules.□io .john.defaults
```

LOGOS displays a message indicating the path copied. For example:

```
.john.defaults.□io[v1]
```

NOTE: Do not include the object's name in the destination pathname.

Copying a Group of Paths

You can also copy a group of paths by specifying the pathnames of all of the objects to copy, and the pathname of the destination directory. For example:

```
copy chart print report .john.system
```

The last pathname is the target, and all others are the source. This displays the message:

```
.john.system.chart[fl]
.john.system.print[fl]
.john.system.report[fl]
```

If the source paths contain any directories, all descendants of each directory are copied as well. This allows you to duplicate hierarchies as easily as single paths. For example:

```
copy .john.modules.?* .john.newsys +makedir
```

You see the message:

```
.john.newsys[dl]
.john.newsys.chart[fl]
.john.newsys.chartsub[dl]
.john.newsys.chartsub.build[fl]
.john.newsys.chartsub.verify[fl]
.john.newsys.print[fl]
.john.newsys.printsub[dl]
.john.newsys.report[fl]
.john.newsys.reportsub[dl]
.john.newsys.util[dl]
.john.newsys.util.mtov[fl]
.john.newsys.util.rcat[fl]
.john.newsys.util.vtom[fl]
```

Paths are always copied below the target directory, which is the last pathname you specify in the command's argument.

HINT: If the target directory doesn't exist, you can use the *+makedir* modifier to create intermediate directories implied by the target path, as was done for *.john.newsys* above. This is necessary only if the target directory itself does not already exist; if the list of paths to be copied contains directories, the directories will be created (if necessary) whether or not *+makedir* is used.

Copying Selected Versions

Normally, the *copy* command copies only the latest version of the desired objects. You can control this behaviour with the *+versions* modifier.

- To copy all versions of the source pathnames, use the *+versions* modifier with no value, or with a value of *all*. For example:

  copy chart +versions

  or

  copy chart +versions=all

- To specify the number of versions you want copied, provide a number as the value to *+version*. For example:

  copy chart +versions=7

  The value behaves like the APL ↑ primitive function: *+versions=¯3* specifies the three most recent versions, and *+versions=5* refers to the first (earliest) five versions.

- To copy specific versions, specify an extended pathname indicating the version you want to copy. For example:

  copy chart[7]

HINT: You can also use the *+versions* modifier in this case. For example, to copy the three most recent versions of *chart* ending at version 4, you could type:

copy chart[*4*] *.john.temp +makedir +versions=¯3*

LOGOS displays a message such as:

.john.temp[*d1*]
.john.temp.chart[*f1*]
.john.temp.chart[*f2*]
.john.temp.chart[*f3*]

Versions 2, 3, and 4 of *chart* are copied as versions 1, 2, and 3 of *.john.temp.chart*, respectively.

Copying One Directory to Another

The *copy* command duplicates all source paths with all their descendants. When you want to copy only the contents of one directory to another (as opposed to the directory itself), you must specify the directory's descendants explicitly.

- To copy the contents of one directory to another, provide the pathnames of objects you want to copy, and the pathname of the destination directory. For example:

```
copy .john.modules.util.?* .john.util
```

You see a message such as:

```
.john.util.mtov[fl]
.john.util.rcat[fl]
.john.util.vtom[fl]
```

HINT: One common mistake with the *copy* command arises out of trying to copy the contents of one directory to another. Suppose you wish to copy the contents of *.john.modules.util* to the directory *.john.util*, which already exists. You might attempt to do the following:

```
copy .john.modules.util .john.util
```

You see the message:

```
.john.util.util[dl]
.john.util.util.mtov[fl]
.john.util.util.rcat[fl]
.john.util.util.vtom[fl]
```

This has the unintended effect of creating a *util* directory beneath *.john.util*. The additional directory was created because the *copy* command duplicates all source paths with all their descendants. To avoid duplication of the source directory itself, you need to specify the directory's descendants explicitly as shown in the preceding example.

Limiting a Directory Copy

You may not always want to copy all descendants of a directory, particularly if the directory you are copying contains further directories. The most straightforward way to limit the copy is to enumerate the source pathnames by using the result of a *list* command. Type:

copy (list **pathname** *+full)* **directory** *+makedir*

For example:

```
copy (list .john.modules[fv] +full) .john.commands +makedir
```

LOGOS displays a message such as:

```
.john.commands[d1]
.john.commands.chart[f1]
.john.commands.print[f1]
.john.commands.report[f1]
```

The full pathnames of the functions and variables in *.john.modules* are computed by the *list* command and passed to the *copy* command, which then copies them to the new directory *.john.commands* (created by the *+makedir* modifier).

Deleting Paths

Objects are removed from the LOGOS file system with the *delete* command. You can use this command to remove particular versions of objects, entire paths, groups of paths, or complete hierarchies.

Deleting All Versions

Unless you specify a particular version, the *delete* command deletes all versions of the specified paths.

To delete all versions of a path, specify the path to delete. For example, to delete the path *.john.defaults.□io*, type:

```
delete .john.defaults.□io
```

LOGOS displays a message indicating how many paths were deleted (and how many versions, if different). For example:

```
1 path deleted
```

In this case, the number of versions deleted (one) matched the number of paths, so the message showed only the number of paths deleted.

Deleting Specific Versions

Rather than deleting all versions of a path, you can delete only one version by supplying an extended pathname indicating the version you want to delete. For example:

```
delete .john.temp.chart[1]
```

This command deletes only one version of the path. If there was more than the one version, then the other versions of the path remain in the file system. You see the message:

```
1 version deleted
```

Deleting the Latest Version

You can delete the latest version of a path by providing an extended pathname, specifying either the version or 0.

For example, if the latest version of *.john.temp.chart* is version 35, you can type:

```
delete .john.temp.chart[35]
```

Or you can type:

```
delete .john.temp.chart[0]
```

Deleting in Multiple Directories

You can delete paths in more than one directory by setting multiple working directories, or by specifying more than one directory in your *delete* command.

To set multiple directories, see the section, *Setting Multiple Working Directories*, in this chapter.

NOTE: The *delete* command operates on one file at a time. If you set multiple working directories, you might specify a command such as:

```
delete a?*.
```

If one path is matched and deleted in each of two directories in two separate files, you do not get a warning message. You simply receive the message:

```
2 paths deleted
```

If you try to delete paths in more than one file-level directory at a time, a different set of prompts is issued for each file. Responding *no* to a warning prompt avoids deletions only within that particular file.

Deleting Obsolete Versions

As you develop a system in LOGOS, you may accumulate several versions of your objects. You can set a retention count on your directories in order to limit the total number of versions stored, but you may want to store many versions while you are changing your system. The *+noncurrent* modifier enables you to remove versions that are no longer needed.

- To delete all but the latest version of a path (and its descendants), specify the *+noncurrent* modifier. For example:

```
delete .john.modules +noncurrent
```

- To delete all paths prior to a certain version, specify an extended pathname indicating the version with the *+noncurrent* modifier. For example:

```
delete .john.modules[~3] +noncurrent
```

This command deletes all but the latest three versions of all objects at and below *.john.modules*. This is particularly convenient when used in conjunction with complementary indexing.

Deleting Unreferenced Objects

After using LOGOS for a while, your object database may become cluttered with objects no longer in use. You can use the *+unused* modifier to remove these objects. For example:

```
delete .john.modules +unused
```

This command selects for deletion only those objects that do not have a current reference (that is, are not used by anyone, anywhere in LOGOS).

The *references* command, described in *Chapter 11: Maintaining and Updating Systems,* determines what objects are in use and where they are used. When the *delete* command finds objects that are still active, you can use the *references* command to find out where and by whom they are being used.

IMPORTANT: Use this modifier with discretion; this form of deletion requires all references to be searched, and can be expensive.

Displaying Results

The result of the *delete* command is normally not displayed. It consists of the pathnames and versions deleted. You may obtain this information by prefixing the command with an assignment. Display the deleted paths on your terminal with ⎕←*delete,* or assign the result to a variable.

Warnings Issued by the *delete* Command

To prevent mishaps with such a destructive command, the *delete* command warns you if the number of paths it is about to delete exceeds the number of paths you specified.

For example, suppose you type:

```
delete .john.temp
```

A warning might appear such as:

```
delete 2 paths?
```

In this case, *.john.temp* contained one descendant, making a total of two paths to delete, while only one was specified. This caused the warning prompt.

The number of paths reported for deletion in a warning prompt is the count of pathnames, not versions, to be removed. Therefore, particular versions being deleted do not affect this prompt, and if only some versions of a path are being deleted, that path has no effect on the warning prompt.

Responding to Prompts

At a warning prompt, you can respond with the commands in the following table:

Table 3.5 Commands You Can Type at the Warning Prompt

Command	Effect
yes (or *y*)	Allows the deletion to proceed.
no (or *n*)	Cancels the delete request for the specific object
stop	Aborts the *delete* command instantly
confirm	Offers you a way to protect your data from inadvertent deletion, by entering an interactive confirmation mode dialogue. In this mode, a separate prompt is issued for each object pending deletion.

If you respond with *confirm,* you enter **confirmation mode**, where prompts are then issued for the requested directory and each of its descendants. For example:

```
υ delete .john.commands
delete 4 paths? confirm
delete <.john.commands>? yes
delete <.john.commands.chart>? no
delete <.john.commands.print>? y
delete <.john.commands.report>? y
unable to delete: .john.commands (not empty)
2 paths deleted
```

In this case *chart* will not be deleted. Because *chart*'s parent directory is therefore not empty, it will not be deleted either.

Controlling the Warning Prompt

You can control the warning prompt issued by the *delete* command with the *+warn* modifier. Provide the number of paths after which you want to be warned. For example if you enter:

```
delete pathname +warn=5
```

LOGOS issues a prompt if the number of paths to be deleted equals or exceeds 5.

To stop the warning prompt from being issued at all, specify *+warn=0*. To prompt every time the *delete* command finds something to delete, specify *+warn=1* .

Enabling Confirmation Mode

You can automatically enable confirmation mode by using the *+confirm* modifier with the *delete* command. For example:

```
delete .john.commands +confirm
```

When you are in confirmation mode, you can enter the following keywords in response to deletion prompts:

Table 3.6 Keywords You Can Enter In Confirmation Mode

Keyword	Effect
yes (or *y*)	Allows the deletion to proceed.
no (or *n*)	Cancels the delete request for the specific object
back	Returns to the previous prompt.
continue	Performs the deletion without further prompting.
stop	Aborts the *delete* command (also applies to warning prompt).

Controlling Access

You have full control over the users who are allowed to see and change objects which you put into LOGOS. Users cannot see any directory or object they can't use. Your data is also carefully protected against unauthorized access (also called permission) or contamination by the LOGOS system itself.

The *share* command controls access. It extends or revokes access, depending on the modifiers you use with it. The access controls take the form of an access matrix, which specifies the users who may access the node and the types of permission to which each is entitled.

Types of Permission

Types of permission include execute, read, write, and control access. They can be assigned independently or in any combination. The meaning of each of these levels of permission differs slightly when applied to a directory as compared with an object. The table below summarizes the permission levels and what they mean, for both directories and objects.

Table 3.7 Types of Permission

Permission	Applied to an Object	Applied to a Directory
Execute (X)	Allows you to execute a script, fetch an object, and list an object's name.	Allows you to list the name of a directory.
Read (R)	Allows you to read the source and other attributes of an object.	Allows you to read the attributes of a directory.
Write (W)	Allows you to write the source and other attributes of an object.	Allows you to write the attributes of a directory, add objects to it, and delete objects from it (if you also have write permission to the objects).
Control (C)	Allows you to alter access to an object.	Allows you to alter access to a directory.

How Access Extends Automatically

When you grant an alias (or a group) access to a directory, the same access automatically extends to new directories and objects that you subsequently create below that directory, in much the same way as retention extends to new objects. Access to a directory defines access to its descendants, unless you specifically change the permission.

When you grant a group access to a directory, the access extends to every member in the group. If you add members to a group, they automatically get the access extended to the group. If you remove members from the group, they automatically lose the access extended to the group.

For example, suppose that the group *invendev* identifies the members of a project working on Inventory System development. If you give *invendev* access to all LOGOS paths pertaining to the project, then the task of keeping access information up to date as people join and depart the project involves only the maintenance of the membership of the group. When the group is changed, the access to all relevant paths changes instantly.

Inquiring about Access

The *share* command without modifiers acts in an inquiry capacity only, and returns the permission of the named aliases to the named paths. You can inquire on access in two ways. You can:

- inquire on a specific alias' access to paths
- inquire on all aliases' access to paths

Inquiring on a Particular Alias

To inquire on a particular alias' access to paths, specify both the alias and the paths in the command. For example, assume your working directory is set to *.john*. To inquire on *invendev*'s access to the node *.john.modules.chart,* type:

```
share invendev modules.chart
```

You see the message:

```
.john.modules.chart          invendev    x
```

The alias *invendev* has execute access.

NOTE: Execute access alone will not allow you to read the source of an object. If you try to read the source, an error message occurs. For example:

```
improper access:
.john.modules.chart
```

If you have no permission whatever to the object, you instead receive the message:

```
not found: .john.modules.chart
```

Inquiring on All Aliases

To inquire on all aliases' access to paths, do not specify an alias in the command. For example:

```
share '' modules.chart
```

You see a message such as:

```
.john.modules.chart   aja              wrx
```

With no aliases specified, LOGOS displays information on all aliases with access to a particular path. For example:

```
.john.modules.chart   invendev          rx
  rohan                                cwrx
  wham                                  wrx
```

Granting Access

You can grant access with the *+permission* modifier to the *share* command. For example, to give *invendev* read access to the path *.john.modules.chart*, type:

```
share invendev modules.chart +permission=r
```

Read permission is now combined with the execute permission that *invendev* already had, to produce both levels of access:

```
.john.modules.chart            invendev   rx
```

Multiple Aliases and Multiple Paths

You can grant access to more than one alias, and to more than one path, simultaneously. For example, you can enter:

```
share 'aja wham' modules.util.rcat modules.chartsub
+permission=wrx
```

LOGOS displays a message such as:

```
.john.modules.chartsub        aja      wrx
                              wham     wrx
.john.modules.util.rcat       aja      wrx
                              wham     wrx
```

IMPORTANT: You must put quotes around the aliases to distinguish them from the pathnames. All names subsequent to the aliases are assumed to be pathnames; quotes are not required around them.

Granting Access to All of a Directory

The *share* command can be used on alias-level paths. Because the alias level is a directory, and because permission set on a directory is inherited by any subordinate directories you create under it, you can establish a global permission for an alias by setting its access.

To establish global permission, type:

share **alias .pathname** *+permission*=**permission**

For example:

share invendev .john +permission=wrx

You see a message such as:

```
.john  invendev      wrx
```

Now, *invendev* will automatically have write, read, and execute permission to any directories created below *.john*. Any directories created below these will inherit the identical permission from their parents. Permission is self-perpetuating as the hierarchy grows through the addition of directories and objects.

Granting Access to Part of a Directory

You can grant access from a particular point in the hierarchy through all nodes below with the *+recursive* modifier. It indicates that LOGOS is to change access not only to the directory given, but to all directories below it.

This way you can widow a portion of the hierarchy for a user. The user has access to a node but not to its parent. For example, if *wham* has some permission to the object *.john.modules.util.rcat* but none to its parent (*.john.modules.util*), *wham* will be unable to name the directory *.john.modules.util*, even though *wham* has access to one of its descendants. This behaviour makes unauthorized probing for paths more difficult.

■ To grant access to part of a directory, type:

share **alias pathname** *+permission*=**permission** *+recursive* **pathname**

For example:

share rohan modules +permission=cwrx +recursive

This command produces access information encountered anywhere along the specified portion of the hierarchy, with aliases appearing in alphabetical order. For example:

```
.john.modules         aja       wrx
                      invendev  rx
                      rohan     cwrx
                      wham      wrx
```

Creating a Public Path

One special alias, ⎕*all*, is a surrogate for all users enroled in LOGOS. If you grant ⎕*all* access to a path, then it becomes public to all users.

To create a public path, type:

share ⎕*all* **pathname** *+permission=***permission**

For example:

share ⎕*all modules.util.rcat +permission=x*

You see a message such as:

```
.john.modules.util.rcat    ⎕all      x
                           aja       wrx
                           rohan     cwrx
                           wham      wrx
```

Granting ⎕*all* access to a path does not remove any specific access permission for specific aliases.

Revoking Access

The *share* command with the *+delete* modifier revokes access. You can revoke:

- all access to a path from an alias
- certain access to a path from an alias
- all access to a path from all aliases

WARNING! The owner of a path has full permission to the path, by default. If you own a path, you can *lock yourself out* of it using the *share* command with the *+delete* modifier.

As the owner, you have control access and hence can extend to others the ability to regulate permission. Therefore, other users to whom you have granted control access can also restrict your permission.

If you or someone else inadvertently removes your access to a path you own, contact your system administrator for assistance. Your system administrator can help you gain temporary access to the path, so that you can restore your own permission.

Revoking Certain Access from an Alias

You can revoke certain access from an alias by specifying the access with the *+permission* modifier. Type:

share **alias path** *+delete +permission*=**permission**

For example:

share aja modules.chartsub +delete +permission=w

You see a message such as:

```
.john.modules.chartsub      aja      rx
                            rohan  cwrx
                            wham    wrx
```

Only write permission is removed from *aja*, leaving the alias with read and execute access.

Revoking All Access from an Alias

You can revoke all access from an alias by omitting the *+permission* modifier. Type:

share **alias pathname** *+delete*

For example, to remove all of *aja*'s access:

share aja modules.chartsub +delete

You see a message such as:

```
.john.modules.chartsub      rohan  cwrx
                            wham    wrx
```

Revoking Access from All Aliases

You can remove all access to a path from every alias using the *+delete* modifier. Type:

share **' ' pathname** *+delete*

For example:

share ' ' modules.chartsub +delete

Making Access Changes Take Effect

LOGOS maintains a certain amount of access information in the workspace during a session. As a result, to cause a permission change to take effect immediately in a session, you may need to issue the *exit +reset* command and then re-enter LOGOS by executing the *logos* function.

You can also achieve the same effect by typing:

alias (alias)

CHAPTER 4: LOGOS COMMAND LANGUAGE

Entering Commands

You enter LOGOS commands in response to the separator character prompt (ᵁ by default) or by passing them in a character vector argument to *∆logos.*

A LOGOS command is made up of several parts. As well as the command itself, it can include **arguments** and **modifiers.** An argument is a value you pass to a command. A modifier is a parameter that modifies the actions of the command. All modifiers begin with a +. Arguments and modifiers can also take values.

This is an example of a typical LOGOS command:

```
list util.formatting +full +recursive=1
```

The command being invoked is *list.* One argument, the string *util.formatting,* is being passed to it. Two modifiers, *+full* and *+recursive,* have been specified; the *+recursive* modifier is assigned a value of 1.

Abbreviating Commands

Although commands are spelled out in full in this manual, you can abbreviate them. For example, you can use *bu* for *build* and *l* for *list.*

For each command, there is a predefined minimum number of characters that you must enter for the name to be recognized. These minimum lengths were chosen so that frequently used commands require fewer keystrokes, and potentially destructive commands require more. So, while *d* is as good as *display,* you must spell *delete* out in full.

For the minimum length required for each command name, use the *?* command to list commands and their minimum abbreviations, or see the command descriptions in the *LOGOS Reference Manual.*

Entering Multiple Commands

You can enter more than one LOGOS command on a single line by delimiting commands with the LOGOS command separator character.

For example:

```
display parse.scan ᵁ xref parse.scan
```

NOTE: You cannot use the following characters to delimit commands:

- alphabetic (including ∆, ⍙, ⍺, ⍵)
- numeric
- any of the reserved characters ( ) + { } ' \ ∇ . [ ] − ↑ ? ← ⍋ ⍒ =
- blank spaces

■ To include the separator character in an argument, enclose it in quotes. For example:

replace '∪'*name* '∪'*names*

■ If the separator character appears in several places, you can specify a different separator character for a particular line by entering a new separator as the first character of the line. For example:

∘ *replace* ∪*name* ∪*names parse.scan* ∘ *display parse.scan*

Here, ∪ is the separator character displayed by LOGOS, and the jot (∘) is the new separator character specified by the user and then used as a delimiter. This command line first replaces the string ∪*name* by ∪*names* within the object *parse.scan*, and then displays the changed object.

Using Commands with Arguments

Most commands accept one or more arguments, separated by blanks. The *locate* command, for example, accepts two arguments. For example:

locate lr↓names util.formatting

The first argument is the search string *lr↓names*, and the second is the pathname *util.formatting*.

Using Quotes in Arguments

LOGOS generally treats quotes in much the same way that APL does. The exception occurs when they are part of a regular expression (a pattern used for searching). For more information on regular expressions, see *Chapter 12: Software Development Tools*, or *Appendix A: Using Regular Expressions*.

■ To provide an argument that contains blanks (for example, a phrase or a list of names), enclose it in quotes. For example:

locate '*1 0 ↓names*' *util.formatting*

Here, the first argument to the *locate* command is the string *1 0 ↓names*, and the second is the pathname *utilformatting*. The APL quotes around the first argument are removed by LOGOS; they are not included in the string passed to the *locate* command. Therefore, in this example, the first character that *locate* tries to find is the character 1, not the character '.

- To include the quote character itself in an argument, enclose the whole argument in quotes and double the embedded quotes. For example:

  locate 'don''t split' .utilformatting

NOTE: You must enclose the following reserved characters in quotes when you provide them in an argument:

```
( ) + { } ' \
```

You must also enclose blanks in quotes.

Argument Scope

Argument scope is a characteristic of every LOGOS command or script. It influences how a command identifies its arguments. A command can have **short scope**, **long scope**, or **unprocessed scope**.

Short Scope

A command with short scope takes a fixed number of arguments and recognizes all unescaped blanks as argument delimiters. It displays an error message if you provide too many arguments.

For example, the *link* command requires exactly two arguments. For example:

link ut .public.util

If the *link* command is issued with more than two arguments, you get an error. For example:

link a b c

The system returns the message:

```
too many arguments
  link a b c
        ^
```

Long Scope

A command with long scope allows its last argument to be extended up to the first modifier, and recognizes blanks as argument delimiters only until it has scanned the expected number of arguments. If additional blank-delimited fields appear at the end of the argument list, they are considered part of the last argument. Most LOGOS commands that work on a list of pathnames position this list as their last or only argument, and have long argument scope.

The *locate* command, for example, takes two arguments and allows the last one to be extended. Because of this, the command:

```
locate names util.formatting parse.scan
```

is equivalent to:

```
locate names 'util.formatting parse.scan'
```

NOTE: When a command has long scope, you do not have to enclose blanks in quotes if they appear as part of the last argument.

Unprocessed Scope

A command with unprocessed scope takes only one argument and does not recognize any special characters other than the command separator (quotes, parentheses, the \ that precedes keywords, or + that precedes modifier names). Therefore, you do not have to enclose reserved characters in quotes.

Two commands have *unprocessed argument scope:* ⍎ and ⍕.

So, the following APL expressions using the ⍎ command work as you would expect:

```
∪ ⍎⍴'hello'
5
∪ ⍎ 1 0 1 0 1\'abc'
a b c
```

All quotes and other special characters are passed directly to APL, without interference from LOGOS. The ⍕ command has this same property, but for a different reason. See the description of the ⍕ command in the next section.

Entering Complex Arguments

The ⍕ command makes it easier to enter complex arguments containing quotes and other metacharacters. The command actually does nothing but return its argument as its result; it is useful because of its unprocessed scope.

Suppose you want to locate occurrences of an APL expression like the one below, which contains several special characters such as parentheses and quotes:

```
'totals: ',(⍕m\v+(n×2)-1),' ytd'
```

The following approach would serve adequately:

```
locate '''totals: '',(⍕m\v+(n×2)-1),'' ytd''' ?⋆
```

But, using the ⍕ command, you can simplify the command as follows:

```
locate (⍕'totals: ',(⍕m\v+(n×2)-1),' ytd') ?*
```

By writing the line using the ⍕ command, you avoid having to double the quotes within the expression.

Using Commands with Modifiers

LOGOS commands also accept **modifiers**, which are parameters that modify a command's action. Modifiers are denoted by + followed by the modifier's name, as in *+recursive*. For example:

```
locate util +recursive
```

A modifier can also specify a value, as in *+flags=s*. For example:

```
locate util +recursive +flags=s
```

Abbreviating Modifiers

In this book, all modifier names are spelled out in full. The modifier names for any particular LOGOS command all have unique first letters, so that you can always abbreviate to one letter. For example:

```
locate util +r +f=s
```

NOTE: Scripts, which are user-defined commands, can have modifiers with non-unique initial letters. In such cases, the modifier names for the script can be abbreviated to the minimum required to distinguish the name. For more information on scripts, see *Chapter 5: Using Scripts.*

Controlling Command Output

LOGOS commands produce two kinds of output:

- **result output**, which is essential information generated by a command
- **message output**, which is ancillary information such as an error message

To pass the result of one LOGOS command as an argument to another command, enclose the command producing the result in parentheses. For example:

```
edit (locate names utilformatting)
```

The *locate* command searches for the string *names* in the *utilformatting* directory. The result output of the command, a list of pathnames of the objects in which *names* is found, is passed to *edit* as its argument.

The result output of a command can be used anywhere in a command line; for example, as the value of a modifier:

```
build budgetpage +exclude=( list budgets.excp )
```

Here, the pathname result of the *list* command is passed as a value to the *+exclude* modifier of *build.*

An expression in parentheses can be made up of several LOGOS commands, delimited by the command separator character. If the first character inside the parentheses is a valid command separator, it is used as the separator just for the expression in parentheses. Otherwise, the normal command separator is used. The result of the expression in parentheses is the result of the last command in the expression.

All or part of a command (including the command name) can be generated by an expression:

```
∪ ⍎cmds←'workdir ∪ list ?* +column +recursive'
∪ ( ⍎cmds )
```

Capturing Output

To capture the result output of a command, assign it to a variable. For example:

```
p←locate names utilformatting
```

In this example, the list of pathnames produced by the *locate* command is placed in the APL variable *p*. *p* is an ordinary variable and remains in the workspace when LOGOS ends.

To capture message output, assign it to an **empty modifier**, +=. The variable to which you want the output assigned is specified by the value of the modifier. For example:

```
locate names utilformatting +=msg
```

Any messages produced by the *locate* command are assigned to the variable *msg*.

Displaying Captured Output

As in normal APL assignment, the result of a command is not displayed if it is assigned to a variable or used within an expression.

To display result output, use ⎕←. For example:

```
edit (⎕←locate names utilformatting)
```

Here, the output of the *locate* command is displayed and passed as the argument to *edit*.

Some commands, like *delete* and *retain*, return results but don't display them by default. You can also use ⎕← to see the results of these commands.

To display message output, specify ⎕ as the value to the += modifier.

This causes messages to be displayed at the terminal. (This is the default.)

Discarding Output

To discard result output, assign it to an empty variable name. For example:

←environment +profile

To discard message output, use an empty modifier with an empty value. For example:

list q.cell +=

Using Keywords

Keywords allow you to store phrases, so that you can call long commands by typing only a few keystrokes. For example, you might find yourself frequently typing:

list .mde.sys.util.tools

You can create a keyword that you type instead of *.mde.sys.util.tools*. For example, if you create the keyword *tools* to represent *.mde.sys.util.tools*, you could then type:

list \tools

NOTE: To use a keyword, you must precede the keyword with a \.

You create and display keywords with the *keyword* command, or with the *environment keyword* construct. See *Chapter 12: Software Development Tools* for more information on environment keywords.

IMPORTANT: Keywords may not be used inside a script. The construct **\name** has a different meaning inside of a script. For more information, see *Chapter 5: Using Scripts.*

Abbreviating Keywords

Like modifier names, keywords can be shortened to any abbreviation which is unique. Depending upon what other keywords you have defined, you may be able to use *\tool*, *\too*, *\to*, or even *\t* to reference the keyword *\tools*.

Defining Keywords

Define keywords with the *keyword* command. Provide the keyword you want to define, and the phrase to be the definition of the keyword. The maximum length of the phrase that defines a keyword is 500 characters.

For example:

```
keyword tools .mde.sys.util.tools
```

Saving Keywords

To make your keywords permanent, you must save them in your profile. To do so, type:

```
environment keyword +profile
```

Displaying Keywords

If you use the *keyword* command without arguments, it simply returns the names of all your keywords. To display all your keywords, type:

```
keyword
```

The system returns a list of your keywords. For example:

```
box cmds tools util
```

To display the definition of one or more keywords, specify the keywords. For example:

```
keyword util
```

The definitions of these keywords are returned. For example:

```
keyword util '.mabra.tools.util'
```

To see the definition of all of your keywords, type:

```
keyword (keyword)
```

You see a result such as:

```
keyword   box    'exit □load ''666 box'''
keyword   cmds   '.proj.tools.cmds'
keyword   tools  '.mde.sys.util.tools'
keyword   util   '.mabra.tools.util'
```

To emphasize that any quotes within the keyword are actually escaped, the result always shows the keyword definition in quotes.

If a second argument is supplied, it is interpreted as the phrase to be associated with the keyword. For example:

```
keyword sced 'edit +command=settype s +Names='
```

Now when you type *keyword*, you get the message:

box cmds sced tools util

Notice that the keyword *sced* uses the argument to *edit*, which is *+Names*, as if it were a modifier. This allows the argument to the keyword (*parse.scan* below) to become the argument to *edit:*

sced parse.scan

Use *environment keyword +profile* to store in your profile any keywords you have defined.

Deleting Keywords

To delete a keyword, specify an empty value as the definition for the keyword (the second argument). For example:

keyword cmds ' '

You see a message such as:

1 keyword deleted

Applying a Sequence of Commands to a List of Arguments

The *with* command applies a sequence of commands to a list of arguments in an itemwise manner.

The *with* command takes two arguments: an expression to evaluate, and an argument list upon which to apply the expression. The expression is a LOGOS command line, which can contain multiple commands. It can also contain one or more occurrences of the argument substitution character, α. The argument list is an arbitrary series of values, often a collection of names separated by blanks.

To use the *with* command, type:

with **'command ᵤ command' argument argument**

The expression is executed once for each argument in the list. Each time the expression is executed, all occurrences of α are replaced by the next argument in the list.

For example, suppose you want to list, summarize, and display two objects. You could enter six separate commands, or you could enter one *with* statement:

with '*list* α *+long* ᵤ *summarize* α ᵤ *display* α' *faccess mfsort*

This is equivalent to the command sequence:

```
∪ list faccess +long
∪ summarize faccess
∪ display faccess

∪ list mfsort +long
∪ summarize mfsort
∪ display mfsort
```

Specifying a Substitution Character

If the expression itself contains ⍺, you can specify an alternate substitution character using the *+surrogate* modifier to the command. For example, if ⎕*sp* is a series of names, you can locate ⍺ in each name, and you can edit those names in which a match was detected using the statement:

```
with 'edit (locate ⍺ ⋆)' (⍋⎕sp) +surrogate=⋆
```

NOTE: You cannot use the following characters as surrogates:

- numeric
- alphabetic (including ∆, ⍙)
- any of the reserved characters ( ) + { } ' \ ∇ . [ ] ↑ ? ← ⍋ ⍒ = ⎕
- blank spaces

CHAPTER 5: USING SCRIPTS

Figures

Tables

What is a Script?

A **script** is a user-written LOGOS command that can take arguments and produce output, just like a native command. Internally, a script is a sequence of APL expressions and LOGOS commands in the same program.

You can use a script anywhere that you can use a LOGOS command. Ancillary functions or variables required by the script can be merged into it to create a self-contained module.

As with other LOGOS objects, scripts can be shared. You can save commonly used scripts in public directories, and the members of a programming team can create project tool libraries for their use or for general application.

Calling Scripts

A script is a LOGOS object, and is therefore referenced by its pathname. To call a script, type the pathname of the script.

For example, to call the script *.public.logos.cmds.search* with the argument *util*, type:

```
.public.logos.cmds.search  util
```

Because a script has the same properties as a native LOGOS command, it can also be used as part of a larger expression:

```
locate '⎕fmt' (.public.logos.cmds.search  mergers)
```

Scripts and Commands with the Same Name

Native LOGOS commands take precedence over scripts, to ensure that LOGOS commands behave identically for every user. If you have a script called *list*, you cannot call it by entering *list* because it will invoke the LOGOS *list* command. You can call the script by supplying its pathname. For example:

```
.mde.scripts.list
```

If a script is in one of your command directories (described in the section, *Using Command Directories*), you can distinguish the script from a LOGOS command by typing a dot after the name. For example, if one of your command directories is *.mde.scripts*, you can type:

```
list.
```

Another alternative is to store the full pathname of the script in a keyword called \list, and then reference it via \list or even \l. For more information about keywords, see *Chapter 4: LOGOS Command Language*.

Predefined Scripts

A number of scripts are already available in the LOGOS utility library. This is a central location for objects that can be useful to many different programmers working on many different applications. All users have access to the utility library. It contains a directory called *.public.logos.cmds,* which contains scripts that extend or supplement the facilities of native LOGOS commands. Subroutines and auxiliary objects for use by the scripts are in the directory *.public.logos.cmds.util.*

To list the directory *.public.logos.cmds,* type:

list .public.logos.cmds

For more information on the utility library, see *Chapter 15: Using the Utility Library.*

Online Help for Scripts

Online help is available for scripts, whether they are predefined or user-defined. To display a summary of a script's syntax, type:

*?***script**

For example:

?clear

■ To display more detailed help for a script, type:

*??***script**

NOTE: This command is actually a shorthand for the command:

display **script**[:*d*]

The help message displays whatever is stored in the documentation attribute of the script. For more information on creating script documentation, see the section, *Creating Script Documentation,* later in this chapter.

Using Command Directories

To make scripts easier to call, LOGOS maintains a **command directory** which the command processor uses when searching for scripts. If your command directory is set to *.public.logos.cmds,* you can call the script *.public.logos.cmds.search* with an argument of *util* using the command:

search util

This also simplifies commands when you are using a command as part of a larger expression. For example, rather than typing:

```
locate ' ⎕fmt' (.public.logos.cmds.search  mergers)
```

You can type:

```
locate ' ⎕fmt' (search  mergers)
```

Inquiring about Your Command Directories

Command directories are set and displayed with the *cmddir* command, which works just like the *workdir* command. When you first use LOGOS, your command directory is set to *.public.logos.cmds.*

To inquire about your command directory, type:

```
cmddir
```

The system returns the message:

```
.public.logos.cmds
```

This is the public LOGOS directory that contains many useful scripts. You will probably want to keep this directory in your list of command directories.

Adding Command Directories

You can add directories to your list of default command directories. LOGOS searches command directories in the order in which you specify them. Therefore, you can add command directories before or after the current command directories.

To add a directory before the current command directory, type:

cmddir **pathname** (*cmddir*)

For example:

```
cmddir .mde.scripts (cmddir)
```

The *cmddir* command returns the new setting. For example:

```
.mde.scripts .public.logos.cmds
```

To add a directory after the current command directory, type:

cmddir (*cmddir*) **pathname**

For example:

```
cmddir (cmddir) .mde.scripts
```

If the current command directory already contains the path that you are adding, the duplicate pathname is removed. Thus, by placing an existing command directory path in front of the (*cmddir*) command, you can change the order of existing pathnames.

NOTE: If only one or two scripts in another user's directory are of interest, you may not want to add the entire directory to your *cmddir* list. If you have read access to a script, you can simply copy it into your private command directory.

Alternatively, you can establish a link in your command directory, which points at the desired script. This method takes less space and ensures that you immediately realize the benefits of any updates made by the owner of the script. For details of this method, see the description of the *link* command in the section, *Creating Links*, in *Chapter 3: Using the File System.*

Saving Command Directories

If you add directories to your list of command directories, you can save them into your profile. Type:

environment cmddir +profile

Creating a Script

You can create a script with either the *edit* command or the *save* command.

Usually, the *edit* command is a more convenient way to create a script. However, one advantage of the *save* command is that it lets you build and save a script under program control. For example, you can write a script that calls a different editor to prepare the source, and then saves this text as a script in the file system.

Creating Scripts with the *edit* Command

You can create a script with the *edit* command by providing an extended pathname indicating that the object type is *script*. For example:

edit .mabra.tools.dispn[*s*]

If you do not specify [*s*], the editor assumes you want to open a new function.

You can also create a script by entering the editor, allowing the editor to assume that you are opening a new function, and then, while in the editor, changing the object type to *script* by typing:

settype s

See *Chapter 6: Using the Editor* for a complete discussion of the editor.

Creating Scripts with the *save* Command

The *save* command expects the source of the script to be a character vector, with lines delimited by carriage returns. The script image can contain line numbers, but should not include opening or closing ∇'s.

The example below shows how the *save* command can be used to create a script. The *s* type specifier is included in the pathname argument, and the character vector value is passed via the *+value* modifier:

```
∪ ⍎source
dispn +Path=
)⎕←display \Path +nopathname
∪ save .mabra.tools.dispn[s]  +v=⍎source
.mabra.tools.dispn[s1]
```

Creating Script Documentation

To make more detailed help available for a script, you must write a help message. Open the documentation attribute of the script using the editor, write the help message, and exit the editor, saving the attribute.

For example, to provide a help message for the script *.mabra.tools.dispn*, write the documentation and save it as *.mabra.tools.dispn*[:*d*] .

Looking at a Simple Script

A script resembles an APL user-defined function. It has a header line and a body consisting of all remaining lines. Unlike a function, however, a script's header is line 1. The body comprises lines 2 onward, and each line is either an APL expression or a LOGOS command. A LOGOS command line in a script begins with a). For example:

```
[1] listfetch;⎕pw
[2] ⎕pw←79
[3] )⎕←list .john.emulator.fetches +column
```

The header of this script, line 1, names the script as *listfetch* and declares ⎕*pw* as local. Line 2 is an APL expression, and line 3 is a LOGOS command.

You can include several LOGOS commands on a single line of a script, if the commands are delimited by a separator. Place the separator character for the line immediately after the). Here is an example showing the use of ∘ to separate two commands on a line:

```
)∘ z←list .mabra.tools ∘ edit (⍎z)
```

The blanks around ∘ are not required; they are included here for clarity.

If you intend to use several LOGOS commands on a single script line, it is a good idea to specify a separator character at the beginning of the line (assuming the default separator character, ∪, will cause problems if the user has changed to some other character).

Executing *listfetch* produces a listing on the terminal of the contents of directory *.john.emulator.fetches*, in columnar format with an effective print width of 79.

NOTE: The ⎕ on line 3 is necessary because, by default, commands do not display their results when invoked from within a script.

The utility of the script *listfetch* is severely limited by the fact that it executes *list* on a constant pathname. Furthermore, the output generated by *list* within the script is always displayed on the terminal; it cannot be captured and processed by another command.

Defining Arguments and Results

This simple script can be made much more useful by defining an argument and a result. The first letter of an argument name must be in the second alphabet. (Script arguments and results are described in detail in the sections *Using Arguments and Modifiers inside Scripts* and *Displaying Script Output* later in this chapter.)

For example:

```
[1] z←listc +Pathname=;⎕pw
[2] ⎕pw←79
[3] )z←list \Pathname +column
```

The header now specifies that the script is named *listc*, and that it takes a single argument, *Pathname*. Line 3 interpolates the value of this argument using the expression *\Pathname*. At first sight this might appear to be a keyword. However, a \ followed by a name within a script refers to an argument to a script. Therefore that argument replaces the backslash expression on the line.

Now, you can use *listc* to produce a listing of the contents of any directory:

```
listc reports.yearly
```

Furthermore, since the header designates a formal result, the result of the script can now be captured and processed:

```
d←listc reports.yearly
⍎d ⎕append 1
```

or:

```
edit (listc emulator.stores)
```

Because *listc* was defined to take an argument, attempting to call it without one will produce an error. For example:

```
listc
```

The system returns the message:

```
<Pathname> argument must be specified
listc
^
```

The *Pathname* argument in the *listc* script can be made optional by enclosing it in parentheses; that is, (*+Pathname=*). A default value for the argument, to be used when the script is called without one can also be specified. For example:

```
[1] z←listc (+Pathname=reports.yearly) ;⎕pw
[2] ⎕pw←79
[3] )z←list \Pathname +column
```

Now, *listc* can be called with or without an argument. When called without one, the script behaves as if you had typed *listc reports.yearly*.

Specifying Arguments and Modifiers in a Script Definition

A script can have any number of arguments or modifiers. They may be required or optional. You can specify default values to be supplied when no value is provided by the user. LOGOS validates the use of the script according to the syntax stated in the script's header. Accordingly, you must adhere to certain rules when constructing a script that includes arguments or modifiers.

Table 5.1 Rules for Specifying Arguments and Modifiers in a Script Definition

Arguments	Modifiers
The first letter of an argument name must be in the second alphabet, as in *Pathname*.	The first letter of a modifier name must be in the first alphabet, as in *pathname*.
Arguments must be specified in a particular sequence and before modifiers.	Modifiers do not have to be specified in any particular order, but must be after the last argument.

For consistency in the header, both arguments and modifiers are preceded with a + before their name, as shown in the previous example for the *listc* script.

There are several different forms that you can use in the header to specify arguments or modifiers when you are writing a script. These forms are summarized in the table below, where the generic term, **parameter**, is used to signify either an argument or a modifier. The form you select determines such things as whether the parameter is optional or required, whether the user can supply a value for the parameter, and what the default value is, if any.

Table 5.2 Forms of Script Parameters

Declaration	Description
+parameter=	The parameter and its value must be specified.
+parameter(=)	The parameter must be specified. Its value is optional.
+parameter(=dv)†	The parameter must be specified. Its value is optional and if no value is specified, its default value is **dv**.
(+parameter)	The parameter is optional. No value is allowed.
(+parameter=)	The parameter is optional. A value is optional.
(+parameter)=	The parameter is optional. A value must be provided if the parameter is specified.
(+parameter=dv)	The parameter and its value are optional, and its default value is **dv**.

† This form of parameter is not fully implemented. While you may specify an argument or modifier with this form, the resultant behaviour is identical to the form **+parameter(=)**.

The = in the previous table indicates that the parameter can accept a value. Arguments always take values, but there is one form of specification for modifiers that does not take a value (**+parameter**).

All of these forms are useful for specifying modifiers. Arguments, however, are most often specified with one of the following forms:

+parameter=

(+parameter=)

(+parameter=dv)

Using Arguments and Modifiers inside Scripts

As noted earlier, a script can contain lines that are executed as LOGOS commands, as well as lines that are executed as APL expressions. Inside a script, there are a variety of ways in which arguments and modifiers can be referenced depending on the information you need to extract in each type of script line.

Arguments and Modifiers in APL Expressions

When a user calls a script, LOGOS defines a local variable of the same name as each argument or modifier that is in the script header. Because these are normal APL local variables, they can be referenced in APL expressions inside the script.

The value that each of these variables will have depends on several factors:

- the form used to specify the parameter in the script header
- the presence or absence of that parameter in the command which called the script
- the value supplied, if appropriate, for a parameter

The following table summarizes the rules by which LOGOS assigns values to these local variables inside the script.

Table 5.3 Rules for Assigning Values to Local Variables in a Script

If:	Then:
an argument or modifier is optional, and the user does not specify it,	the corresponding APL variable is set to a Boolean 0.
a modifier does not take a value, or takes an optional value, and the user specifies the modifier but does not supply a value,	the corresponding APL variable is set to a Boolean 1.
the user supplies an argument, or provides a value for a modifier,	the corresponding APL variable is a character vector of the value specified by the user.
the parameter specification in the script header provides a default value, and the user does not specify that parameter,	the APL variable is set to the default value specified in the header.

Note how the first two cases provide for a natural logic control inside the script through the use of the Boolean values to control branching, typically accomplished by specifying valueless modifiers in the script header.

For example, the following script displays an object and then optionally displays a cross-reference of the same object, based on the presence of the *+xref* modifier.

```
[1] dx +Pathname= (+xref)
[2] )⎕←display \Pathname ⍝ display object
[3] →xref↓0 ⍝ exit if <+xref> not specified
[4] )⎕←xref \Pathname ⍝ cross reference object
```

Note the simplicity of the branching logic on line 3. In this case, you are interested only in whether or not the user has requested a cross-reference, so there is no need for the user to specify a value for that parameter. Indeed, the form of the specification in the script header does not allow the user to provide a value for the *xref* modifier.

Arguments and Modifiers in LOGOS Commands

Two different situations arise when you refer to arguments or modifiers in a script line that is a LOGOS command.

- You may want to use the actual value of the parameter provided by the user, for example if they have provided a pathname.
- You may want to pass on the modifier and the value supplied as a modifier and value specification to the LOGOS command being executed in that line.

To support these two requirements, LOGOS supports two special reference constructs inside scripts: **\parameter** and **\+parameter**.

The following modification of the *listc* script illustrates the use of these two constructs. The *list* command accepts an optional *+full* modifier that causes *list* to return the full pathnames of each object listed. Let's add the *+full* modifier to the script:

```
[1] z←listc (+Pathname=) (+full) ;⎕pw
[2] ⎕pw←79
[3] )z←list \Pathname +column \+f
```

Note that:

- there are parentheses around *+full* in the header so that the modifier is optional
- there is no = after so that it can't accept a value
- the reference to *full* in the \+ construct is abbreviated to *f* (*f* is enough information to uniquely identify the parameter in this script)

Now the following behaviour applies to the *listc* script:

Table 5.4 *listc* Script Behaviours

If:	Then:
listc is invoked without *+full,*	the \+*f* construct on line 3 will be replaced by an empty string (that is, it will disappear) when the line is evaluated.
listc is invoked with *+full,*	\+*f* will be replaced by *+full* when the line is evaluated.

When the command processor encounters the construct \+**name** in a LOGOS command within a script, it chooses one of the replacement rules in the following table.

Table 5.5 Replacement Rules for \+ in a Script

If:	Then:
the parameter +**name** is not specified in the line that called the script,	\+**name** is replaced by an empty string (it is removed).
the parameter +**name** is specified without a value,	the string +**name** is interpolated.
the parameter +**name** is specified with a value,	the string +**name=value** is interpolated.

The LOGOS command processor follows a similar set of rules when it encounters the **parameter** construct in a script command line. The rules appear in the following table. .

Table 5.6 Replacement Rules for \ in a Script

If:	Then:
the parameter +**name** is not specified on the line that called the script,	**name** is replaced with an empty string (it is removed).
the parameter +**name** is specified without a value,	**name** is replaced with an empty string (it is removed).
the parameter +**name** is specified with a value,	**name** is replaced with a character vector of the value.

This construct enables you to pass the argument or modifier values to a LOGOS command automatically, without having to verify that a value was entered. (You may wish to examine the APL variable to determine if the user's value is a valid one for your application.)

NOTE: The \ construct can be used only on names defined as parameters in the script header. To interpolate the value of an ordinary APL variable into a command line, use the ⍎ command. For example, to interpolate the value of the variable *dir*, which is not a script parameter, into a *list* command, you could type:

```
[2] )z←list (⍎dir) +column
```

Script Argument Scope

Like commands, scripts also have argument scope. The scope is specified by including a bracketed flag immediately after the script's name in the header. Recall that there are three kinds of command argument scope: **short** (the default), **long**, and **unprocessed**. The corresponding scope flags are as follows:

Table 5.7 Scope Flags

Flag	Meaning
s	Short scope (default)
l	Long scope
u	Unprocessed scope

For more information on the types of scope, see the section, *Argument Scope,* in *Chapter 4: LOGOS Command Language*.

Advantages of Long Scope

Most scripts you write will have short or long scope. Long scope has the advantage of allowing you to pass a list which contains blanks as the last argument of your script. This might be useful, for example, in a script that takes a list of pathnames as one of its arguments. By assigning the script long scope and making the pathname argument the last one, you can avoid the need to put quotes around the list.

To allow *listc* script to take an arbitrary number of paths as its argument, change the header to specify long scope:

```
[1] z←listc[l] +Pathnames= (+summary)
```

Now you can invoke *listc* with a list of pathnames without resorting to quotes. If you gave *listc* short scope and attempted this, you would get the error *too many arguments*.

Pros and Cons of Unprocessed Scope

If you assign a script unprocessed scope, it will not recognize any special characters in its argument. The script:

- can have only one argument (the blanks that delimit arguments won't be recognized)
- cannot have modifiers (the + won't be recognized)
- cannot be called using keywords or evaluated arguments (the symbols \, and (or) will be ignored). Unprocessed scope has the advantage of making it convenient to enter arguments for commands or scripts which routinely contain LOGOS metacharacters that you might not want interpreted.

For example, the script *rx* aids in the preparation of regular expressions on terminals that require several keystrokes to enter a left or right brace. It is a simple filter which removes any leading blank from its argument, encloses the remainder of the argument in braces, and returns the string as the result. Because a regular expression might contain characters to which LOGOS is sensitive (+ or blank, for example), *rx* has unprocessed scope.

```
[1] z←rx[u] +Arg=
[2] z←'{',((' '∈1↑Arg)↓Arg),'}'
```

Using *rx*, you can enter:

```
replace (rx . +⊣) .⊣ ?⋆
```

rather than:

```
replace {. +⊣} .⊣ ?⋆
```

Note that the first expression is slightly longer, but that it will be easier to enter on certain terminals.

Displaying Script Output

Scripts can produce output. They generate:

- error message output
- message output
- a quadprime prompt
- status line output
- result output

You can display the various classes of output using the *output* command. This command takes a text as its argument, and uses modifiers to determine what type of output to produce.

Displaying Error Message Output

The *+error* modifier provides you with the ability to signal an error from within a script. To produce error message output, specify in your script:

)*output* **text** *+error*

For example:

```
[2] )output cannot define: (⍎⎕er[⎕io+1;]) +error
```

The operation of this modifier is similar to the APL system function ⎕*signal*. *output +error* causes LOGOS to abandon execution, exit the script, and display the specified message along with the line invoking the abandoned script.

Displaying Message Output

The *+message* modifier causes the text to be displayed as standard message class output. To produce message output, specify in your script:

)*output text +message*

For example:

```
[2] )output processing complete +message
```

This is the default class if you do not specify a modifier.

Displaying a Quadprime Prompt

The *+quadprime* modifier causes output to be generated as if it were a quadprime (⍞) prompt. To produce a quadprime prompt, specify in your script:

```
)output text +quadprime
```

The next piece of output displayed will appear on the same line. For example:

```
[2] )output 'the answer is: ' +quadprime
[3] )output thirty six
```

generates the output:

```
the answer is: thirty six
```

Displaying Status Line Output

The *+status* modifier causes the text to be displayed on the status line of the terminal device if *environment status* is set to *full*. To produce status line output, specify in your script:

```
)output text +status
```

For example:

```
[2] )output (⍎'now processing ',name) +status
```

The message in the status line will be displayed until another status line message overwrites it or until script execution is finished.

If the current device does not support a status line, no output is generated.

Displaying Result Output

The *+result* modifier allows you to generate buffered result output. To produce result output, specify in your script:

```
)output text +result
```

For example:

```
[2] )output (⍎rs) +result
```

If a script uses several *output +result* calls, the actual result of the script is the catenation of all calls.

You can also generate result output by declaring a formal result in the script's header, as described in the preceding section.

NOTE: These techniques are mutually exclusive. You cannot use *output +result* in a script whose header includes a formal result variable.

There are advantages to each of the techniques for generating script result output. The *output +result* technique is useful with scripts that produce a stream of output in a loop. This is because output can be displayed in segments, each time the *output* command is invoked. When a result variable is used, no output is displayed until the script exits and returns a result.

The formal result method is more efficient and is particularly appropriate with scripts that produce a single piece of result output. Also, the value returned by a script using this technique does not have to be a character vector. Higher dimension arrays, numeric data, packages, and enclosed arrays are all allowed. LOGOS converts these values to character vectors for you automatically. The converted values are the display image of the original values. In particular, a package is converted to:

package

Using Composite Scripts

You may often need to call a user-defined function from a line of APL code within a script. If the function is not already in your active workspace, you can use the *get* command from within the script to fetch the required functions or variables. (The *get* command is described in detail in *Chapter 8*.) You can even localize the names of the fetched functions and variables to protect the outer environment.

A simpler way to make a user-defined function available for use in a line of APL code within a script is to localize its pathname in the header of a script, just as you localize simple names in the header of an ordinary APL function.

Scripts that have pathnames localized in their headers are called **composite scripts**. When the script is called, each object specified by a pathname in its header is fetched and defined as a local object within the script. This happens automatically and invisibly to the caller.

Sample Composite Script

The following composite script illustrates how to call the functions *sqz* and *vtom*, and use the variable *CR*. All of these objects are in the *.public.util* directory.

```
[1] z←tseg +Pathnames=;.public.util.sqz|vtom|CR
[2] ⍝ return terminal segment of pathnames
[3] z←(' ',CR) vtom Pathnames
[4] z←sqz (,1,⌽∧\⌽z≠'.')/,' ',z
```

The alternation character (|) is used in the pathname to avoid typing the directory name three times. The pathname pattern matches the pathnames:

- *.public.util.sqz*
- *.public.util.vtom*
- *.public.util.CR*

For more details on regular expressions, see *Chapter 12: Software Development Tools,* or *Appendix A: Using Regular Expressions.*

The pathnames in the header can refer to functions, variables, or clusters. The use of regular expressions makes it possible to include an entire directory of objects within a script. These objects are truly local to the script. They become undefined or assume their global definitions (if any) when the script completes execution.

Pathnames in a script's header are fetched only the first time a new or modified script is used. The composite object formed by the script and its local objects is saved for future use. It is then no longer necessary for LOGOS to fetch the local objects when the script is executed; they are an integral part of the script. If the script is edited and then invoked, the objects are fetched again and the composite script is rebuilt.

Note that the definitions of local objects are frozen when the script is compiled. If you later edit one of the pathnames referenced in the header, the new definition of this object is not reflected in the composite script until the script itself is changed.

Shortening Pathnames in a Script Header

You can shorten the pathnames you include in the script's header by associating a *workdir* (*w*) compilation directive with the script.

The argument to *w* is a set of working directories to be used when fetching the objects referenced in the header. The compilation directive is set using the *save* command. (For more information on compilation directives, see *Chapter 10: Using the Compiler.*)

For example, if the pathname of the script is *.mabra.tools.tseg*, the directive is saved as follows:

```
save .mabra.tools.tseg[:c] +value=w=.public.util
```

The header of this script can then be shortened to:

```
[1] z←tseg +Pathnames=;sqz.;vtom.;CR.
```

or:

```
[1] z←tseg +Pathnames=;sqz|vtom|CR.
```

The trailing dots in the locals list are required to distinguish the pathnames from ordinary locals.

Using Clusters in Scripts

By including the pathname of a cluster in the locals list of a script, you can take advantage of the calling tree analysis features of LOGOS.

The *build* command accepts a script as the root of a potential cluster. It performs an analysis of the script, and generates a calling tree as it would with an ordinary function. The script itself is excluded from the cluster that is constructed.

To see how this works, suppose you want to write a script that is a cover for a function called *avam*. This function references several other functions and variables, all of which reside in the directory *.rhl.sys.avam*. The script itself is called *.rhl.scripts.avam* and might look something like this:

```
[1] avam +Argument=;.rhl.sys.avamcluster
[2] avam Argument
```

The cluster *avamcluster* can be constructed using the *build* command:

```
build .rhl.sys.avamcluster .rhl.scripts.avam +depth=all
+workdir=.rhl.sys.avam
```

The resulting cluster contains all of the objects required by the *avam* script. The script automatically localizes these objects and defines them when it is invoked. For more information on clusters and the *build* command, see *Chapter 8: Building Applications with LOGOS.*

Controlling Local Environments

Scripts often need to change some aspect of the LOGOS environment temporarily. The environment includes your:

- working directories
- command directories
- separator character
- status area control
- other dynamic properties of the session

For example, a script that focuses on a certain set of working directories may want to pre-empt the caller's working directories for the duration of its execution. If it runs a lot of private commands from the same directory, it might want to establish its own local command directories. If it issues a lot of LOGOS commands, particularly in a loop, having the status line refreshed for each command might be tedious; disabling the status area on a screen display can avoid unnecessary and time-consuming output.

A script can achieve its own local environment by storing any values it plans to change. For example:

```
[1] buildfile (+Dir=.john.modules.chartsub) ;wkdir
[2] )wkdir←workdir ⍝ retain present value
[3] )workdir \Dir ⍝ set new value
[.]  .     .
[.]  .     .
[.]  .     .
[n] )workdir (⍎wkdir) ⍝ restore previous value
```

Although this works in principle, it is cumbersome if several aspects of the environment are being changed. Moreover, if the script terminates abnormally and never executes its last line, it can be difficult to predict the status of the caller's environment.

Capturing an Environment

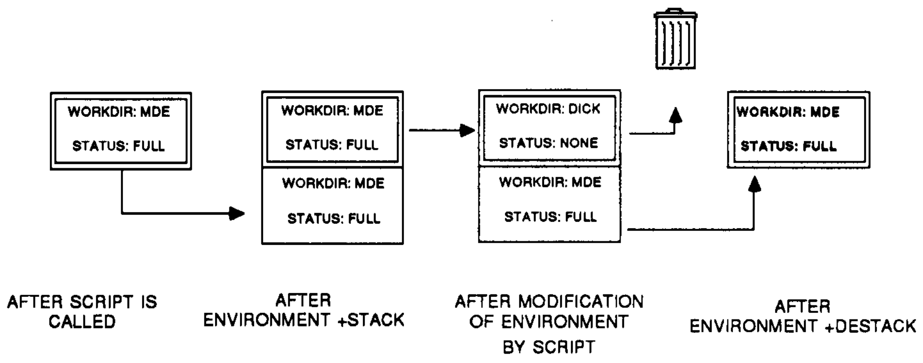
To remove the burden of saving environment values before altering them, LOGOS maintains an **environment stack**. This stack provides a simple and effective means of saving and restoring environments. For more information on stacking environments, see *Chapter 14: Profiles and Environments*.

■ To capture the caller's environment, issue the command *environment +stack* from within your script.

The following figure illustrates the operation of the environment stack at four different times. These are just after a script:

- is called
- issues *environment +stack*
- modifies environment parameters
- issues environment *+destack*

Figure 5.1 Stacking an Environment



Because virtually any script which saves the environment will want to restore it, LOGOS handles the restoration for you automatically: If you have written an *environment +stack* into a script and not destacked it, LOGOS will destack it for you (thereby restoring the previous values) at the point when the script finishes. The implicit destack happens even if the script aborts or fails with an error, so you can be sure you won't be leaving your caller in an unpredictable state.

Example

The script *cmddoc* fetches the long form description of each LOGOS command and appends it to a file which it creates. To avoid unnecessary writing of the status area, the script disables it by setting *environment status none*. Here's what *cmddoc* looks like:

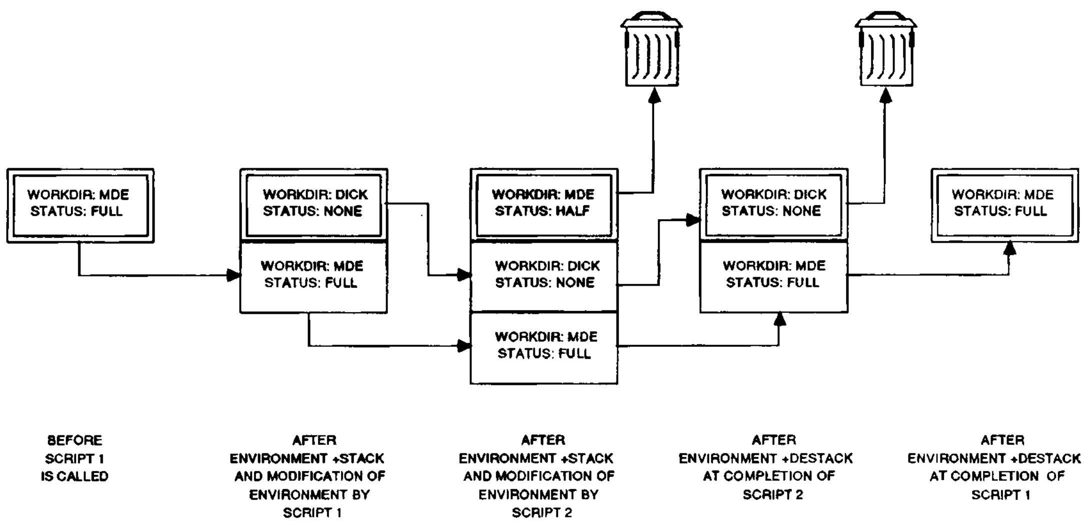
```
[1] z←cmddoc;tn;txt;.public.util.sys.hsp
[2] )environment status none +stack ⍝ save environment, adjust status
[3] ('t',⍕256⊥¯4↑⎕ts) ⎕create tn←((⍳⍴tn)⌊tn←0,⎕nums)∊0 ⍝
create file
[4] )with '∪ txt←??⍺ ∪ ⍎txt ⎕append tn' (?) ⍝ append help messages
to file
[5] z←tn hsp 'erase∘please hold for mary lee' ⍝ submit hsp request
```

The *environment* command on the second line performs two functions: it snaps a copy of the caller's environment, and then causes subsequent status area activity to be ignored. Establishing a *status* setting of *none* has no effect unless the status line was enabled. The *with* command on line 4 calls *??* for each command, and then appends this result to the file created on the preceding line.

If the status area were not disabled, LOGOS would show that the script called *??* and ⍎ repeatedly (once per LOGOS command). No destack is necessary in the script (although you can include one), because LOGOS restores the saved environment implicitly.

The environment stack is actually a little more general than this. You can stack multiple snapshots of the environment settings, and restore them at will. Stacking more than one entry is important if you are writing a script that needs its own environment, and your script calls another which needs its own environment. The following figure shows how the stack might appear after two snapshots of it have been taken via *environment +stack*, with intervening changes to the working directory.

Figure 5.2 Stacking Two Environments



WORKDIR: MDE
STATUS: FULL
WORKDIR: DICK
STATUS: NONE
WORKDIR: MDE
STATUS: FULL
WORKDIR: MDE
STATUS: HALF
WORKDIR: DICK
STATUS: NONE
WORKDIR: MDE
STATUS: FULL
WORKDIR: DICK
STATUS: NONE
WORKDIR: MDE
STATUS: FULL
WORKDIR: MDE
STATUS: FULL
BEFORE SCRIPT 1 IS CALLED
AFTER ENVIRONMENT ←STACK AND MODIFICATION OF ENVIRONMENT BY SCRIPT 1
AFTER ENVIRONMENT ←STACK AND MODIFICATION OF ENVIRONMENT BY SCRIPT 2
AFTER ENVIRONMENT ←DESTACK AT COMPLETION OF SCRIPT 2
AFTER ENVIRONMENT ←DESTACK AT COMPLETION OF SCRIPT 1

Using Script Debugging Mode

Normally, if a line of a script engenders an error, an error message is printed, the line of the script is displayed, and execution of the script is terminated. For example:

```
test
```

This could produce a message such as:

```
syntax error
.mabra.tools.test[3] )txt←list 'a b
                               ^
u
```

In the above example, the error is obvious; to correct it, edit the script and then re-execute it. The problem may not always be as obvious, however. You might want to examine the values of local variables, perhaps modify one or more of them, and then resume execution of the script, possibly on a different line. You can do all of these things by enabling **script debugging mode.**

Enabling Script Debugging Mode

The environment *debug* parameter controls script debugging mode. This parameter can assume the values *on* or *off*; the default is *off*.

To enable debugging mode, type:

```
environment debug on
```

To save this in your profile, type:

```
environment debug +profile
```

The *environment* command is described more fully in *Chapter 14: Profiles and Environments.*

With *debug* enabled, errors in scripts do not cause the execution of the script to be abandoned. Instead of returning to the LOGOS system prompt after the error message is displayed, you are placed in debugging mode. To remind you of this, the system prints the message:

```
⋆debug⋆
```

It also prompts you for input with six blanks, and awaits input. For example, if you ran the script *test* which resulted in an error, you would see a message such as:

```
value error
.mabra.tools.test[5] txt←txt,cr
                            ^
*debug*
```

Working in Debugging Mode

Debugging mode is an immediate execution mode. You can enter an APL expression and it will be executed. You can branch to any line of the script. You can enter a naked branch arrow to terminate the script and return to the ᴜ prompt. In the current example, you might continue your session as follows:

```
⋆debug⋆
      cr←⎕av[157]
⋆debug⋆
      →⎕lc
```

The script resumes execution and completes normally.

Recalling and Editing Input

You can also recall the last line you entered, go into editing mode, edit and re-execute your last line of input. This is analogous to the manner in which the APL interpreter lets you edit the last line you entered in immediate execution mode.

To recall the last line you entered and go into editing mode, type:

```
)
```

You can recall a line and position the cursor under a specific character within the line by supplying a number with the `)`. For example:

```
)7
```

This recalls the last line and places the cursor under the seventh character in the line.

Executing LOGOS Commands

To execute LOGOS commands while in debugging mode, precede them with `)`. For example:

```
*debug*
      )list +column
misc      tools      utilities
*debug*
      )edit tools.scan
```

Inquiring on Scripts on the Stack

The LOGOS *si* command is very similar to the APL)*si* system command. While the APL command displays information only about each function on your execution stack, the LOGOS command also provides information about each script on the stack. For example:

```
 *debug*
      )si
fcreate[1] *
.mde.logos.mycmds.putfile[2]
.mde.logos.mycmds.update[3]
logos[17]
maint[5]
```

This shows that you are suspended on line 1 of the function *fcreate*, which was called from line 2 of the script *.mde.logos.mycmds.putfile*, which in turn was called from line 3 of the script *.mde.logos.mycmds.update*, which was invoked from the LOGOS function which was called from line 5 of a function named *maint*.

Points to Note about Debugging Mode

Although debugging mode offers many of the facilities of APL immediate execution mode, there are some points to keep in mind when using it:

- If you use ⎕*trap*, avoid trapping events in the 400-500 range. You can adversely affect the LOGOS event traps and partially disable debugging mode. Avoid ⎕*signal* entirely. When you resume your script with →⎕*lc* or another line number, traps you have set previously may no longer be in effect.

- Ordinary APL system commands such as)*load* and)*copy* are not available in this mode. Input lines preceded by) are interpreted as LOGOS commands. This means that)*copy* is interpreted as the LOGOS *copy* command, not the APL system command of the same name. Script names as well as ordinary command names are recognized.

- You can edit a suspended or pendent script. If you are running Release 19 of SHARP APL, any changes you make to the script are saved both in LOGOS and in the workspace. The modified script can be restarted with a branch to the appropriate line. With earlier releases of SHARP APL, your changes will be saved in LOGOS, but will have no effect on the suspended script.

- If you invoke a script from debugging mode via)**script**, and this script also suspends and leaves you in debugging mode, entering a naked branch (→) will cut you back to the level of the first script suspension. To return to the system prompt, issue → once for each suspended script (one more time in this example).

Sample Scripts

As you become more familiar with the LOGOS environment, you will discover useful applications for scripts on your own. The following examples illustrate some of the principles presented in this chapter.

Substitute for the *list* Command

The personal preferences you develop after using LOGOS for a while will likely incline you toward certain command/modifier combinations. A script is a good way to construct a user-defined command that behaves almost identically to a particular LOGOS command, but with different defaults. The *listc* script used throughout this chapter is an example of this.

The *listc* script, as developed so far, has a few glaring deficiencies.

Table 5.8 Deficiencies with the *listc* Script

Problem	Solution
Its name is longer than the name of the command it mimics.	Change the name of this script to *lc*.
It does not accept the full range of modifiers that the *list* command does. To obtain a columnar listing of full directory names recursively, you would be forced to abandon *lc* and go back to the native *list* command. For example: *list +column +full +recursive*	Add all of the *list* modifiers to the header of *lc*, and use the \+ construct to conditionally interpolate them into the script.

The following script is an exact substitute for the *list* command, except that the *+column* modifier is enabled by default, and output is constrained to width 79. Note that *+column* is included in the header of the script, even though it is not referenced within the body of the script. If the user accidentally invokes *lc* with *+column*, it is ignored.

```
[1] z←lc[1] (+Pathnames=) (+column) (+data=) (+full)
      (+headings) (+long) (+overhead) (+recursive=)
      (+summary) (+type) (+ultimate) (+versions=);⎕pw
[2] ⎕pw←79
[3] )z←list \Pathnames +c \+d \+f \+h \+l \+o \+r
      \+s \+t \+u \+v
```

The *profile* and *fork* Scripts

Chapter 14: Profiles and Environments includes a description of how to define an expression to be executed each time you enter LOGOS. Briefly, this is done by setting the *environment entry* parameter to the desired expression and saving the environment in your LOGOS profile.

One handy entry expression is one which invokes a *profile* script. You will probably not want to perform an extensive amount of processing here, as this script is executed every time you enter LOGOS. But you might want to take two different courses of action in LOGOS depending upon the type of task you are on. This would give you a background processing capability.

The following script illustrates:

```
[1] z←profile;.public.util.ts.date|DAYS|MTHS
[2] ⍝ logos entry profile
[3] ⍝
[4] →(¯1=⌶28)⍴l0 ⍝ branch if we are not a t-task
[5] ⍝
[6] ⍝ t-task profile
[7] ⍝
[8] z←date ⋄ →0 ⍝ get time and date, and exit
[9] ⍝
[10] ⍝ non t-task profile
[11] ⍝
[12] l0:)(⍎⎕sp) ⍝ non t-task, execute ⎕sp
[13] ).public.logos.cmds.off ⍝ sign off
```

If the task entering LOGOS is a T-task, your profile function prints the time and date. Note that this is a composite script; it uses a function called *date* and two objects called *DAYS* and *MTHS* from the *.public.util.ts* directory of the LOGOS utility library.

If the task is an N-task or a B-task, the contents of ⎕*sp* are executed as a LOGOS command, then the task is signed off via the *.public.logos.cmds.off* script. Now any sequence of LOGOS commands can be executed by assigning a character vector expression to ⎕*sp* and starting an N-task which enters LOGOS.

You can make this facility easier to use by writing another script, which sets up ⎕*sp* and starts the N-task:

```
[1] z←fork[u] +Commands=;⎕sp
[2] ⎕sp←Commands
[3] z←⎕run ': 1 logos Clogos 0 0'
```

The *fork* script takes a single argument containing a list of expressions to be executed. Note that this script has been assigned unprocessed scope. This means that the argument to the script can contain quotes, + symbols, parentheses, and other reserved characters, which will not be interpreted by LOGOS before the argument is passed to the script.

As an example of how to use the *fork* script, consider a situation in which you want to run a script that performs a lengthy update on a number of workspaces and files. You could tie up your terminal waiting for this process to complete, or you could execute the following line:

```
fork bigupdate +file=sysfiles +ws=testfns
```

This initiates an N-task to perform the update and frees your terminal immediately. Note that because *fork* has unprocessed scope, the *bigupdate +file* and *+ws* modifiers can be entered without confusion.

Logging LOGOS Session Output

A possible shortcoming of the facility just developed is that there is no record of what happens in the N-task. The following script uses ⎕*out* to log LOGOS session output:

```
[1]  z←log[l] (+Fname=logoslog) (+clear) (+echo)
    (+off) ;.public.utilfiles.∆fopen;tn
[2] ⍝ log logos session output to named file
[3] ⍝
[4] ⍝ default file name is 'logoslog'
[5] ⍝
[6] ⍝ modifiers:
[7] ⍝
[8] ⍝ +clear: drops all components from end of file
[9] ⍝ +echo: causes output to be echoed locally
[10] ⍝ +off:  turns off logging
[11] ⍝
[12]  →off↓l0 ⍝ request to disable logging?
[13] tn←⎕out 1 0 ⍝ if so, restore default
[14] z←'session logging deactivated' ◇ →0
[15] l0:tn←Fname ∆fopen 0 ⍝ tie or create file
[16] →clear↓l1 ⍝ clear file?
[17] ⎕drop tn,−/2⍴⎕size tn ⍝ drop all components
[18] l1:⊣⎕out echo,tn ⍝ output to file, optional echo
[19] z←'session logging to ''',Fname,''' activated'
```

All of the modifiers in this script are Boolean switches, because they do not accept any arguments. Notice how the associated variables are used in branching statements, such as line 12, and in implicitly conditional logic, such as line 18. The ∆*fopen* function used on line 15 is a LOGOS utility, provided to assist in the generation and manipulation of files. It is described in *Appendix B*.

Making Sure Your Scripts have Room to Run

LOGOS is a paged application. It moves modules in and out of your workspace on demand. Without direct user control over the page-out of LOGOS objects, it is possible for a script executing a line of APL code to encounter a *wsfull*, even when there are a number of LOGOS modules in the workspace which could be paged out. The function *.public.logos.cmds.util.∆lreclaim* and the script *.public.logos.cmds.lreclaim* provide a means of reclaiming this space.

Invoke *∆lreclaim* or *lreclaim* at the beginning of a script which requires significant amounts of free workspace. In addition, the *∆lreclaim* function can be called from a *⎕trap* expression in response to a *wsfull*.

∆*lreclaim* versus *lreclaim*

The advantage of the function over the script is that it can be embedded in an event trap recovery expression. However, you must materialize the function in your workspace, either by localizing it in the header of your (composite) script, or by using a *get* command. This is unnecessary if you use the script, which is always directly accessible.

A good rule of thumb is to use the function form if you want to invoke it from a trap expression, or if the script is already a composite one; otherwise, use the script form.

CHAPTER 6: USING THE EDITOR

Tables

Terminal Support

LOGOS supports two classes of terminals: standard asynchronous terminals, and members of the IBM 3270 family of full screen display devices. Although a 3270 device may be more convenient to use certain features, the editor has the same functionality regardless of terminal type. LOGOS determines your terminal type automatically when you enter the editor.

If you are using an asynchronous device such as a PC running Reuter:file's CONNECT, the editor runs in line mode, mimicking the SHARP APL *DEL* editor. In particular, ∇ and the [*n*⎕*m*] family of commands can be used as they would be in the system *DEL* editor. (The *DEL* editor is, in turn, based on the VS APL Extended Editor and Full Screen Manager. For more information, see the document, *VS APL Extended Editor and Full Screen Manager*, IBM publication SH20-2341-1.)

If you are using a device that supports AP124 (for example, an IBM 3270 display station or a PC with 3270 emulation) the LOGOS editor runs in full screen mode. It also allows you to use program function keys.

Editing Objects

Invoking the editor on an object is called opening an object for editing. You can invoke the editor using the *edit* command or the ∇ command. These two commands are synonymous. The *edit* command takes a list of names or pathnames as its argument. These names can refer either to existing objects or to new objects that you want to create.

For example, to edit the object *modules.chart*, type:

```
edit modules.chart
```

Patterns are not allowed in the argument, because they have no meaning if you are defining a new object. But you can use the *list* command to resolve a pattern against objects that already exist in LOGOS:

```
edit (list ∆r?*[f])
```

Here, the *edit* command opens the definition of all the functions in your working directory that begin with ∆*r*.

Editing Objects In Your Active Workspace

You can edit objects in your active workspace by prefacing the name of the object with =. For example:

```
edit =table
```

If the object was opened from the workspace, the *edit* command saves it back there by default.

Editing Attributes

To edit an object attribute (its compilation directives, documentation, journal, note, or tag) specify an extended pathname indicating the attribute. For example, to edit the documentation attribute of the path *.john.modules.chart*, type:

```
edit .john.modules.chart[:d]
```

When you are already editing an object and want to access another attribute of the same object, you can omit the name and provide only the bracketed qualification; the editor assumes you are referring to the current path. For example, to edit the journal attribute of the current object, use:

```
∇[:j]
```

Editing Versions

To edit a particular version of an object, specify an extended pathname indicating the version. For example, to edit the second version of path *.john.modules.chart*, type:

```
edit .john.modules.chart[2]
```

When you are already editing an object and want to access another version of the same object, you can omit the name and provide only the bracketed qualification; the editor assumes you are referring to the current path. For example, to edit the third version of the current object, type:

```
∇[3]
```

Here, version 3 of the object is opened for editing.

On 3270 devices, function key F12 is defined to move you among the source, journal, and documentation attributes of an object. This makes it especially convenient to record changes made to an object at the same time as you alter the the object itself.

Executing Commands as You Open an Object

You can pass a series of commands to be executed by the editor before it solicits input from the keyboard using the *+command* modifier. For example, to begin editing *modules.chart* with the reference line at the first occurrence of the string *title←*, you could use:

```
edit modules.chart +command=locate/title←
```

The following example shows how multiple commands can be passed to the editor.

```
edit modules.chart +command='locate/title←∪ add3'
```

The quotes are required so that LOGOS does not interpret the ∪ within the line as the beginning of a new command. You could also accomplish this by providing an alternate separator character. For example:

edit modules.chart +command= ⊂ locate/title←⊂ add3

Registering Objects Out

Use of the registration facility with the editor can be implicit or explicit. If an object has registration potential set, the editor will automatically register the object out when opening it; no special action is required on the part of the person editing the object.

Alternatively, you can explicitly request that an object be registered out during your editing session, by using the *+register* modifier to *edit*. For example:

edit .john.modules.chart +register

In either case, LOGOS prints a message reminding you that the object is registered. For example:

.john.modules.chart: ⋆now registered

Overriding Registration

You can use the *+override* modifier to override registration set on an object by another user. This allows you to save a new version of an object, even if it's registered by someone else. For example:

edit .john.modules.chart +override

When the original registrant next accesses the object, he or she is immediately notified that you overrode the registration.

Looking at Objects in the Editor

When you open an object for editing, each line is given a line number. If you open a variable or a script, the first line is 1. Functions begin on line 0.

By default, the first line of the object is the **reference line.** You use the reference line to manipulate the object during editing.

The status line displays:

- the name of the object opened
- the version of the object opened
- the type of the object opened
- a browse indicator to let you know whether you are in browse mode

It appears on the screen automatically when you edit in full screen mode. When you are editing in line mode, you can display it using an editor command.

Objects in Line Mode

When you invoke the editor in line mode, the system returns the pathname and version of the object you have opened and enters edit mode. For example:

edit .john.practice.createfile

The system returns the message:

```
function: .john.practice.createfile[2]
∇∪
```

∇∪ is the editor command prompt. The ∪ indicates your current separator character.

To display the object, type:

```
[□]
```

The object scrolls on your terminal. Stop the scrolling by pressing Ctrl-S. Start it again by pressing Ctrl-Q.

When the object finishes scrolling, the cursor rests on the next blank line under the object, where you can use other editor commands.

■ Display the status line by typing:

status

A status line might look as follows:

```
.john.modules.chart[4] type: f mod
```

The *f* indicates that the object you are editing is a function, and *mod* indicates that you have modified the object.

```
.john.modules.opt[2] rank: 1 type: vc cd
```

This status line shows that you are editing a character variable (*vc*). The variable is a vector because its rank is 1. The absence of the *mod* flag indicates that you have not modified the object. The letters *cd* at the end of the status line tell you that this object has non-default compilation directives and documentation.

Objects in Full Screen Mode

When you open an object using full screen mode, the system enters the editor screen:

```
.john.modules.chart [0]                                          type: f


[0] chart










new function: .john.modules.chart[0]
∪ _
                                              --- logos r2.0 editor ---
```

The top line of the screen is the status line. Beneath the status line is the input area containing the object. At the bottom of the input area is the editor separator character ∪, and beside it rests the cursor. This is the command line, where you enter editor commands.

Changing the Editor Command Separator

You can specify a different separator character by beginning the line with it. For example:

∪!*change/*∪/n

This makes it possible to use the default separator character in a command argument.

Using Editor Commands

Like LOGOS commands, editor commands can be abbreviated to a few characters. Each command has a minimum number of characters that must be entered for it to be recognized. For example, the *again* command can be abbreviated to *a*.

Command Output

In line mode, command output is displayed directly on your terminal. In full screen mode, output is displayed in a small window at the bottom of the editor screen. The window appears only when it is required. Otherwise, the display area expands to fill the space.

Using Commands with the Reference Line

Several commands, such as *add*, use the reference line as a starting point. For example, if you move the reference line to line 20 and use the command *add 5*, the editor inserts the new lines after line 20.

Several editor commands also have an *up* version that reverses the direction of the command. For example, to tell the editor to add 5 blank lines above the reference line, rather than below the reference line, use the command:

```
addup 5
```

Many commands move the location of the reference line.

To change the reference line to a specific line number, use the command [**n**] where **n** is the line number. For example, to move it to line 6, type:

```
[6]
```

To move the reference line a number of lines up or down from the current location, use the *up* or *down* commands. For example, to move the line down 5 lines, type:

```
down 5
```

The *locate* command moves the reference line to the next line containing a given string or pattern.

Entering LOGOS Commands

While the editor environment is distinct from the LOGOS environment, it is possible to issue LOGOS commands from within the editor. You can do this using the *logos* command.

The *logos* command takes a LOGOS command line as its argument, and executes it as if you had typed it at the command prompt outside the editor. For example:

```
logos list .john.modules +column
```

You can invoke any command or script, with the exception of the *edit* or *exit* commands, in this manner. If you are using a 3270 device, output from the command is displayed in the editor window.

- To make the editor *logos* command easy to type, you can use) as a surrogate for it. For example:

) list .john.modules +column

  This is equivalent to:

  logos list .john.modules +column

Using Function Keys

In full screen mode you can also use function keys. Function keys have the following definitions.

Table 6.1 Function Key Definitions

Key	Function
F1	Invokes help.
F2	Toggles line number protection on/off.
F3	Closes a window, or saves the object and exits the editor.
F4	Enters input mode after the reference line or the current line.
F5	Switches to the next object in the edit stack.
F6	Makes the current line the reference line.
F7	Scrolls backward one page.
F8	Scrolls forward one page.
F9	Recalls the last command line.
F10	Splits the current line at the cursor position.
F11	Moves the cursor to the end of the current line.
F12	Switches between source, journal, and documentation attributes.

Getting Help

The editor help system provides a detailed explanation of all editor commands. You can display a summary of editor commands by simply typing *help*. This lists each editor command, its short form, and its function.

You can get help for individual commands by typing *help* followed by a command. For example:

help add

You see the message:

The <add> command adds one or more lines to an object. If you do not provide an argument, <add> inserts one line. If you provide the first argument, a positive integer, <add> inserts that number of lines below the reference line. If you provide a second argument, a string of text, <add> places the text on the added lines.

To insert the new lines before the reference line, use the <up> variant of <add>.

Examples:

Insert two empty lines following the reference line:

> *add 2*

Add five empty comment lines above the reference line:

> *add 5* ⍝

Full Screen Help

In full screen mode, pressing F1 produces a list of topics. For example:

```
                           select help topic
command summary                                  pf keys
keywords                                         line number commands
separator characters                             utility interface
the <add> command                                the <again> command
the <apl> command                                the <aplw> command
the <bottom> command                             the <browse> command
the <change> command                             the <copy> command
the <delete> command                             the <diamond> command
the <display> command                            the <down> command
the <edit> command                               the <end> command
the <format> command                             the <get> command
the <header> command                             the <help> command
the <highlight> command                          the <input> command
the <insert> command                             the <join> command
the <lastline> command                           the <locate> command
the <logos> command                              the <move> command
the <names> command                              the <next> command
the <put> command                                the <putnum> command
the <quit> command                               the <renum> command
the <replace> command                            the <resequence> command
the <sepchar>                                    the <setname> command
----------------------------3=back 7=up 8=down --------------------------
```

To choose a topic, type any character beside the topic and press Enter. The help message appears on the screen.

Using the *help* command produces a split screen. The help message appears in a window on the bottom half of the screen. This allows you to look at the help message and the object you are editing at the same time. For example:

help add

The following screen appears.

```
.john.modules.chart [0]                                            type: f



[0] chart



-----------------------------------------------------------------------------
The add command (has up variant)
Short form: ad
Parameters: [INTEGER] [STRING]

The <add> command adds one or more line to an object. If you do
not provide an argument, <add> inserts one line. If you provide the
first argument, an integer, <add> inserts that number of lines
below the reference line. If you provide a second argument, a
string of text, <add> places the text on the added lines.
To insert the new lines before the reference line, use the <up>
help add
U _
                                                 --- logos r2.0 editor ---
```

To scroll through a help message using F7 and F8, you must move the cursor inside the display window. By default, help messages appear in a window of twelve lines. You can change the size of the display window using the *window* command. For example, to change the window to 20 lines, move the cursor to the command line and type:

window 20

The maximum window size is 12 lines less than the maximum number of lines your terminal can display.

Exiting the Editor

Saving your changes to an object and exiting the editor is called closing an object. You can do this using the *end* or ∇ command. For example:

```
end
```

If you are using an IBM 3270 device, you can also exit by pressing F3.

NOTE: If you are using the full screen version of the editor and you have something displayed in the window, you must press F3 twice to close the object. The editor closes the window first, and then the object.

These methods have the same effect: the object you are editing is saved and then removed from the current editing stack.

If you have more than one object open, the editor moves to the next object; otherwise, it terminates and returns you to the LOGOS command prompt.

- To save an object without incrementing its version number, type:

```
end[0]
```

Changing Object Name

You may sometimes want to save an object in a different path or under a different name from the one you started with. There are two ways to do this.

- You can specify the new name as the argument to the *end* command. The argument can be a pathname or a simple name. For example, to save the current object under the name *comp.linedit*, use:

```
end comp.linedit
```

If *linedit* is a function or a script, the name *linedit* must agree with the name in the object's header. For example, you can change the name in the object's header from *pool* to *loop* and then use the command:

```
end loop
```

The copy associated with the original name is unaffected by the operation.

- You can also change the name of an object without exiting the editor with the *setname* command. For example:

```
setname comp.linedit
```

This command changes the pathname under which the object will be saved, and if the object is a function or script, automatically updates its header to reflect the change.

NOTE: If the object is a function or script, typing over its name in the header line will not cause the name to change.

IMPORTANT: The object is not saved yet. You must still use the *end* or ∇ commands to save it.

Changing Object Destination

You can change the destination of an object by changing its name using the *setname* or *end* commands. Thus, an object opened for editing from the LOGOS file system can be resaved in your workspace.

- To save the object into the workspace under a new name using *setname*, precede the object name with =. For example:

  setname =complinedit

  When you press F3 or type *end*, the object will be saved in the workspace.

Changing Object Type

The datatype and rank of an object can be changed with the *settype* command. For example, if you are editing a function and you want to convert it to a script, you can type:

settype s

For example, to convert an object to a character vector (even if it were, for example, a numeric matrix), type:

settype vc 1

The first argument, *vc*, indicates a variable of character type; the second argument, *1*, indicates an object of rank *1* (vector).

Discarding Changes

If you decide not to save your changes, you can use the *quit* command to discard them and exit without saving a new version of the object. To protect you from inadvertently discarding important work, the *quit* command refuses to exit if the current object has been modified in any way.

- To discard modifications to the object and, if there are no other objects open, exit the editor, type:

  quit imm

- To discard modifications to several objects and exit the editor, type:

  quit all

Registering Objects In

If the object has registration potential set, it's automatically registered back in again when you close it. But if the object was already registered by you, registration potential is ignored.

You can register the object in yourself using the *+register* modifier to the editor *end* or ∇ command. For example:

end +register

Overriding Registration

The *end* and ∇ commands also support a *+override* modifier to override registration. This modifier is more convenient than the parallel modifier to the *edit* command. To override registration, type:

end +override

Using Editor Tools

The editor contains its own set of tools to help accelerate the program development process. For details on the commands, and all other editor commands, see the *LOGOS Reference Manual*. Be careful not to confuse the editor commands with the LOGOS commands of the same names.

Importing and Exporting Blocks of Text

The editor uses two commands to move blocks of text in, out, and around an object: *get* and *put.*

The *put* command writes all or part of the object being edited to the editor clipboard, or to a variable in your workspace as a character matrix. The *vput* command is similar, except that it writes the variable as a character vector with embedded carriage returns delimiting lines.

For example, to put the reference line and two lines after it into the editor clipboard, type:

put 3

To put the reference line and all lines after it into the clipboard, type:

put ⋆

To put the reference line and two lines after it into a variable in your workspace, specify the variable name after the number. For example:

put 3 foo

The *get* command extracts text from the editor clipboard, an object in your workspace, or a LOGOS path, and inserts it into the object you are editing.

To fetch the contents of the editor clipboard and insert it after the reference line, type:

get

To fetch the value of a variable or the canonical representation of a function that exists in your workspace, and insert it into your object, specify the variable or function name. For example:

```
get foo
```

The *get* command recognizes abbreviated pathname syntax. To fetch the contents of a path and insert it into your object, supply the pathname. For example:

```
get .john.tools.report
```

You can also copy all or a number of lines from the cursor on, and insert them after the reference line. This allows you to copy text from a window into your object. For example, to copy all lines from the cursor on, type:

```
get *
```

To copy a specific number of lines from the cursor position down, for example 5, and insert them after the reference line, type:

```
get * 5
```

The cursor may be located in either the editor window or the display window.

Locating Strings in Objects

The editor *locate* command searches for a given string and positions the editor reference line at the next occurrence of the string. This command is an enhanced version of the *locate* command available in the *DEL* editor. There are two major improvements: the command accepts an optional **syntactic search qualifier**, and it supports regular expression patterns.

The syntactic search qualifier modifies the action of the *locate* command, so that it takes context into consideration when searching for a string. When searching syntactically, names and numbers are treated as units and either match fully or not at all.

To illustrate, a simple *locate* of *tr*, as in:

```
locate/tr
```

will find *tr* in strings such as *translate* or *control*. However, if we include the syntactic search qualifier _, only the name *tr* is sought:

```
locate_/tr
```

This distinction is very important when editing functions or scripts.

The use of regular expressions enhances the utility of the *locate* command significantly. For example, to search for strings such as *l0*:, *l1*:, and so on, you could use:

```
locate/{l[0-9]+:}
```

See *Chapter 12: Software Development Tools* and *Appendix A: Using Regular Expressions* for a detailed description of regular expressions.

Highlighting What You Locate

The *locate* command finds only the next occurrence of a string or pattern; it does not find all occurrences. The *highlight* command identifies all occurrences of a pattern.

On an asynchronous terminal, the *highlight* command simply displays all lines that contain the specified string. On a 3270 device, the *highlight* command allows you to specify the screen attributes (colour, intensity, and highlight) to be used to identify matches. For example, to highlight all instances of the name *beta* in inverse red, use:

```
highlight_/beta/ir
```

Here, the syntactic flag is used to limit the context of the search. The *highlight* command is particularly effective when searching for a regular expression. For example, you can highlight all branching statements in a program, and thereby reveal its control flow, using the command:

```
highlight/{→?*}/w
```

The resulting matches are highlighted in white.

You can have different patterns highlighted on the screen concurrently, each with its own distinct attributes. Suppose you are debugging a function and want to identify all references and assignments to the variable *ctl*. You can highlight references in yellow and assignments or indexed assignments in red with the following:

```
highlight_/ctl/y ⋄ highlight_/{ctl(¨[?*¨])-←}/r
```

To cancel all highlighting, type:

```
highlight/
```

Cross-referencing an Object

The editor *xref* command produces a cross-reference of the function currently open; it is very similar to the LOGOS command of the same name. Careful examination of the cross-reference of an object can be extremely helpful in avoiding and tracking errors.

On 3270 devices, the cross-reference is displayed in the window.

Locating Suspicious Names in Headers

The *header* command automates some of the duties normally performed with a cross-reference table. The *xref* command is commonly used to detect identifiers that are unintentionally global, or identifiers that are localized but not referenced. The *header* command isolates these identifiers and asks you on a name-by-name basis if you want the name added to or removed from the header. On 3270 devices, the list of suspicious names is presented as a full screen menu. If the header is altered, it is also sorted automatically.

Formatting Objects

The *format* command puts an object into its canonical display format. It also ensures that the object can be defined (has a valid header) and renumbers its lines. Use the *format* command to clean up a function or script that you have changed extensively, and verify that it can be saved. If the object you are working on is a numeric array, the *format* command validates the array's contents and regularizes its spacing.

Resequencing Line Labels

The *resequence* command is useful if you use program line labels such as *l0*, *l1*, and so on. When you insert new labels into a function and disrupt their numeric order, you can use *resequence* to restore the sequence. For example, if you insert a line labelled *l99* between two lines labelled *l1* and *l2*, *resequence* will rename *l1*, *l99*, and *l2* to *l1*, *l2*, and *l3*, respectively. If you have inadvertently assigned a particular line label twice, *resequence* prints an error message. All references to the labels are altered also to match the new sequence, unless the reference is inside quotes.

The *+prefix* modifier to this command controls which labels in a function or script are resequenced. The default value is *l*, and causes labels of the form *l*n to be processed. When *+prefix* is specified, its value is used in lieu of *l*. For example:

```
resequence +prefix=err
```

would cause line labels of the form *err*n to be resequenced.

You can optionally specify the starting label number (the default is 0) and increment (the default is 1) as arguments to the command.

NOTE: The *resequence* command does not affect line labels that are enclosed within quotation marks, as often occurs in ⎕*trap* statements. You might consider rewriting traps to put the line label outside of quotes. For example:

```
⎕trap←'∇ 2 e →',⍕log∆error
```

Sorting the Header's Locals List

The *sort* command helps keep the header of a function or script well-ordered. Many programmers find it convenient to organize local names in alphabetical order, as it simplifies locating a particular name in a long header. You can use the *sort* command to achieve this; it sorts the contents of the header's locals list and removes any duplicate entries.

Searching and Replacing

The *change* command is the editor's general purpose find-and-replace primitive; it allows you to replace one string or pattern with another. Like the *locate* command, *change* supports the syntactic search qualifier _.

For example, to change all occurrences of the identifier *title* to *subtitle* from the reference line to the end of the object, type:

```
change_/title/subtitle/⋆ ⋆
```

The characters ⋆ ⋆ indicate that all occurrences on all lines following the reference line will be changed. For more information on the third field containing ⋆ ⋆, see the *LOGOS Reference Manual.*

The ability to use regular expression patterns as arguments makes this command very flexible and powerful. For more information, see *Chapter 12: Software Development Tools.*

Displaying Objects

The *display* command displays the source of an object stored in LOGOS or in your active workspace. If you are using a 3270, the object is displayed in the window.

You may find yourself in a situation where you are editing an object and need to consult the source of another object. For example, you might be writing a line that invokes a function whose calling conventions you don't recall. Or, you might need to check a piece of online documentation. The *display* command makes these operations very simple and straightforward.

The *display* command recognizes the abbreviated pathname notation discussed earlier, which sets the command apart from using *logos display* or)*display.* For example, to display the change journal for the object you are presently editing, you can type:

```
display [:j]
```

To display the previous version, type:

```
display [¯1]
```

If you have made changes to the source of an object and want to examine a copy of the object as it was before your changes, you can display the latest version using:

```
display []
```

CHAPTER 7: USING AUXILIARY TASKS

Starting an Auxiliary Task

An **auxiliary task** is an S-task started by you under the auspices of LOGOS. It can be thought of as a logical extension of your LOGOS session. There are a number of auxiliary task commands in LOGOS; these enable you to initiate an auxiliary task, communicate with it interactively or under program control, and inquire upon its status. Other commands, such as *get* and *save*, support options that allow you to reference objects in an auxiliary task workspace as easily as you reference objects in your active workspace.

Auxiliary tasks are initiated using the *signon* command. To initiate an auxiliary task, type:

```
signon
```

Each task has a task name associated with it. You can have more than one auxiliary task signed on at a time, and the task name is used to differentiate among them.

By default, auxiliary tasks are signed on to the account associated with your alias, with a task name of *aux*. For example:

```
auxiliary task 4110 <aux> signed on 11apr86 20:25
```

The *signon* command can also take an argument, specifying the alias or user number under which the auxiliary task is to run, and possibly a password. For example:

```
signon proj:secret
```

If you supply an alias as the argument, LOGOS determines the primary account associated with that alias, and signs the task onto that number.

Specifying a Task Name

To specify a task name other than *aux*, use the *+task* modifier. All of the auxiliary task commands accept an argument or modifier that specifies the name of the task you want to reference. For example:

```
signon proj:secret +task=inst
auxiliary task 4118 <inst> signed on 11apr86 20:26
```

Here, an auxiliary task named *inst* is deployed under the alias *proj* (which has a lock of *secret*). If *+task* is specified without a value, it always refers to the default auxiliary task, *aux*.

If you provide an alias that is not your own and do not specify the alias' signon lock, LOGOS asks for it with a protected prompt:

```
signon proj +task=inst
password:
auxiliary task 4118 <inst> signed on 11apr86 20:26
```

Obtaining a Clear Workspace

It is generally important that an auxiliary task start off in a known state. The *signon* command guarantees that the newly-spawned task is in immediate execution in a clear workspace when the command completes. However, in some cases, the S-task may be forced into a *Profile* or *continue* workspace left by a previous session.

If LOGOS finds the task in immediate execution in another workspace, it issues a *)clear* command to obtain a clear workspace. If the task is not in immediate execution, LOGOS attempts to get the task into that state for you.

The *signon* command tells you of any actions it took in attempting to obtain a clear workspace. For example, if *1234567* has a *continue* workspace, the following might occur:

```
signon 1234567:secret +task=slave
<continue> loaded, cleared
auxiliary task 7812 <slave> signed on 12apr86 03:58
```

If the *signon* command is not able to obtain a clear workspace, the command itself fails. Therefore, if *signon* completes without producing an error report and terminating abnormally, you are guaranteed that the auxiliary task has a clear active workspace.

The *signon* command returns as its result the name of the task created. Normally, this result is not displayed. You can see it using ⎕←, or capture it using assignment to a variable.

Lifespan of an Auxiliary Task

An auxiliary task has the same properties as any other S-task. LOGOS uses a global shared variable for each active auxiliary task. If you retract any of these shares by erasing one of the variables, loading another workspace, or by using ⎕*svr*, the associated task terminates.

The *signon* command supports a *+retract* modifier that allows you to request retract permission for the S-task. This permits you to retract the share without terminating the auxiliary task. However, even with retract permission granted, the task will terminate if it requests input while the share is retracted.

You can also use the *send* command (described below) to request retract permission.

LOGOS uses a temporary workfile to keep track of auxiliary task information across sessions. This workfile is named ⍙0*logostemp*; it is created on your account when you issue a *signon* command. If a workfile already exists on your account, it is simply tied and reused.

When you exit LOGOS, the workfile is erased if you don't have any active auxiliary tasks. If you do have active auxiliary tasks, a reminder message is displayed and the workfile is preserved. You can always erase the workfile yourself, but if you do so, you will lose information regarding any active auxiliary tasks and will not be able to re-establish communications with them.

The workfile is also used by the *transfer* command, and can be put to further use in future releases of LOGOS. As a safeguard against accidental erasure, LOGOS ties the workfile with a passnumber of ¯*1*.

Communicating with an Auxiliary Task

Two commands are provided for communication with auxiliary tasks: *send* and *talk*. The *send* command is most useful for managing an auxiliary task as a brief interactive session, or under the control of a script. The *talk* command initiates a true interactive session, creating the illusion that you are signed onto the auxiliary task as an asynchronous T-task.

The *send* Command

The *send* command transmits one line of input to an auxiliary task and returns the output generated by it as its result. To transmit a line of input using *send*, type:

send **input** *+task=***task**

For example:

```
send '(1 ⎕ws 2)[⍳5;]' +task=inst
comms
control
file
menu
snap
```

Here, the names of the first five variables in the active workspace of the auxiliary task *inst* are returned as the result of the command. The argument is enclosed in quotes so that the parentheses around *1* ⎕*ws 2* are not evaluated by LOGOS, but rather are evaluated by APL in the auxiliary task.

The result of the *send* command is all of the output generated by the S-task, up to the next request for input. Normally, an input prompt is returned as part of the result. However, if the next request for input is an immediate execution prompt (carriage return followed by six blanks), this prompt is not included in the result of the *send* command.

The *send* command fails if the expression transmitted to the task fails. If the expression is an APL statement, an error occurring during the execution of the line is considered a failure. If the expression is a system command, any response other than a completely successful one is considered a failure. For example, *incorrect command* is always an error, and anything other than a timestamp from *)save* indicates an error.

If you want *send* to ignore errant conditions, you can specify the *+suppress* modifier. *+suppress* causes the *send* command to return its result without inspecting it for error messages, allowing execution of your command line or script to continue even if the argument to *send* fails.

Signalling Break or Attention

You can signal Break or Attention to an auxiliary task by using the *+break* modifier with the *send* command.

Break is sent before the line itself is transmitted. If the task is in input mode, *+break* tries to interrupt the input request using *o* **bs** *u* **bs** *t*.

Getting into Immediate Execution

When you are sending a system command or an expression that must be evaluated in immediate execution, you can include the *+immex* modifier to specify that the task be in the desired state:

```
send )copy reports subtotal +immex +task=inst
saved  1986-04-09 19:39:32
```

If the auxiliary task is not already in immediate execution, the *send* command tries to get it into that state by transmitting *break* or *o* **bs** *u* **bs** *t*. The command fails without transmitting the line if LOGOS cannot get the task into immediate execution.

Freeing up Your Terminal

Auxiliary tasks are often used as slave tasks for lengthy operations. For example, you may want to begin a long update without tying up your terminal waiting for completion. To free up your terminal, use the *+asynch* modifier in your *send* command.

With *+asynch*, the *send* command transmits the line without waiting for a response. To receive output at a later time, issue the *send* command again. If you don't provide an argument to the *send* command, it simply returns any pending output from the task.

Running a Task Autonomously

You can sever your connection with an auxiliary task and let it run autonomously by specifying *+retract* with the *send* command.

If you have specified *+retract*, you can load another workspace or sign off without affecting the task, as long as it is processing. The task will continue to run until it requests input, returns to immediate execution, or explicitly signs itself off. If none of these has happened, you can re-enter LOGOS and re-establish communications with the task.

The retract option is useful only while your auxiliary task is in a pure processing state. If the task requests input or enters immediate execution after the share has been retracted, it will terminate even with *+retract* set.

Retrieving Pending Output

You can check on the processing within the auxiliary task by using the *send* command with no argument.

This retrieves pending output generated by the task.

Interrupting an Auxiliary Task

To interrupt an auxiliary task, signal an interrupt by pressing Break or Attention while waiting for the *send* command to produce output.

LOGOS prompts you with:

abandon, resume, or break:

Enter one of the following (each of which can be shortened to the first character):

Table 7.1 Commands You Can Enter at the Warning Prompt in *send*

Command	Result
abandon	Terminates the *send* command immediately.
resume	Resumes execution of the *send* command and ignores the interrupt.
break	Propagates the interrupt to the auxiliary task.

The *talk* Command

The *talk* command allows you to carry on an interactive session with an auxiliary task. The argument to the *talk* command is the name of the task with which you wish to communicate. It is optional, and as with the other auxiliary task commands, defaults to *aux*.

- To initiate an interactive session with auxiliary task *aux*, type:

  talk

- To initiate an interactive session with another auxiliary task, type:

  talk **name**

The *talk* command issues an immediate execution prompt (carriage return followed by six blanks) when the auxiliary task is awaiting input. You can change the prompt so that sessions look a little different from ordinary APL sessions.

■ To initiate an interactive session with an auxiliary task, and change the immediate execution prompt, use the *+prompt* modifier. For example:

talk +prompt=**prompt**

Your session continues as if you were signed on to the auxiliary task rather than to the spawning task. You can do anything you would in an ordinary terminal task, including starting another LOGOS session.

NOTE: If you are signed on using a terminal that supports a status line, and you have the full status line enabled, the second line displays a message reminding you that you are communicating with an auxiliary task.

Using LOGOS Commands in a *talk* Session

The *)logos* command allows you to perform LOGOS commands directly from your *talk* session. Type:

)logos **command**

)logos executes the command issued in your primary task, not in the auxiliary task. Here is an example showing use of the *)logos* pseudo-system command. The ∘ character is used as a statement separator in the second entry:

)logos workdir
.john.modules.chartsub
)logos ∘ edit verify ∘ delete oldverify

A shorthand for *)logos* is available. If the character following the) is a valid separator character, the remainder of the line is treated as a LOGOS command. For example, the second entry shortens to the following:

)∘ edit verify ∘ delete oldverify

To recall the last LOGOS command you executed from *talk* mode, type:

))

Leaving the *talk* Session

There are three ways to exit a *talk* session:

- using the *)disconnect* command
- pressing Break or Attention
- signing off the auxiliary task

Each method is described in more detail below.

■ To exit using *)disconnect*, type:

)disconnect

The)*disconnect* pseudo-system command results in an immediate exit from the *talk* session. The auxiliary task itself is not affected, and you can resume your session with it by executing *talk* again.

- To exit using Break or Attention, press the appropriate key. Either key causes the prompt:

  abandon, resume, or break:

  To exit at this prompt, type:

  abandon

  LOGOS immediately exits your *talk* session.

  Table 7.2, below, summarizes the effect of each response to the warning prompt.

Table 7.2 Commands You Can Enter at the Warning Prompt in *talk*

Command	Result
abandon	Exits your *talk* session, but leaves the task connected.
resume	Returns to your *talk* session and ignores the interrupt.
break	Propagates the interrupt to the auxiliary task.

- To exit by signing off your auxiliary task, type:

)*off*

Sample *talk* Session

Here is a sample session using the *talk* command. The session:

- investigates a workspace
- fixes an error by changing the source in LOGOS
- ends by sending a MAILBOX message informing someone of the change

```
∪ signon ∪ talk
auxiliary task 7160 <aux> signed on 21mar86 19:02
      )load 1234567 crashws
saved  1986-03-21 17:40:44
      ⎕er
   6 value error
picktitle[16] title←fread tn,¯1+2×cn
                    ^
      )si
picktitle[16] *
reports[1]
getcommand[3]
start[5]
⍎
      )logos replace title←fread title←∆fread picktitle
.budget.repfmt.picktitle[5]
      )load 666 box
saved  1985-01-07 23:50:04
      unread
no messages for fixit
       1470214  t ⎕prof
      send
to bdev
text:
I just fixed the bug in the <crashws> workspace.  it was
caused by a slight typing error in the function
<.budget.repfmt.picktitle>.  I have not generated a new
copy of the ws yet.     /bill
Space Enter
action: send
no. 2621440 filed 19.07.34  fri 21 mar 1986
to Space Enter
mail service complete
      )disconnect
∪ ←send )off
auxiliary task <aux> terminated
```

Monitoring Auxiliary Tasks

The *tasks* command is used to obtain information about your auxiliary tasks.

- To return a list of the names of all active auxiliary tasks you created, type:

  tasks

- To report detailed information on tasks, supply an argument of one or more task names. For example:

  task one two

  The report includes:

  - the APL task id
  - user number
  - sign-on time
  - CPU time
  - connect time
  - an indication of whether or not retract permission has been granted

- To display headings above each column of the report, use the *+heading* modifier. For example:

  tasks aux +headings

  You see a message such as:

  ```
  name--id----account--signon time----cpu-connect-retract
  aux  4110  mde     11apr86 20:25  105    257    yes
  ```

- To report on tasks running on the accounts of particular users, use the *+users* modifier and provide a list of aliases with it. For example:

  tasks aux +users=mde dpm rbk

- To get a report on all your tasks, type:

  tasks (tasks)

You see a message such as:

```
aux   4110  mde     11apr86 20:25  105    290    yes
ptask 4118  mde     11apr86 20:26   86    251    no
```

Transferring Objects between Workspaces

The *transfer* command makes it possible for you to copy objects from one active workspace to another. Objects can be transferred between the active workspaces of any two tasks you have initiated, in either direction, using the *+to* and *+from* modifiers.

The argument to the *transfer* command specifies a list of objects to be transferred. You can use a *+from* modifier to specify the task whose active workspace is the source for the transfer. You can use a *+to* modifier to specify the task whose workspace is the destination. Both the *+from* and *+to* modifiers take a task name as an optional argument.

If you specify only one of these modifiers, the other workspace involved in the transfer is assumed to be your current active workspace. If you specify a modifier without an argument, the default task *aux* is assumed for that modifier.

WARNING! The *transfer* command assumes that any auxiliary tasks involved in a transfer operation are in immediate execution mode. The *transfer* command operates by sending APL expressions to the auxiliary tasks to be executed. If a task is not in immediate execution, the expression cannot be evaluated, and the *transfer* command will fail.

- To transfer objects from the current workspace to the active workspace of a task, specify a value for the *+to* modifier.

  For example, to transfer the objects *rcat* and *verify* from the current workspace to the active workspace of the *inst* task, type:

  transfer rcat verify +to=inst

- To transfer an object from the current workspace to the active workspace of the *aux* task, do not specify a value for the *+to* modifier.

  For example, to transfer the object *report* from the current workspace to the active workspace of the *aux* task, you could type:

  transfer report +to

- To transfer objects from the active workspace of a task to the current workspace, specify a value for the *+from* modifier.

  For example, to transfer the objects *compact* and *table* from the active workspace of the *inst* task to the current workspace, type:

  transfer compact table +from=inst

- To transfer an object from the active workspace of the *aux* task, to the active workspace of another task, specify a value for the *+to* modifier and do not specify a value for the *+from* modifier.

NOTE: If you specify both the *+from* and *+to* modifiers, the current active workspace is not involved in the operation at all.

For example, the command below transfers the object *vtom* from the active workspace of the *aux* task to the active workspace of the *inst* task:

transfer vtom +from +to=inst

Protecting Objects from Overwrites

You can ensure that an object never overwrites one of the same name in the destination workspace by specifying the *+protect* modifier with the *transfer* command.

When you specify *+protect*, the behaviour of this command is similar to the behaviour of the APL)*pcopy* system command.

Ignoring Errors if Transfer Fails

Normally, the *transfer* command fails if it is unable to transfer any of the requested objects. If you are anticipating this condition, you can cause the error to be ignored by using the *+suppress* modifier.

The *+suppress* modifier is often used in conjunction with the *+protect* modifier, so that failure of the command is avoided if any of the objects are already defined in the destination workspace.

Using Auxiliary Tasks with other Commands

Commands you can use with auxiliary tasks are *build, distribute, edit, get, save, shell,* and *wssave.* Specify them with the *+task* modifier to identify the auxiliary task involved.

LOGOS also has a global *environment task* parameter, set using the *environment* command, that can be used to precondition these other commands. The default value of this environment parameter is ⋆. By convention, ⋆ is the user's T-task (the one that originally began running LOGOS itself). Although these commands normally deal with the active workspace by default, you can change their default behaviour by setting the *environment task* parameter to the desired task name. For more information on the *environment* command, see *Chapter 14: Profiles and Environments.*

When the *build* or *get* command is directed to use an auxiliary task (through either its *+task* modifier or the global *task* parameter), it fetches objects from the LOGOS hierarchy and deposits them directly into the active workspace of the named auxiliary task. In a similar fashion, the *save* command, if so directed, fetches objects from the active workspace of the named auxiliary task and saves them as LOGOS paths in the hierarchy.

Sample Sessions

The short session below illustrates how commands work with auxiliary tasks.

First, an auxiliary task named *proj* is created. Then, the task loads a workspace called *util.* Finally, it saves all of the functions and variables in that workspace into the directory *.mabra.tools.util.* Note the use of the *send* command to obtain the list of function and variable names in the auxiliary task.

```
∪ signon +task=proj
auxiliary task 9198 <proj> signed on 12apr86 12:02
∪ send )load util
saved  1986-04-08 13:00:28
∪ save (send 1 ⎕ws 3 +task=proj)  +workdir=.mabra.tools.util
+task=proj
```

Here is the same example, using the environment *task* parameter:

```
∪ signon +task=proj
auxiliary task 9198 <proj> signed on 12apr86 12:02
∪ environment task proj
∪ send )load util
saved  1986-04-08 13:00:28
∪ save (send 1 ⎕ws 3 )     +workdir=.mabra.tools.util
```

Now you can use the *get* command issued from within a *talk* session to fetch a function named *faccess* from the LOGOS hierarchy into the active workspace of the auxiliary task. Because of the global *task* environment parameter, you do not need to specify the destination for *get.*

```
∪ talk proj
      )fns
ascan   convert model   rcat    vtom
      )logos get tools.util.faccess
.mabra.tools.util.faccess[4]
      )fns
ascan   convert faccess model   rcat    vtom
```

Like the *transfer* command, the *build, get,* and *save* commands require that the auxiliary task be in immediate execution. If it is not, the behaviour of the command is unpredictable.

Using Auxiliary Tasks in Scripts

Auxiliary tasks are very useful in scripts; they let you write scripts that install workspaces or perform general library management functions with ease. Several examples are presented in this section.

The *lib* Script

The *lib* script returns as its result the workspace library of the specified user. If the argument *User* is not specified, it defaults to the user number of the caller.

```
[1]  z←lib (+User)=
[2]  )signon \User +task=libtask ⍝ sign on task
[3]  )z←send )lib +task=libtask ⍝ fetch  library
[4]  )send )off +task=libtask += ⍝ sign off task
```

The *drop* Script

The *drop* script uses the *signon, with,* and *send* commands to drop a list of workspaces from a specified library. An optional *+user* modifier selects the alias or user number of the library; if not specified, the user number of the caller is assumed. The script has long scope, so that multiple workspace names can be provided without the need for quotes. At the completion of the script, the auxiliary task is signed off.

```
[1]  drop[1] +Wslist= (+user)=
[2]  )signon \user +task=drop ⍝ sign on task
[3]  )⎕←with 'send )drop ⍺ +task=drop' \Wslist ⍝ drop wss
[4]  )send )off +task=drop ⍝ sign off task
```

The *startgen* Script

The *startgen* script uses an auxiliary task to begin the execution of a script that generates a large system. After the generation script has been started, the auxiliary task is disconnected and the generation continues autonomously. The script that actually performs the generation is called *maint.gensys.*

```
[1]  startgen
[2]  )signon +task=gen ⍝ sign on task
[3]  )send )copy 1 logos +task=gen ⍝ copy <logos> fn into clear ws
[4]  )send logos +task=gen ⍝ enter logos
[5]  ⍝ invoke script asynchronously and request retract permission
[6]  )send maint.gensys +asynch +retract +task=gen
```

The *savewss* Script

The *savewss* script saves the contents of one or more workspaces into similarly named LOGOS directories. It starts an auxiliary task on the user number or alias indicated by the *+user* modifier (which defaults to the caller's number if not specified). Then, the script loads each workspace specified in its argument.

Note how the *send* command is used on line 17 to obtain a list of object names to save. The objects in each workspace are saved in a directory whose name is formed by appending the workspace name to the working directory given in the *+workdir* modifier. After the last workspace is processed, the auxiliary task is signed off. The script's result is all of the pathnames that it saved.

```
[1]  savewss[1] +Wss= (+user)= (+workdir)=;⎕io;t;wsid
[2]  ⍝ save a list of workspaces into logos file system.
[3]  ⍝
[4]  ⍝ parameters:
[5]  ⍝
[6]  ⍝  Wss      : list of ws names (no library numbers)
[7]  ⍝  +user    : user number owning workspaces
[8]  ⍝  +workdir: working directory to prepend to ws
[9]  ⍝
[10] ⎕io←1 ⍝ set local origin
[11] )signon \user +task=sw ⍝ sign on dedicated task
[12] )environment task sw +stack ⍝ set task parameter
[13] )workdir \workdir ⍝ set working directory
[14] lp:→(⍴Wss←(∨\Wss≠'  ')/Wss)↓end ⍝ any wss left?
[15]  wsid←(¯1+t←Wss⍳'  ')⍴Wss ⋄ Wss←t↓Wss ⍝ get next ws
[16] )send )xload (⍎wsid) ⍝ load it
[17] )output (save (send 1 ⎕ws 3) +workdir=(⍎wsid)
      +makedir) +result ⍝ save contents of ws
[18] →lp ⍝ go back for more
[19] end:)send )off ⍝ sign off task
```

CHAPTER 8: BUILDING APPLICATIONS WITH LOGOS

Tables

Figures

You have seen how LOGOS can be used to organize and manage the objects that form the building blocks of an application. At this point, you should be familiar with the techniques necessary to get an object into LOGOS, to manipulate it within the hierarchy, to edit it, to delete it, and so forth. These capabilities are pointless, however, unless there is a means for assembling these objects into the workspaces and files that comprise application systems. The term **end environment** is used to describe these workspaces and files.

This chapter will show you how to use LOGOS to build applications. Some familiar tools will be used in new ways and some new tools, designed specifically for application building, will be introduced. As with many other aspects of LOGOS, there are a variety of alternative approaches to application building. This chapter explores several of these techniques. The chapter ends with a detailed discussion of a methodology for establishing and generating applications in LOGOS.

End Environments

Probably the ideal execution environment for an APL system is a single saved workspace, containing all of the functions and variables the system needs. Few large-scale systems achieve this ideal in practice, for more often than not, the use of files to store or transport data becomes an important part of the system's design. But every APL system has at least some components which are stored in a saved workspace. This part of the application is workspace resident. Workspace-resident systems face three important limitations:

- Sharing. Communicating information through saved workspaces is usually awkward. Files facilitate the sharing of information. Changes are immediate, and all users can take advantage of them instantly.

- Permanence. Data changes which must be preserved are better handled with files. Workspaces require an explicit *)save* operation, whereas any change made to a file is taken by the system to be permanent without further action.

- Workspace size. Because many applications are too large to fit entirely in one workspace and still leave sufficient working storage for the application to perform reliably, many systems keep a miniature database of functions and variables in a file.

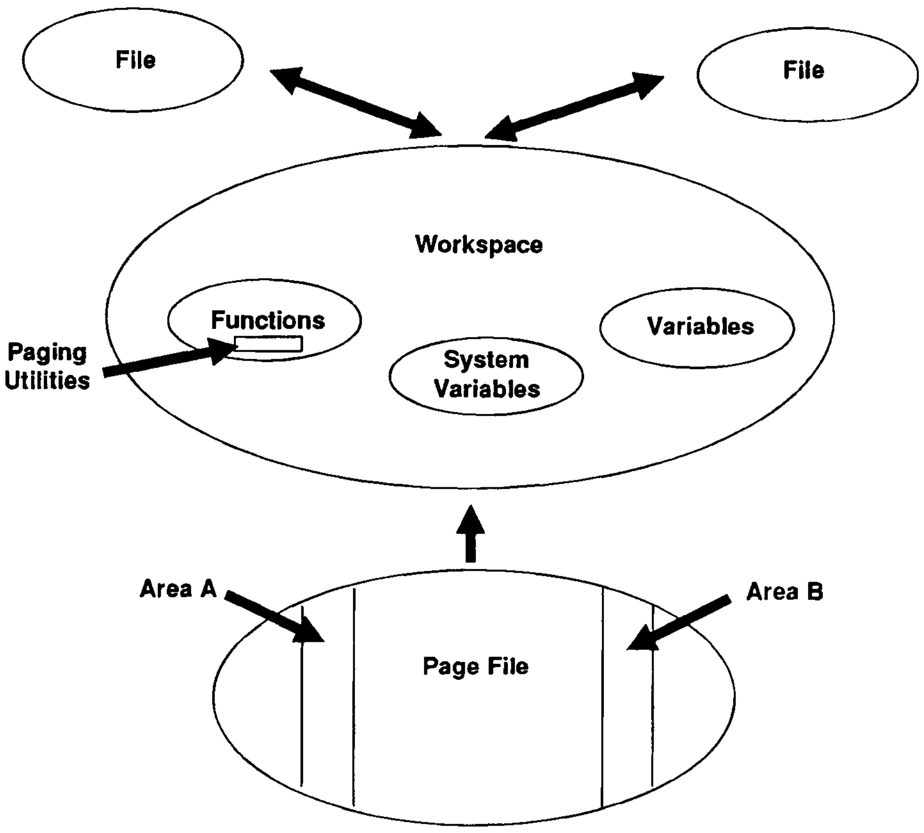
To work around these limitations, parts of the system are usually stored in a file, brought into the workspace when needed, and then either written back to the file or expunged when no longer required. These systems have file-resident components, and they represent most applications written in APL. Even a system that will fit in a workspace and is not shared may keep data on file for permanence, or programs on file to prevent or exploit name conflicts.

The general structure of an application is illustrated in Figure 8.1.

At the center is a workspace, containing functions, variables, and system variables (such as ⎕*io*). The workspace and its contents may entirely compose a small system. A more complex application may use data and program files, and one or more **paging files**. A paging file can contain arbitrary application data, as well as multiple independent areas from which objects may be paged. These multiple areas allow you to control a system's behavior; when there are similar objects in several areas, one set can be used in preference to another, or several sets can "overlay" to produce in the workspace exactly the combination of functions and variables you need.

Shells provide another means of running a large system in a small workspace. A shell is a function that acts as a data "umbrella", and materializes required objects as locals beneath it. When the work of the function is complete, the materialized objects disappear. Shells are also useful for avoiding name conflicts in the workspace. See the section on shells later in this chapter.

Figure 8.1 General Structure of an Application



The *get* Command

The *get* command provides a basic transport mechanism between the LOGOS hierarchy and the workspace. Given a list of paths, *get* will materialize them in the workspace. Example:

```
      )clear
clear ws
      )copy 1 logos
saved 1987-07-27 12:46:00
      logos
 ⋆logos⋆ r2.0
∪ get inventory general.⎕ps|⎕pw +compile=x,d,p
```

All objects directly subordinate to the *inventory* directory, and the user's standard values of ⎕*ps* and ⎕*pw*, are compiled and materialized into a clear workspace. In this example, the entire workspace-resident system is defined by a single LOGOS path, *inventory.*

inventory is a directory which contains only functions, variables, and links -- and no subordinate directories. Organizing the directory this way makes system generation easy.

The *+protect* modifier prevents overwriting of objects of the same name in the target environment:

```
∪ get inventory
∪ get general.∆?⋆[f] +protect
unable to define: general.∆help general.∆qi
```

In this example, the first command materializes all objects one level below *inventory;* the second materializes all functions in *general* whose names begin with ∆ except those already defined by *inventory.* If *inventory* is a directory containing other directories below it, the *+recursive* modifier may be used to specify the levels from which objects will be fetched.

To define objects at all levels below *inventory,* use:

```
∪ get inventory +recursive
```

The *+workdir* modifier establishes a temporary set of working directories, for just the single command:

```
∪ get inventory general.⎕ps|⎕pw util.?⋆[f] +workdir=.invent.rel3
```

Specifying multiple working directories allows you to overlay one directory's contents with another:

```
∪ get inventory general.⎕ps|⎕pw util.?⋆[f] +workdir=.invent.rel3 (workdir)
```

This establishes a working directory list of *.invent.rel3* followed by the current working directories. Since the directories are searched in the order of their specification, objects found in the earlier directories shadow objects of the same names in the subsequent ones.

A Simple Approach to Building Workspaces

The *get* command provides enough functionality to build a workspace. Consider a simple application consisting of a single end environment: a workspace containing a handful of functions. Here is a manual procedure for building the workspace using only tools discussed thus far:

1. Obtain a clear workspace.
2. Copy the *logos* function from *1 logos* into the workspace and execute it.
3. Set the working directory to the directory containing the source for the workspace.
4. Issue one or more *get* commands to bring the required objects into the workspace.
5. Exit LOGOS.
6. Erase the LOGOS function.
7. Save the workspace with an appropriate workspace-id.

While this manual procedure would accomplish the job, it's very crude. An obvious improvement would be to encapsulate the *workdir* and *get* commands within a script. This would save some typing and would guarantee consistent results. The *)copy* and *)save* steps would still have to be executed manually. These steps can be automated as well through the use of an auxiliary task. The following script illustrates these techniques:

```
[1] z←genreport1ws
[2] )signon +task=genreport1
[3] )workdir .plan.wss.report1 +stack
[4] )get runrep1 format1 calc1 totals +task=genreport1
[5] )z←send )save report1 +task=genreport1
[6] )send )off +task=genreport1
```

This script signs on an auxiliary task, establishes a set of working directories, defines the relevant objects into the active workspace of the auxiliary task, saves the workspace by transmitting a *save* command to the task, and then logs the task off. Maintenance of the workspace has been simplified greatly, since a single LOGOS command (script) will now create a new copy of the workspace containing the latest versions of all objects.

Building a Simple File

A similar technique can be used to build a file. The following script builds a file named *report2* which has a character vector file description in component 1 and a package containing the function *sum* and the variables *accumtab* and *∆acc* from the directory *.planfiles.report2* in component 2:

```
[1] genreport2file;accumtab;sum;tn;∆acc;.public.util.files.∆fstie | ∆fappend
[2]  )get .planfiles.report2.accumtab |filedesc | sum | ∆acc
[3]  tn←'report2' ∆fstie 0
[4]  filedesc ∆fappend tn
[5]  (⎕pack 'accumtab sum ∆acc') ∆fappend tn
[6]  ⎕untie tn
```

The file utility functions in *.public.utilfiles* are quite useful for writing scripts of this nature. The utilities are summarized below (see *Appendix B: File Utilities* for full details):

Table 8.1 File Utility Functions

Function	Description
Δ*fappend*	Appends data to a file.
Δ*fbackup*	Backs up a file by copying it to another file.
Δ*fcopy*	Copies all or specific components from one file to another, filling components if necessary.
Δ*fcreate*	Creates a file.
Δ*fcwrite*	Conditionally writes data to a particular file component (if the component doesn't already exist), appending components if the file is not long enough.
Δ*ffirst*	Drops components from a file so that a particular component number is the first in the file.
Δ*flast*	Appends or drops components so that a particular component number is the last in the file.
Δ*fmappend*	Appends data to a file a specified number of times.
Δ*fname*	Returns a file identifier in the canonical 22-character format.
Δ*fopen*	"Opens" a file; that is, the file is share-tied or, if necessary, created.
Δ*freplace*	Replaces data into a specified location.
Δ*fstie*	Share-ties a file.
Δ*ftie*	Exclusively ties a file.
Δ*fwrite*	Writes data to a particular file component, appending components if the file is not long enough.

Improving Upon the Simple Approach

The techniques presented so far work well in limited situations. LOGOS provides additional functionality that makes it possible to improve these techniques. What are the restrictions of the simple approach just described?

- The contents of an end environment must be static.

  That is, the items generated into the environment come from a fixed list which must be maintained by the developer. This list may be a constant within a script or it may be represented by the contents of a directory. The composition of the list must be arrived at through detailed analysis.

  If a function is edited so that it references a new object, care must be taken to update the list. Likewise, if all references to an object are removed from a system, the object must be manually deleted from the list or it will continue to appear in the end environment. This bookkeeping can become tedious and error-prone. As applications grow larger, this approach becomes unmanageable.

- Most real-world applications are not as structurally simple as the examples presented so far.

  Many consist of numerous interrelated workspaces and files with overlapping members. A great number of modern applications use files to page code and data into workspaces. Quite a few applications are built around a single visible function which localizes all the objects in the system and pages them in when needed. The simple techniques discussed so far do not effectively deal with these architectural issues.

- Questions such as "in which end environments is this function used?" must be answered through analysis of the scripts which generate the environments.

  This analysis is difficult and cannot answer more complex questions such as "which version of function *calc1* is in workspace *report1*?"

  It is often desirable to build testbed versions of an application where new features can be evaluated in isolation from production users. This requires the ability to retarget the results of a generation into an alternative end environment as well as facilities for identifying and processing "test" versus "production" code.

A simplistic approach does not provide the flexibility required to generate multiple end environments of this nature. The features and techniques described in the remainder of this chapter overcome these shortcomings and provide the flexibility and power required to generate real-world applications conveniently.

Clusters and the *build* Command

LOGOS includes facilities for almost totally automating the maintenance of the list of objects to be placed in an end environment. The key component in this automation is the *build* command. Given a minimal list of important objects and a working directory list, *build* will perform static calling tree analysis and deduce the names of all of the objects required by the application to any specified depth in the application calling tree.

Think about the manner in which you might generate this list yourself. You would:

- run off a WSDOC listing of all of the code you knew or even suspected might be part of the application.
- start with the top-level function or functions in the application and examine the source or more conveniently, the cross reference listing.
- record the names of the global variables and functions referenced.
- examine each of the referenced functions and add any globals it referenced to your list.
- continue in this manner until you processed the entire calling tree.

LOGOS performs its calling tree analysis in an similar manner. But instead of running off a WSDOC of all of the suspect objects, you provide *build* with a list of directories containing those objects. This is both more convenient and more powerful, since a few directory names can encompass hundreds or thousands of objects.

There are two commands in LOGOS that perform calling tree analysis: *build* and *calls*. The primary difference between the two is the form of output they generate. The *calls* command produces a character vector list of pathnames that belong in the calling tree of the object named in the argument. The *build* command actually produces a package containing the members of the calling tree. This package is called a **cluster**.

Clusters

Clusters are simply packages constructed by the LOGOS *build* command. You can direct *build* to place a cluster in any one of a number of different locations. You can have *build* place the cluster in your active workspace or in the active workspace of an auxiliary task. You can also request that a cluster be placed in a specified component of a named file. The *build* command can also write a cluster to a LOGOS paging file. Finally, *build* can save a cluster in the LOGOS file system. "Cluster" is a valid LOGOS datatype. Each of these variations, except for paging files, will be described in this section. Paging files are covered in the section, *Paging Files*.

The *build* Command

The *build* command is a very flexible command and has many options associated with it. Let's examine the syntax of the *build* command and explore some of these options. The basic syntax of *build* is:

build **[destination] [source]**

The source argument is used to provide a list of pathnames from which the cluster is to be built. If tree analysis is enabled (see the section *Tree Analysis* in this chapter), only the objects at the root of the calling tree must be included in the list. The destination argument determines where the resultant cluster is to be placed. This argument has several forms:

- If you provide a name in angle brackets as the first argument, *build* creates a package with the specified name and places it in the active workspace or the active workspace of an auxiliary task. If you omit the name in the angle brackets, the objects in the cluster are dispersed in the workspace. For example:

  ```
  build <utilpkg> .public.util.?*[f]
  build <> .public.util.?*[f]
  ```

- If you provide a pathname, LOGOS creates the cluster and saves it in the LOGOS file system with the type "cluster". Any name which includes a dot is considered a pathname; if necessary, a trailing dot may be included in the name to indicate that it is a pathname. For example:

  ```
  build .cb.src.util.utilcluster .public.util.?*[f]
  ```

- You can instruct *build* to write the resultant cluster to a file component by providing an argument consisting of a component number in angle brackets. The file name is specified through the argument to the *+file* modifier. The cluster is written to the file component as a package. For example:

  ```
  build <3> .cb.src.files.dirfns.?* +file=report2
  ```

- In some cases you may want to write the contents of a single path to a file component without placing it in a cluster (package). This can be done by providing an argument consisting of a component number without angle brackets. If the second argument yields a single object, the value of this object will be written directly to the specified component. If a cluster with more than one object is generated, the cluster is written to the file as a package. For example:

  ```
  build 1 .cb.src.files.dirfns.filedesc +file=report2
  ```

If the destination argument is a simple (non-path) name, or the symbol ⋆, it is treated as the name of a node in a LOGOS paging file. Paging is discussed in the section *LOGOS Paging*. The *build* command has a large number of modifiers. Some of these will be described in this section. For a full discussion of all of *build*'s options, refer to the *LOGOS Reference Manual*.

Tree Analysis

The *+depth* modifier controls *build*'s tree analysis capabilities. The optional argument to *+depth* must be an integer or the keyword *all*. This integer determines the depth to which the calling tree is computed. An argument of 0 or *all* stipulates that the tree is to be computed down to the leaves. An argument of 1 causes *build* to include only the objects included in the source argument. This is equivalent to eliding *+depth* altogether. If you specify *+depth* but elide its argument, the default argument is *all*.

Tree analysis works by fetching each function named in the source argument and examining it for global references. Each of these global references is searched for in your global work directories or those set with the *+workdir* modifier. If an object matching the reference is found, it is added to the cluster. If the object is a function, it is examined for global references in an identical manner. This continues until the calling tree is generated to the requested depth or no unresolved references remain.

It is important to understand how LOGOS identifies global references. LOGOS considers any name within the body of a function which is not part of a comment or a quoted string an identifier. For an identifier to be considered global, it must not appear in the header of the function or be used as a label. There are circumstances in which tree analysis is unable to detect a reference. A name appearing in a quoted string may be a legitimate global reference if the string is used as an argument to ⍎ or ⎕*pdef*. A "hidden" name such as this can cause a discontinuity in the calling tree LOGOS generates.

Fortunately, the system provides a means for dealing with these identifiers. You can alert LOGOS to their presence by declaring them in a ⍝∇∪ code tag. Example:

```
[2] ⍎('flag',⍕n),'←bflg∧sum' ⍝∇∪ flag0 flag1 flag2 flag3
```

The code tag on this line will cause LOGOS to act as if *flag0, flag1, flag2,* and *flag3,* appeared outside comments and quotes on line 2. Note that other facilities, such as the *xref* command, the *calls* command, and WSDOC also recognize these references. It is a good idea to acquire the habit of adding a ⍝∇∪ code tag to the end of any line of code you write which contains a hidden reference. This includes references passed in name lists to system functions such as ⎕*svo* and ⎕*fd*. Variants of this code tag make it possible to create references to names stored in tables or in LOGOS paths (see the section *Using Code Tags* in *Chapter 10: Using the Compiler* or the *LOGOS Reference Manual* for details.

Using the *build* Command

The following script is a modified version of the script used earlier to generate a workspace called *report1*. This script is almost identical, except for the use of *build* rather than *get*. Note that the argument list to *build* includes only one object, that tree analysis is turned on with the *+depth* modifier, and that *build* supports a *+task* modifier. In this simple example, tree analysis removes the need to remember the names of the handful of objects called by *runrep1*. In a real-world application, the savings could be considerably larger.

```
[1] z←genreport1ws
[2] )signon +task=genreport1
[3] )environment workdir .plan.wss.report1 +stack
[4] )build runrep1 +depth +task=genreport1
[5] )z←send )save report1 +task=genreport1
[6] )send )off +task=genreport1
```

Note that tree analysis has another important benefit: even if significant changes are made to the calling structure of the application the script will continue to work without modification. Using *build* to generate a file is also straightforward. Here is a modified version of the script developed earlier:

```
[1] genreport2file
[2] )environment workdir .plan.files.report2 +stack
[3] )build +file=report2
[4] )build 1 filedesc
[5] )build 2 sum +depth
```

Note that this script is considerably simpler than the original version. LOGOS takes care of several tasks that the original script had to deal with manually: tying the file, determining the ancillary objects called by *sum*, and updating the file. Note the use of *build* with no arguments on line 3. This is a special feature: when *build* is invoked with only modifiers and no arguments, it establishes those modifiers as default settings. Setting the file name on line 3 relieves the script of the responsibility of specifying a file name on subsequent lines that call the *build* command.

Audit Files

While the *build* command addresses many of the shortcomings of the simplified approach discussed earlier, it leaves several problems unresolved. The tools discussed so far do not help with questions about the use of objects within application environments. Clearly, to provide answers to such questions, some kind of database containing object usage information must be established. LOGOS provides these databases in the form of audit files.

An **audit file** is an APL file in which LOGOS records information about the placement of objects within end environments. Each audit file contains a series of individual **audit records**. Each record describes a generation of the application and contains a list of the end environments built, the objects placed within them, and, in some cases, their interrelationship.

Information is kept on the way objects were compiled, and on the version of each object used. Symbol table information is kept for functions, indicating which objects are the results, left arguments, right arguments, locals, labels, and so on. Time, date, and alias information is kept when audit file records are created. Multiple instances of a single object are tracked by audit files. An instance is defined as a unique combination of version number and compilation directives.

For example, suppose you have two versions of a function named *scan*. The first version is used in a workspace in an uncompiled form. The second version is in two different paging files. In one file, *scan*[2] is decommented, and in another file, *scan*[2] is decommented and diamondized. An audit file would track three unique instances of *scan*: *scan*[1], *scan*[2] decommented, and *scan*[2] decommented and diamondized.

For every pathname involved in the application, the audit file makes possible the rapid determination of where it is used, and what version was picked up during a particular generation. The *references* command, described in the section *Registering Out Objects* in *Chapter 11: Maintaining System,* reports on audit file information and can be used to analyze a system generation.

Audit files are an important record-keeping tool when they accompany the generation of a large system. They contain extensive usage information beyond that which LOGOS keeps in its file system, and they allow parts of a system to be regenerated without rebuilding the entire application. When an audit file has been used, the *distribute* command can place changed objects into the specific end environments in which they are used (see the section *Distributing Changes* in *Chapter 11: Maintaining Systems* for details on the *distribute* command).

Specifying Audit Files

Commands such as *build*, which have the propensity to update audit files, have a *+audit* modifier. You specify an audit file by providing a file name as the argument to the *+audit* modifier. You can use the fact that *build* sets defaults if it is invoked with no arguments to establish an audit file for the duration of a script. Example:

```
[4] )build +audit=8221210 cbaudit +depth
```

This line sets the default audit file to *8221210 cbaudit* and turns on calling tree analysis with infinite depth (remember that *+depth* is equivalent to *+depth=all*). If the audit file does not exist, LOGOS automatically creates it for you, providing the command is executed on the file owner's account.

Using and Maintaining Audit Files

Despite the fact that audit files are stored in SHARP APL files, you should never access one directly from APL; always use LOGOS commands or scripts to access audit files. Tampering with audit files can lead to trouble. To help you avoid accidentally damaging an audit file, LOGOS always uses a passnumber when creating or accessing an audit file. If it becomes necessary to erase an audit file, you may do so by tying the file yourself outside of a LOGOS session, then erasing it.

Chapter 11: Maintaining Systems describes how you can use the *distribute* command in conjunction with audit files to update end environments. The *LOGOS Reference Manual* documents the audit file utility scripts available in the public script library. Some of the tasks you can perform using these scripts are:

- display formatted reports containing detailed information from audit files
- delete references to obsolete objects from audit files
- share audit files with other users
- generate end-environment/pathname cross reference listings

The *wssave* Command

The simple examples presented so far have used the *)save* system command to save workspaces. LOGOS provides a more convenient and powerful alternative to *)save*: the *wssave* command. This command saves the active workspace using a specified name. Example:

```
u wssave inventory
 1989-04-12 11:61:10 inventory
```

If you call *wssave* without an argument, the workspace will be saved under its current name. When using this method, remember that you're in LOGOS, and that the *logos* function will be present in the saved workspace. This can be avoided by running the generation in an auxiliary task. The *wssave* command maintains reference entries for the workspace, noting the functions and variables from LOGOS used to build it. The *references* command (see *Chapter 11: Maintaining Systems*) can then easily track the use of the objects.

Audit files will retain even more information about the placing of objects; the *+audit* modifier to the command specifies that an audit file should be used to record information about a saved workspace generation. Use of the *+user=* modifier with the *wssave* command will save an active workspace in another user's workspace library. The owner's alias (or user number) and his password must be supplied. For example:

```
u wssave '501 budget' +user=bmaint:jellybean
1989-04-21 10:06:10 501 budget
```

Here, the active workspace is saved under the name *501 budget*. The name must be enclosed in quotes to be treated as a single argument. The workspace owner is the user number associated with the LOGOS alias *bmaint*, whose signon password is *jellybean*. If the password is needed but not provided, LOGOS will issue a secure prompt for it. When you use a script to generate a workspace, you may specify the *+information* modifier to *wssave*. The newly saved workspace will then contain a variable called *⍙LINFO* which holds the name of the script.

For example:

```
ᵘ locate wssave .bmaint.cb.maint.genbudws +display
.bmaint.cb.maint.genbudws[s14] (1 occurrence)
)wssave '501 budget' +user=invent +information
1 occurrence found.
ᵘ .bmaint.cb.maint.genbudws
generating budget system
password:
system generated
ᵘ exit
      )load 501 budget
saved 1986-02-14 14:01:37
      ∆LINFO
.bmaint.cb.maint.genbudws
```

The benefit of using the *+information* modifier is that the *∆LINFO* variable it generates saves maintenance time by documenting the name of the generating script. The *wssave* command supports auxiliary tasks through a *+task* modifier. Using *wssave* with an auxiliary task is the preferred method of saving a workspace, since it allows workspace construction to begin with a truly clear workspace, free from the overhead incurred by the LOGOS function. (For more information, see the checklist item "If you are generating a workspace, will you need an auxiliary task?" in the section *Generation Script Checklist.*)

The *wsbackup* Script

This script, located in the LOGOS directory *.public.logos.cmds*, backs up a saved workspace:

```
ᵘ wsbackup utilities oldutils
saved 1989-02-01 09:29:17
1989-01-13 11:22:14 oldutils
```

In this example, the current version of the workspace *utilities* is saved as *oldutils.* The *wsbackup* script starts an S-task which loads the specified workspace (without triggering the latent expression, *⎕lx)* and saves it with the new name. You may use *wsbackup* to back up a workspace owned by another user if you specify that user's primary alias or user number and signon password. If you omit the password, LOGOS will issue a secure prompt for it. Now the original workspace can be modified safely:

```
ᵘ get utilities general.⎕??
ᵘ wssave utilities
1989-02-14 14:34:08 utilities
```

To illustrate the utility of this script, suppose a copy of the budget system had been saved before regenerating it:

```
∪ wsbackup '501 budget' cb14feb bmaint:jellybean
saved 1989-02-13 19:44:08
1989-02-14 13:07:29 cb14feb
```

Now, if the budget workspace which was subsequently generated had to be backed off, the workspace *cb14feb* could be used:

```
      )load 501 budget
saved 1989-02-13 19:44:08
value error
Qlx[1] ⍎
       ∧
      )load 1 logos
⋆logos⋆ r2.0
∪ wsbackup '501 budget' badcheck bmaint:jellybean
saved 1989-02-13 19:44:08
1989-02-14 14:06:27 badcheck
∪ wsbackup cb14feb '501 budget' bmaint:jellybean
saved 1989-02-14 13:07:29
1989-02-14 14:06:43 501 budget
```

The first call to *wsbackup* saves the defective copy of budget as *badcheck* for later investigation. The second call saves the backup copy as the primary copy.

The *genbudws* script is a good example of a script that uses most of the workspace generation features discussed here:

```
∪ display .bmaint.cb.maint.genbudws
.bmaint.cb.maint.genbudws[s1] :
[1]  genbudws (+production) ;audit;wsid;savews
[2]  ⍝ generate the budget system workspace
[3]  ⍝
[4]  ⍝ if +production is specified, a production system is
[5]  ⍝ generated, otherwise a test system is generated.
[6]  ⍝
[7]  ⍝ step 1: initiate an auxiliary task
[8]  ⍝
[9]  )signon bmaint +task=genbudws
[10] ⍝
[11] ⍝ step 2: Establish working directories and
[12] ⍝        defaults based on test/production status
[13] ⍝
[14] →production⍴prod
[15] )environment workdir .bmaint.budget.src.dev  .bmaint.budget.src.util|input|report +stack ⍝ test
     workdirs
[16] audit←'2231840 tbudaudit' ⍝ test audit file
[17] wsid←'2231840 cbdev' ⍝ test wsid
[18] )build +audit=(⍎audit) +compile=e,p +depth
[19] →l0
[20] prod:)environment workdir .bmaint.budget.src.util|input|report +stack ⍝ production workdirs
[21] audit←'2231840 pbudaudit' ⍝ production audit file
[22] wsid←'501 budget' ⍝ production wsid
[23] )build +audit=(⍎audit) +compile=d,e,r,p,x +depth
[24] ⍝
[25] ⍝ step 3: back up the production budget workspace
[26] ⍝
[27] savews←wsid,, 'zi2'⎕fmt ⎕ts[1 2 ⍴ ⎕io+1 2]
[28] )wsbackup (⍎wsid) (⍎savews) bmaint
[29] ⍝
[30] l0:⍝ step 4: fetch objects in calling tree of root function <startbud>
[31] ⍝
[32] )build <> startbud +task=genbudws
[33] ⍝
[34] ⍝ step 5: save the workspace
[35] ⍝
[36] )wssave (⍎wsid) +audit=(⍎audit) +task=genbudws
[37] )send )off +task=genbudws
```

Note that this script uses a *+production* modifier to choose between two sets of workspace configuration data. The workspace name, audit file, compilation options, and working directory list used to generate the file are all determined by the absence or presence of this modifier when the script is invoked.

Building Files

The reasons for using files as part of a large APL system were discussed in the introduction to this chapter. Examples of data which are better stored on file are:

- data used by the system (for example, a directory of the system's users)
- data written or read by the system's users
- functions and constants that can be materialized in the active workspace.

  A file used for this purpose is often called a program file. The program file may be built from objects stored in LOGOS, or elsewhere.

LOGOS supports a special kind of program file called a **paging file**. These files are discussed in detail in the section *Paging Files - A New Environment.* An example was given earlier to demonstrate a simple way of building a file with a LOGOS script. Here is that sample script once again:

```
[1] genreport2file;accumtab;sum;tn;Δacc;.public.utilfiles.Δfstie | Δfappend
[2] )get .plan.files.report2.accumtab |filedesc | sum | Δacc
[3] tn← 'report2' Δfstie 0
[3] filedesc Δfappend tn
[4] (⎕pack 'accumtab sum Δacc') Δfappend tn
[5] ⎕untie tn
```

Using the file-generating capabilities of *build*, this can be rewritten as:

```
[1] genreport2file
[2] )environment workdir .plan.files.report2 +stack
[3] )build +file=report2
[3] )build 1 filedesc
[4] )build 2 sum +depth
```

Note that *build* simplifies matters noticeably. It is no longer necessary, for instance, to localize the list of objects or define them in the workspace. Nor is it required that the script know the name of every object in the calling tree of *sum*, *build*'s calling tree analysis takes care of this automatically.

Generating Foreign Paging Files

Sometimes it is desirable to move a paging-based system into LOGOS without changing the paging mechanism. It may be, for example, that the system is being migrated into LOGOS gradually, and that a switch to LOGOS paging is not possible until the entire system has been migrated. This problem is usually solved easily, since generation scripts can be written to generate any file format. As an illustration, suppose an application is maintained using a paging system that uses a file with the following format:

Component	Contents
1	File format description
2	Paging directory
3	Package containing system start-up code
4-10	Unused
11- n	Packages containing objects to be paged

Further suppose that the paging directory is a complex data structure, and that the function *builddir*, which is part of the maintenance system, builds this directory by scanning the file. Now suppose that it is necessary to move this application into LOGOS while maintaining the existing paging mechanism. Given that the file structure is well-documented, the only additional information that must be gathered is a list of names of the root functions of each package. Presumably, the mechanism used to generate the file in the foreign system contains this information.

With this information in hand, the following script can be written:

```
[1]  genfpfile (+production);tn;.fpsys.maintfile.builddir;.public.utilfiles.∆fstie | ∆freplace
[2]  ⍝ generate foreign paging file
[3]  ⍝
[4]  ⍝ step 1: Establish working directories and
[5]  ⍝        defaults
[6]  ⍝
[7]  )environment workdir .fpsys.app.src.util | drivers | ask | plot | calc
[8]  )build +audit=2231840fpaudit +compile=e,p +depth +file=2231840fpage
[9]  ⍝
[10] ⍝ step 2: put the file description variable in component 1
[11] ⍝
[12] )build 1 filedesc
[13] ⍝
[14] ⍝ step 2: build each package using calling tree analysis
[15] ⍝
[16] )build <3> start
[17] )build <11> init∆task
[18] )build <12> scan
[19] )build <13> plot
[20] )build <14> select∆driver
[21] )build <15> menu
[22] )build <16> shutdown
[23] ⍝
[24] ⍝ step 3: build a directory and place it in component 2
[25] ⍝
[26]  tn←'2231840fpage' ∆fstie 0
[27]  (builddir tn) ∆freplace tn,2
```

Shells

A **shell** is an APL function that defines a package or a cluster from a file and executes the root function of the cluster or some expression that you specify. The objects that have been defined from the file are localized in the header of the shell, so when the work of the shell function is complete, the functions and variables used in the shell disappear from the workspace. The source may be a package that resides in an ordinary APL file, or it may be a node of a LOGOS paging file. The sections following *Arguments to shell* describe the use of shells with packages that are components of an ordinary APL file. Using shells with LOGOS paging files is discussed later, in the section *Using Shells with Pagefile Nodes.*

Shells are useful for two different kinds of systems. In an open system with a finite (if large) set of commands, establishing these commands as shells creates the illusion of a clean, workspace-resident system. The *)fns* system command displays the names of the shells, but not of their many subfunctions. Shells are also useful at the top of a large closed system that you must be able to copy. LOGOS is itself such a system, and the *logos* function is a shell.

Shells are a very efficient way to encapsulate all or a portion of an application, because the analysis required to build them is performed at the time you generate them, not at the time the application is run. The *shell* command is used to build a shell and to place it in a workspace or a file. The resulting shell can include all subfunctions and variables that are part of the package or LOGOS paging file node around which the shell is built.

Arguments to *shell*

The *shell* command takes two arguments, **destination** and **source**. The source may be either a LOGOS paging file node, or it may be a package in a component of an ordinary APL file. The valid destinations depend on the source, as shown in the following table:

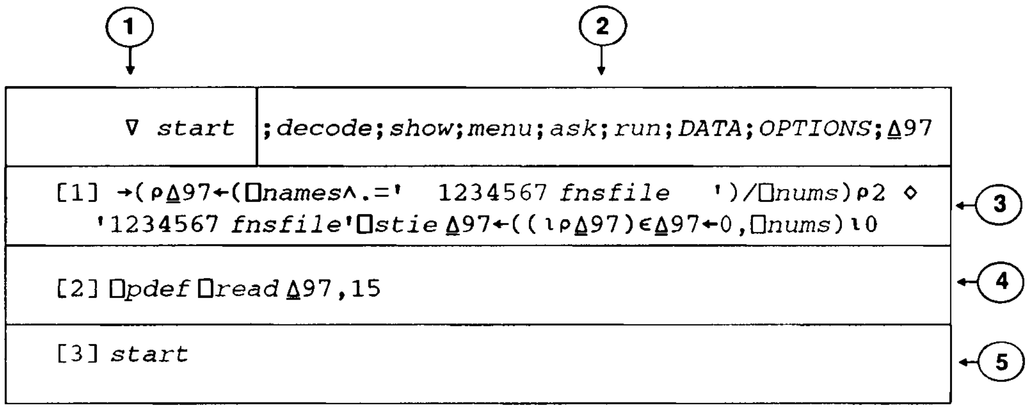
Table 8.2 *shell* Command Sources and Destinations

If the source is:	The destination may be:	
a paging file node	the same node	The source and destination node names must be the same. In fact, only the destination need be specified; the source will be assumed. Use *+name* if you want to give the shell function a different name from that of the node's root function. This combination must be used in conjunction with the *build* and *filesave* commands that create the node.
an APL file component	<>	The shell function will be placed into the workspace - either your active workspace, or in that of an auxiliary task. Requires *+name* to provide the name of the shell function, unless *+skeleton* is specified. Requires *+file* to specify the name of the source file.
	another component number	When the destination is another component number, the shell function will be placed into that component. The two numbers must be different. Requires *+name* to provide the name of the shell function, unless *+skeleton* is specified. Requires *+file* to specify the name of the file that is both the source of the package and the destination for the shell.
	a LOGOS pathname	If the destination is a LOGOS pathname, the shell function is stored in that path. Requires *+name* to provide the name of the shell function, and this name must match the terminal segment of the pathname. Requires *+file* to specify the source file.

Anatomy of a Shell Around a Package

Figure 8.2 shows the definition of a shell created around a package stored in component 15 of an APL file, divided into its individual segments.

Figure 8.2 The Parts of a Shell



```
       ∇ start  ;decode;show;menu;ask;run;DATA;OPTIONS;⍙97
[1] →(ρ⍙97←(⎕names∧.=' 1234567 fnsfile ')/⎕nums)ρ2 ⋄
   '1234567 fnsfile'⎕stie ⍙97←((ιρ⍙97)∊⍙97←0,⎕nums)ι0
[2] ⎕pdef ⎕read ⍙97,15
[3] start
```

1 - Function name
2 - Locals list, objects in the package
3 - File tie logic
4 - Page in statement
5 - Execution of root object

This shell could have been generated by the following command:

ᵁ *shell <> 15 +file=fnsfile +name=start*

Part 1 is the shell function's name and is taken from your argument to the *+name* modifier (in this example, *start*). It is important to note that shell functions built around packages from an ordinary APL file assume by default that the root function in that package (your argument to *+name*) does not take any arguments and does not return a result. If you want to create a shell function that returns a result, or accepts arguments, then use *+skeleton* to provide the pathname of a function with the appropriate syntax and definition to which *shell* can add the locals list.

Part 2 is the computed locals list. In its simplest form, it is the names of all the objects that were in the package at the time the shell function was created. If you alter the contents of this package at some later time, you will need to recreate the shell function so it can reflect those changes. When you create the shell function, you may use the *+header* modifier to add names to the locals list that are not in the package, or to remove names from the locals list that are part of the package.

Part 3 is the logic that either ties the file or determines the tie number if it is already tied. The local variable created in this step, ∆97, is automatically localized in the shell function header.

Part 4 defines the contents of the package from the component specified as the source when the *shell* command was executed.

Part 5 executes the root function of the package. By default, this is taken to be the same as the argument provided to the *+name* modifier, so it matches the shell function's name. If you wish to specify a different name, or some APL expression to be executed instead, provide that name or expression as the argument to the *+qlx* modifier and it will appear in part 5 instead.

Options with *shell* Used on Packages

In addition to the *+name* and *+file* modifiers that are required when building shells around packages in an APL file, several other modifiers may also be used in that situation.

+compile

With *+compile*, you can specify compilation directives to be applied to the pathname provided as an argument to the *+skeleton* modifier. See below for more information on using *+skeleton*.

+header

Use *+header* to alter the locals list of the shell function. There may be some objects in the source package whose names you do not want localized in the shell's header. Use, for example, *+header=/describe driver* to remove the names *describe* and *driver* from the shell header. Similarly, there may be objects that are not a part of the package whose names you want to have added to the shell's locals. Then use, for example, *+header=,name who* to add the objects *name* and *who* to the list of locals. These may be combined in the same argument:

+header=,name who/describe driver

+lock

If the file in which the package resides requires a passnumber for access, specify it as the argument to the *+lock* modifier.

+qlx

Normally, the startup function in the package is assumed to be the same as the name used for the shell. If you want to provide a different startup function name, or simply an APL expression to be executed by the shell, use the alternate name or the expression as the argument to this modifier. This modifier is ignored if *+skeleton* is specified.

+skeleton

The argument to *+skeleton* must be the pathname of a function whose definition forms the complete shell function. File tie logic, page-in logic and execution of the correct root function must all be included in the skeleton function. When *+skeleton* is specified, the *shell* command only adds the names of objects in the package to any locals already specified in the skeleton function's header. Use a skeleton function if you want to build a shell that accepts arguments or returns results. The skeleton function is subject to compilation according to any directives used as arguments to the *+compile* modifier.

+task If the destination of the shell is specified as <>, meaning to place the shell in the active workspace, you can direct LOGOS to put the resulting shell function into the workspace of an auxiliary task by using this modifier.

+variant The only variant that may be used when building shells around packages from an APL file is the *e* variant. This causes the shell function to be created with ⎕*ec* localized in its header. This is ignored if *+skeleton* is specified.

Paging Files - A New End Environment

As APL applications have grown more ambitious in scope over the years, the resources of the APL workspace have been strained. One approach to this limitation has been to provide virtual workspaces that can be stretched to accommodate expanding applications. While this solution requires a minimum of preparation to use, it often has unfortunate consequences with respect to performance because virtual workspaces are a brute-force approach to the problem. At any given time, it is quite likely that a large part of the workspace is consumed by objects that are not required by the process in execution.

Another approach to this problem is paging. Paged systems store organized collections of related objects in files and bring these collections into and out of the workspace as required. Objects are maintained in the workspace only when it makes sense for them to be there. We have seen how generation scripts can be used to populate arbitrary file structures from the contents of a LOGOS hierarchy. LOGOS also provides native support for an efficient paging system of its own. You can use this paging system to create paged systems from your own applications. This section describes paging in detail and demonstrates how the paging file end environment is built and maintained.

Paging File Structure

A LOGOS paging file is an ordinary SHARP APL file distinguished only by the nature of its contents. It may have any valid file name you find appropriate and any access matrix you wish to establish. Paging requires only ⎕*read* permission to the file. For the purposes of operating as a LOGOS paging file, it is divided into **paging areas.** Each paging area is further subdivided into **nodes** (or **pages**).

Paging Areas

A paging area is a set of contiguous components in a LOGOS paging file. An area is referred to by its starting component number within the paging file, as in *cbpage 40.* A paging file may contain as many areas as you wish, and those areas may be used by entirely different applications. In the application's workspace, a particular paging area is selected by using the *∆lpagefile* function, by executing, for example, *∆lpagefile*'*cbpage 40*'.

The first component of each paging area contains information describing the area. The second component is the **base node**. The third and following components in a paging area contain the rest of the nodes.

Nodes

Each paging area within a LOGOS paging file is divided into nodes. A node is a collection of objects from the LOGOS hierarchy and is created as a result of using the *build* and *filesave* commands. Nodes are referred to by name, which usually corresponds to the name of the function used as the root for the *build* command that created the node. Each paging area may have a special node called the **base node** (or **base page**). The contents of this node are brought into the application's workspace when the pagefile is initialized with the *Δlpagefile* function. The contents of this node are never paged out of the workspace. See the section *Generating a Paged System* for more information on building nodes and the base page.

Paging Concepts

What Is Paging?

The fundamental idea behind paging is simple: instead of storing all of the parts of an application in the active workspace simultaneously, objects are maintained in secondary storage (the paging file) and brought into the workspace as they are needed. When an object is no longer required, it is removed from the workspace, freeing space for other objects. During paging, objects move from a paging file into a workspace. LOGOS allows you to define multiple **paging areas** within a single file.

Every paging area has a special page (the first one), called the **base page**, associated with it. This page contains objects that must remain in the workspace at all times. The top-level functions and global variables that drive a system are usually stored in the base page.

It is important that the paging process be an efficient one. It is also important that the paging mechanism be transparent as possible, so that only minimal effort will be needed to set up or convert a paged application. The tools provided with LOGOS help the LOGOS paging system achieve both of these goals. The paging utilities are located in the directory *.public.logos.paging*; these functions are summarized later in this section, and described fully in *Appendix C*.

Page-in

There are two types of paging: **demand paging** and **request paging**. The basic difference between these types is the manner in which a paging event is triggered. As its name implies, demand paging is demand-driven; paging takes place when a type '*6 i*' event trap is triggered by a value error in the application code. The expression invoked by the trap calls the *Δlpagein* utility. *Δlpagein* searches the paging area for a node whose name matches the object that triggered the value error. If such a node is found, then its contents are materialized within the workspace. As the '*i*' event trap action code dictates, execution resumes at the point where the value error occurred. This is completely transparent to the user and makes it appear as if the missing object was in the workspace all along. Note that a workspace running a demand-paged system might initially contain only a single function (the LOGOS paging initialization utility) invoked from ⎕*lx*.

Request paging is less spontaneous, occurring only when a call to Δ*lpagein* is deliberately made by the application. A request paging system might have a structure, for example, where an outer-level menu program waits for a user to select an item from the menu. When a choice is made, the program would call the paging utility to bring in the relevant collection of objects and then execute the appropriate function.

Note that, in general, the adoption of demand paging requires fewer changes to existing code, since missing nodes can be brought in automatically through the action of the trap. Request paging, on the other hand, necessitates the insertion of calls to Δ*lpagein* into application code at critical points. Usually, the only code that must be added to an application to utilize demand paging is the initialization procedure. An exception to this is an application that uses event traps that interfere with the global '*6 i*' trap that demand paging employs.

Page-out

The other critical facet of a paging system is the **page-out mechanism**. It does little good to conserve space by storing objects outside the workspace, bringing them in only when required, if there is no mechanism to dispose of these objects when they are no longer needed. The function Δ*lpageout* handles page-out activities. There are several circumstances that might warrant the paging out of one or more nodes:

- The amount of free workspace in the application has fallen below a critical level.
- An operation which requires as much free workspace as possible is about to be executed.
- A "workspace full" event has occurred. Δ*lpageout* can handle each of these circumstances.

As with page-in, page-out may be demand driven based on the occurrence of a workspace full or insufficient free workspace, or it may be request driven by explicit calls to Δ*lpageout*.

Designing a Paged System

When you design a system to use paging, there are three operational characteristics to consider: What paging areas are to be selected, and in what order? What nodes are to be paged in, and when? What nodes are to be paged out, and by what criteria?

Selection of Paging Areas

The selection of paging areas is done with the Δ*lpagefile* function, usually at the time the application is initialized. An application can use the selection of paging areas or their nodes to control its behavior. For example, you can explicitly page in one node or select one paging area. Alternately, you can establish an implicit precedence of nodes by selecting more than one paging area, so that nodes in the latter areas are shadowed by nodes of the same name in the former.

Explicit Selection

Explicit selection is useful when you have a repertoire of similar nodes, only one of which is to be selected. For instance, a system might use explicit selection to bring in the appropriate terminal driver for a particular session, where the system can be certain which is to be used. Suppose you have a different display function in each of several nodes. One of these nodes is paged in, based upon a known or computed terminal type:

```
Δlpagefile '501 budgetpage'
Δlpagein Termtyp
```

Instead of just paging in one node, a system might select a terminal-dependent paging area at system startup:

```
[1] ⍝ system startup function
...
...
...
[7] ⍝ select paging area
[8] Δlpagefile '501 inventory' ,⍕ 500 600 700 800
    500[('typewriter'⊃'hds108'⊃'ap124'⊃ 'ibm3279')⍳
Termtype]
```

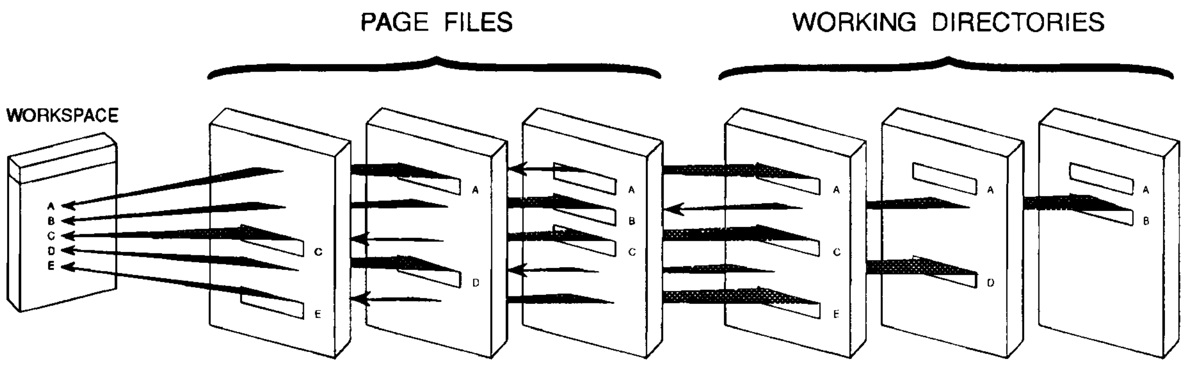
This selects a paging area beginning at component 500, 600, 700, or 800, depending on the terminal type. The base node of the selected paging area, which is materialized by *Δlpagefile*, may contain all the needed objects; if so, a separate *Δlpagein* isn't required.

Implicit Selection

Implicit selection allows a system to fully customize its behavior for the user. Through selection of paging areas, radically different objects can be materialized for different users. One user might receive the production software; another a semi-public test version; and still another a private development version. The choice affects only the initial *Δlpagefile* invocation, and not any other part of the system. The rest of the system need not be aware that paging is occurring, or from where. The paging files themselves might be built by overlaying multiple working directories, as shown in Figure 8-3.

Here, the directories overlay to produce one or more paging files during the building of the system. When a user starts the application, multiple paging files overlay to produce a unique selection of objects to be materialized in the workspace. Paging files follow the same ordered precedence scheme as working directories, in that those which are specified first are used first, and objects in earlier paging files shadow the same objects in later ones. This layering makes it possible to control what objects a specific user sees.

Figure 8.3 Overlaying Paging files and Working Directories



A fragment of an application's initialization routine below illustrates what is involved in specifying multiple paging areas:

```
[1] ⍝ system startup function . . .
. . .
. . .
[8] Δlpagefile (dev,test,1)/'501 inventoryd 100'⊃'501 inventory 1100'⊃'501 inventory 100'
```

dev and *test* are two flags, which the program has presumably computed. If the development flag *dev* is on, any required node is first sought in the **development area** (components 100 and onward of the file *501 inventoryd*). If the test flag *test* is on, any required node is sought in the **test area** (components 1100 and onward of the file *501 inventory*).

Finally, nodes are always sought in the **production area** (components 100 and onward of the file *501 inventory*). With a system like this, the developer need put only those objects he's working on in the development and test areas. He can do a complete integration test at the same time as his development testing, and he can allow other users to access his test system. Moreover, the user of the production version doesn't suffer any performance degradation or instability, because he isn't affected by any unreleased code.

Page-In of Nodes

Demand Paging

Demand paging is based on the premise that the commands being entered by the user are arbitrarily chosen from the repertoire of operations in the application. To use demand paging, you need a global value error trap that executes *∆lpagein* to resolve the error in the immediate environment:

```
⎕trap←' ∘ 6 i ∆lpagein ⎕er[⎕io+3;] '
```

Request Paging

Request paging is based on at least some advance knowledge of the likely order of processing. In a request-paged system, a single user command usually triggers a requirement for processing of many modules of the application, which may be brought in all at once. Request paging is particularly applicable to closed, prompting systems where the issuing of one command is statistically unrelated to, or biases against, subsequent issuing of the same command. However, even demand paging can be readily applied to such systems.

To use request paging, your application need invoke only the *∆lpagein* function with specific node names; the application is expected to know (or compute) the nodes to page in. For example, your system might proceed through a series of operations, one after another, following a well-defined and invariant procedure. As it moves through these operations, it might page in the required nodes explicitly, and then reference a function brought in with the node:

```
[1] ⍝ workspace documentation system -- main driver. . .
. . .
. . .
[35] ∆lpageout 'sums' ⍝ finished with summaries
[36] ∆lpagein 'defs' ⍝ object definitions
[37] ∆fndef ⍝ contained in <defs>
[38] ∆vardef ⍝ also contained in <defs>
```

Alternatively, several commands in your system might be invoked in the same way, but require different programs to execute them. Here, you can use request paging to select the command node:

```
[13] ⍝ now, <CMD> is validated command name.
[14] Δlpagein 'CMD',cmd ⍝ page in command node
[15] doit ⍝ execute main function -- which differs according to the node paged in
```

Page-out of Nodes

Objects can be paged out explicitly by the *Δlpageout* utility, or implicitly by the page-in mechanism when ⎕*wa* has fallen below a certain threshold. Although we talk of "paging out", the objects that make up paged-out nodes are in fact expunged from the active workspace.

Consequently, suspended or pendant functions cannot be "paged out". An alternative technique for removing objects from a workspace is to use shells. With a shell, the contents of a node are defined locally to the root function of the node. When the function ends, everything local to it disappears, leaving only the shell itself. The shell can be paged out as an ordinary object. The contents of the base node are never paged out.

Explicit Page-out

Some systems -- typically closed systems that use request paging -- page out unneeded nodes deterministically. Before the system pages in a node for the next operation, it pages out the node from the previous one; this is illustrated in the example in the preceding section. Generally, explicit page-out is used with explicit page-in (request paging). A system might use explicit page-out to expunge objects that would confuse a user issuing the *)fns* command. Explicit page-out might also be used to avoid name conflicts.

Implicit Page-out

You may, if you prefer, leave page-out to LOGOS. In a system that uses demand paging, implicit page-out is really the only viable choice, but you can use this form of page-out with request paging as well. Implicit page-out can be triggered in two ways: when your application pages a node into a fairly full workspace, or when your application encounters a *ws full*. What constitutes "fairly full?" Every time a node is paged in, LOGOS automatically pages out some nodes if the available storage is less than a value known as the **trigger threshold**. Page-out continues until the available storage rises above a second value, known as the **storage margin**.

Both of these values are attributes of your application; you can set these attributes with the utility function ∆*lpageset*. Settings of less than 1.00 are interpreted as a percentage of maximum workspace size. If not explicitly set, values for these parameters are assumed by the LOGOS paging utilities. By default, the trigger threshold is one third of the maximum workspace size; the storage margin is one half of the maximum workspace size.

If you condition your application appropriately, page-out can also occur when the application encounters a *ws full*. To enable this, you need a global trap for *ws full* that executes →∆*lpwsfull* 0:

```
⎕trap←'∘ 1 e →∆lpwsfull 0'
```

A typical demand-paged system combines both the *value error* and *ws full* traps, giving:

```
⎕TRAP←'∘ 6 i ∆lpagein ⎕er[⎕io+3;] ∘ 1 e →∆lpwsfull 0'
```

Any other traps you need can be appended to the end of this set. By default, LOGOS also keeps track of the reference count for each object it has paged in. If a node that contains a particular object is paged out, the object is nonetheless exempt from page-out until its reference count falls to 0.

You can use ∆*lpageset* to disable tracking of reference counts. Reference counts are important if any of your nodes has objects in common with any other. Without these counts, paging out a node might expunge an object expected to be present by another node that is already paged in. With them, only when all nodes that use an object have been paged out, does the object actually disappear.

Heuristics for Implicit Page-out

There are a number of theoretical approaches to page-out heuristics. An application using LOGOS to perform implicit page-out may choose from least recently used, least recently paged, largest objects first, or paging priority.

Least Recently Used (LRU)

A node is increasingly liable to page-out as the real-time interval since it was last used increases. **LRU paging** requires setting the *q* compilation directive for each function subject to it. This option causes LOGOS to introduce a subroutine call at the start of the first unlabelled line of the program. The subroutine efficiently maintains information on the time the function was last called. LRU paging is capable of noting only the call to a function, not the exit from it. Consequently, the algorithm may not perform optimally if a long period of real time is spent in a particular function.

Least Recently Paged (LRP)

A node is increasingly liable to page-out as the real-time interval since it was paged in increases. **LRP paging** is less complicated than LRU, but is also less optimal because time of page-in and time of last use may be very different.

Largest Objects First (LOF)

A node is increasingly liable to page-out as the amount of workspace it occupies increases relative to that of other pageable objects in the workspace. **LOF paging** is the simplest scheme, but the likeliest to cause excessive page movement.

Paging Priority

A node is increasingly liable to page-out as its priority decreases relative to that of other pageable nodes in the workspace. A node's priority is set with the *+keepin* modifier to the *build* command.

When the page-out logic is activated, the order in which nodes are expunged is based upon some or all of the following factors, in decreasing order of importance:

- Paging priority
- Time of last use
- Node size
- Time paged in

If priority has not been set for any of the nodes, it has no effect on page-out liability. If the *q* compilation directive has not been specified for any function, time of last use has no effect on page-out liability.

Paging Utilities

The LOGOS directory *.public.logos.paging* contains the paging utility functions. Table 8-2 presents a brief synopsis of each. See *Appendix C: Paging Utilities* for a more detailed description.

Table 8.3 Paging Utility Functions

Functions	**Description**
Δ*lfnstop*	Sets stop controls on paged functions.
Δ*lfntrace*	Sets trace controls on paged functions.
Δ*lkeepin*	Prohibits page-out of specified nodes.
Δ*lpage*	Returns a specified node as a package.
Δ*lpagefile*	Initializes paging from a specified area.
Δ*lpagein*	Pages a specified node into the workspace.
Δ*lpageout*	Expunges a specified node from the workspace.
Δ*lpageset*	Defines paging attributes.
Δ*lpagestop*	Sets stop controls on specified nodes.
Δ*lpagetrace*	Sets trace controls on specified nodes.
Δ*lpcmprs*	A subfunction used by Δ*lpagein* and Δ*lpageout*. You must insure this function is in the workspace if you intend to call Δ*lpagein* or Δ*lpageout*, but you should not invoke this function directly.
Δ*lprestart*	Reties paging files after an interrupted session resumes.
Δ*lpwsfull*	Expunge nodes from workspace based on page-out heuristics in effect.

Structuring a Paged System

The design of a paging area is similar, in some ways, to the design of a LOGOS hierarchy: both involve the grouping of APL objects into logically related collections. However, unlike the hierarchical structure of the LOGOS file system, the nodes in a paging file form a rectangle, the two dimensions being paging areas and nodes. That is, you can split objects among separate paging areas, or among nodes within a paging area. Splitting the objects among separate paging areas is better suited to controlling the system's behavior; splitting the objects among nodes in one paging area is better suited to reducing the amount of code in the active workspace at one time.

Isolating Subtrees

To reduce a system's workspace requirements, split the system's objects within a given paging area into separate nodes. This is accomplished by isolating subtrees of the system's calling tree, and putting each subtree in a separate node. The *calls* command can help you to examine and assess subtrees that might be candidates. The *build* command splits off a subtree into a node when generating a paging file. Deciding which subtrees deserve to be separate nodes involves a trade-off between workspace size and processing time.

When fewer subtrees are isolated, the nodes in a paging area will be larger, and the likelihood of filling the workspace during execution is increased. When more subtrees are isolated, the system must page that many more nodes in and out. These are guidelines to keep in mind:

The external modular structure of your system -- such as its individual commands -- is a good place to start dividing your system into subtrees.

Subtrees whose use is infrequent -- such as initialization routines that are always used, but only once -- are also good candidates for separate nodes.

Subtrees which are used by many nodes may be appropriate for separate nodes. The code in these subtrees can then remain in the workspace, while the callers are being paged in and out.

Any subtree in the system's calling tree that may not be required during a session is a good candidate for a separate node. A rare path in a common operation may be such a subtree.

Controlling Behavior through Paging

Both explicit and implicit paging can be used to control a system's behavior. This section discusses design considerations for each.

Explicit Page-in

Many applications have modules that exhibit different but related behaviors, depending on circumstances. The example discussed earlier in this chapter in which a system must support different terminal drivers, is a case for explicit page-in. You could structure this part of your system this way:

- Isolate the section of code required for each behavior.
- Define a common interface between the application and these groups. For a terminal driver, you might have a general *io* function, which is given arguments to display information, and to prompt for input.
- Define different nodes for each terminal supported, and include the associated *io* function in each, along with any required subfunctions.
- Have the application determine at startup the terminal type in use, and explicitly page in the required node using *∆lpagein*.

Another case for the above technique is the support of different languages. The common interface would be a table of messages; separate nodes would contain this table in different languages. Using separate nodes for separate states is advantageous because the required node can be paged in at the beginning of the session. If the interfaces are well defined, the rest of the application need not be concerned with special cases. This technique applies to paging area selection as well. The code required for each state can be split into a separate paging area, and the application can then select the desired one.

Implicit Page-In

You can alter the behaviour of a system by altering the selection of paging areas. Two different paging areas may contain nodes with the same name. When a node is to be paged in, selected paging areas are searched for that node, in the order that they were specified to *∆lpagein*. The node found in the first paging area searched is used, and nodes with the same name in following paging areas are shadowed from use.

As mentioned earlier, shadowing is useful when a system has several different available versions. It's also applicable to software that supports different languages, if the language tables are too extensive to be located in a single node. In the latter instance, you can set up the system so that it is built from parallel paging areas, each of which reflects the text segments for a different language. The nodes across areas contain objects of the same name, whose values are the language parts in different tongues. When the application is initialized, it selects the appropriate language tables by specifying the related paging area to *∆lpagefile*. From then on, the system's activities can proceed oblivious to the actual language in use.

Generating a Paged System

A paging area is built by a series of *build* and *shell* commands, followed by a *filesave* command. By default, every root object of a node and its associated calling tree are excluded by inference from other nodes in the paging area.

Defining Nodes Using *build*

The *build* command generates a cluster and places it into a specified end environment. For a detailed description of *build* and clusters, see the section, *Clusters and the build Command,* earlier in this chapter. This section concentrates on the use of *build* as it applies to paging files. Once you've decided on the structure for your paged system, you can use the *build* command to analyze the system's calling tree and to assemble a node from each root.

When generating a paging file, the *build* and *shell* commands you enter are buffered and are not executed immediately. They are recorded and acted upon only when you issue a *filesave* command. *Build* assumes that you are building a node of a paging file if the command's first argument -- the destination -- is a simple name. Thus:

```
[5] )build startup
```

specifies a node name, whereas:

```
[5] )build <startup>
[5] )build <10>
```

and

```
[5] )build 10
```

do not.

For example:

```
[10] )build startup +depth=all
```

This builds (or more accurately, *buffers* a request to build) node *startup* from the LOGOS file system object of the same name. The node contains all objects needed by the function *startup* except those whose trees form other nodes in the paging area. (See the section *Exclusion by Inference* for more detail.) *+depth=all* specifies that *startup*'s calling tree is to be analyzed to its farthest branches for objects to include. The name of the node and the name of its root function need not be the same.

To build the *startup* node based upon the tree of a function called *init*, you may use:

```
[11] )build startup init +depth = all
```

A special node you can specify is the base node, identified by ⋆. The contents of the base node are never expunged from the workspace. You can force objects to be included in the base node by use of the A∇U code tag in any function which is in the node's calling tree. A∇U is described in the section *Tree Analysis*. Here is a possible specification for the construction of the base node:

```
[4] )build ⋆ qlx +depth=all
```

If you supply a single pathname argument to *build*, the terminal segment of the pathname becomes the node, and the full pathname is used as the root. This is particularly useful if the pathname you give is a pattern; you can select and build many nodes with one command this way.

Another way to build more than one node with a single command is to specify multiple node names as the first argument. To do this, you must enclose the argument in quotes. If you specify multiple roots in the second argument, the node is built so that each root can cause it to be paged in. This is useful in cases where a node has two or more logical entry points.

A *build* command is the same regardless of the paging method the system will use. What is likely to differ is the node name. For request paging, the nodes will probably be paged only by your program code, and so can have names which are meaningless outside the context of your programs. For demand paging, however, the node names must match the names of the programs that reference them because the trap statement looks for a node that has the same name as the object causing the value error; usually, a node has the same name as the root function of the node.

Most modifiers of *build* work for node clusters exactly as they do for clusters built to workspaces or files. When you're building a paging file, it's particularly advantageous to specify an audit file; this allows generation to proceed more rapidly and also allows efficient updates to be performed. Audit files are specified using *+audit*, as described in the section *Audit Files*.

Because the generation of a paging file usually involves a series of similar *build* statements, it's handy to first set global defaults using *build* without arguments, and then to issue the statements necessary to define the nodes.

Generating the Paging File with *filesave*

The *filesave* command executes the buffered *build* and *shell* commands. It creates the paging file (if necessary), writes the paging area header, and assembles and writes the nodes to the paging area. The remainder of this section relates *filesave* to *build, shell,* and the other aspects of paging file generation.

Identifying the Paging Area

The paging area may be specified either in the *+file* modifier to the *build* command or in the argument to the *filesave* command. In either case, the paging area identification may include the library number, the file name, the starting component number, and the ending component number. For example, *501 inventory 100* indicates the area beginning at component 100 of the file *501 inventory*, and *501 inventory 100 499* specifies an upper bound on the size of that area. The upper bound is useful if you plan to have multiple paging areas in the same file, or to include some of your application's own data in the paging file.

If you do not specify a starting component number, the first component of the file is assumed. If you omit the ending component number, the paging area is extended as far as necessary. You may specify a passnumber to the paging file with the *+lock* modifier to the *build* or *filesave* command. If you omit *+lock*, 0 is assumed (that is, no passnumber).

Exclusion by Inference

For a paging file, the various node definitions are reconciled at the time of the *filesave* command. Each cluster deposited in the paging area contains the root object and every object needed by the root object, subject to four factors:

- the depth of tree analysis
- the size of the node
- exclusion by specification
- exclusion by inference

The depth of tree analysis is controlled by the *build* command's *+depth* modifier. The size of a node is controlled by the *+size* modifier, described in the section *Specifying Node Size*. Specific objects can be excluded if you name them in the *+exclude* modifier. Exclusion by inference is a property of clusters that applies only to paging files. When you're building a paging file, inference causes each node to be built without subtrees whose roots form another node. The analysis required to do this is performed at the time of the *filesave* command.

The exclusion by inference process structures a system as you would expect: each node in the paging area is essentially a different portion of the system's calling tree, and no node is wholly contained within another. Without exclusion by inference, nodes near the "top" of your system's structure would contain all nodes beneath it.

Let's look at a simple example to see how exclusion by inference works. Suppose several nodes require the *menu* function. This sequence will exclude *menu* from all nodes but the one of which it is the root:

```
[5] )build read        ⍝ (Requires menu)
[6] )build compose      ⍝ (As does this)
[7] )build menu
```

The nodes *read* and *compose* are built without the calling tree of *menu*. When, during execution, the function *menu* is needed, its node is paged in separately (and is thus eligible to be paged out separately). You can turn off exclusion by inference by specifying *+inference=no* in a *build* command. If you do, then every object needed by the root will be included (subject to tree analysis depth, and specific exclusion).

Specifying Keep-In Priority

Keep-in priority is the most significant page-out characteristic of a node. The priority is a relative measure of how important it is that a node remain in the workspace. All other things being equal, the node with the highest keep-in priority will be paged out last. Keep-in priority is specified with the *+keepin* modifier to the *build* command.

To build a node called *input* with priority 100, you may use:

```
[10] )build input +keepin=100
```

The priority values you choose are meaningful only as they compare with other values in your application. Often, selecting three or four discrete priorities for the application is sufficient. Think about those nodes that are heavily used, and set their priorities accordingly. For nodes of less importance, set somewhat lower priorities.

Leave the priority of the bulk of your nodes unassigned, so that they will all have the same priority of 0. It is very important that you apply priorities consistently and thoughtfully across your application. Otherwise, you may end up with a system that pages excessively.

Specifying Node Size

It is sometimes necessary to limit the size of a node. If you know that a particular node is large, or is likely to be paged in when the active workspace is inescapably crowded, you'll want to make sure that the node is not so large that it can't be paged in. You can specify a size limit to a particular node with the *+size* modifier to the *build* command. To ensure that the node *input* is at most 10000 bytes, type:

```
[10] )build input +size=10000
```

If the construction of the node exceeds the size you've specified, LOGOS will automatically split the calling tree for you. This will occur as many times as necessary. You may set a general size limit with the *+size* modifier to the *filesave* command, or with the *+size* modifier to a *build* command without arguments. These limits apply to nodes for which no specific size was set.

For example:

```
[10] )build input
[11] )filesave 501 inventory +size=15000
```

Overwriting and Updating a Paging File

When you regenerate a paging area, you must indicate that the existing paging area can be overwritten. For example:

```
[12] )filesave 501 inventory +audit=invaudit +overwrite
```

This causes the audit file and the paging area to be overwritten. If you omit *+overwrite*, nothing is overwritten and the entire existing paging area is rebuilt. You may specify the *+overwrite* modifier in either a *build* command or a *filesave* command. *+overwrite=audit* limits the overwriting to the audit file, while *+overwrite=dest* limits it to the destination paging area.

In a *build* command, you may also enter *+overwrite=buffer*. This causes all of the buffered but unexecuted *build* and *shell* commands to be discarded. Typically, you will include this only if a paging area generation has ended abnormally, to avoid duplicate specifications for nodes when you retry the generation. You may make changes to only certain nodes in a paging area by specifying the *+update* modifier to the *build* or *filesave* command. For more information on updating files, see *Chapter 11: Maintaining Systems.*

Building Paging Files With a Script

Here is a script that might be used to build a paging file for an inventory system:

```
∪ display .invent.cmds.gen.invpage
.invent.cmds.gen.invpage[s5]:
[1]  invpage
[2]  ⍝ generate inventory system paging file.
[3] )build +audit=invaudit +compile=x,d,p +depth=all
     +overwrite +recursive +size=10000 +workdir=(list .invent.src[d]
     +full +recursive)
[4] ⍝
[5] )build * startup ⍝ define base node
[6] )build CMD?* +keepin=20 ⍝+ all commands
[7] )build cmdparse +keepin=30 ⍝ define other nodes
[8] )build init
[9] )build grreq
[10] )build update +keepin=10
[11] )build l1 lang.english
[12] )build l2 lang.french
[13] ⍝
[14] )shell CMDreport
[15] )shell CMDprint
[16] ⍝
[17] )filesave 501 inventory 100 499
```

The first *build* command, on line 3, establishes new default values for subsequent *build* commands. It identifies an audit file; specifies compilation directives and tree-analysis depth; indicates that the audit file, the build buffer, and the paging area are to be overwritten (via *+overwrite* without an argument); specifies that all subordinate objects are to be included in any node; defines the maximum size of any node, and specifies the working directories.

The next *build* command, on line 5, defines the base node. This node is paged in when the paging area is selected and is never paged out. Line 6 builds each command as a separate node, using a single *build* statement with a pattern. These nodes have a medium keep-in priority. Lines 7 through 12 build other nodes, with varying priorities. The commands on lines 11 and 12 define nodes whose roots differ from their names. Lines 14 and 15 specify that two nodes are to be shells.

Finally, line 17 issues the *filesave* command that actually generates the paging file. Here, the paging area begins at component 100 and ends at component 499 (or earlier) of the file *501 inventory.*

Diagnostics

You might think that it would be almost impossible to debug a system that uses paging, especially a demand-paged one, since objects materialize and dematerialize in the workspace transparently. How can you use the traditional tools of the APL programmer, stop and trace vectors, when the function you want to observe is not yet in the workspace? Fortunately, LOGOS furnishes a set of diagnostic tools that let you use familiar concepts to debug paged systems.

These tools allow you to set stop and trace vectors on paged functions as well as paging file nodes themselves. Facilities are also provided for monitoring paging activity and for optionally recording this information in files. These tools are described fully in *Appendix D: Stop and Trace Controls* as well as in their documentation attributes. Here is a synopsis of each which will help you identify the tools you are interested in. Each of these functions is stored in the directory *.public.logos.paging*:

Table 8.4 Diagnostic Tools

Function	Description
∆*lfnstop*	set/clear a trace vector on a paged function
∆*lfntrace*	set/clear a stop vector on a paged function
∆*lpagestop*	set/clear a stop on a node
∆*lpagetrace*	set/clear a trace on a node
∆*lpageset*	set/clear monitoring of paging activity

Using Shells with Pagefile Nodes

Any node of a paging file (but not the base page) can be turned into a shell, using the *shell* command.

To build the node *startup* as a shell, you may use:

```
[7] )shell startup
```

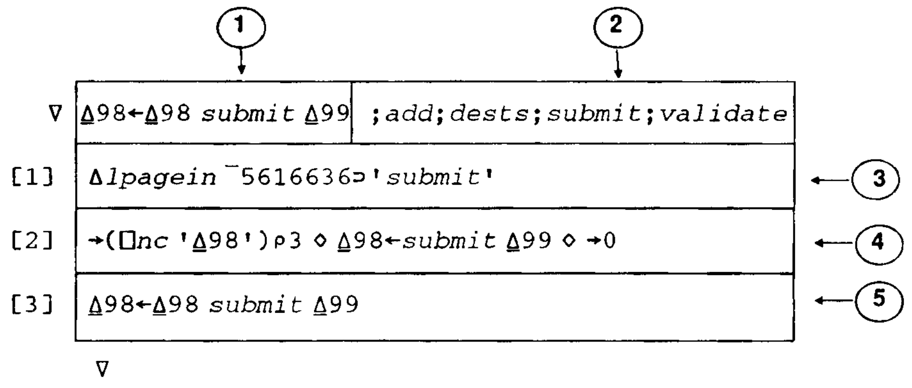
Because both *build* and *shell* commands are buffered, you may issue the *shell* command before or after the *build* command associated for the node. Aesthetically, it may be desirable to group all *build* statements together, and then perform the *shell* commands afterward. By default, a shell contains every object referenced by the root function or its calling tree. You may use the *+exclude* modifier to specify particular nodes to be excluded from the header's composition.

The resultant shell will no longer define the entire calling tree local to it, but this behaviour may be desirable if the shell is being used in a paged environment where part of the calling tree is already in the workspace. By allowing part of the calling tree to be global, you avoid having two copies of it in the workspace.

Anatomy of a Page File Shell

Figure 8.4 shows the definition of a shell created for a node in a LOGOS paging file, broken into its five individual segments.

Figure 8.4 Parts of a Page File Shell



```
    ∇ ⍙98←⍙98 submit ⍙99 ;add;dests;submit;validate
[1]   Δlpagein ¯5616636⊃'submit'
[2]   →(⎕nc '⍙98')⍴3 ⋄ ⍙98←submit ⍙99 ⋄ →0
[3]   ⍙98←⍙98 submit ⍙99
      ∇
```

1 - Function name and syntax
2 - Computed locals list
3 - Page-in logic
4 - Ambivalence
5 - Bootstrap logic

This shell was generated by the command:

∪ *shell submit +variant=a*

Part 1 is the shell's header, and is based on the root function's calling tree. The shell has the same name as the root function by default, and the same syntax. However, the arguments and result are renamed to avoid name conflicts with your function; specifically, the result and the left argument are named *⍙98,* and the right argument is named *⍙99*. For the same reason, absolute line numbers are used in the shell where branching is required.

Part 2 is the computed locals list. It identifies the other objects in the cluster which are local to the shell. You can use the *+header* modifier to add or delete names from the locals list.

Part 3 is the page-in logic -- in this case, a call to the *∆lpagein* function, with a precomputed node hash. This shell is not self-contained, as it requires the presence of *∆lpagein* in the workspace. If you want a shell to be self-contained, specify *+variant=s*.

Part 4 tests for the presence of a left argument and contains the code which is executed if the shell is called with only one argument. You may incorporate such ambivalence code by specifying *+variant=a*. By default, shells assume the strict syntax of their root function.

Part 5 contains the code which is executed if the shell is called with two arguments.

Options With *shell*

Normally, a shell takes its name from its root function. You can use the *+name* modifier to alter that. For example, to create a shell named *input* around a root function named *Input*, you may use:

```
∪ shell Input +name=input
```

With the *+header* modifier, you can add or delete names from the locals list. For example, *+header=,ask ∆qi* adds two names to the shell's header, while *+header=/∆qi* deletes one. As the example in the previous section showed, a basic shell does very little; it brings in the required node, and calls the root function, ambivalently if necessary.

A more complex shell can be generated, if you wish. With the *+qlx* modifier, you can replace the bootstrap logic by any code fragment. For example, if you generate *submit* with:

```
+qlx=⍙98←⍙98 submit 1⊃⍙99
```

That expression will replace the fifth part of the shell. The ambivalency specification is ignored when *+qlx* is specified. Three variants handle common cases of errors arising within the shell. *+variant=e* causes the shell to behave as a primitive function (via localizing and not setting ⎕*ec*). *+variant=r* causes result error to be trapped and ignored, for shells which do not return a result in all circumstances. *+variant=t* indicates that errors are to be signalled to the shell's caller, as if the shell itself were transparent.

A shell created around a node of a LOGOS created paging file depends on the presence of the LOGOS paging utilities (specifically *Δlpagein*) to page in the required code. You can make such a shell self-contained, and thus can copy it from one workspace to another, using *+variant=s*. With this variant, the shell is built with all the bootstrap code in it, and without reference to any global objects. A shell created around a package in an ordinary function file is always self contained.

If you want the shell to be locked, specify *+variant=l*. You can combine more than one variant, as in *+variant=lst*.

The *+skeleton* modifier allows you to specify a pathname to be used as a frame for the shell. The pathname identifies a function. If you use this modifier, the computed locals list is added to your function header, and your function becomes the shell. This allows you to write your own custom page-in and bootstrap logic, and still let LOGOS worry about handling the local names.

A Method for Maintaining and Generating Applications with LOGOS

Clearly, LOGOS provides a great deal of flexibility in terms of generating end environments. There are many approaches to structuring application source code within LOGOS and to writing generation scripts to build end environments. *Chapter 9: Generating End Environments* discusses in detail some strategies for organizing source within LOGOS. This section describes a method for getting a system into LOGOS and for creating the scripts necessary to build end environments. This approach has been used successfully on several projects and is general enough to be easily tailored to special situations.

The major steps in the method are:

1. Understand the structure of the application.
2. Decide on a LOGOS directory structure and populate it.
3. Create references to objects already established in the LOGOS hierarchy.
4. Introduce naming conventions where appropriate to group objects based on regular expressions.
5. Thread together the calling tree of the application using code tags.
6. Use code tags and compilation directives to handle special cases such as locked functions and configuration dependencies.
7. Build generation scripts.

A Prototype Application

The following scenario will be used to illustrate the practical application of the method:

Janice, an experienced LOGOS user, has been given a new assignment: the ongoing maintenance of an operational checkbook balancing system. This application is not currently maintained with LOGOS. Janice has been asked to move it into LOGOS, to fix several bugs that have been reported by users, and to effect a general clean-up of the code. Janice has been told that she will be asked to make some major enhancements to the system in the near future. Each step of the method will be discussed in terms of general principles and will be illustrated by observing Janice as she performs her job.

1 Understand the structure of the application

You must have a grasp of the structure of the application if you are to make intelligent decisions about maintaining it in LOGOS. Obviously, you must read the code carefully. It is often useful to generate a WSDOC listing including a global cross-reference and a calling tree listing, especially if you are not the original author of the application. This is invaluable in helping you to identify the modules in a system and to classify individual objects by the modules to which they belong.

Pay particular attention to special architectural features of the application, such as the use of functions stored in files, the use of ⎕*load* or ⎕*qload* to chain workspaces, or the use of server tasks. Try to understand *why* as well as *how* these features are used. Often, you will find that you can improve the performance of the system through the judicious use of LOGOS features.

For example, if an application brings functions in from a file because it requires as much free workspace as possible, you may be able to free even more workspace by compiling functions or by using the demand-based paging system provided with LOGOS. If an application uses several workspaces to hold production, test, and steward versions of a system, you can employ a paging file with multiple areas and an intelligent initialization program to run everything from one workspace without resorting to the use of ⎕*load*.

How does Janice go about understanding the checkbook system? The first thing she does is generate a full WSDOC listing of the workspace (if your site does not have a copy of WSDOC, there are usually similar tools available). She examines the listing and discovers that the root function of the application is a function named *cb*. She suspected this, since she noticed earlier that this function is invoked from ⎕*lx*, and the tree section of the WSDOC listing confirms that it is indeed at the root of the largest calling tree. Janice also makes note of the fact that there are several suspicious functions that are not part of any calling tree.

Next, she inspects the global cross-reference listing looking for references to system functions such as ⎕*read*, ⎕*fd*, ⎕*fx*, ⎕*pdef*, and ⎕*load* which indicate the use of features that might have an impact upon the architecture of the application. She examines any functions which contain references to these facilities and learns about the general structure of the system. At the close of her investigation Janice concludes that this is an architecturally straightforward application, that all of the code resides in the workspace, and that no paging or auto-loading of workspaces takes place.

2 Decide on a LOGOS directory structure and populate it.

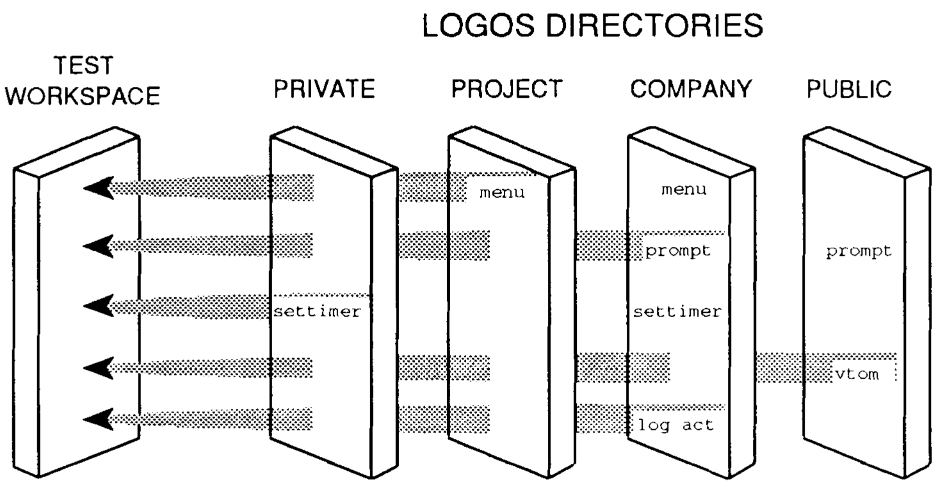
You must decide if you are going to duplicate the existing application architecture or if you are going to use the opportunity to restructure things. LOGOS can be used to build any application architecture, even those based on foreign paging systems. In either case, you must map out the structure you plan to build. Note that you don't have to concern yourself with the specific contents of every end environment, since calling tree analysis can do most of this work for you.

To make the most effective use of LOGOS you will want to do more than simply copy workspaces and files to LOGOS directories on a one-to-one basis. You will want to take advantage of the hierarchical nature of LOGOS and organize objects at a finer level of granularity than the workspace. Decide if you are going to build multiple environments for test versus production versions of an application.

This is a very powerful feature of LOGOS and does not require much effort to set up. Nevertheless, it is best if you make the decision before you get too far into the process of putting the system in LOGOS. Your decision will affect subsequent steps, particularly those related to putting code tags in source and defining generation scripts. If you are moving the first of a suite of applications into LOGOS, you will want to give some thought to a superstructure for the entire suite. If the applications share several utility functions you may want to set up a utility directory to service the entire set.

Often, when developing large systems, you will find you want to access a hierarchy of utilities. For example, at the most global level you might use the utilities found in *.public.util*. You share these with all users of LOGOS at many installations. At the next level, your organization may have established a company-wide utility library. Your project team may have a project library. Finally, if you are generating a test version of a system, you may have private debugging utilities you wish to include.

Figure 8.5 Utility Directory Hierarchy



Copying the contents of a workspace or file into a LOGOS directory is a simple task. There are several ways of accomplishing this using the *save* or *snap* commands or the *filetologos* script. Refer to the documentation for these commands for examples. Often, it is more productive to move all of the code from workspaces or files into one holding directory and then use LOGOS commands to copy objects to their ultimate destination within the hierarchy. After copying each object, delete the copy in the holding directory.

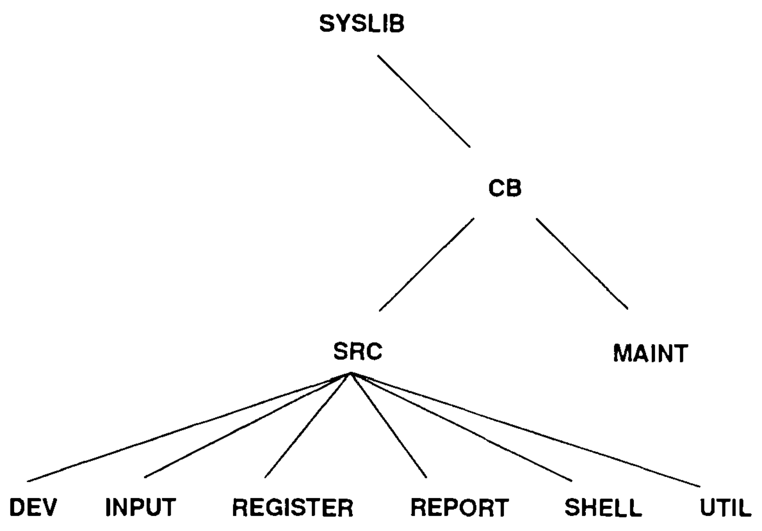
Using this approach, you know every object has been accounted for when the holding directory is empty. For example, if you have snapped a workspace with many functions into a directory and you want to move every function whose name begins with ∆*f* into a directory named *fileutils*, you can do this with the command sequence:

```
∪ copy (d←list ∆f?⋆) fileutils
∪ delete (⍎d)
```

In this example, the objects to be moved could be easily classified by matching their names against a pattern. A useful variant of this technique is to use the editor as a "filter". That is, you list the holding directory and save the output of the *list* command in a variable, as above. Then, use the editor to edit the variable, retaining only the names you wish to move into a given directory.

After you have finished editing the variable, you can pass it to *copy* and *delete*, as was done in the preceding example. Using techniques like this, you can quickly migrate a large number of objects into a directory structure. After some careful consideration, Janice decides that she is going to make a change in the operational environment of the application. She decides that she is going to add a test workspace to serve as an environment for evaluating bug fixes and future enhancements. Taking this into consideration, she develops the following directory structure:

Figure 8.6 Sample Directory Structure



The *maint* directory is designed to contain scripts used to maintain the system, for example the script which generates the workspaces associated with the system. The *dev* directory holds development versions of objects until they have been debugged. Janice's plan is to copy a function she needs to work on from its original directory into the *dev* directory.

The generation script will place *dev* at the beginning of the working directory list whenever the test version of the workspace is to be generated. This means that the development version of the function will appear only in the test workspace. When the function has been debugged, it will be copied back into its original directory and the copy in *dev* will be deleted. The *register, report, input*, and *shell* directories contain code unique to the check register, report generator, data input, and command shell modules of the system, respectively.

The purpose of the *util* directory is to hold utilities used throughout the system. Janice moves the application code into a temporary holding directory named *.syslib.cb.src.tmp*, by signing an auxiliary task, instructing the task to load the application workspace, and using the *save* command:

```
ᵘ signon
ᵘ send )load 510 checkbook
ᵘ save (send ⎕nl 2 3) +task +workdir=.syslib.cb.src.tmp +makedir
```

She then selectively moves objects out of the *tmp* directory and into their proper directories using the techniques described above. She finds, however, that there are a number of anomalous objects. Two of these are locked functions with the suspicious names *del* and *xref*. She remembers that each of these objects appeared in the WSDOC listing outside any calling tree.

A check of the cross-reference listing confirms that these functions are not called by any other functions in the application. It would appear that these are development tools that were inadvertently left in the production workspace. Janice knows, however that it is possible to fool WSDOC by calling a function in an indirect manner, for instance by using ⍎ to execute a row of a character matrix.

As a check, she uses the LOGOS *locate* command to search for the strings "xref" and "del" throughout all the objects in the application. She finds a match for "del" in a character matrix named *menucmds*. Further investigation reveals that the function *∆menu*, uses ⍎ to invoke functions. Therefore, this is a legitimate reference and *del* is indeed part of the application. Janice deletes *xref* and uses the same technique to investigate the other suspicious objects.

3 Create references to objects already established in the LOGOS hierarchy.

One of the most important reasons for using LOGOS is to re-use and share code. You will want to eliminate redundancy wherever possible. If you are the developer of an important utility used by many programmers in your organization, it is in your best interest and the best interest of the organization to maintain one central copy of the utility. This means that when you copy a workspace or file into LOGOS, you must identify objects that already exist within the LOGOS file system and replace the new copies with references to the existing code.

There are three ways to create such a reference to an external object. The first is to explicitly reference the object by pathname in the generation script. The second is to use *build*'s calling tree analysis capability and include the directory the utility appears in within the list of working directories used by the script. The third is to actually create a link in your application directories that points at the object in question.

Each approach has its own advantages and disadvantages. Before you can create such a reference, you must first identify those objects that appear elsewhere in the LOGOS hierarchy. The script *.public.logos.cmds.matchobj* is a useful tool for accomplishing this. If you present this script with a list of objects in LOGOS and a list of utility directories, the script will identify objects in the list which also appear somewhere in the utility directories.

The script works by first using *.public.logos.cmds.search* to identify objects in the directories with identical names and then by calling the *compare* command to verify that the objects are indeed identical. The script allows you to specify a *+flags* modifier to be used with *compare*. This allows you to match objects without regard to comments, local identifier names, etc. by specifying the *s* (syntactic) flag.

Janice plans to use two public utility libraries to generate the checkbook application: *.public.util* and *.public.camco.util* The former directory is the familiar utility library distributed with LOGOS, the latter is a utility library Janice's company has established. To identify any objects she recently added to the *.syslib.cb.src* hierarchy that might also appear in one of these utility libraries, she uses the following call to *matchobj*:

```
∪ ⎕←d←.public.logos.cmds.matchobj .syslib.cb.src .public.camco.util .public.util +flags=s
.syslib.cb.src.input.∆prompt
.syslib.cb.src.report.fmtcols
.syslib.cb.src.util.rcat
.syslib.cb.src.util.vtom
```

Janice is going to use *build* and tree analysis, so she knows that the public versions of these objects will be picked up automatically if she includes the proper utility directories in the working directory list set up by her generation script. Therefore, the pathnames produced by *matchobj* can be deleted. She does this with the command:

```
∪ delete (⍎d)
```

4 Introduce naming conventions where appropriate to group objects based on regular expressions

Regular expression text patterns are a useful means for identifying and organizing collections of objects within a LOGOS hierarchy. For example, you might be working with a system that contains a dozen primary commands. A likely hierarchical structure for such a system might include a directory for each command, with each directory containing the root function for the command and all of the objects referenced only by that command. Objects referenced by more than one command would be placed in a directory dedicated to the kind of processing performed by the object, or simply in a utility directory. This is a logical structure easily navigated by a developer looking for a particular object.

Suppose, however, for purposes of generating an end environment through tree analysis, you need to identify the root function in each directory. A frequently used technique for doing this is to give each command function a name that fits a defined pattern. For example, you might precede each command function name with *Δc*. Using this technique, the command:

```
∪ list Δc?⋆
```

will yield the names of all command functions. From her examination of the checkbook system code, Janice knows that the "command" functions in this system are all called from a driver function named *Δmenu*. *Δmenu* solicits input from the user and then executes a row of the character matrix *MENUCMDS*, based on this input. She observes that most of these function begin with the word *do*, for example *doprint*. There are a few exceptions however. She also notes that there is a function in the system named *doslink* that is not a command function. This name will produce an undesirable match the pattern *do?⋆*.

Janice decides that she will rename all commands functions so that their names begin with *Δdo*. This will address the exceptions and will establish a name prefix that is not likely to collide with other function names.

5 Thread together the calling tree of the application using code tags.

If you plan to use calling tree analysis to generate any part of your application, you must insure that LOGOS can deduce the calling tree by examining the application's code. The use of special features such as ⍎ and ⎕*fd* to fix functions dynamically, can introduce discontinuities in the calling tree. Fortunately, LOGOS provides a means for you to identify these discontinuities: ⍝∇∪ code tags.

This was discussed earlier in the section entitled *Tree Analysis*. This code tag allows you to declare simple names as well as reference regular expression patterns and objects such as name tables stored in LOGOS. Sometimes you will be faced with a different situation: references to names that you do not want LOGOS to treat as global references.

For example, you might call a public utility function called *printdef* that ties a file, selects a component based on your terminal type, and then defines a *print* function stored in a package in the selected component. This print function is outside your control and you do not want LOGOS to try to find it in the hierarchy during generation.

You can instruct the calling tree analysis process to ignore the name by using the ⍝∇; code tag:

```
[4]  printdef ⋄ print ⍝∇; print
```

The mechanism used by the checkbook application's ∆*menu* function to invoke command functions is one place where Janice needs to tie together the calling tree. Janice uses the LOGOS *locate* command to find the line containing the execute construct and adds a code tag:

```
[10] ⍎MENUCMDS[index;] ⍝∇∪ ⍎.syslib.cb.src.shell.MENUCMDS
```

This construct causes tree analysis to fetch the *MENUCMDS* table, to examine it for identifiers, and to generate a global reference to each of these identifiers. This will provide linkage between ∆*menu* and the functions it invokes indirectly through *MENUCMDS.*

6 Use code tags and compilation directives to handle special cases such as locked functions and configuration dependencies.

Generally, if you plan to compile any of your application, you will want specify this globally, in your generation script. You do this by using the *+compile* directive with the *build* or *get* commands in your script or you use the *environment compile* command. This is preferable to setting the compilation directive attributes of individual objects because it allows you to use the same function in other applications with different compilation directives in effect.

There are some cases, however, where you will want to control compilation on an object-by-object basis through compilation directive attributes or code tags. A few situations that warrant this treatment are:

- Objects that contain constructs that differ depending upon the configuration of the end environment into which they are generated. A good example of this is a function that must behave differently in a test and production environments. You use the ⍝∇∊ code tag to mark these lines. *Chapter 10: Using the Compiler* contains more information on this technique.

- Functions that should not be processed by certain compilation directives under any circumstances. For example, you may know that a function will not operate properly if its local identifiers are processed by the *r* (rename) compilation directive because it uses ⍎ and ⎕*fd*. You can inhibit this by including the directive *~R* in the function's [:*c*] (compilation directive) attribute.

- Functions that can be processed by certain compilation directives with some local exceptions. For example, if you use the *d* (diamondize) directive globally, you must add ⍝∇⋄ directives to any function lines that cannot be joined to adjacent lines.

- Functions that you wish to lock in the end environment. By specifying the *l* (lock) compilation directive, you can insure that *get* and *build* always place locked versions of the function in end environments. If you restrict other developers' access to the object to just *x* (execute), only you will be able to examine or modify the function's source.

Janice has already determined that she will be generating both a production and a test environment for the checkbook application. To support this she makes several changes to the objects in the *.syslib.cb.src* hierarchy. One important change she makes is to add a second line to the ∆*menu* function to reference a test copy of the *MENUCMDS* matrix stored in the *dev* directory:

```
[10] ⍎MENUCMDS[index;] ⍝∇∪  ⍎.syslib.cb.src.shell.MENUCMDS ⍝∇∊  prod
[11] ⍎MENUCMDS[index;] ⍝∇∪  ⍎.syslib.cb.src.dev.MENUCMDS ⍝∇∊  test
```

Janice intends to set her generation script up so that functions will be compiled with the directive *i=prod* when the production environment is being generated and compiled with the directive *i=test* when the test environment is being generated. Therefore, line [*10*] will not appear in test versions of the system, and line [*11*] will not appear in production versions. Each version will include the correct *MENUCMDS* table.

If Janice were to use either the *r* (rename) or *y* (relabel) directories, she would also have to look for occurrences of local variables or line labels inside quoted strings. In those instances, LOGOS would not rename that occurrence, so, when that line was executed, there would no longer be a local variable or line label of that name, and a value error would occur.

Trap statements are particularly likely to contain line labels inside the quoted trap expression. Janice could use the *locate* command to find trap expressions so she could examine them to see if the form should be changed.

For example, she might change a typical trap statement from:

```
[3] ⎕trap←'∘ 3 e →error'
```

to:

```
[3] ⎕trap←'∘ 3 e →', ⍕ error
```

so the line label *error* occurs outside of the quoted string. LOGOS would now appropriately change the label if the *r* or *y* directives were used.

Another situation that may require adding code tags arises when a function makes a global reference to a variable that is localized in a higher level function. Using the *r* compilation directive would cause that variable to be renamed in the higher level function, where it is localized, but not in the later subroutine where it appears to be a global object.

Janice could include a ⍝∇~ code tag on a line in the higher level function where the variable is localized to indicate that the variable whose name she includes after the code tag should not be renamed. In this way, the later global reference would still be valid.

7 Build generation scripts

Now you are ready to write scripts that will generate the end environments required by your application. Several examples of this kind of script were presented earlier in this chapter. The following checklist will serve as a review of the important principles discussed there.

Generation Script Checklist

☐ Is your application best served by a single generation script or a collection of scripts?

If your application consists of a single end environment, you probably only need one generation script. If your application architecture is more complex however, and it is comprised of several environments, you may want to write scripts for each of the environments. This gives you the flexibility to regenerate one part of the system without the necessity of rebuilding everything. You can write a "master" script which invokes the other scripts for situations where you do want to rebuild the entire application.

☐ Which aspects of the end environment do you want to make "soft"?

There are a number of aspects of an end environment that you may want to customize at the time you generate the environment. Usually, the best way of specifying these parameters is through arguments and modifiers to the generation script. You can use script syntax to provide default values for these parameters.

Some features of end environments that are commonly given this kind of treatment are:

- the end environment name
- audit file names
- the test or production status of the end environment
- compilation directives
- the value of global variables interpolated into source via the ⍝∇⍎ code tag (for example, file names, account numbers, etc.)
- global flags which control the inclusion of debugging code via the ⍝∇∊ code tag
- global flags which control the inclusion of special features via the ⍝∇∊ code tag (for example, does a workspace generated for a given user include interfaces to special steward code?)

Note that you do not always need a separate script parameter for each aspect of the end environment you wish to control. For example, it is common practice for scripts to determine audit file names, compilation directives and other global flags based on the setting of the test/production flag.

☐ Will you be using audit files? Will you want to use multiple files?

Audit files make application maintenance much easier by providing linkage between LOGOS source and application code. You will want to use audit files in all but the smallest applications. Many developers find it convenient to use different audit files for different versions (e.g. test vs. production) of their end environments. This makes it possible to easily update only one set of environments when using maintenance commands such as *distribute*.

☐ If you are generating a workspace, will you need an auxiliary task?

Although you can generate and save workspaces using the active workspace and the *wssave* command without an auxiliary task, it is generally preferable to use the task approach. This is because the task starts out with a completely clear workspace, whereas the active workspace always has a copy of LOGOS in it.

LOGOS places demands upon the resources of the workspace in terms of working area and symbol table space. If you move objects into the active workspace during generation and then save that workspace without using an auxiliary task, the saved workspace will usually have a larger symbol table overhead associated with it than necessary.

- ☐ What kind of compilation will you use?

  Compilation can provide very tangible benefits in end applications. Some very large applications would not run at all without the space-savings compilation affords them. You must use some compilation directives with caution, however. The *r* (rename), *y* (relabel), and *d* (diamondize) directives can be particularly dangerous if used indiscriminately.

- ☐ If you are generating a paged environment, what kind of paging will you use, demand or request?

  If you use request paging, you must predetermine the contents of each node and include calls to the LOGOS paging utilities at critical points in your application code. If you decide to use demand paging, you need to identify the root functions of each page and amend your application's start-up code so that the proper call to the paging initialization utility and ⎕*trap* assignment take place.

- ☐ Do you want to be able to perform "update" generations?

  If you are building a paged environment, most likely you will want to provide a modifier to the generation script that will allow you to specify a node or nodes to be regenerated during an update generation. The value of this parameter, the node names, can be passed to the *+update* parameter in your call to the *filesave* command.

- ☐ Does your environment require any special features, such as shells?

  If you want to include any shells in your application, you will need to identify the contents of each shell and include a call to the *shell* command in your generation script. In some cases, you will also need to provide a skeleton for the shell (see the section *Shells* in this chapter).

Janice writes the following script for generating the checkbook system workspace:

```
[1]  z←gencbws (+compile=p,r,) (+production) ;wsid
[2]  ⍝ generates the "checkbook" workspace used by the
[3]  ⍝ checkbook system.
[4]  ⍝
[5]  →production⍴l0
[6]  )environment workdir .syslib.cb.src.dev
     .syslib.cb.src.util|report|input|register|shell
     .public.camco.util .public.util +stack ⍝ establish test working directories
[7]  )environment audit 4949321 tcbaudit ⍝ establish test audit file
[8]  wsid←'4949321 checkbook' ⍝ establish test wsid
[9]  compile←compile,'i=test'
[10] →l1
[11] l0:)environment workdir .syslib.cb.src.util|report|input|register|shell
     .public.camco.util .public.util +stack ⍝ establish production working directories
[12] )environment audit 4949321 pcbaudit ⍝ establish production audit file
[13] wsid←'510 checkbook' ⍝ establish production wsid
[14] compile←compile,'i=prod'
[15] l1:)output generating workspace (⍎wsid) ⍝ inform user of our intentions
[16] )signon +task=gencbws ⍝ we will need an aux task to generate the workspace, sign
     it on now
[17] )env task gencbws ⍝ make it the default task
[18] )build <> start +depth +compile=x,\compile
[19] )z←wssave (⍎wsid) ⍝ save the workspace
[20] )send )off ⍝ dispose of aux task
```

Make note of the following: Janice wants to be able to build both the test and production environments using the same script. To accomplish this, she provides the script with a *+production* modifier. The branch on line 5 uses the local variable established by the script from the parameter to control which block of initialization code gets executed. Each block defines a set of working directories, an audit file, a target workspace id, and augments the compile modifier.

Note that the list of development working directories begins with *.syslib.cb.src.dev.* This means that any test objects Janice places in this directory will be placed into the end environment rather than their production counterparts. An auxiliary task is used to build the workspace. This task is signed off when the generation has completed. Tree analysis is used to determine the contents of the workspace.

All Janice needed to know was the name of the root function (*start*) and the list of working directories where referenced objects could be found. A *+compile* modifier controls how objects are compiled. By default, the *x* (decomment), *p* (parent pathname tag), and *r* (rename) directives are in effect. The *p* and *r* directives can be overridden by providing a different value to the modifier when invoking the script.

Janice knows that some of the functions contain voluminous comments and that the application will be severely constrained by the working space these comments will consume unless they are removed when the application is generated. Therefore, she embeds the *x* directive directly in the build command on line 18, enabling it permanently.

In the future, Janice would like to generate the application using the diamondize compilation directive as well. She knows however, that doing so now would jeopardize the restartability of some of the functions. What she plans to do at a later date, is examine these functions and either add ⍝∇⋄ code tags to protect sensitive lines, or in extreme cases, add the *~D* directive to the compilation directive attribute of functions that should not be diamondized at all.

The *build* command is used to generate a cluster using calling tree analysis and to disperse the contents of this cluster into the active workspace of the auxiliary task. The inclusion of the *+depth* modifier with no arguments specifies that calling tree analysis is to be carried out to infinite depth.

CHAPTER 9: GENERATING END ENVIRONMENTS

Figures

Tables

Designing a Hierarchy

The hierarchy is the vehicle for organizing data in LOGOS. It allows you to keep together related information such as the objects belonging to individual users, the applications they use, and the data associated with a specific project.

Principles of Choosing a Structure

A hierarchy should make your system easy to maintain. Several factors influence how you should organize your hierarchy. They are:

- the number of objects involved
- the number of people who require access, and to what parts
- whether parts of the system are common to other systems
- whether parts of the system can be functionally isolated

If the system you are storing in LOGOS consists of a large number of objects, avoid putting them all in the same directory. Isolate functional units (or *modules*) and store them in separate directories.

If parts of your system contain *subsystems* (one or more modules) that are common to another system, isolate the subsystems from the rest of your system. Store any group of objects that receives similar treatment (because they are conceptually related, or accessible to the same users, or exported as a unit) in its own directory.

Types of Structures

There are three basic forms of hierarchical organization:

Table 9.1 Forms of Hierarchies

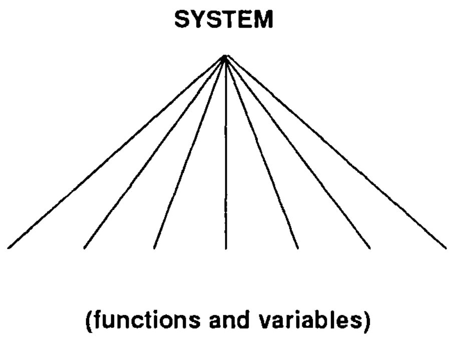
Form	Description
Flat Structure	Stores all objects under a single directory.
Module Structure	Separates functional units of a system into individual directories.
Common Structure	Groups objects based upon some common characteristic.

These forms can be employed alone or in combination, to yield an effective systemic structure.

Flat Structure

The simplest organization, a *flat structure*, has all objects stored under a single directory. This reflects the "flat" format of a normal APL workspace. It is sufficient for storing a small, self-contained systems.

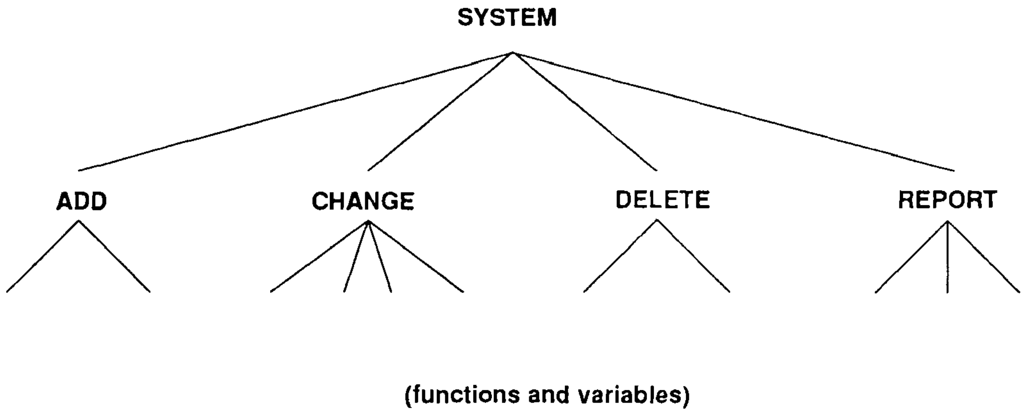
Figure 9.1 Flat Structure



Module Structure

A *module structure* separates functional units of a system into individual directories. For example, if your system consists of subsystems that add records to a database, change records, delete records, and produce reports, you might store each of these modules in a separate directory. These, in turn, would be stored under a single directory unifying the entire system.

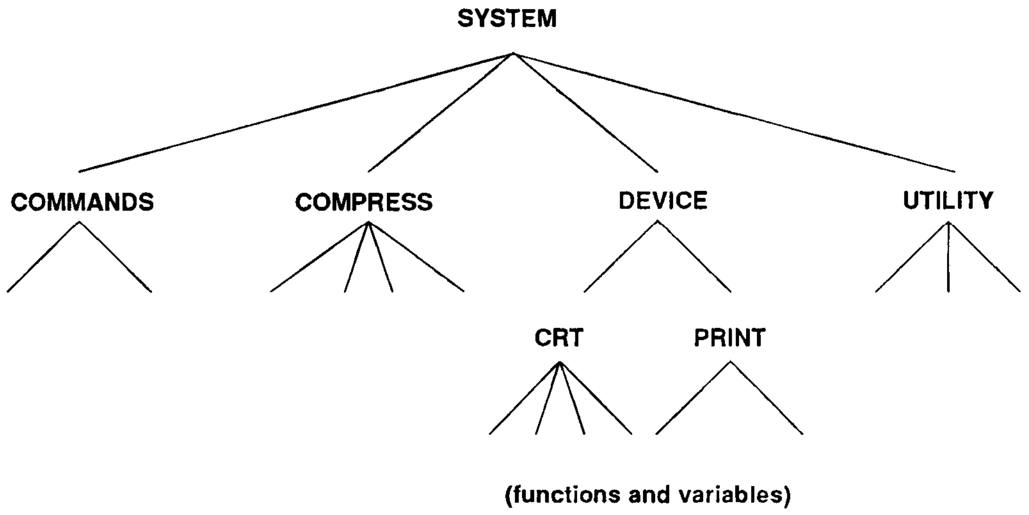
Figure 9.2 Module Structure



Common Structure

A *common structure* groups objects based upon some common characteristic, such as related functionality or operation on common data structures. For example, you might segregate general utilities, device handlers, data compression functions, and command routines, each in its own directory. These directories are unified by a single parent directory for the system as a whole.

Figure 9.3 Common Structure



Maintaining Multiple Versions of a System

With critical software, you can maintain two parallel versions of a system: a **production** version, which is the "official" system, and a **development** version, in which changes are tested and evaluated.

There are at least two ways to accommodate multiple versions in your system's hierarchy.

- Keep parallel hierarchies if the system is widely distributed: one for the distribution version of the system, and one for the development version. This technique involves duplication and the concomitant use of additional file space, but can be the simplest approach if you are maintaining the distributed version while developing the new one.

- Save changes to development directories. Have the same directory structure for development, but have these directories be empty initially.

When you save changes to development directories, you can then perform the following activities.

1 As you change a program or a piece of data, save the changed copy into the development directories.

2 When you generate the test version of your system, use multiple working directories to overlay the production versions with the test versions.

3 When a change is finally ready for release, simply copy the appropriate objects to the production directories and delete the development versions.

Putting Your System Into LOGOS

Once you have designed a hierarchy, you can put your system into LOGOS. You can use the *save* command, described in *Chapter 3: Using the File System,* to create your hierarchy of directories.

For example, to store your entire system in a single directory (a flat structure), you could type:

```
save (⍎⎕nl 2 3) +makedir  +workdir=system
```

This creates a directory called *system* and stores all functions and variables in your active workspace beneath it. (This method of storing a system is similar to use of the *snap* command, described in the section *Snapping a Workspace* in this chapter).

Saving Large Workspaces

If your workspace is large, you can use the auxiliary task facility to save it in LOGOS.

1 Sign on an auxiliary task.

2 Load the appropriate workspace under that task.

3 Save the objects in the auxiliary task's workspace in LOGOS. For example:

```
save (send ⎕nl 2 3) +makedir +task=aux  +workdir=system
```

NOTE: The *+task* modifier is not specified with the *send* command; it assumes the target is the default auxiliary task. Normally, however, the *save* command stores objects from the workspace in which LOGOS is running, so you must use *+task* to alter its behaviour. Alternatively, you could set the *environment task* parameter to *aux,* as described in *Chapter 14: Profiles and Environments.* This eliminates the need to specify the task name to *save,* or to any other command that works with auxiliary tasks.

Saving Systems Stored on File

There are several ways to store a system that is partially or completely file-resident in LOGOS.

- The most straightforward technique is to bring a group of objects into the workspace, use *save* as described in the previous section, and erase the objects. You can automate this process using a script.
- You can use the *disperse* script in *.public.logos.cmds* to save the contents of a SHARP APL package as separate paths in LOGOS. (See *Chapter 15: Using the Utility Library* for details about this public directory.) For example:

  disperse +value=⎕read 1 2

  The argument to *+value* is an expression that must produce a package when executed. In this example, *disperse* saves the contents of the package found in component 2 of the file tied to 1, as objects in your primary working directory.
- You can transfer an entire program file to LOGOS by using the *disperse* script in combination with the *with* command. For example:

  with 'disperse +value=⎕read 1 ⍺' (⍕((1⍴⎕size 1)-⎕io)+⍳-- /2⍴⎕size 1)

  Here, all components of the file are saved as objects in your primary working directory.
- Systems written before packages were available can have functions stored on file as character vectors or matrices. You can move these into LOGOS using the *+value* modifier to the *save* command. For example:

  save device.print.page[*f*] *+value=⍎⎕read 2 1*

  Normally, if *+value* is specified, *save* assumes that a variable is being saved. In this example, the extended pathname, [*f*], tells LOGOS to save the path as a function.

Creating New Objects

You can create new objects by:

- establishing variables in the workspace, then transferring these new objects to LOGOS with the *save* or *snap* command. For more information on creating new objects with *save*, see *Chapter 3: Using the File System.* For a discussion on the *snap* command, see the section, *Snapping a Workspace,* in this chapter.

- editing functions. For more information on creating new objects with the editor, see *Chapter 6: Using the Editor.*

- using the *save* command with the *+value* modifier to directly specify an object to be saved. This method is appropriate whenever you are saving something that is derived from a calculation or a value on file. For a discussion on *save +value,* see *Chapter 3: Using the File System.*

To illustrate the last method, suppose your application is one that allows configuration based upon user load, workspace size, and other similar parameters. You can use a simple script to automate storing the parameters for the application, and letting the user change them. The script might look like this:

```
[1]  config[1] +Wa= +Ul= +Intro=
[2]  ⍝ set modern configuration parameters.
[3]  )save config.Wa +value=⍎Wa ⍝ workspace  size
[4]  )save config.Ul +value=⍎Ul ⍝ expected  max user load
[5]  )save config.intro +value=\Intro ⍝ welcome message
```

You can run this script whenever the parameters need to be altered. You can regenerate your system whenever you like using the values stored by the most recent execution of the script.

Snapping a Workspace

The *snap* command is another means of interaction between LOGOS and a workspace. It examines the contents of the workspace, and stores in LOGOS all objects that are new or have changed since they were last stored. Changed objects are resaved in the directories from which they were fetched, while new ones are stored in your primary working directory.

The *snap* command can also save a script from which the workspace can be recreated, and can build an **audit file** (see *Chapter 8: Building Applications with LOGOS*) so LOGOS can automatically update the workspace.

Using the *snap* Command

To store the contents of your active workspace in your current working directory, type:

```
snap
```

LOGOS prints the list of paths saved.

If you do not want to store the entire workspace, you can supply an argument. For example, to pick up changes to *a*, *b*, or *c* only, type:

```
snap a b c
```

To select all objects beginning with either Δ or ⍙, type:

```
snap Δ?⋆|⍙?⋆
```

Snapping Compiled Objects

The *snap* command ignores compiled objects. (A compiled object has been altered by compilation. Inclusion of the parent pathname comment in an object does not make the object a compiled object.)

This is normally not a problem, but if an object has neither a parent pathname comment nor a referent in the tracking table, the *snap* command cannot tell if compilation occurred and stores a new version of the object. If you plan to use the *snap* command, try to avoid using compilation except where absolutely necessary.

Snapping Attributes

The *snap* command ignores attributes you bring into the workspace as variables, such as documentation or tags. However, if *snap* finds these objects by searching your working directories, it may inappropriately save the attributes as the source for the objects. You can use the command's argument, or modifiers such as *+confirm* and *+exclude*, to help control this.

Excluding Objects From the Snap

You can exclude some objects from being saved in LOGOS. For example, if commands in your system begin with a normal alphabetic character, subroutines begin with Δ, and global variables begin with ⍙, you can store each of these groups in a separate directory with the following commands:

```
u snap +exclude=Δ?⋆ ⍙?⋆ +makedir +workdir=commands
u snap Δ?⋆ +makedir +workdir=subroutines
u snap ⍙?⋆ +makedir +workdir=globals
```

The first command uses the *+exclude* modifier to remove objects beginning with Δ or ⍙ from consideration. This is often more convenient than itemizing what to save in the argument to the *snap* command, as is done in the second and third commands. For more complex control, you can use the selection list of the argument and the exclusion list of *+exclude* at the same time.

Confirming the Snap

If you are not sure that you want everything the *snap* command selects to be saved, you can use the *+confirm* modifier. As with the *delete* command, this modifier selects a *confirmation mode* in which you are prompted with each object about to be saved. For example:

```
snap ∆f?* +confirm
```

LOGOS then prompts you before snapping each object. For example:

```
save <.mde.tools.util.∆fread>? yes
save <.mde.tools.util.∆fver>? no
save <.mde.tools.util.∆fwrite>? y
.mde.tools.util.∆fread[f1]
.mde.tools.util.∆fwrite[f2]
```

While you are in this mode, the following keywords can be entered in response to the prompts:

Table 9.2 Keywords Available when Confirming a Snap

Keywords	Description
yes (or *y*)	Saves this object.
no (or *n*)	Does not save this object.
back	Returns to the previous prompt.
continue	Saves all selected objects without further prompting.
stop	Aborts the *snap* command.

Previewing the Snap

You can preview the effects of the *snap* command by using the *+view* modifier. For example:

```
snap object +view
```

The *snap* command returns a list of pathnames, but does not actually move any objects into LOGOS. This is also useful for determining which objects in your workspace have changed.

Snapping Changed Objects

If you work with objects that are already stored in LOGOS, *snap* and other commands maintain information that allows LOGOS to determine where they came from and whether you have changed them in the workspace.

For example, if you have a workspace containing only the *logos* function, you can use the *get* command to materialize some objects:

```
∪ get .public.util.files.∆fappend | ∆fcreate util.copyfile
.public.util.files.∆fappend[f1]
.public.util.files.∆fcreate[f1]
.john.modules.util.copyfile[f2]
```

If you invoke the *snap* command at this point, it would not save any workspace objects because the versions currently in LOGOS are identical to those in the workspace. However, if you edit *copyfile,* the *snap* command picks up the copy in the workspace and saves it as version 3 of *.john.modules.util.copyfile.*

The *snap* command's ability to determine what has changed in your workspace can save you time after a complicated editing session. For example, if your system is largely or completely stored in LOGOS, you can change and debug many objects directly in the workspace without using LOGOS, and the *snap* command will find the objects you changed and store them back wherever they came from.

Rebuilding a Snapped Workspace

If you want to rebuild the workspace you snap, include the *+script* modifier in your *snap* command, and supply a pathname as its argument. The *snap* command builds and stores an image of the workspace in the form of an executable script. When you run the script, it recreates the workspace exactly as it was when you snapped it.

The following example shows how to use the *snap* command to capture the workspace in which you changed the object *copyfile,* and display the script produced by the operation.

```
∪ workdir
.john.modules
∪ snap +script=gen
.john.modules.util.copyfile[f3]
.john.modules.gen[s1]
∪ display gen
.john.modules.gen[s1]:
[1]  gen
[2]  ⍝ logos-generated script
[3]  ⍝
[4]  ⍝ compiled or unchanged objects
[5]  ⍝
[6]  )get ∆fappend ∆fcreate  +w=.public.util.files
[7]  ⍝
[8]  ⍝ newly saved objects
[9]  ⍝
[10] )get copyfile +w=.john.modules.util
[11] ⍝
[12] ⍝ re-save the workspace
[13] ⍝
[14] )wssave ''
```

The *snap* command saves version 3 of *.john.modules.util.copyfile* and saves the script *gen* under the current working directory. The generated script is broken into several sections, with different *get* commands for objects that were not saved by *snap* (those that were unchanged or compiled), and objects that were saved by *snap* (those that were new or changed). Finally, the *wssave* command saves the workspace (because this workspace was *clear,* no workspace name is saved).

A script generated by the *snap* command should be considered a working document. It is not necessarily suitable for a real workspace generation, but instead provides a solid starting point for editing and revision. In this script, for example, you would probably want to include a proper workspace name as the argument to *wssave.*

Subsequent use of *snap* with the same script specified will add lines to the script for any new or changed objects.

How the *snap* Command Works

The *snap* command is a process that:

1. Relates workspace objects to LOGOS pathnames.
2. Determines whether objects present in both environments have been changed in the workspace.
3. Confirms through user interaction that each new or modified object should be saved.
4. Stores changed objects in the directories in which they had previously been saved.
5. Stores new objects in the primary working directory.
6. Builds and stores a script capable of regenerating the workspace.
7. Stores an audit from which the workspace can be updated.

Steps 3, 6, and 7 of this process are optional, and are controlled through the selection of the modifiers *+confirm, +script,* and *+audit,* respectively.

How the *snap* Command Determines Relationships

The *snap* command relates workspace objects to their counterparts stored in LOGOS. It determines these relationships from three sources:

- the tracking table
- parent pathname comments
- your working directories

Information in the Tracking Table

The **tracking table** is a variable, *∆LTRACK*, that is put into your workspace by the *get* and *build* commands. It contains information on every object fetched from LOGOS, including:

- the directory from which the object came
- the type of object
- the version number of the object
- an indication of whether the object was modified by compilation
- a **cyclic redundancy check** (CRC), which is used to determine whether the object was modified by you since fetched from LOGOS.

The tracking table is stored and referenced only in the workspace in which LOGOS is running. Even if you are defining objects in another workspace using *get* via an auxiliary task, the tracking entries for the objects are added to the tracking table in the active workspace.

The *snap* command does not interact with an auxiliary task, so it cannot take advantage of these tracking entries. (You can use the APL command *)copy* or the LOGOS command *transfer* to move *∆LTRACK* from the active workspace to the auxiliary workspace, save that workspace, load it from your active session, and then use *snap*.) Also, if you copy objects from workspace to workspace, the tracking table remains unaware of the derivation of those objects.

Information in the Parent Pathname Comment

A **parent pathname comment** is a special comment affixed to the end of the last line of a function by the LOGOS compiler (see *Chapter 10: Using the Compiler*). If you have selected the *p* compilation directive, functions fetched contain such a comment, distinguished by the prefix ⍝*. Parent pathname comments include the same information as entries in the tracking table, in a form that is easy to understand. If both a parent pathname comment and an entry in the tracking table are present, the former takes precedence.

Parent pathname comments have limitations. They are generated only if you request it by specifying the *p* compilation directive. If you plan to use *snap* frequently, consider setting this directive in your global *environment compile* directive list when you generate workspaces or within the scripts that do the generation for you (see *Chapter 14: Profiles and Environments*).

Unlike tracking table information, parent pathname comments stay with your functions even if they are copied from workspace to workspace, or if they are saved in a file. They are automatically updated by the *snap* and *save* commands, and they provide a valuable visual tracking tool for the programmer. However, they exist only in functions and don't apply to variables.

Information in Working Directories

If the *snap* command cannot deduce an object's pathname from either the tracking table or parent pathname comments, it searches your working directories for an object with the same name. If it finds one, it assumes the workspace object is a version of the LOGOS object.

Determining Whether Objects Have Changed

If the *snap* command does not find an object pathname by searching the tracking table, the parent pathname comment, or the working directories, it considers the object new and stores it in your primary working directory.

Figure 9-4. Operation of the *snap* Command

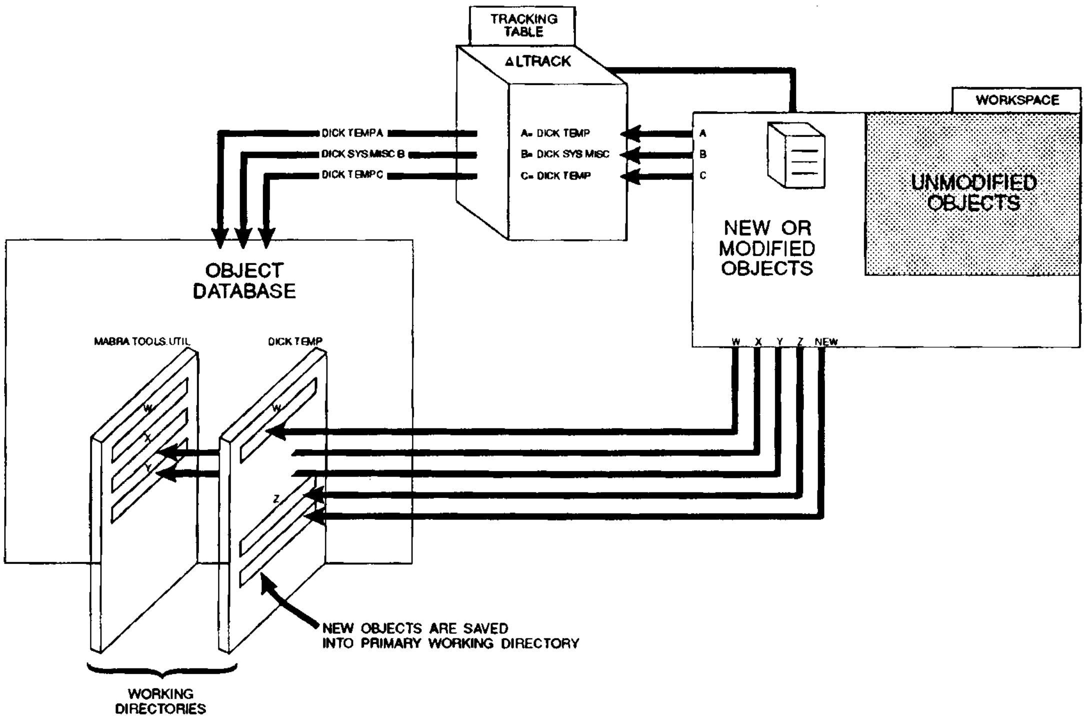


Figure 9-4 illustrates how *snap* uses the tracking table and your working directories to update the LOGOS object database. *a, b, c, w, x, y,* and *z* are objects that have been modified in the workspace. *new* is an object not yet stored in LOGOS. *a, b,* and *c* are found in the tracking table, and are therefore stored in the directories so implied. The working directories *.dick.temp* and *.mabra.tools.util* are then searched for the remaining objects. *w, x, y,* and *z* are located and are determined to have been modified; these objects are saved. *new* has no current referent, so it is stored in the primary working directory.

If the *snap* command does find an object's pathnames, it uses one of two techniques to determine whether an object found both in LOGOS and in the workspace has been changed.

If the object has an entry in the tracking table or a parent pathname comment, the *snap* command compares the object's CRC from the tracking table or its parent pathname comment, with a CRC computed from the version in the workspace. If they are the same, the object is considered unmodified and is not stored by *snap*.

If no CRC is available, the *snap* command searches the working directories for an object of the desired name in the working directories. If it finds one, *snap* compares it with the version in the workspace. If they are the same, the object is not resaved.

Using This Information To Your Advantage

You can use the *snap* command more effectively if you know how it works. For example, if you have designed an extensive hierarchy and stored your workspace-resident system in LOGOS, and then discover that a later version of the workspace is available, you can use the *snap* command and avoid repeating much of the work you had just completed. Set your working directories to reflect your hierarchy, and perhaps include a catch-all directory at the front of the list (for new objects), and you can capture the updated version of the workspace and have it stored in the appropriate directories, with a single command.

CHAPTER 10: USING THE COMPILER

Tables

About the Compiler

Whenever LOGOS deposits an object into an end environment, it passes the object through a filter. This filter, the LOGOS **compiler,** may take no action at all, or it may transform the object in a prescribed manner. LOGOS then places the transformed object into the end environment.

The original object is called the **source form** of the object, while the transformed version is called the **object form.** Much of the time, the compiler does nothing and these two forms are identical. In certain cases, however, the ability to alter an object before saving it in an end environment is extremely useful.

You could remove the comments from a function to make it run faster or to save space. You could customize an object for each environment in which it is placed. For example, you might have a function to tie an application system file whose name varies from installation to installation. If you write the function with a placeholder substituted for each occurrence of the file name, the compiler can fill in the blanks with the correct file name whenever the function is placed into an end environment.

The compiler also allows you to store the source in a form that is more easily maintained than the object form, and let LOGOS format it for you whenever the object is fetched.

Compilation Directives and Code Tags

The actions taken by the compiler are controlled through **compilation directives** and **code tags.** Compilation directives specify in a general way the actions you want the compiler to take; they can be applied to a single object or to arbitrary groups of objects.

Code tags are special comments that appear in the body of an object; they qualify or fine tune the actions specified by the compilation directives. Because code tags appear in the source of an object, they are processed object by object. To the APL interpreter, code tags are ordinary comments beginning with ⍝∇.

When Does LOGOS Compile Objects?

The source form of an object is compiled whenever it is processed by the *build, distribute*, or *get* commands. Unless you have specified compilation directives, compilation does not change the object; the source form and the object form are identical. Objects are transformed by compilation when you:

- save compilation directives with the object
- establish temporary environmental compilation directives
- specify compilation directives in the appropriate command

LOGOS generally saves the compiled form of an object automatically. This avoids the cost of recompiling the object the next time it is fetched.

LOGOS does not save the compiled form when:

- the use of special user-defined compilation directives inhibits saving of the compiled form of an object

- the *e* compilation directive is in effect

The *e* compilation directive causes specially delimited expressions in the source form of the object to be executed. The results of these expressions are interpolated into the source

Using Compilation Directives

Compilation directives enable or disable the various options provided by the compiler. The following table lists the tasks you can perform with directives; the directives you use to perform each task; the name of each directive; and the types of objects with which you can use each directive.

Table 10.1 Compilation Directives

Tasks You Can Perform	Directive	Full Name	Object Type
Specify a user-defined prologue	*a*	User-defined prologue	function, script, variable
Define a context for evaluation	*c*	Context	function, script, variable
Diamondize objects	*d*	Diamondize	function
Evaluate expressions marked by ⍝∇⍎ *code tags*	*e*	Evaluate	function, script, variable
Format variables	*f*	Format	variable
Control the inclusion of specially marked source lines	*i*	Inclusion	function
Lock a function or script	*l*	Lock	function, script
Include the Parent Pathname	*p*	Parent Pathname tag	function
Include the LRU Page-out Statement	*q*	Implant tracking statement	function
Rename locals	*r*	Rename locals	function
Specify a set of working directories	*w*	Working directory	function, script
Decomment a function	*x*	Excise comments	function
Remove line labels	*y*	Remove labels	function
Specify a user-defined epilogue	*z*	User-defined epilogue	function, script, variable

The directives are explained later in this chapter.

In general, a compilation directive list has the syntax:

directive[=value],directive[=value], directive[=value],...

For example:

l,x,z=.mabra.tools.tscom,d

The order in which you specify directives has no effect on the order in which they are evaluated. LOGOS evaluates compilation directives in the following order:

a, i, c, w, e, x, d, y, r, f, q, p, z, l

You can specify a list of compilation directives in any of three places:

- as the object's *:c* attribute
- with the *environment compile* command
- as a modifier to a command (for example, *get .john.modules.chart +compile=d,R=2,y*)

Precedence of Directives

LOGOS computes the set of effective directives using the following precedence rules (listed in order of most dominant to least):

- a strong directive (a directive in the second alphabet) saved with the object
- a strong directive in the user's profile
- a strong directive passed as a command parameter
- a weak directive (an ordinary directive) saved with the object
- a weak directive in the user's profile
- a weak directive passed as a command parameter

Turning Off Directives

You can turn off directives by specifying them in their negative form; that is, *~x*, or *~X*. The strong negative directive takes precedence over the weak negative directive.

Specifying a User-defined Prologue (*a*)

A user-defined prologue is a function or cluster to be executed immediately before compilation of the object begins. To specify one, use the *a* directive. The syntax of the *a* compilation directive is:

a=**pathname**

Using this directive, you can write your own custom compilation filter, as described in the section *Writing User-defined Compilation Directives*, later in this chapter.

Displaying a Context for Evaluation (*c*)

To define a context for evaluation (the objects to be available for use during compilation), use the *c* directive. The syntax of the compilation directive is:

c=**pathname**

The argument **pathname** must be a cluster. It indicates the objects to be available for use during compilation. Specifically, these objects are visible during the processing of the *a*, *e*, and *z* directives. They are also visible when names preceded by ⍎ within code tags are evaluated.

For more information, refer to the description of the *e* directive and to the section, *Writing User-defined Compilation Directives*, later in this chapter.

Diamondizing Objects (*d*)

To diamondize an object (merge each line with the next, separated by ⋄), use the *d* (diamondize) compilation directive. A line is not merged with its successor if:

- it contains a comment
- the following line is labelled
- either line contains the ⍝∇⋄ code tag

For this reason, the *d* directive is most effective when used in conjunction with the *x* (excise comments) directive.

Because there is a fixed overhead for each line of a function, use of *d* can save space.

NOTE: This directive can adversely effect the restartability of a function. You can use the ⍝∇⋄ code tag to suppress diamondization on specific lines that must remain intact if restartability is to be preserved. Before you apply the *d* directive to a group of functions, be sure to consider the effect upon each individual function.

Evaluating Expressions (*e*)

To evaluate expressions marked by ⍝∇⍎ code tags, use the *e* directive. To mark an expression for evaluation, specify a character to be used as a delimiter after the ⍎ in the code tag. Then surround the expression with a pair of delimiters. LOGOS executes marked expressions and replaces them with their results. For example, on the following line, the character ! is used as a delimiter:

```
'!sysfilename!' ⎕stie tn ⍝∇⍎!
```

The string !*sysfilename*! is replaced by the value of the APL expression *sysfilename*. For this to work properly, the expression *sysfilename* must be a variable or function available at the time of compilation. It may be resident in the workspace at the time of compilation, or you can use the *c* directive to materialize the execution environment automatically. Typically, *c* is used to supply the pathname of a cluster, built by the *build* command. The cluster is composed of objects referenced within expressions marked by ⍝∇⍎.

NOTE: You can use the *e* directive to compile character vector variables, as well as functions or scripts.

Formatting Variables (*f*)

To format a variable (which must be a character vector) with the SHARP APL text editor (from workspace *4 edit*), use the *f* (format) compilation directive.

This directive allows you to maintain formatted text variables, such as *describe* or *how*, in unformatted form.

Controlling the Inclusion of Source Lines (*i*)

To control the conditional inclusion of specially marked source lines, use the *i* directive. The syntax of the tag is:

i=**name**

Each line of source code can potentially have an **inclusion tag,** indicated by ⍝∇∊**name** (see the section *Using Code Tags* in this chapter). The *i* directive controls which of these tagged segments appear in the compiled object. A tagged line is included in the compiled object if any of the words in the tag appears in **name**, or if **name** is empty, or if no *i* directive was specified.

There are many applications for this directive. You might, for instance, include certain lines of code only in the development version of a system, or only in a particular level of the development version.

Locking a Function or Script (*l*)

To lock a function or script, use the *l* (lock) compilation directive.

Including the Parent Directory Name (*p*)

To include the parent directory name, use the *p* (parent pathname tag) compilation directive. A comment beginning with ⍝⋆ is appended to the last line of a function. The comment contains the parent directory name from the root, and the object's version number in brackets.

Use *p=0* to add the parent directory at the end of the last line.

Use *p=1* to add the parent directory name on a new line by itself at the end of the function.

For example:

```
⍝⋆.public.logos.paging.[3]------
```

The segments do the following:

.public.logos.paging.	Identifies the directory. (Note the terminal dot, implying extension of the path.)
[3]	Is the source's version number.
-----	Is a cyclic redundancy check (CRC) for the *snap* command.

This directive provides visual tracking information, and improves the operation of the *snap* command. You should use it if you plan to develop or maintain a workspace using *snap*.

Including the LRU Page-out Statement (*q*)

To include the least-recently-used (LRU) page-out statement, use the *q* (implant tracking statement) compilation directive. This statement generates a record of a function's use for page-out control.

When each function is placed in its end environment, it is automatically altered to include a call to the function *∆lpagelru*. This is placed at the beginning of the first line of the function.

Renaming Locals (*r*)

To rename local variables and line labels, use the *r* (rename locals) compilation directive.

The syntax of the directive is:

```
r[=[_]0|1|2[_]]
```

If you specify _ at the beginning or end of the value, names beginning with letters in the second alphabet are not changed.

r=0 indicates that locals and labels are to be renamed *⍙99*, *⍙98*, *⍙97*, ..., *⍙00*, *⍙199*, ... This is also the scheme used if you specify just *r*.

r=1 indicates that locals and labels are to be renamed *a*, *b*, *c*, ..., *aa*, *ab*, ...

r=2 indicates that locals and labels are to be given random seven-character names.

Local renaming can reduce the total number of symbols in an application workspace, reducing the requirement for symbol table space. If you use renaming schemes 0 or 2, the compiled functions use unlikely local names. This is an important benefit for utilities that need to reference objects in their caller's environment. For example, suppose you have a function whose purpose is to define the contents of a package it reads from a file globally. Using the *r* directive reduces the likelihood of a name that is local to the function conflicting with the name of one of the objects in the package.

NOTE: References to labels that occur inside of a quoted string, such as a *⎕trap* statement, or after ⍎, are not altered. Such expressions could be changed to a form like this, for example:

```
⎕trap← '∇ 1 e →',⍕ L5
```

Specifying Working Directories (*w*)

To specify a set of working directories, use the *w* compilation directive. Working directories are used when searching for objects to be included in a composite script, and when evaluating pathnames in code tags.

The syntax of the directive is:

w=**workdirs**

workdirs is the list of directories to be used as working directories.

Decommenting Functions (*x*)

To decomment a function, use the *x* (excise) compilation directive. To remove all comments (the default behaviour), specify *x*= or simply *x*.

NOTE: If a comment in a function starts with ⍝⍫, LOGOS never removes these symbols. LOGOS recognizes them as an APL indication of proprietary code.

To retain the opening ⍝ of whole-line comments, use *x=1*. This prevents renumbering of function lines. End-of-line comments are still removed in their entirety. When you are dealing with functions that contain branching statements with references to absolute line numbers, use *x=1*.

- To remove all LOGOS code tags, use *x*=∇.

This leaves regular comments intact.

This directive allows you to make extensive use of comments in your source, without sacrificing space or execution speed in your production application workspace.

Removing Line Labels (*y*)

To remove line labels and replace all references to line labels with constants, use the *y* (remove labels) compilation directive.

The syntax of the directive is:

y[=_]

Specifying *y*=_ causes line labels beginning with letters in the second alphabet to be left untouched.

Removing labels can reduce the number of symbols in your workspace, and thereby conserve symbol table space.

NOTE: References to constants consume marginally more workspace than do references to labels. They are also slightly less efficient in terms of processing speed. Be careful if you edit a function that was compiled with the *y* directive, because branching statements may be incorrect if you insert new lines into the function or delete existing ones. In practice, this is not a problem because generally it is the source form that is edited, not the object form.

NOTE: References to labels that occur inside of a quoted string, such as a ⎕*trap* statement, or after ⍎, are not altered. Such expressions could be changed to a form like this, for example:

```
⎕trap← '∇ 1 e →' ,⍕ L5
```

Specifying a User-defined Epilogue (*z*)

A user-defined epilogue is a function or cluster to be executed at the end of compilation of the object. To specify one, use the *z* (user-defined epilogue) compilation directive. The syntax of the directive is:

z=**pathname**

Using this directive, you can write your own custom filter.

Overriding Compilation Directives

The *environment compile* parameter (described in *Chapter 14: Profiles and Environments*) lets you temporarily override some of the directives set by a script or stored with an object. You can do this using strong directives (compilation directives in the second alphabet) without having to modify the script or the object.

■ To override compilation directives, type:

environment compile **directives**

The **directives** parameter is a list of compilation directives you want to use to override the defaults, specified in the second alphabet.

For example, suppose you have a function you want to decomment, and have saved it with the directive *x* set. This indicates a desire for that directive to occur (or not occur, in the case of a preceding ~).

To run a test generation of the application with nothing decommented, specify:

environment compile ~X

This overrides any directives whose letters are in the first alphabet. The function with the *x* directive will be commented. If you had used ~*x* as your environment compilation directive, the higher precedence of the directive bound to the function itself would have caused it to be decommented despite the global directive.

Preventing Overriding of Directives

To prevent a compilation directive from ever being overridden, save it with the object's compilation directives in the second alphabet.

For example, suppose you have a function that will not run if it is diamondized. Save the function with a directive of ~*D*. The function will never be diamondized.

Then you can set a global directive of *d* and run a test generation of an application, with the code diamondized. Your function is generated undiamondized; even a global directive of *D* cannot override this. Functions that are diamondized are those with:

- no mention of the *d* directive in any form
- *d* or *D* set locally (the former indicates a desire to be diamondized, the latter a requirement)

If you use a global value of *D*, functions diamondized are those above, plus those with:

- ~*d* set locally, indicating a desire not to be diamondized

Prohibiting Others from Overriding Directives

Compilation directives attached to an object cannot be overridden unless a user has read or write access to the source of the object. If you have a public function that you would like locked for most users, you need only save it with directive *l*, and give general execute access to the object. Users cannot fetch the unlocked copy of the object, no matter what global compilation directives they set.

Using Code Tags

A code tag is a special comment, distinguished by its opening ⍝∇. ⍝ marks it as an APL comment, and ∇ as a code tag. A symbol immediately following ⍝∇ identifies the tag's type.

Table 10.2 Code Tags

Name	Tag	Argument	Description
Union	⍝∇∪	**namelist**	Treats the names in **namelist** as if they were explicitly referenced in this line. This allows them to appear in cross-references or be involved in calling-tree analysis, even if they are quoted or computed via a table look-up.
Local	⍝∇;	**namelist**	Treats the names in **namelist** as local to this function so that calling-tree analysis is not performed on them.
Exclude	⍝∇~	**namelist**	Excludes names in **namelist** from the actions of the *r* (rename) and *y* (excise labels) directives. Effects of this code tag are not limited to the line on which it appears.
Inclusion Set	⍝∇∊	**name**	Includes the entire line on which this code tag occurs only if **name** has any values in common with the set defined by the *i* directive.
Evaluate expressions	⍝∇⍎	**dlm**	**dlm** is the evaluated expression delimiter for the line on which the code tag appears; expressions appearing between pairs of delimiters are executed and their results replace the original expressions in the source. Objects referenced within the delimiters must exist either in the workspace or in a path specified by the *c* compilation directive.
Non-diamondize	⍝∇⋄		Does not diamondize this line; it is not to be merged with the preceding or succeeding line.
User-defined	⍝∇⍺ ⍝∇⍵	**text** **text**	These are reserved for use in user-defined compilation directives. The compiler does not process these tags, except to remove them at the conclusion of compilation if the *x* directive has been set.

Names included in these code tags will appear in the output of the *xref* command.

NOTE: The *xref* command will report the line number of the line on which the code tag appears. For this reason, it is a good idea to place the code tag on the same line as the indirect (quoted) reference.

Specifying Code Tags

Code tags can appear in:

- functions
- scripts
- character vector variables

A line can contain more than one tag and an ordinary comment as well. The ordinary comment must appear last; otherwise, tags following the comment are interpreted as part of the comment and ignored. For example, two code tags are recognized in the line:

i←i+1 ⍝∇◇ ⍝∇∪ *j* ⍝ *increment index*

However, when the following line is compiled, the ⍝∇∪ tag is considered part of the comment and is ignored:

i←i+1 ⍝∇◇ ⍝ *increment index* ⍝∇∪ *j*

Interpretation of Names in Code Tags

The **namelist** arguments provided in the ⍝∇∪, ⍝∇;, and ⍝∇~ code tags can be interpreted in several ways, depending upon the format of the names. The following general forms illustrate (**x** can be any one of ∪, ;, or ~):

Table 10.3 Interpretation of Code Tags

Format of Argument	**Interpretation of Code Tag**
⍝∇**x name**	Interprets a name literally.
⍝∇**x name.**	Fetches and interpolates LOGOS paths. (Any name that includes dot (.) indicates that **name** is a LOGOS path or a regular expression.)
⍝∇**x ⍎name** ⍝∇**x ⍎name.**	Uses the value of the object name. The argument **name** must be in either the active workspace, or the path specified by the *c* compilation directive. If the argument name contains a dot, it is interpreted as a LOGOS path or a regular expression; the values of the implied paths are used.

All forms of a name can be used in the same code tag. For example:

⍝∇; *Δtie ⍎cmdtab ⍎.mabra.tools.locals.draw*

Writing User-defined Compilation Directives

The *a* and *z* directives enable you to write your own compilation filters. You do this by providing the pathname of an explicit, dyadic function (or a cluster with an explicit, dyadic function as its root) as an argument to the *a* or *z* directives. When the object is compiled, the compiler fetches and invokes your function, passing the object's source and some ancillary information to it. LOGOS expects you to return the processed source as the filter's result.

Prologue versus Epilogue Directives

The prologue and epilogue directives have different characteristics. If you invoke a function as a prologue directive (*a*), you are given source that has not yet been edited by the compiler. This means you can reliably scan the source for comments, labels, and so forth, without interference from other directives.

On the other hand, any modifications you make to the source in an *a* directive are subject to further processing by other directives. For instance, if you were to add a comment to the object, and the *x* directive were in effect, your comment would be removed before compilation completed.

This is where the epilogue (*z*) directive is useful; it allows you to examine the edited source and to make final modifications to it before it is disbursed.

For example, suppose you are generating a workspace and you want to gather statistics on the number of lines and comments in each function. You also want to add a comment containing a timestamp to the end of each function. Use both directives. The prologue directive gathers the statistics (before comments and labels are removed by *x* and *y* directives). The epilogue directive adds the timestamp comment (after *x* has removed existing comments).

Creating a Compilation Directive

The user-defined directive functions are expected to be dyadic and explicit. Source is passed to your function as its right argument.

Table 10.4 Rules for Creating a Compilation Directive

If the object is:	then the source is:	and the result you return:
a function or script	a character vector without bracketed line numbers and a carriage return terminating each line but the last	must be in the same format
a variable	the variable	can be a different datatype

You cannot compile a function or script into a variable, but you can compile an integer variable into a package or an array.

Syntax of the Left Argument

The left argument passed to your function is a vector of enclosures in the following format:

pathname cdtype ltype masks

The parts of the arguments are described in the following table.

Table 10.5 Parts of a Left Argument

Part	Description
pathname	A character vector representing the full pathname of the object being compiled.
cdtype	A character scalar, *a* or *z*, indicating the type of compilation directive.
ltype	A character scalar representing the LOGOS datatype of the object being compiled. It can be one of function, script, or variable.
masks	A Boolean matrix with four rows, and as many columns as there are characters in the source. If the source is not a character vector, **masks** has no columns.

masks[*0*;]	Marks carriage returns with 1's.
masks[*1*;]	Marks comments and quoted strings with 0's.
masks[*2*;]	Marks comments with 0's.
masks[*3*;]	Marks LOGOS code tags with 1's.

The Boolean masks can help when scanning an object's source. For example, if you have used the ⍝∇α or ⍝∇⍵ user-defined code tags, most of the work required to locate the code tags is already done for you.

The Compilation Environment

The pathname provided as an argument to the *a* or *z* directive can represent a function or a cluster. If a cluster is used, it must have a suitable function as its root.

When the directive is evaluated, the root function is fetched and executed in an environment where all of the other objects in the cluster have been localized and defined. This provides a convenient means of including all of the ancillary objects required by the directive function.

The execution environment also includes the objects contained in the environment specified by a *c* directive.

Sample Compilation Directive

The following sample application shows you how to count lines and comments in functions as they are compiled, and add a timestamp comment to the end.

Gathering Statistics

First, you need a function to gather statistics:

```
display .mabra.tools.stats
.mabra.tools.stats[fl] :
  ∇ s←m stats s;⎕io
[1]  ⍝ user-defined 'a' directive for gathering
[2]  ⍝ function statistics
[3]  ⍝
[4]  ⍝ counts the number of lines and adds this
[5]  ⍝ number to first element of the global
[6]  ⍝ integer vector <Stat>. also counts the
[7]  ⍝ number of comments and adds this number to
[8]  ⍝ the second element of <Stat>
[9]  ⍝
[10] ⎕io←0
[11] →('f'=>m[2])↓0 ⍝ exit immediately if not a fn
[12]  m←>m[3] ⍝ disclose mask array
[13]  Stat[0]←Stat[0]++/m[0;],1 ⍝ count lines
[14] Stat[1]←Stat[1]++/m[2;]<¯1↓1,m[2;] ⍝ count remarks
∇
```

Adding a Timestamp Comment

Next, you need a function to add the timestamp comment:

```
display .mabra.tools.tscom
.mabra.tools.tscom[fl] :
 ∇ s←m tscom s
[1] ⍝ user-defined 'z' directive that adds a
[2] ⍝ comment containing a timestamp to the
[3] ⍝ end of each compiled function.
[4] ⍝
[5] ⍝ CALLS: <datefmt> - timestamp formatting fn
[6] ⍝
[7] ⍝ GLOBALS: CR - carriage return character
[8] ⍝
[9] →('f'=>m[⎕io+2])↓0 ⍝ exit immediately if not a fn
[10] s←s,CR,'⍝ compiled: ',,datefmt ⎕ts ⍝ append remark
   ∇
```

Notice that this function references two global objects: a function named *datefmt* and a variable named CR. You must ensure that these objects are defined whenever *tscom* is executed. These two objects are part of the public utilities. Therefore, you can build a cluster with *tscom* as its root:

```
ᵁ build .mabra.tools.tscomdir .mabra.tools.tscom +d +w=.public.util
.public.util.ts
```

Initializing the STAT Variable

Finally, initialize the *Stat* variable, used by *stats*:

```
⍎Stat←0 0
```

To demonstrate the effect of these directives, you can compile the following object:

```
∪ display .public.util.vtom
.public.util.vtom[f1]:
 ∇ z←dlm vtom a;b
[1] ⍝ form a left-justified matrix from a character vector.
[2] ⍝ <dlm> ←→ hard-delimiter. adjacent delimiters are treated as a single one.
[3] ⍝           eg:  '/' vtom 'top/to//bottom'
[4] ⍝            ←→ 'top
[5] ⍝                 to
[6] ⍝                 bottom'
[7] z←a∊dlm ⋄ b←(b,⎕io+⍴a)−⎕io,1+b←z/⍳⍴z ⋄ b←(b≠0)/b ⍝ lengths of each row
[8] z←(⍴b)⍴(,b←b∘.>|⎕io−⍳⌈/0,b)\(~z)/a ⍝ fill and shape into a matrix
∇
∪ ⍎ Stat ←0 0
∪ get .public.util.vtom +compile=a=.mabra.tools.stats,x,z=.mabra.tools.tscomdir
.public.util.vtom[1]
∪ ⍎1 ⎕fd 'vtom'
  ∇ z←dlm vtom a;b
[1]  z←a∊dlm ⋄ b←(b,⎕io+⍴a)−⎕io,1+b←z/⍳⍴z ⋄ b←(b≠0)/b
[2]  z←(⍴b)⍴(,b←b ∘.>|⎕io−⍳⌈/0,b)\(~z)/a
[3] ⍝ compiled: 13 jan 1986 19:42:37
 ∇
∪ ⍎Stat
9 8
```

Using the Compiler Utility Function

The function *∆lcomputil* in *.public.logos.util* provides an interface between the compiler and user-defined compilation directives. By calling *∆lcomputil* from within a user-defined directive function, you can effect changes in the compilation process. The right argument to *∆lcomputil* must be a character vector command name.

Currently, one command is supported. The *save* command causes the compiler to retain the compiled version for future use. By default, the compiler does not retain the compiled version whenever the *a*, *z*, or *e* directives are in force. Executing this command overrides this behaviour.

The result is always a Boolean scalar 1. If you pass an invalid argument, LOGOS signals an error.

Typically, you use the *build* command to build a cluster containing your user-defined directive function, *Δlcomputil,* and any other object required by your function. You then specify the pathname of this cluster as the argument to the *a* or *z* directive when defining the `[:c]` attribute of the object you want to compile.

Applications of User-defined Directives

This section suggests some ways you might use user-defined compilation directives. The last of these is illustrated with an example.

- Generating statistics

  You can generate statistics describing a given system. The statistics might include:

  - the number of functions and variables
  - the number of variables of each datatype
  - the counts of lines, comments, and labels

- Enforcing programming standards

  You can enforce programming standards by scanning source code and noting exceptions to the standards. As examples, you might look for such things as:

  - uncommented functions
  - branches to constant line numbers
  - duplicate line labels
  - syntax errors

- Writing functions in direct-definition format

  You can write functions using direct-definition format, and compile them into executable APL.

- Replacing inefficient constructs

  You can replace handy but inefficient constructs by more efficient ones. For example, if your programming style includes the use of branching functions such as *if* and *unless,* you can write a directive that compiles these into expressions without function calls. (**→label** *if* **expression** becomes **→(expression)⍴label**, and **→label** *unless* **expression** becomes **→expression↓label.**) This allows you to make use of these programming aids without sacrificing execution speed in the application workspace.

- Defining language extensions

  You can define your own language extensions, and compile them into executable APL.

- Defining a datatype

  You can define your own user-defined datatype. The following example illustrates this.

Example

Suppose you have an application that uses a full-screen display device. The device is driven by an auxiliary processor (AP), and the application issues commands to the AP. You describe the format of a screen to the AP using an integer matrix table. The columns of this table describe the position, size, colour, and attributes of each field on the screen.

While this integer matrix representation can be efficiently processed by an APL function, it is an unwieldy representation. If you want to change the colour of a given field from green to red, you must scan the columns describing the fields' positions, locate the correct field, recall the proper integer code for red and make the change. You can improve this situation by defining your own language for describing a screen. For example:

position(2,2),size=(1,80),colour=red,attr=reverse

Next, you write a function, *compilescreen*, that compiles these descriptions into their integer counterparts. Once you have saved this function in LOGOS, you can maintain all of your screen format variables as descriptive character vectors. LOGOS will automatically compile these variables into machine-readable form whenever an application is constructed:

```
∪ display .invent.src.screens.inpfmt
.invent.src.screens.inpfmt[f2] :
home(1,1),size(1,79),colour=red
home(2,1),size(31,39),colour=green,attr=reverse,type=char
home(2,41),size(31,39),colour=blue,attr=reverse,type=char
∪ display .invent.src.screens.inpfmt[:c]
.invent.src.screens.inpfmt[v2:c] :
a=.invent.src.util.compilescreen
∪ get .invent.src.screens.inpfmt
.invent.src.screens.inpfmt[2]
∪ ⍎inpfmt
1   1  1   79  2   1   2   0   0   0   1
2   1  31  39  0   1   4   0   2   0   1
2   41 31  39  0   1   1   0   2   0   1
```

CHAPTER 11: MAINTAINING SYSTEMS

About Registration

Registration allows you to signal to other users that an object is in use. While an object is registered as being out, other users are prevented from altering it, and are notified that you have the object registered. For example, if Bob registers the object, *chart,* before working on it, then Joe's attempt to change the same object produces the message:

```
modules.chart:*registered by bob (20jan86 18:36)
```

The warning displays for any reference to the object, including those by the *display, get, save,* and *edit* commands. This means that registration applies universally; not just to those individuals accessing the object by a particular means.

The warning message looks much like the message produced by a broadcast note. However, the warning message produced by registration contains the character ⋆ immediately after the :.

Registration information is displayed only for those users with write access to an object.

Unlike the broadcast note, which produces a warning only, registration blocks access to objects.

Registering Out Objects

You can register out objects with the *register out* command or with the *+register* modifier to the *edit* command. If the object is already registered out by someone else, you can override the registration using the *edit* command.

When you apply either registration or registration potential to a directory, any objects that you subsequently create below that directory automatically inherit the registration status of their parent. This allows the registration status of objects within the same application to remain consistent.

The following examples illustrate ways you can register out objects.

- To register out the object *modules.chart* with the *register out* command, type:

  ```
  register out modules.chart
  ```

  You see the message:

  ```
  1 object registered out
  ```

- To register out the object *modules.chart* using *edit +register*, type:

  edit modules.chart +register

- To register out all objects below the directory *modules.chartsub* simultaneously, type:

  register out modules.chartsub +recursive

Overriding Registration

If you must change an object while someone else has it registered, you can override registration by specifying the *+override* modifier to the *register* command. For example:

register out modules.chart +override

The following message appears:

1 object registered out

- To override someone else's registration using the *edit* command, type, for example:

  edit modules.chart +override

The next time the other user references the object, the following message appears:

*modules.chart:*registration overridden by joe (21jan86 02:12)*

NOTE: The *+data*, *+long*, *+summary*, and *+versions* forms of the *list* command remind you that an object is registered out by the presence of *r* in the attributes column. The *+long* form also tells you by whom.

Displaying Results

The result of the *register* command is a list of pathnames whose registration was changed. To display this result, use ⎕← before the command.

Registering In Objects

When changes to the object are complete, you can register it back in with the *register in* command. You might do this just after finishing your changes; you might test them first; or you might even wait until they have run for some time without difficulty. For example, to register in the object *modules.chart*, type:

register in modules.chart

You see the message:

1 object registered in

■ If you changed more than one object, you can register them all in (or out) with a single command. For example:

register in modules.chart utils.mtov

You see the message:

2 objects registered in

Setting Registration Potential

LOGOS recognizes that people can forget to use the *register* command. Therefore, you can set the **registration potential** of an object.

Registration potential works with the *edit* command. If a path has registration potential on, then any time the object is opened via the *edit* command, it is automatically registered for the duration of the editing session. When the object is closed, it is registered back in but its registration potential remains in effect for the next time it is modified.

NOTE: Registration potential does not affect whether the object is registered in or out. You can register out a group of objects even if they have registration potential set. In fact, this is quite useful if you are going to be changing an object or a series of objects over an extended period of time.

■ To set registration potential on, use the *register on* command. For example:

register on modules.chart

You see the message:

1 object registered on

- To set registration potential off, use the *register off* command. For example:

  register off modules.chart

  You see the message:

  1 object registered off

  The registration potential of an object is displayed by the *+data, +long, +summary* and *+versions* options of the *list* command, as the flag *p*.

Registration with Other Commands

Registration options appear on commands other than *register*. The following commands recognize registration potential, and support modifiers to facilitate registration activities.

Table 11.1 Commands Supporting Registration

Command	Modifiers	Allows you to:
copy	*+override*	Copy over objects registered by others.
	+in	Reset the registration of objects as you copy them.
delete	*+override*	Delete registered objects.
distribute	*+override*	Distribute registered objects.
edit	*+register*	Register objects while editing.
	+override	Edit objects registered by others.
export	*+override*	Export registered objects.
replace	*+override*	Perform search and replace operations on registered objects.
save	*+override*	Copy over objects registered by others.
	+in	Reset the registration of objects as you save them.

Tracking References to Objects

LOGOS maintains a **used list** for every object stored in the file system. The used list is a record of where each object in the file system is used. It is primarily to help developers locate systems that contain specified objects. You can interrogate the used list using the *references* command.

If changes to utilities or public objects can affect other people's systems, you can determine who they will affect. You can find problems relating to a particular version of an object, and find the systems that include it.

For example, to interrogate the path *.john.util.vtom,* type:

references .john.util.vtom

You see a message such as:

```
.john.util.vtom     l    .john.util.modules.vtom
                    s    .mabra.tools.search
                    w    1234567 util
                    w    1234567 syslog                 1234567 logaudit
                    f    1234567 functions       23
                    p    7654321 sysbase         10     7654321 audit
                    p    7654321 sysbasetest     10     7654321 auditdev
```

For each pathname argument you supply, the *references* command checks the used list for every entity that refers to:

- the pathname
- the pathname's type
- the component location (if a file)
- an audit file (if there was one)

The letters indicate the type of information the command is returning, summarized in the following table.

Table 11.2 Information Returned by the *references* Command

Type	Meaning	End Environment	Command that Created the Object
c	Cluster	Pathname	*build*
f	File	Filename	*build*
l	Link	Pathname	*link*
p	Page file	Filename	*build, filesave*
s	Script	Pathname	*edit, save*
w	Workspace	Workspace name	*snap, wssave*

User files and page files have a component offset associated with them. For a user file, this value is the component number in which the object resides. For a page file, it is the start of the paging area.

Each data type except a link or a script can also have an audit file associated with it, if the operation that produced the reference specified one. Where an audit file was specified, LOGOS retains its name, and shows it in the references report.

NOTE: If you don't have write access to an object, LOGOS doesn't necessarily display all references to it. When you query workspaces, files, and page files, LOGOS displays only references made by you or by an application in the public libraries. This keeps usage information away from people who are not authorized to see it, even if they have execute or read permission to it and can use it. The amount of reference information disclosed about links and scripts is governed by the LOGOS access controls.

Examples

You can use the *+headings* modifier displays the columns of the report. For example:

```
u references .john.util.vtom +headings
```

This displays a report similar to the following:

```
--------pathname---------- typ --------end environment------ ----loc---- -----audit file-----
.john.util.vtom             l     .john.util.modules.vtom
                            s     .mabra.tools.search
                            w     1234567 util
                            w     1234567 syslog                              1234567 logaudit
                            f     1234567 functions            23
                            p     7654321 sysbase              10             7654321 audit
                            p     7654321 sysbasetest          10             7654321 auditdev
```

The *references* command normally resolves links in its arguments. You can disable link resolution by specifying [/] at the end of the pathname. If *util.vtom* is a link, *references util.vtom* lists the references to the object to which the link refers, and *references util.vtom*[/] lists the references to the link itself.

If you specify the *+audit* modifier to the *references* command, information from audit files is included in the report. For example:

```
u references .john.util.vtom +audit
```

This displays a report such as the following:

```
.john.util.vtom          l      .john.util.modules.vtom
                         s      .mabra.tools.search
                         w      1234567 util
                         w      1234567 syslog                      1234567 logaudit
                         f      1234567 functions        23
                         p      7654321 sysbase          10         7654321 audit
version=25          references=7              saved=14jan86 11:08
pages=*       allocate  deleterow error         match       runline      zebra
                         p      7654321 sysbasetest      10         7654321 auditdev
version=27          references=7              saved=15jan86 14:51
pages=*       allocate  deleterow error         match       runline      zebra
```

Additional detail is given for each end environment found within each audit file, including:

- the type of object
- the version number of the object
- the directives used to compile the object
- the total number of references to the object by all end environments in the audit
- the timestamp of the audit record and the alias that saved the object
- the names of the nodes in which the object appears (if the record is for a page file)

Each paging area counts as one end environment. A node name of * refers to the base node of the paging area.

Distributing Changes

If you built your application with the assistance of an audit file, LOGOS allows you to move changes from LOGOS into the end environments that use them with the *distribute* command. You can distribute changes to your application and other people's applications without rebuilding the application from scratch, or having to determine where the objects need to go.

To use the *distribute* command, you must specify the names of objects in LOGOS, and the name of the audit file used to record the original generations. (The audit file was built by earlier *build, filesave, snap,* and *wssave* operations.)

The *distribute* command searches the audit file for all end environments that reference the objects, and updates them one by one. The update process depends upon the type of end environment, described in the following table:

Table 11.3 Rules Determining the Update Process of the *distribute* Command

If:	Then:
the environment is a file,	the *distribute* command introduces the changes into the component.
the environment is a page file,	the *distribute* command introduces the changes into the node.
the environment is a workspace,	the *distribute* command loads the workspace, materializes the objects, and then saves the workspace. To accomplish this, you need an auxiliary task (created using the *signon* command) that is signed on to the user number that owns the workspace.

NOTE: If you don't have an audit file describing where the object belongs, you can't use the *distribute* command. Instead, you can regenerate your application using the same commands or script that you used to build it originally.

■ To use the distribute command, type:

distribute **pathname** *+audit*=**audit**

The **pathname** argument is the object you want to distribute, and the value **audit** is its audit file name.

For example, to release a new version of the object *.john.util.vtom,* type:

distribute .john.util.vtom +audit=7654321 audit

You see a message such as:

```
1 object distributed to 7 end environments
```

The *distribute* command works on various types of end environments. For example, if the same function is tied to a workspace and a file by virtue of the audit file *1234567 logaudit*, one command can update both. Because a workspace is involved, you first need to sign a task onto the owner's number:

```
u signon 1234567
password:orange
auxiliary task 1729 <aux> signed on 11jan86 15:20
```

```
u distribute .john.util.vtom +audit=1234567 logaudit +show +task=aux
.john.util.vtom to workspace: 1234567 syslog
.john.util.vtom to file: 1234567 functions, 23
1 object distributed to 2 end environments
```

Displaying Output

You can display the end environments affected by the *distribute* command using the *+show* modifier. For example:

```
distribute .john.util.vtom +audit=7654321 audit +show
```

You see the message:

```
.john.util.vtom to file: 7654321 sysbase, 10; page: *
.john.util.vtom to file: 7654321 sysbase, 10; page: allocate
.john.util.vtom to file: 7654321 sysbase, 10; page: deleterow
.john.util.vtom to file: 7654321 sysbase, 10; page: error
.john.util.vtom to file: 7654321 sysbase, 10; page: match
.john.util.vtom to file: 7654321 sysbase, 10; page: runline
.john.util.vtom to file: 7654321 sysbase, 10; page: zebra
1 object distributed to 7 end environments
```

Inquiring on End Environments to be Affected

You can see where an object will go before distributing it using the *reference* command. Type:

references **pathname** *+audit*

Replacing Objects

Sometimes, a change made to an object might affect the calling tree of the object. Suppose *vtom* were changed to reference the variable *CR*, which it previously did not use. With the *+replacement* modifier, you can insert both objects anywhere *vtom* is used. The example below uses the earlier audit file again:

```
distribute .john.util.vtom +audit=7654321 audit
    +replacement=.john.util.vtom .john.util.CR
```

You see a message such as:

2 objects distributed to 7 end environments

LOGOS updates the used lists of new objects so that the *references* command will reflect the changes caused by the *distribute* command. If one object is replaced by another with the same name but from a different directory, the used list entries are removed from the old path's used list and added to the new path's used list. The audit file associated with the original generation of the end environment is updated to reflect the new objects placed into it.

If you use the *+replacement* modifier with *+show*, the *distribute* command also reports which versions of the replacement objects were picked up and inserted into the end environments.

Setting Compilation Directives

The *distribute* command supports a *+compile* modifier that allows you to set compilation directives for objects before they are distributed. If you specify *+compile* without a value, the objects are compiled in the same way as the last time they were placed into their end environments. You must supply directives for new objects or objects saved into workspaces.

Using *distribute* in a Script

Because the *distribute* command always needs an audit file, you can embed it within a script that provides one. For example, suppose that:

- a cover for the *distribute* command is being prepared for the application built around the audit file *7654321 audit*
- there is an identical development copy of the application audited by the file *7654321 auditdev*
- the compilation evaluation environment (for objects that need one) is either *.john.modules.util.prod.eval* or *.john.modules.util.dev.eval*, depending upon the system being generated
- functions that need an evaluation environment have a compilation directive of *c=e* attached to them
- all functions are to be decommented and, if the production system is being generated, locked as well

A script to do this might look like this:

```
[1] put[1] (+Pathnames=) (+production) (+replacement=) (+task=);c
[2] ⍝ distributes a set of pathnames to <audit> or <auditdev>.
[3] c←'c=.john.modules.util.',3 ¯4[⎕io+production]↑'devprod'
[4] c←c,',e,x',production/',l' ⍝ compilation directives
[5] )distribute \Pathnames +audit=(⍎>('7654321 auditdev'⊃
   '7654321 audit')[⎕io+production]) +compile=(⍎c)
   \+replacement +show \+task
```

When called with the *+production* modifier, the *put* script updates the production version of the application. When called without it, the script updates the development version. Lines 3 and 4 set up the compilation directives to be applied to the objects. Line 5 does the *distribute* operation, first selecting the appropriate audit file, and including the computed compilation directives in the command's modifiers.

Using the Application Debugging Assistant

When you open an object in the workspace for editing while the Application Debugging Assistant is active, LOGOS attempts to locate the source of the object within the LOGOS hierarchy. If it is successful, LOGOS opens the source version for editing instead of the copy in the workspace. When you finish editing and close the object, LOGOS compiles it and distributes the compiled copy to the relevant end environments, including the active workspace. You can resume execution of the application using the edited code now in the workspace.

For example, suppose you have generated a system and in the process of testing it, discover an error in the function *∆Test*. To correct the error, you would:

1 Copy LOGOS to your workspace. For example:

)copy 1 logos

2 Start LOGOS. For example:

   logos

3 Tell LOGOS which audit file is in use for this application using the *environment audit* parameter. For example:

   environment audit 1234567 myaudit

For more information on the *environment audit* parameter, see the *LOGOS Reference Manual.*

4 If you have more than one end environment to be tracked in your audit file, you can select particular end environments to update using the *environment update* parameter. For example:

environment update /w 1234567 myws/p ? ?**

For more information on the *environment update* parameter, see the *LOGOS Reference Manual.*

NOTE: You must set the *environment audit* and *update* parameters every time you invoke the Application Debugging Assistant for the first time in a workspace. Like the *environment task* parameter, *audit* and *update* are workspace session properties that are not saved with your profile. If you find you are using a particular set of *audit* and *update* parameters frequently, you may want to write a script to set these up for you.

5 If you want the object to be distributed to your current active workspaces, or to any other workspace in which it resides, then you must sign on an auxiliary task to be used for the workspace distributions. For example:

signon +t=wsupdate

6 Open the object for editing. If you signed on a task in step 5, then specify the task name as the argument to the *+disttask* modifier. For example:

edit =ΔTest +disttask=wsupdate

NOTE: The editor opens the source from LOGOS rather than the object in the workspace. The editor deduces the source pathname of the object in the workspace by executing a series of searches:

- First, the editor searches the object in the workspace for a parent pathname comment generated by the parent pathname (*p*) compilation directive.
- If none exists, the editor then searches the LOGOS tracking tables in the workspace to see if it can find entries for the object. From this tracking table, the editor deduces the source pathname of the object.
- If still unsuccessful, the editor searches the LOGOS paging table for audit files that can be searched for references to the object.
- Last, the editor examines each audit file in use for references to the object.

If the object name is duplicated elsewhere, LOGOS will find more than one object by the name you specified. In this case, the editor presents a menu of objects it found, and prompts you to select the correct object.

7 Edit and close the object.

The editor redistributes the new version of the object, including to the active workspace if the *+disttask* modifier was specified. Unless you set new compilation directives with the *edit* command, the editor uses the same compilation directives used the last time the object was distributed, using the specified audit file.

You'll see the new version saved into the LOGOS file system followed by a list of the distribution activities as they progress. For example:

.myalias.myfile.directory.test.∆Test[f11]
∆Test to workspace 1234567 myws
∆Test to file 1234567 pagefile 100; ∆testshell
1 object distributed to 2 end environments.

The object is also redefined in the active workspace. Now you can exit LOGOS and resume execution of the application where it stopped, using the changed version.

Updating Paging Areas

If you have changed or added nodes within your paging system, you can update them without rebuilding the nodes that have not changed. This is called an **update generation**. Use the *filesave* command with the *+update* modifier to do this.

■ To use *filesave*, type:

filesave **filename component** *+audit*=**audit** *+update*=**pathnames**

The *+update* modifier specifies the names of the nodes to be regenerated, and signals that this generation is to overlay the previous generation of the paging area.

NOTE: The system must have been generated initially using an audit file. You must specify the name of the audit file when you issue the *filesave* command with *+update*.

You can run update generations just like ordinary ones. All of the *build* and *shell* calls originally used to generate the system are still required, so that LOGOS can re-analyze parts of the system in their proper context. In addition, if you have added any new nodes, you must specify *build* and possibly *shell* commands for them.

LOGOS first finds where in your application each node included in the value given to *+update* is referenced. If the node is new, include both the node's name and the names of any that call it. If a new node is not called by any other node, it is assumed to be called from the base node.

After each new or changed node has been located in the application's systemic structure, LOGOS recomputes its calling tree. This takes into account the depth setting you requested for the node in its *build* statement, and the location of other nodes in the application. For example, if you requested *+depth=all*, the trees of all objects referenced by the node (except those parts that are separate nodes) are included in it.

Example

Because an update requires the same information as the original generation did, the best way to accommodate updates is to provide for them in your standard generation procedure. The script below shows how a system might be generated from scratch, or as an update to the last generation, with minimal operational difference:

```
[1]  gen (+update=)
[2] ⍝ builds <sysbasetest> paging area, with update capability.
[3] )build +depth=all +workdir=(list .john.modules[d] +full +recursive) ⍝ establish defaults
[4] ⍝
[5] )build ⋆ start
[6] )build allocate
[7] )build change
[.]    .  .  .
[.]    .  .  .
[.]    .  .  .
[.] )shell change
[.] )filesave 7654321 sysbasetest 10 +audit=7654321 auditdev \+update
```

To perform an ordinary generation, call the script *.john.modules.util.gen* directly.

To perform an update generation, call the script with the *+update* modifier. For example:

```
.john.modules.util.gen +update=allocate change
```

You see the message:

```
updating 2 nodes of 16 in paging area 7654321 sysbasetest, 10
2 nodes (including 1 shell) generated using 9 objects
generation 12, update 1 of 7654321 sysbasetest, 10 saved 15jan89 18:36 by john
```

The following table lists the differences between this operation and performing the *distribute* command on the page file.

Table 11.4 Differences between Using the *distribute* and *filesave* Commands

The *distribute* Command	**The *filesave* Command**
Moves the objects you specify into the pages that reference them.	Performs the same kind of structure analysis that the original *filesave* did to generate the system.
Does not perform calling tree analysis.	Rebuilds calling trees.
Does not rebuild shells.	Rebuilds shells. Does not include objects that are no longer referenced. Includes objects that are new to the calling trees.

NOTE: If you create a new shell around a new node, this will result in two nodes in the paging file rather than one (see *Chapter 8: Building Applications with LOGOS*).

If you specify a small enough node size using the *+size* modifier, the *filesave* command may split a single node being updated into two.

During an update generation, the value of the *+overwrite* modifier to *filesave* is partially ignored. To allow you to discard prior *build* statements that you might not want processed, *+overwrite=buffer* is honoured. However, *+overwrite=destination* or *audit* is not honoured. These settings contradict the notion of an update, and would destroy information required for the *filesave* command to function meaningfully.

CHAPTER 12: SOFTWARE DEVELOPMENT TOOLS

Displaying and Summarizing Objects

The *display* and *summarize* commands provide you with information about objects in the LOGOS file system to which you have read permission. The *display* command shows you the object's value or definition, or the value of any of its attributes. The *summarize* command returns identifying information about an object, such as type, syntax, size, number of lines, shape, rank and so on.

Displaying Objects

The *display* command takes a single argument: the pathname(s) of the object(s) whose source or attribute is to be displayed. The *display* command may be used to test the effects of various compilation directives on an object.

■ Display the source of the function *elemreplace:*

```
∪ display .public.util.elemreplace
.public.util.elemreplace[f1]:
  ∇ result←vector elemreplace elemstring;b;c;i;n
[1] ⍝ replace, in <vector>, all occurrences of 1↑elemstring by
1↓elemstring. ⎕io-independent.
[2] c←,vector∊1↑elemstring
[3] i←c/⍳⍴c
[4] n←¯2+1⌈×/⍴elemstring
[5] b←((×/⍴vector)+n×⍴i)⍴0
[6] b[(i+n×(⍳⍴i)-⎕io)∘.+(⍳n+1)-⎕io]←1
[7] result←(~b)\(~c)/vector
[8] result[b/⍳⍴b]←(+/b)⍴1↓elemstring
  ∇
```

■ Display the object with compilation directives specified:

```
∪ display .public.util.elemreplace +c=x,p=1,r=0
.public.util.elemreplace[f1]:
  ∇ ⍙99←⍙98 elemreplace ⍙97;⍙96;⍙95;⍙94;⍙93
[1] ⍙95←,⍙98∊1↑⍙97
[2] ⍙94←⍙95/⍳⍴⍙95
[3] ⍙93←¯2+1⌈×/⍴⍙97
[4] ⍙96←((×/⍴⍙98)+⍙93×⍴⍙94)⍴0
[5] ⍙96[(⍙94+⍙93×(⍳⍴⍙94)-⎕io)∘.+(⍳⍙93+1)-⎕io]←1
[6] ⍙99←(~⍙96)\(~⍙95)/⍙98
[7] ⍙99[⍙96/⍳⍴⍙96]←(+/⍙96)⍴1↓⍙97
[8] ⍝⋆.public.util:[1]--ugd
```

- Display an object attribute:

```
∪ display .public.util.elemreplace[:d]
.public.util.elemreplace[fl:d]:
    result ← vector elemreplace elemstring

replaces, in <vector>, all occurrences of 1↑elemstring by 1↓elemstring.
<vector> may be character or numeric; it should not contain enclosures.

examples:

etc. ...
```

- Display a variable with a surrogate substituted for the Carriage Return character:

```
∪ display .public.util.elemreplace[:d] +s=c
.public.util.elemreplace[fl:d]:
    result ← vector elemreplace elemstring¨¨replaces, in <vector>, all
occurrences of 1↑elemstring by 1↓elemstring.¨<vector> may be
character or numeric; it should not contain enclosures.¨¨examples:

etc. ...
```

On some terminals, those that can display it, the surrogate will consist of a dieresis character (¨) overstruck with one of the following symbols representing the character being replaced:

b	backspace	⎕av[⎕io+158]
c	carriage return	⎕av[⎕io+156]
i	idle character	⎕av[⎕io]
l	linefeed	⎕av[⎕io+159]
n	null	⎕av[⎕io+1]

Some terminals will display only the dieresis characters, others will display a "squish-quad."

Summarizing Objects

The *summarize* command displays a variety of information about objects in the LOGOS hierarchy. For all objects, the result includes the object's name, type, version number and size. Additional information is returned depending on the type of object. For functions and scripts, the result includes the syntax and the number of lines. For variables, the result includes an indication of the type of variable (character, boolean, package and so on), its rank, and shape. If a variable is a package or a cluster, *summarize* includes a count of the number of objects it contains. The objects may be individually listed by using the *+expand* modifier.

- Summarize objects in *.public.util* beginning with the letter *a*, and include a heading line:

```
u summarize .public.util.a?* +headings
---------pathname--------- type    ver    attr    size    shape
.public.util.ah            f (ed)  1      [2]     124
.public.util.alf           v (ch)  1      (1)      64     64
.public.util.alloceq       f (ed)  1      [3]     252
.public.util.allocfifo     f (ed)  1      [3]     220
.public.util.assert        f (nd)  2      [3]     172
```

- Display a summary of some scripts:

```
u summ .public.logos.cmds.t?* +headings
---------pathname--------- type    ver    attr    size    shape
.public.logos.cmds.tom     s (ep)  3      [2]      76
.public.logos.cmds.tree    s (ep)  8      [18]   1268
.public.logos.cmds.type    s (ep)  1      [5]     172
```

- Display a summary of a cluster that contains 4 objects and expand the contents of the cluster to display information about those objects:

```
u summ .public.logos.cmds.util.ctlclust +headings +expand
---------pathname--------- type    ver    attr    size    shape
.public.logos.cmds.util.ctlclust
                           c (pk)  2      <4>     972
.public.logos.cmds.util.ctlclust°ctlmsg
                           v (ch)  2      (2)     288     13 22
.public.logos.cmds.util.ctlclust°ctlparm
                           v (ch)  2      (2)     120     13 9
.public.logos.cmds.util.ctlclust°ctlqfi
                           v (bl)  2      (1)       4     13
.public.logos.cmds.util.ctlclust°ctlvalid
                           v (ar)  2      (1)     472     13
```

Locating and Replacing Strings or Patterns

The editor lets you search for a string in a function and change it. Using the *locate* and *replace* commands with **pattern matching**, you can make more sweeping changes to a system that are tedious with the editor. You can find and replace strings in any collection of LOGOS objects.

Using Pattern Matching

LOGOS supports a shorthand, called a **regular expression**, for generating a series of names or strings which match a particular pattern.

There are two kinds of regular expression:

- *Limited regular expressions*, which can appear only in pathnames.
- *Full regular expressions*, which cannot appear in pathnames but are useful in commands such as *locate* and *replace*.

Full regular expressions consist of a *locator template*, which specifies the pattern to be sought, and an optional *action template*, which defines an action to be performed when a match to the locator template is found. The action template comes into play only in replacement operations.

Using Metacharacters

You can use the following metacharacters (and several others) in full regular expressions as special searching characters:

?	Matches any character.
⋆	Performs an action as many times as possible.
|	Denotes alternation.

Unless these characters are escaped, they are interpreted literally and have no special properties in the regular expression.

■ To escape one or more characters (and enable their special searching properties), enclose the characters in braces, as in {*str?*⋆}.

Full regular expressions in command lines must be enclosed in braces.

NOTE: The dieresis (¨) escapes the character which immediately follows it. You can use it as an alternative to braces to enable searching properties. If you use a dieresis within braces, it robs the character which follows it of its special properties.

Using Non-metacharacters

Not all characters are special. The letters of the alphabet and the digits, for example, are not metacharacters. These characters are treated literally whether or not they are escaped. For example, consider the following pattern:

a⋆{b+}c

If ⋆ and + are metacharacters, the + is treated as a metacharacter, and the ⋆ is not. The converse is true in the pattern:

a••⋆{b••+}c

This last pattern is equivalent to the shorter pattern:

a••⋆b+c

The letter b is treated literally in all cases.

Specifying Locator Templates

The locator template specifies the pattern of strings you are searching for. You can express families of strings using:

- special single-character patterns
- alternation, elision
- closure
- certain other special metacharacters

By combining these single-character patterns, complex patterns can be created. For more information on each of these patterns, see the *LOGOS Reference Manual.*

Specifying Action Templates

An action template specifies the processing to take place when a locator template encounters a match of the pattern. It can be used to define complex replacement strings for the *replace* command and for the editor's *change* command.

You can use the following metacharacters the same way you would in a locator template:

{	Enables escaping of characters.
}	Disables escaping of characters.
••	Complements escaped or literal treatment of next character.
⊣	Carriage return character.

If an action template contains only normal text, then that text simply replaces the matched substring in the object string.

An action template provides three important capabilities:

- Allows references to be made to parts of the action template.
- Allows expressions to be evaluated and optionally inserted into the text.
- Provides information about where the match occurred.

■ A portion of matched text can be marked and referred to in an action template. You do this by enclosing the relevant portion of the locator pattern in the metacharacter pair ⊂ ⊃.

These "tagged" strings can then be referenced in the action template by referring to them as ⊂**n**⊃, where **n** is an integer. For each locator template match, each tag in the action template is replaced by the text that matched the **n**th tagged pattern.

For example, with the locator template { ⊂*?*⋆⊃ , ⊂*?*⋆ ⊃ , ⊂*?*⋆⊃ } and the action template { ⊂*3*⊃ , ⊂*2*⊃ , ⊂*1*⊃ }, the object text *first,middle,last* becomes *last,middle,first.*

NOTE: Pattern tags may be nested in a locator template, as in { ⊂*jan* |*feb* ⊂ω⊃⊃ }, but not in an action template. A particular tag may be referenced any number of times in an action template, or not at all.

■ You can specify that parts of an action template are APL expressions to be executed when a match to a locator template is found. Indicate that an expression is to be evaluated by enclosing it in dels (∇).

Optionally, the result of the evaluation can be included in the replacement template if the opening ∇ is immediately followed by an assignment arrow. For example, the action template {*file*∇←⎕*av*[*cnt*←*cnt*+*1*]∇} causes all matches to be replaced by *file* followed by a different letter from ⎕*av*. (*cnt* is assumed to be a global variable, initialized before the template is used in a command.) Pattern tags in evaluated strings are replaced by the appropriate text before the string is executed.

■ Strings in evaluated mode can also reference three template *descriptor variables,* called ⎕*ln*, ⎕*cp*, and ⎕*cn*.

These variables are integer scalars which contain information about the relative location of the current match within the object string.

⎕*ln* (line number) is the line number on which the match was found.

⎕*cp* (cursor position) is the origin-0 index of the first character of the match relative to the beginning of the line.

⎕*cn* (character number) is the origin-0 index of the first character of the match relative to the beginning of the searched text.

NOTE: These are not true APL system variables, but rather are special names which are recognized by the pattern matching processor.

For a formal summary of the full regular expression notation, see the *LOGOS Reference Manual.*

Examples

1 Suppose you've written a function that references a global integer vector named *State*. When the function was originally written, *State* had nine elements. Now you want to add a tenth element, but you want to insert it in the fifth position rather than append it to the end of the vector. The difficulty is that this will invalidate many of the indexed references to the variable in the function.

Even if the indices that reference the variable are constants (normally the difficult case), a single command can be used to correct all of them at once:

```
replace State[{⊂⍵⊃}] State[{∇←⊂1⊃ +⊂1⊃≥5∇}] .syslib.cb.src
```

Here, a pattern is used to locate all indexed references to the variable, and another pattern is used to increment the index by one if it's 5 or more.

As another example, suppose you have character vector objects, or attributes of objects, in which you want to capitalize the first letter following periods or carriage returns. The problem is complicated by the fact that a period may be followed by one or more blanks, or one or more carriage returns, which might also be followed by blanks. The following sequence of commands will perform the replacement.

First, create two character vectors that contain the upper and lowercase alphabets:

```
∪ ⍎upper←⎕av[(¯1+⎕av⍳'A')+⍳26]
∪ ⍎lower←⎕av[(¯1+⎕av⍳'a')+⍳26]
```

Then, perform the replacement, for example, on the documentation attribute of the object *start*:

```
∪ replace {.⊂( +)|(⊣+ ⋆)⊃⊂[a-z]⊃}
{.⊂1⊃∇←('⊂2⊃'=lower)/upper∇} start[:d]
```

In the first regular expression, the locator template has two pattern groups, as indicated by the two pairs of ⊂ and ⊃, and matches a period (.) followed by those groups. The first pattern group, ⊂(+)|(⊣+ ⋆)⊃ , indicates either one or more spaces or (indicated by the | character) one or more carriage returns followed by zero or more spaces (see the *LOGOS Reference Manual* for a discussion of the closure concepts represented by the symbols + and ⋆). The second pattern group, ⊂[*a-z*]⊃ , matches any single character from the sequence: *a b c ... y z.*

In the second regular expression, the action template refers to each of the pattern groups by number. The first reference, ⊂*1*⊃, simply inserts the pattern that was matched by the first pattern group - whatever it was. The second reference occurs in an expression to be evaluated:

```
∇←(' ⊂2⊃ '=lower)/upper∇
```

The lower case character matched by the second pattern group is substituted into the expression where ⊂2⊃ occurs, then the expression is evaluated. Because of the ∇←, the result (a selection from the vector of uppercase characters,) is placed into the object and the capitalization is complete.

3 Suppose you wanted to add a ⍝∇⋄ code tag at the end of any line that used ⎕*lc* in a branch statement, like:

```
[5] →(options≠5)/1+⎕lc  ⋄ dowork
```

The following replace command would add the code tag to statements like the example above:

```
replace {⊂→?⋆¨+⎕lc ¨⋄?⋆⊃⊂( ⍝|⊣)⊃} {⊂1⊃ ⍝ ¨∇¨⋄ ⊂2⊃}
.syslib.cb.src
```

The dieresis characters are used so that each character following the dieresis is interpreted literally, not as a special character.

Cross-Referencing Functions

A function cross-reference is a powerful aid to program development, debugging, and documentation, as it allows quick identification of all uses of certain function and variable names within the functions of a system. The *xref* command computes and displays a cross-reference table for functions stored in LOGOS, which gives the location and type of each reference to each identifier used in each named program.

The *xref* command takes a pathname argument, and returns a cross-reference table as the result:

```
∪ xref .public.util.vtom +s=⍳∊⍴⌈|
.public.util.vtom[f2]:
⎕io   *     7     7     8
a     ra    0     7     7     8
b     lv    0     7←    7     7←    7←    7     7     8                 8
c     lv    0     8←    8     8
dlm   la    0     7
z     rs    0←    7←    7     7     8←    8
⍳     *     7     8
∊     *     7
⍴     *     7     7     8     8
⌈     *     8
|     *     8
```

As shown above, you may include any APL symbols in the cross reference by using them as arguments to the *+symbols* modifier. Cross references will tell you of line labels that are in the function but are never branched to, possibly alerting you to the presence of "dead" code in a function. It will also indicate variables that are localized in the header of the function but never referenced. Suspicious references such as these are indicated by a *?* in a column following the indication of the reference type, left argument, right argument and so on.

Looking at Calling Trees

The *calls* command performs calling tree analysis on a function and returns the names of functions and variables that are referenced, either directly or indirectly, by the function being analyzed. While performing the analysis, the command searches for objects in your work directories, and informs you of any that cannot be found in the current set of working directories. You can instruct the command to return only a list of objects not found, to enable you to track down the missing objects. The depth to which the analysis is carried out is determined by the *+depth* modifier.

For example, to determine the functions and global variables referenced directly by the function *start*, use the following LOGOS command:

```
∪ calls start
ask
menu
run
Loctable
Options
```

To examine the next level, include the *+depth* modifier with an argument:

```
∪ calls start +d=2
ask
checkΔaccess
input
menu
run
share
Ctl
Loctable
Options
Res
```

To examine the complete calling tree for *start*, include *+depth* with no argument, *+depth=all*, or *+depth=0*.

You may exclude objects from being scanned during the analysis, so that objects they reference are not included in the result, and you may specify a set of working directories that are to be searched for the referenced objects:

```
∪ calls start +exclude=menu +w=.sys.cb.ws +d=2
ask
input
run
share
Loctable
Options
```

The calling tree analysis that LOGOS performs during execution of the *calls* command is responsive to the presence of ⍝∇∪ code tags in a function to create a reference to objects that would otherwise be hidden inside quoted strings and executed expressions. *calls* does not analyze quoted strings, and could not analyze expressions such as ⍎*Options*[*ndx*;] for identifiers.

Code such as the execute statement above create breaks in the calling tree. These can be threaded together, however, by using ⍝∇∪ code tags to create the references that LOGOS needs to complete the analysis. For instance, you might have a variable, say, *.sys.cb.ws.Options*, that contains the names of several functions that the user might execute:

```
∪ display Options
.sys.cb.ws.Options[v4]:
calculate
report
enter
change
reset
```

You might also have a line of code in the *menu* function that executes a row of that matrix:

```
⍎Options[ndx;]
```

There are several ways in which you could add a code tag to this line to keep the calling tree intact. You could edit *menu* to explicitly list each function in *Options*:

```
[5] ⍎Options[ndx;] ⍝∇∪ calculate report enter change reset
```

Now the calling tree analysis can continue correctly, since the LOGOS will detect the reference to each of the functions that was previously hidden by the execute statement.

However, if you should change *Options* to add new function names, or remove some, you would have to edit the *menu* function to correct your code tag. A better method for creating the references is to use the pathname of the option matrix itself in the tag:

```
[5] ⍎Options[ndx;] ⍝∇∪ ⍎.sys.cb.ws.Options
```

Now, whenever the calling tree needs to be analyzed, LOGOS will examine the variable *.sys.cb.ws.Options*, for possible identifiers. If changes are made to *Options*, you will not have to edit the code tag. The new references will be picked up any time you perform calling tree analysis.

Documenting Objects with WSDOC

The wstofile command allows you to build a source file suitable for input to the SHARP APL Workspace Documentation Facility (WSDOC). This command also allows you to document other LOGOS attributes, such as compilation directives, documentation, etc.

IMPORTANT: You must be using version 2.2 (or later) of WSDOC to use the *wstofile* command.

To use the command, specify:

wstofile **pathnames**

The argument **pathnames** specifies the objects to be incorporated in the WSDOC source file.

For example:

```
wstofile test.wsfns +attributes +pathnames +wsid=wsfns
srcfile 20 - 1234567 wsfns
28 functions
20 variables
```

This command creates a WSDOC file. When processed by WSDOC, the summary and definition reports contain the full LOGOS pathnames, a LOGOS header line, and any attributes associated with the objects.

For example:

```
wstofile test +attributes=dj +recursive
srcfile 21 - 1234567 clearws
145 functions
42 variables (including 11 scripts)
```

This command creates a WSDOC file from all objects below *test*. Eleven scripts found are included as variables. When processed by WSDOC, the definition reports will contain a LOGOS header line and the documentation and journal attributes of the objects, if set.

For more information on using the *wstofile* command, see the *LOGOS Reference Manual*. For more information on using WSDOC, see the *WSDOC User's Guide*.

Using the *syntax* Command

The *syntax* command computes a report describing static errors within a program. It tests conditions such as illegal characters, symbol juxtaposition problems, mismatched parentheses, brackets, or quotes, and suspicious use of names. The *syntax* command does not actually execute the program; consequently, a tool such as this command should be used to supplement but not replace careful program and system testing.

The syntax of the command is:

syntax **pathnames**

The **pathnames** argument is a list of objects on which you want to compute reports.

Errors are classified into a number of categories. If you specify the *+lines* modifier, the category is represented by a symbol following the line number on which the error was detected. If you specify the *+show* modifier, the category is represented by the symbol under the location where the error was detected. (For more information on the modifiers you can use with the *syntax* command, see the *LOGOS Reference Manual.*)

The syntax command reports:

- Generic syntax errors, which include most incorrect uses of symbols. For example, a dyadic symbol used monadically; an improper outer product; an improperly labelled line; use of branch not as the root function of a statement; or redundant use of a diamond, all constitute syntax errors.

- Parentheses, bracket, and quote errors. For example, mismatched instances of the paired delimiters `(...)`, `[...]`, and `'...'`, respectively.

- Domain errors from apparent use of a character argument where a numeric one was expected. As the *syntax* command does not execute the program, only a limited number of such cases is detected.

- Constant errors, which are illegal formation of numeric constants. For example, *4..1* and *8je4* are illegal constants, whereas *4.1* and *8j8E4* are legal ones.

- Suspicious references, or names which are unusual but may or may not be erroneous in the running application. For example, a local variable which is not assigned a value, or a name which is used to define a line-label more than once, is considered suspicious.

For example:

```
∪ syntax .sys.cb.ws.test +s
.sys.cb.ws.test[f6]: (5 errors)
[5] l3:x←53+⍳ndx)+⎕io
                ^
[8] y←3j.j5
       .
[14] x←9○
        ^
[15] →L7:
      ^
[22] r←'the answer is: ,⍕x
       ^
```

Using the *compare* Command

The *compare* command compares two versions of the same object or directory, or two distinct objects or directories in LOGOS. The syntax of the *compare* command is:

```
compare primaries [secondaries]
```

The argument **primaries** is a list of pathnames to be used as the reference in performing the comparison. The argument **secondaries** is a list of pathnames which are compared against the **primaries**. The differences shown by the *compare* command describe the changes that would need to be made to the secondary paths to attain the primary definitions.

The arguments to the *compare* command can be specified in several ways for several types of comparisons. For example, to compare an object with the previous version of the same object, type:

```
u compare object
```

This command is equivalent to:

```
u compare object[0] object[¯1]
```

To compare one object to another:

```
u compare object1 object2
```

In this case, only one object or directory can be specified for both primary and secondary arguments. If the names of the objects do not match, the types must.

You can also compare each object in a directory to its previous version. For example:

```
u compare directory
```

This command is equivalent to:

```
u compare directory.?*[0] directory.?*[¯1]
```

If you use the *+recursive* modifier, you can compare all of the objects in a hierarchy to their previous versions:

```
u compare directory +recursive
```

To compare two hierarchies, type:

```
u compare directory1 directory2
```

The directories are compared as well as the objects within the directories. If used with the *+recursive* modifier, this command compares entire structures.

To compare an object with any object of the same name in a particular directory, type:

ᵁ *compare* **object directory**

More generally, the first argument can be a list of objects and directories, quoted if necessary. There must be at least one object in the list. The second argument can also be a list of objects and directories. The objects in the primary list, including the objects within each directory in the list, are compared to the objects with the same name in the secondary list. Entries in the primary list that do not have matching objects in the secondary list are reported as *not found in secondaries.* Entries in the secondary list that are not found in the primary list are ignored. For more information on the modifiers you can use with *compare*, see the *LOGOS Reference Manual.*

CHAPTER 13: MOVING DATA BETWEEN SYSTEMS

About Export Files

In LOGOS, you move data between systems using a special kind of file, called an **export file.** Export files are ordinary LOGOS files, but can also be transported to other machines and connected to other LOGOS file systems. An export file can be dumped to tape, transferred, and then retrieved at the receiving site using the standard SHARP APL file utilities.

Exporting Data

Export files are created and built using the *export* command, which copies a list of pathnames to a specified export file. The *export* command differs from the *copy* command in two important ways:

- The destination path must be a LOGOS export file, rather than any LOGOS directory. Because it represents a file, the destination must be a two-level pathname, as in *.john.transfer.*
- The copied paths have the same name as the original paths, with the addition of the export file name as a prefix.

Creating an Export File

You can create an export file using the *export* command with the *+makedir* modifier. Type:

export **path path** *+makedir*

The *+makedir* modifier allows the *export* command to create any directories that don't already exist as it copies the paths.

For example, to export the path *.vp.sys.install* to the as yet non-existent export file *.john.transfer*, type:

export .vp.sys.install .john.transfer +makedir

You see a message such as:

```
.john.transfer[d1]
.john.transfer.vp[d1]
.john.transfer.vp.sys[d1]
.john.transfer.vp.sys.install[f1]
```

The first three paths are newly-created directories (including the export file itself, *.john.transfer*). Version 1 of the object *install* is copied.

By using the export file name as a prefix, the *export* command essentially preserves the original names of the paths. The first two levels of a source path become the third and fourth directory levels of the export file. Therefore, the object *.vp.sys.install* exported to the file *.john.transfer* becomes *.john.transfer.vp.sys.install.*

You can export entire directories. For example:

```
export .vp.sys.newrel .john.transfer +makedir
```

You see a message such as:

```
.john.transfer.vp.sys.newrel[dl]
.john.transfer.vp.sys.newrel.base[dl]
.john.transfer.vp.sys.newrel.base.appendr[fl]
.john.transfer.vp.sys.newrel.base.checklimits[fl]
.john.transfer.vp.sys.newrel.base.checkquotas[fl]
.johm.transfer.vp.sys.newrel.base.fetchhelp[fl]
.john.transfer.vp.sys.newrel.base.fretie[fl]
.john.transfer.vp.sys.newrel.base.init[fl]
        .         .
        .         .
        .         .
```

NOTE: If the directories already exist, you don't need the *+makedir* modifier. To export a new version of the object *init* into the file, you need only type:

```
export .vp.sys.newrel.base.init .john.transfer
```

Any LOGOS operation, not just the *export* command, works on an export file. For example, you can save objects into the file using the *copy* or *save* commands, and you can modify objects using the *edit* command. To display the contents and structure of the file, use the *list* command.

After the export file has been set up satisfactorily, ask your operations staff to dump it from the system. You can then move it to another machine and retrieve it.

Importing Data

Once attached to LOGOS, an export file becomes an ordinary LOGOS file. The *import* command doesn't move any data out of the file being imported. To do this, you can use the *copy* command to distribute the file's contents elsewhere in the file system. You may want to do this after the new software has been vetted by generating and experimenting with test versions of the affected programs or applications.

Importing a File

Once the export file has been retrieved at the receiving site, the *import* command attaches it to the LOGOS file system. For example:

import .john.transfer

You see a message such as:

.john.transfer imported

If you know the name of the file, you can use it as the argument to the *import* command. For example:

import '1234567 ∆transfer'

You see a message such as:

.john.transfer imported

(The ∆ here is optional; LOGOS will provide it if necessary.) These examples assume that the alias and filename on both the sending and receiving sites are the same. That needn't be the case; the file might have been renamed during either of the dump or retrieve operations.

Changing the Filename

The *import* command allows you to change the name of the file by specifying the new name as the second argument.

For example, if you retrieve the export file to another account as *1234567 ∆transfer*, and your alias on the current account is *dave*, you could import the file as *.dave.new* by typing:

import '1234567 ∆transfer' .dave.new

You see a message such as:

.john.transfer imported as .dave.new

Retrieving a File Never Exported

When you are importing a file that was never exported (the file might have been retrieved from an archive tape, for example), you must also specify the original name of the file, if it is different from its retrieved name. Type:

import **path1 path2 path3**

The arguments are:

path1	the file's retrieved name
path2	the name you want the file to have in LOGOS
path3	the file's original name on the archive tape

For example, suppose you retrieve the LOGOS file *1234567 ∆transfer* as *1234567 trans2feb*. You can import the file as *.john.transold* by typing:

import .john.trans2feb .john.transold .john.transfer

The arguments are:

.john.trans2feb	the file's retrieved name
.john.transold	the name you want the file to have in LOGOS
.john.transfer	the file's original name on the archive tape

If the file's retrieved name is the same as the name you want it to have in LOGOS, that is, **path1** is the same as **path2**, you can use `''` instead of specifying **path2**.

Installing New Software

If your export file is designed to install or upgrade an application, include a script called *install* at the level just below the export file name. (Use the *copy* or *edit* commands to save the script there.) The receiving site can simply run this script after the file is transferred.

The script *.public.logos.cmds.install* does the import operation for you, and then runs the script called *install* within the imported file. Here's what *.public.logos.cmds.install* looks like:

```
[1]  install +Pathname= +Newpathname= +Oldpathname=;pn
[2]  )pn←import \Pathname \Newpathname \Oldpathname  ⍝ import file
[3]  )(⍎pn).install (⍎pn)      ⍝ perform custom installation
```

The *install* script within the imported file can be arbitrarily complex. As its argument, it takes the name you chose for the file just imported. This allows the script to deduce the directories within the file to permit the rebuilding of applications using those directories instead of (or on top of) the application's production directories. If you write an installation script for use with *.public.logos.cmds.install*, remember to allow it to take an argument.

Dispersing Objects

At some point, possibly as part of the installation script itself, you will probably want to disperse the objects within the export file elsewhere into your directories. Selective use of the *copy* command can do this. If you are moving everything, you can use a single *copy* command to perform the inverse of all *export* commands used to build the file in the first place.

For example:

copy .john.transfer .

This copies all paths within *.john.transfer* (for example, *.john.transfer.vp.sys.install*) to the target path (for example, *.vp.sys.install*).

If all directories do not exist, include the *+makedir* modifier with the *copy* command. Be sure you are satisfied with the contents of the file you imported before issuing the command.

Exporting for Archival

Export files are useful for archiving applications kept in LOGOS. An export file can be built, dumped to tape, and then deleted from the LOGOS file system using the *delete* command. Then, if the need arises, this tape can be retrieved and the file imported back into the active LOGOS system.

Archiving provides two important facilities. It allows you to:

- move inactive applications from expensive, online storage to cheaper, offline storage

- back up critical applications independently and store them in a secure location

LOGOS files are backed up as part of ordinary system procedures, but you might want to archive an application separately. If you require a LOGOS file that was not exported, you can still use the *import* command to attach it to the hierarchy. If you have changed the name of the file, you may need to provide the third argument to the *import* command, as shown in the preceding section, *Retrieving a File Never Exported.*

CHAPTER 14: PROFILES AND ENVIRONMENTS

Environment Parameters

Many aspects of LOGOS are controlled by environment parameters that are a part of your session. For example, your command separator character and your terminal type are values you can set.

Collectively, you can save most of these parameters as your LOGOS profile so they automatically take effect each time you begin a new session. A few are parameters are workspace session properties; they cannot be saved in your profile. They are indicated in the following table of environment parameters and their functions:

Table 14.1 Environment Parameters

Parameter	Full Name	Workspace Session Properties
audit	Audit file	X
cmddir	Command directories	
compile	Compilation directives	
debug	Debug setting	
entry	Entry command line	
exit	Exit command line	
field	Screen field attributes	
keyword	Keyword definitions	
sepchar	Command separator character	
status	Status line detail	
task	Task identity	X
terminal	Terminal type	
track	Tracking setting	
update	End environment	X
workdir	Working directories	

For a complete description of each of these parameters, see the *LOGOS Reference Manual.*

Default Parameters

Every new LOGOS user has a set of default parameters. These are shown in the following table:

Table 14.2 Default Environment Parameters

Parameter	Default Setting
environment sepchar	ᴗ
environment cmddir	.public.logos.cmds
environment workdir	†
environment compile	
environment debug	off
environment track	on
environment task	*
environment entry	
environment exit	
environment terminal	unspecified
environment status	full
environment field	status=yellow high,title=white high, message=red high,command=green,frame=blue, input=green, output=turquoise

† Users' alias is the default working directory.

Displaying Parameters

You can display your current environment parameters, which may not be the same as your default parameters. If you change any parameters during your LOGOS session, this command shows the current values.

To display all of your parameters, type:

environment

LOGOS returns a list of your current environment parameters.

To display a single parameter, specify the parameter. For example, to display your debug setting, type:

environment debug

Setting Parameters

You set environment parameters with the *environment* command. To set a parameter, type:

environment **parameter value**

The new value of the parameter is immediately set and, unless you change it again, remains at that value for the rest of your session. The new setting is not maintained across sessions unless it is a parameter you can save into your profile. (See the section, *Saving Parameters*, in this chapter.)

For example, to set your debug parameter on, type:

environment debug on

NOTE: If you exit LOGOS and re-enter in the same workspace, your environment parameters (and your alias) are maintained across those sessions.

Resetting Parameters

If you change any environment parameters during your session, you can change back to values saved in your profile. To reset all parameters, type:

environment +reset

To reset a single parameter, specify the parameter. For example, to reset your debug parameter, type:

environment debug +reset

Saving Parameters

You can change the default of an environment parameter by changing the parameter during a LOGOS session and saving it to your profile. Type:

environment **parameter value** *+profile*

For example:

environment workdir .mde.util +profile

This command will not affect the settings of any other parameters.

Stacking Environments

LOGOS can maintain copies of your environment in the environment stack. This is so you can make temporary changes to your environment parameters. For example, you can change your working directory, task, or compilation directives temporarily. You can make a local modification to your environment from within a script and when the script completes execution, revert to the original environment, or the previous environment on the stack.

You stack an environment using the *environment* command with the *+stack* modifier. To stack an environment, type:

environment +stack

This puts your current environment parameters aside, and allows you set new ones. You can proceed to change the environment parameters. For example, you might change your working directory:

environment workdir .blue.blue

To stack an environment and change your working directory at the same time, type:

environment workdir .blue.blue +stack

The environment is stacked first, then the working directory is changed.

When a script completes execution, it destacks automatically. To destack environments manually, type:

environment +destack

You can also stack multiple environments. See *Chapter 5: Using Scripts* for more information on stacking different environments from several scripts.

CHAPTER 15: USING THE UTILITY LIBRARY

Figures

Tables

About the Utility Library

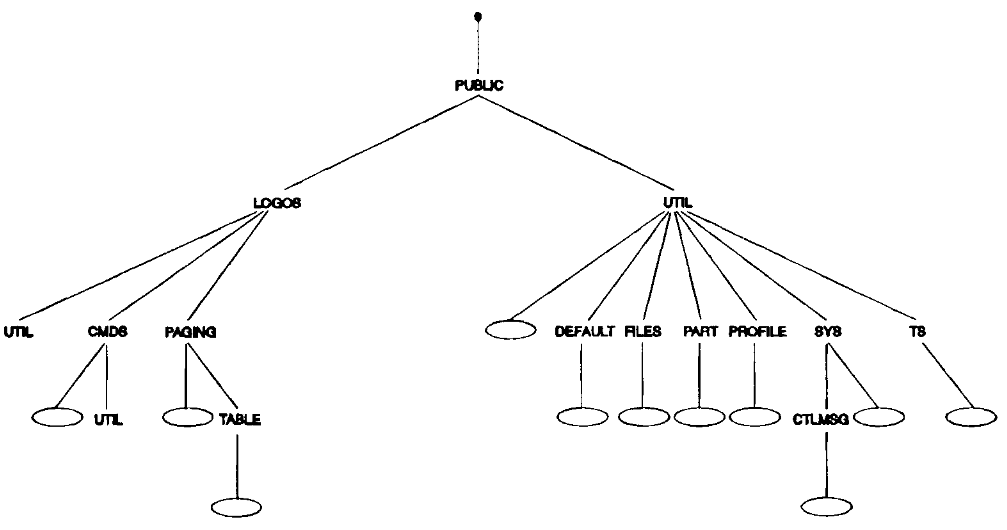
The primary purpose of the utility library is to provide a central location for objects that may be useful to many different programmers working on many different applications. Most of the utilities are fast, efficient functions that you can use in most applications.

Many objects in the utility library are related to the SHARP APL system. Examples are the *hsp* function from workspace *1 hsprint*, the *clearout* function from workspace *1 wsfns*, and the Backspace, Linefeed, and Carriage Return characters. These objects are brought together in one place, to make it easier for you to include them in an application.

Structure of the Utility Library

The following diagram illustrates the structure of the public directories.

Figure 15.1 LOGOS Utility Library



Virtually every object in the utility library, including directories, functions, variables, scripts, and clusters, is documented in its documentation attribute. Several subroutines of scripts, located in *.public.logos.cmds.util* are not documented because they are not intended to be used directly.

For a description of each directory in the utility library, see the section *Contents of the Utility Library* in this chapter.

Using Objects from the Utility Library

You can use objects from the utility library:

- in end environments
- during compilation of objects
- during a LOGOS session

Descriptions and examples of these are presented below. To include the *logos* function itself in an application, you'll find it through the path *.public.logos.logos.*

Library Objects in End Environments

To use objects from the utility library in end environments, you can:

- include the utility directories in your list of working directories
- establish links from your working directory to the objects you want to use
- explicitly reference the objects you want to use via the *get* or *build* commands

Using Working Directories

When generating an application system, you can include the utility library directories in your list of working directories, and access any of the objects contained in them.

For example, while generating a system with references to file functions such as *Δfstie* and *Δfappend* or more esoteric ones such as *Δfcopy* and *Δfmappend,* you can specify the *.public.utilfiles* directory in your argument to the *workdir* command. A subsequent *build* command (with calling tree analysis) will then make the required functions available to your system.

Using Links

Links are appropriate when you need the occasional object from a directory.

For example, suppose the working directories for your generation are *.invent.src.general* and *.public.util.* There is a function *prompt* in the *.public.util* directory that you know you'll call several times. Instead of having LOGOS first look in *.invent.src.general* and then search *.public.util,* you can specify a link to the object:

link .invent.src.general.prompt .public.util.prompt

Now, you can set your working directories to just *.invent.src.general.* When LOGOS performs tree analysis, it locates the *prompt* link in *.invent.src.general* and resolves it to the function in *.public.util.* You could also copy the function directly into your own directory, but by linking to an object, you are always accessing the latest version of it. This is especially relevant when using objects from public directories.

Using Explicit Referencing

Another common technique for accessing needed objects in the utility library is to make an explicit reference to the objects, using the *get* or *build* commands. For example:

```
get .public.util.cr +task=genws
```

or

```
build <def> .public.util.default
```

Library Objects with Scripts

A script can include pathnames in its header. When you execute the script, it fetches the objects specified in these pathnames. For example, if the functions *sqz* and *vtom*, and the variable *CR* were required by the script *compose,* its header might look like this:

```
[1] txt←compose (+output=) ; .public.util.sqz|vtom|CR
```

You can also specify an object with a relative pathname. These are resolved from the working directories set with the *w* compilation directive. For example, with a working directory of *.public.util*, all of the objects needed to execute the following script are available during its compilation:

```
[1]  z←tom +Vector=;cr.;default.⎕ps
[2]  z←⍕<⍪¯1⍤<cr,Vector
```

Library Objects and Compilation

The source form of an object is compiled whenever it is processed by the *build, distribute*, or *get* commands. The LOGOS compiler allows the specification of pathnames as arguments to the *c* compilation directive, so that certain objects can be guaranteed to be available for use during compilation. For example, objects that are required during execution of ⍝∇⍎ code tags must exist either in the workspace or, more conveniently, in a path specified by the *c* directive.

Utilities are often useful when evaluated expressions are compiled. To illustrate, suppose the following line appears in a secure function:

```
[3] 'sysfile' ∆fstie 0 ⍠' ',prompt 'passno? '⍠ ⍝∇⍎⍠
```

If the function's compilation directives (or those specified through *environment compile*, or the *+compile* modifier to the relevant command) include *c=.public.util.prompt*, the compilation of the object prompts for the file passnumber and includes it directly in the program. Suitable compilation directives might be as follows:

+compile=e,c=.public.util.prompt,l

The *e* directive requests evaluation of ⍝∇⍎ code tags; the *c* directive specifies the evaluation context; and the *l* directive locks the function.

Library Objects in a LOGOS Session

The scripts in *.public.logos.cmds* provide an extension to the LOGOS commands. You can include the directory itself in your list of command directories using the *cmddir* command. (It is there by default.) You can use the scripts as if they are ordinary LOGOS commands. For example, you could use the *.public.logos.cmds.submit* script like this:

submit 12 +specs=erase +delivery=deliver to john +type=hsprint

This line submits the file tied to 12 for printing with the processing specifications and delivery instructions as specified by the appropriate modifier.

If the script is not accessible through your command directories, you can still call it directly:

.public.logos.cmds.submit 12 +specs=erase +delivery=deliver to john +type=hsprint

Contents of the Utility Library

The table below summarizes the contents of the directories in *.public*.

Table 15.1 Contents of the Utility Library

Directory	**Contents**
.public.logos	The *logos* and *∆logos* functions, and the two directories *cmds* and *paging*.
.public.logos.cmds	Scripts that extend or supplement LOGOS commands and scripts that supply system interface, session manager, or general capabilities. One directory, *util*, is located at this level.
.public.logos.cmds.util	Subroutines and auxiliary objects for use by the scripts in *cmds*.
.public.logos.paging	Utility functions for use with LOGOS paging applications. These functions are discussed in detail in *Appendix C*. One directory, *table*, is located at this level.
.public.logos.paging.table	A single variable, *LPT*, which is a prototype of the one used by the LOGOS paging utilities to control and monitor paging. It is not necessary to include this variable in a user application, as it is dynamically generated whenever paging is used. It is included here mainly for the user's information.
.public.logos.util	Three utility functions: *∆lcomputil*, *∆ledutil*, and *∆lreclaim*.
.public.util	The majority of the objects in the LOGOS utility library. There is also a series of directories at this level, which subdivide specialized utilities into separate categories.
.public.util.default	The default values for all APL system variables.
.public.util.files	Utility functions useful in the generation and manipulation of SHARP APL files. Most of the functions extend the concept of an existing file system primitive. These functions are discussed in detail in *Appendix B: File Utilities*.
.public.util.part	Functions that are useful for dealing with partitioned vectors.
.public.util.profile	Functions and variables necessary to implement and use an APL profile function in a workspace.
.public.util.sys	APL system constants and utilities, many of which are interfaces to other facilities such as the B-task scheduler, *hsprint*, *hcprint*, and *filesort*. One directory, *ctlmsg*, is also located at this level.
.public.util.sys.ctlmsg	Functions used to append control messages to APL files. These messages are most frequently used with the *hsprint* and *hcprint* facilities.
.public.util.ts	Utility functions and variables used to manipulate system timestamps.

Documentation for Library Objects

The documentation for a directory describes the purpose of the directory, and summarizes the objects within it (including subordinate directories). When you are searching for a utility, check the documentation on the likely parent directory.

The documentation for an object other than a directory specifies general usage information about the object (such as its syntax if a function, or its characteristics if a variable); describes how the object is used; and often gives an example.

Using Tag Attributes

Many of the objects in the public directories have a tag (*:t*) attribute. This tag is a character vector describing features or side-effects of the utility. The tag is analogous to a keyword (or a series of them), but is designed so that it can be analysed by a program.

You can search for utilities with a specific tag using the *locate* command. For example, to find all utilities under *.public.util* that have side effects, use:

```
locate side .public.util[:t] +recursive
```

The table below summarizes the interpretation of each utility tag used. Utilities to which more than one characteristic apply have a tag that includes all relevant characteristics, separated by blanks, in alphabetical order.

Table 15.2 Utility Tags

Tag	Significance
⎕io-resp	The utility is *⎕io* responsive. It may return an origin-sensitive result, or it may have an outcome that is directly affected by the value of *⎕io* in the calling environment. Such utilities assume that *⎕io* has a valid setting.
restartable	The utility can be safely restarted with the expression →*⎕lc*.
conflict	The utility may be susceptible to name conflicts. For example, it may need to refer to an object in the caller's environment by name, or it may need to execute an expression that is passed to it. For such utilities, the chance of conflict is generally reduced through the use of local names such as *∆99*.
global	The result or effect of the utility is, or can be, affected by aspects of the caller's environment (other than system variables). Any utility that uses the file system or requires the value of *⎕ts* (not given as an argument) is tagged with *global*.
input	The utility may stop for input from the user (including full screen reads).
output	The utility may produce output (including full screen writes).
side	The utility may have side effects. For example, the *clearout* function has the side effect of giving a *clear ws* the next time the session returns to immediate execution mode.
sv	The utility requires the shared variable processor to be available. Possible effects of this are failure due to inability to obtain a unique clone id, and possible waits on the shared variable processor. Utilities tagged with *sv* likely also have a *global* tag, because there is generally an external partner upon whom the program depends.
term	The utility may take advantage of special features of the device on which it is running, or may fail if the device is deficient.

APPENDIX A: USING REGULAR EXPRESSIONS

Tables

About Regular Expressions

LOGOS supports a shorthand for generating a series of names or strings which match a particular pattern. This shorthand is called a *regular expression,* because the patterns are built of simple expressions which obey a small and simple set of rules. While the components of a regular expression are straightforward, the resulting pattern can have tremendous power. For example, a simple pattern might allow the matching of all names beginning with the same prefix, or of all words containing the same sequence of letters anywhere within them.

You may specify a regular expression in an argument or in the value of a modifier to a Logos command. The use of regular expressions saves typing, and also makes possible the selection of families of strings which would otherwise be awkward to express. It encourages systematic naming of directories and objects, and allows sets of names to be processed as easily as single ones.

There are two kinds of regular expression:

- *Limited regular expressions,* which can appear only in pathnames.
- *Full regular expressions,* which cannot appear in pathnames but are useful in commands such as *locate* and *replace.*

Using Limited Regular Expressions

Limited regular expressions are used in pathnames to generate a sequence of names which match a particular pattern. The *metacharacters ?*, ⋆, and | combine with partial pathnames to create these expressions. For example, *list temp?⋆* lists those names in the current working directory that begin with *temp.* Here, *?* stands for "match any character," and ⋆ means "as many times as possible."

To find all two-letter names in *.dick.util*, you would type:

```
list .dick.util.??
```

This returns a result such as:

```
.dick.util.ah
.dick.util.sh
```

You may also use | to specify alternatives. For example, to list all names beginning with the string *util* or *tool*, type:

```
list util?⋆ | tool?⋆
```

To display the source form of the objects *device.io.utils* and *device.io.tools*, type:

display device.io.utils | tools

Using Full Regular Expressions

Full regular expressions are more general and, correspondingly, more powerful than limited ones. They can be used in the arguments to the *locate* and *replace* commands, as well as within the LOGOS editor when performing string searches or replacements. Full regular expressions are not presently permitted for selecting pathnames.

Full regular expressions consist of a *locator template*, which specifies the pattern to be sought, and an optional *action template*, which defines an action to be performed when a match to the locator template is found. The action template comes into play only in replacement operations.

Metacharacters such as *?*, ⋆, and |, and several others which will be discussed below, can be used in full regular expressions to represent special searching characters. Unless these characters are escaped, they are interpreted literally and have no special properties in the regular expression.

To escape one or more characters, and thereby enable their special searching properties, enclose the characters in braces, as in {*str?*⋆}. Full regular expressions in command lines must be enclosed in braces.

Dieresis (¨) has the effect of escaping the character which immediately follows it, and may be used as an alternative to braces to enable searching properties. If dieresis is used within braces, it has the effect of robbing the character which follows it of its special properties.

Not all characters are special. The letters of the alphabet and the digits, for example, are not metacharacters. These characters are treated literally whether or not they are escaped. For example, if ⋆ and + are metacharacters, then in the pattern *a*⋆{*b*+}*c*, + is treated as a metacharacter and ⋆ is not. The converse is true in the pattern *a*¨⋆{*b*¨+}*c*. This last pattern is equivalent to the shorter pattern *a*¨⋆*b*+*c*. Of course, "*b*" is treated literally in all cases.

In the sections which follow, *regular expression* is intended to mean *full regular expression*.

Locator Templates

The locator template specifies the pattern of strings you are searching for. You can express families of strings using special single-character patterns, alternation, elision, closure, and certain other special metacharacters. By combining these single-character patterns, complex patterns can be created.

Single-character Pattern Components

The metacharacter *?* matches any character except CARRIAGE RETURN. Thus the pattern {*t?n*} matches the strings *tan, tbn, tcn,* and so on. When searching a function, {*?* } matches any character except statement end or line end.

The CARRIAGE RETURN character is represented by the metacharacter ⊣, and is used to locate occurrences of a pattern at the extreme of a line. For example, {⊣*110*} matches occurrences of *110* which begin a line, and {*house.*⊣} matches occurrences which end a line.

The pattern [ccl] specifies any character from the character class represented by **ccl**. For example, {*t*[*aeio*]*n*} matches the strings *tan, ten, tin,* and *ton*. {*t*[~*aeio*]*n*} matches any three-character strings beginning with *t* and ending with *n* other than those listed. You can also specify a range of characters in a class, as in { [*a-e*] } which matches the first five letters of the alphabet, or { [*a-*<u>Δ</u>] } which matches any letter. { [*a-z67*] } matches any letter of the standard alphabet or the digits 6 or 7. Ranges are based on the following ordered set:

abcdefghijklmnopqrstuvwzyzΔ
ABCDEFGHIJKLMNOPQRSTUVWXYZ<u>Δ</u>
0123456789

Finally, { [*a-9~01*] } matches any letter of the extended alphabet or any digit, except 0 or 1.

Alternation, Grouping, and Elision

The metacharacter | denotes alternation. For example, the pattern component {*up*|*down*} matches the string *up* as well as the string *down*. Parentheses may be used to group expressions, as in {*frow*(*sty*|*zy*) }.

A component followed by the metacharacter – denotes an optional expression. For example, {*the* (*old*)–*dog*} matches the string *the old dog* as well as the string *the dog*. Note that, unless the optional phrase is a single-character component, it must be grouped inside parentheses.

Closure

The metacharacter ⋆ specifies closure, and matches the phrase it follows any number of times (including zero). For example, the pattern {*the* (*old*)⋆*dog*} matches the strings *the dog, the old dog, the old old dog,* and so on. One particularly valuable closure is {*?*⋆}, which matches a string of non-CARRIAGE RETURN characters.

The metacharacter + specifies a different kind of closure, called positive closure, and matches the phrase it follows one or more times.

A closure pattern matches the longest possible sequence of characters. For example, with the pattern {*ab*⋆} and the string *abbb*, the closure component matches all three *b*s, and not just the first one.

Miscellaneous Components

Several regular expression metacharacters have been specifically designed to facilitate processing of APL functions.

The metacharacter ⋄ is the statement delimiter character, and matches the beginning or end of an APL statement.

The underscore _ is used as the context delimiter character. A pattern bordered by _ is treated as a *syntactic element,* and the context in which a potential match appears determines if it is an acceptable match. For example, the pattern {*_yes_*} matches the word *yes,* but not *yesterday* or *eyes.* If _ appears on only one side of a phrase, only the left or right context of a potential match is checked. For example, {*_yes*} would match *yes* and *yesterday,* but not *eyes.* The context delimiter works in a similar manner with numeric strings.

The α metacharacter matches any identifier, and is functionally similar to the pattern {¨α|¨ω|([*a*− ⍙][*a*−*9*]⋆)} for valid names. Also, {ω} matches any numeric string, and is similar to the pattern {[¯*0*−*9*][¯.*ej0*−*9*]⋆} for valid numbers.

Action Templates

An action template specifies the processing to take place when a locator template encounters a match of the pattern. It can be used to define complex replacement strings for the *replace* command and for the editor's *change* command.

The escaping metacharacters {, }, and ¨, and the CARRIAGE RETURN character ⊣, have the same meaning in an action template as they do in a locator template. If an action template contains only normal text, then that text simply replaces the matched substring in the object string.

What does an action template do that is special? It provides three important capabilities:

- Allows references to be made to parts of the action template.
- Allows expressions to be evaluated and optionally inserted into the text.
- Provides information about where the match occurred.

A portion of matched text can be marked and referred to in an action template. You do this by enclosing the relevant portion of the locator pattern in the metacharacter pair ⊂ ⊃. These "tagged" strings may then be referenced in the action template by referring to them as ⊂n⊃, where **n** is an integer. For each locator template match, each tag in the action template is replaced by the text that matched the **n**th tagged pattern. For example, with the locator template {⊂*?*⋆⊃,⊂*?*⋆ ⊃,⊂*?*⋆⊃} and the action template {⊂*3*⊃,⊂*2*⊃, ⊂*1*⊃}, the object text *first,middle,last* becomes *last,middle,first.* pattern tags may be nested in a locator template, as in {⊂*jan* |*feb* ⊂ω⊃⊃}, but not in an action template. A particular tag may be referenced any number of times in an action template, or not at all.

You may specify that parts of an action template are APL expressions to be executed when a match to a locator template is found. Indicate that an expression is to be evaluated by enclosing it in dels (∇). Optionally, the result of the evaluation can be included in the replacement template if the opening ∇ is immediately followed by an assignment arrow. For example, the action template *{file*∇←⎕*av*[*cnt*←*cnt*+*1*]∇} causes all matches to be replaced by *file* followed by a different letter from ⎕*av*. (*cnt* is assumed to be a global variable, initialized before the template is used in a command.) Pattern tags in evaluated strings are replaced by the appropriate text before the string is executed.

Strings in evaluated mode may also reference three template *descriptor variables,* called ⎕*ln*, ⎕*cp*, and ⎕*cn*. These variables are integer scalars which contain information about the relative location of the current match within the object string. ⎕*ln* (line number) is the line number on which the match was found. ⎕*cp* (cursor position) is the origin-0 index of the first character of the match relative to the beginning of the line. Finally, ⎕*cn* (character number) is the origin-0 index of the first character of the match relative to the beginning of the searched text. Note that these are not true APL system variables, but rather are special names which are recognized by the pattern matching processor.

Examples of Using Full Regular Expressions

```
∪ locate l{[0-9]+}: .john.util
```

This expression locates all line labels of the form lxx, where xx is any integer.

```
∪ locate '⍎('{?⋆}')/''→' .john.util
```

This expression would find any executed statement that is used to activate a branch.

```
∪ replace {⊂⎕trap←'?⋆⊃→⊂α⊃'} {⊂1⊃→',⍕⊂2⊃} .john.util +s
```

This expression converts occurrences of simple ⎕*trap* statements like ⎕*trap*←'∇ *1 e* → *wsfull*' to the form ⎕*trap*←'∇ *1 e* →',⍕ *wsfull.*

When using full regular expressions to make significant changes to objects in your hierarchy, test their effectiveness on a copied subset of your hierarchy before applying the expression to all the objects.

For additional examples, see *Chapter 12: Software Development Tools.*

Template Metalanguage Summary

This section provides a formal summary of the full regular expression notation.

Table A.1 Summary of Template Metalanguage

Type	Language	Description
Locator template components	rx	Any regular expression
	rrx	Restricted regular expression: any regular expression not containing a closure component
	cp	Any single-character pattern
Escape mechanisms	{	Enable escaping of metacharacters
	}	Disable escaping of metacharacters
	¨	Complement escaped or literal treatment of next character
Single-character components	c	The character specified by c
	?	Any character except CARRIAGE RETURN (or statement end if searching function text)
	[c1c2c3]	Any of the characters c1 c2 c3
	[c1-c2]	Any letter of the alphabet or digit between and including c1 and c2 (character range)
	[cs1~cs2]	All characters in set cs1 which do not appear in set cs2 (cs1 and cs2 may include a character range)
	[~cs2]	All characters in the global character set which do not appear in character set cs2
Alternation, grouping, elision and closure		
	rx1 | rx2	Either of the regular expressions rx1 or rx2 (alternation)
	(rx)	Groups the regular expression
	cp- (rx)-	Zero or one occurrences (elision)
	cp⋆ (rrx)⋆	Zero or more occurrences (closure)
	cp+ (rrx)+	One or more occurrences (positive closure)
Miscellaneous components	⋄	Statement delimiter
	⊣	Line delimiter
	_	Context delimiter
	α	Any identifier
	ω	Any number
	rx?	Pattern tag
Action template components	⊣	CARRIAGE RETURN
	⊂n⊃	String matching pattern tag n in locator template
	∇	Evaluated string delimiter
	←	Replace string with evaluated expression (recognized only when appearing immediately after the first of a pair of ∇'s).

APPENDIX B: FILE UTILITIES

The file utility functions are used to assist in the creation and manipulation of SHARP APL files, particularly in scripts. The functions described are in the LOGOS public directory *.public.utilfiles*, and may be accessed by any user of LOGOS.

r← *time* **t**

Formats a single ⎕*rdci* style timestamp as mm/dd/yy hh:mm:ss:ms

ts←*timen* **ts**

Formats one or more ⎕*rdci* style timestamps to ⎕*ts* format:

yyyy mm dd mm hh ss ms

The result is a seven column matrix if more than one timestamp is provided as an argument.

cno ← data Δ*fappend* **tn**

Appends **data** to the file tied to **tn. tn** may be a two-element vector, whose second element is the file's passnumber.*file full* is trapped and recovered from. **cno** is the number of the new component in the file.

cnt ← src Δ*fbackup* **dest**

Backs up the file tied to **src** into the file tied to **dest**. Either argument may be a two-element vector whose second element is the file's passnumber.*file full* is trapped and recovered from. After copying, the last component of the destination file is an additional component containing the access matrix of the source file. **cnt** is the number of components actually copied. For example:

101 Δ*fbackup 202*
284

This backs up all of the file tied to 101 into the file tied to 202.

cnt ← src Δ*fcopy* **dest**

Copies the file data specified by **src** into the file area specified by **dest**. **src** is a vector containing the source file tie number, starting component, ending component plus 1, and passnumber; only the first element must be given. **dest** is a vector containing the destination file tie number, starting component, and passnumber; only the first element must be given. In either argument, zeroes may be used to indicate elided elements. **dest** may be enclosed, and linked to the fill element for appends. If a fill value is required but not specified, ' ' is used. **cnt** is the number of components actually copied.*file full* is trapped and recovered from. For example:

101 Δ*fcopy 202*
284

This copies all of the file tied to 101 into the file tied to 202.

```
101 200 Δfcopy 202
84
```

This copies all components of file 101 from 200 onward to file 202.

```
101 200 Δfcopy 202 300 ⊃ 0
84
```

This copies all components of file 101 from 200 onward to file 202, starting at component 300. File 202 is padded, if necessary, with components containing numeric scalar 0.

atno ← fileid *Δfcreate* **dtno**

Creates the file identified by **fileid**, tied to **dtno**, if possible. If **dtno** is 0, the lowest available tie number is used. **dtno** may be a two-element vector whose second element is the file's passnumber. If you specify a passnumber, an access matrix is set for the file giving you *¯1* permission with that passnumber. **atno** is the tie number the file is actually tied to. For example:

```
'501 inventory' Δfcreate 1 +⌈/0,⎕nums
203
```

Creates file *501 inventory*, tied to one number greater than the highest current tie number.

data *Δfcwrite* **locn**

Conditionally writes **data** to a particular file component, appending components to make the file the required length. If the specified component already exists, it is not replaced. **locn** is a two- or three-element integer vector containing the destination file's tie number, component number, and (optionally) passnumber; **locn** may be enclosed and followed by the fill element for padded components. If no fill element is specified, `''` is used. *file full* is trapped and recovered from.

For example:

```
items Δfcwrite 101 301
```

This writes *items* to component 301 of file 101. Any padded components contain `''`.

```
items Δfcwrite 101 301 ⊃ 0
```

This writes *items* to component 301 of file 101. Any padded components contain numeric scalar 0.

cno ← *Δffirst* **first**

Drops components from a file so that a particular component number is the first in the file. **first** is a two- or three-element vector containing the file's tie number, the component number to be first, and, if necessary, the file's passnumber. **cno** is the number of the first component in the file when the program is finished. The program fails if the file has a first component greater than that requested.

```
∆ffirst 101 200
200
```

This drops components 1 through 199 of file 101; the first component in the file is number 200.

cno← [fill] ∆*flast* last

Appends or drops components so that a particular component number is the last in a file. **fill** specifies the value to be appended to the file, if necessary; if you omit **fill**, ' ' is used. **last** is a two- or three-element integer vector containing the file's tie number, the component number to be last, and, if necessary, the file's passnumber. **cno** is the number of the last component in the file when the program is done.

```
∆flast 101 300
300
```

This appends or drops components so that component 300 is the last component in file 101.

```
0 ∆flast 101 300
300
```

As above; but any appended components contain numeric scalar 0.

cno ← data ∆*fmappend* control

Appends **data** to a file, a specified number of times. **control** is a two- or three-element integer vector containing the file tie number, count (number of times to append the data), and, if necessary, passnumber. **cno** is the number of the last component in the file when the program is done. *file full* is trapped and recovered from.

```
'reserved' ∆fmappend 101 10
310
```

This appends *reserved* ten times to file 101. The last component in the file is now 310.

cid ← ∆*fname* fileid

Returns the file identifier in **fileid** in the canonical 22-character format. If **fileid** does not identify a file, **cid** is empty.

```
ρ⎕←∆fname '501 inventory'
      501 Inventory
22

  ρ⎕←∆fname '501'

0
```

atno ← fileid Δ*fopen* dtno

"Opens" file **fileid**, tied to **dtno**, if possible; that is, the specified file is share-tied or, if necessary, created. **dtno** may be a two-element vector whose second element is the file's passnumber. If you specify a passnumber and the file is created, an access matrix is set for the file giving you ¯1 permission with that passnumber. If **dtno** is 0, the lowest available tie number is used. **atno** is the actual tie number of the file.

```
'test' Δfopen 0
2
```

This ties (or creates) file *test* tied to the lowest tie number available, which was 2 in this case.

data Δ*freplace* locn

Replaces **data** into a specified location. **locn** is a two- or three-element integer vector containing the file tie number, component number, and, if necessary, passnumber.*file full* is trapped and recovered from.

atno ← fileid Δ*fstie* dtno

Share-ties file **fileid** to tie number **dtno. dtno** may be a two-element vector whose second element is the file's passnumber. If **dtno** is 0, the lowest available tie number is used. **atno** is the actual tie number of the file. For example:

```
'501 inventory' Δfstie 0
3
```

atno ← fileid Δ*ftie* dtno

Exclusively ties file **fileid** to tie number **dtno. dtno** may be a two-element vector whose second element is the file's passnumber. If **dtno** is 0, the lowest available tie number is used. **atno** is the actual tie number of the file. For example:

```
'501 inventory' Δftie 0
4
```

data Δ*fwrite* locn

Writes **data** to a particular file component, appending components if the file is not long enough. **locn** contains the tie number, component number, and optional passnumber. **locn** can be enclosed and followed by the fill element for padded components. If no fill element is specified, '' is used. For example:

```
items Δfwrite 101 300
```

This writes *items* to component 300 of file 101. Any padded components contain ''.

```
items Δfwrite 101 300 ⊃ 0
```

This writes *items* to component 300 of file 101. Any padded components contain numeric scalar 0.

APPENDIX C: PAGING UTILITIES

The paging utility functions are used to assist in the running and debugging of applications which use LOGOS paging. The functions described below are in the public directory *.public.logos.paging*, and may be accessed by any user of LOGOS.

linenumberlist←[**linenumberlist**] ∆*lfnstop* '?' | **fnnamelist**

Sets stop vectors for the specified functions whenever they are paged in. If a function already exists in the workspace, its stop vector is set immediately. *?* indicates that the names of functions which already have stop control set are to be returned. **fnnamelist** specifies the functions on which stop control is to be displayed or set; it may be a partitioned vector or a matrix. **linenumberlist** may be a simple vector (for one function) or a vector of enclosures (for more than one function); if you omit it, the current stop vectors are returned.

linenumberlist←[**linenumberlist**] ∆*lfntrace* '?' | **fnnamelist**

Sets trace vectors for the specified functions whenever they are paged in. If a function already exists in the workspace, its trace vector is set immediately. *?* indicates that the names of functions which already have trace control set are to be returned. **fnnamelist** specifies the functions on which trace control is to be displayed or set; it may be a partitioned vector or a matrix. **linenumberlist** may be a simple vector (for one function) or a vector of enclosures (for more than one function); if you omit it, the current trace vectors are returned.

∆*lkeepin* **nodelist**

Indicates that the specified nodes are to be exempt from paging out. The nodes in **nodelist** need not be in the workspace; if they are not, they become exempt when they are paged in.

package←[**arealist**] ∆*lpage* **nodelist**

Returns a package containing a composite of the contents of the nodes specified in the right argument. If you omit **arealist,** the page areas specified by the ∆*lpagefile* function are searched. Objects that are returned by ∆*lpage* are not considered paged in, and hence are not subject to page-out.

[table←] [flags] Δ*lpagefile* **arealist**

Specifies the page areas to be used by your application, causes the base nodes of the specified areas to be paged into the active workspace, and sets up the global paging table in a variable named *LPT*. If object names from several base nodes conflict, the objects first paged in take precedence. Page areas are searched in the order that they are specified. This function is usually present in the application workspace in order to start the paging process.

arealist is in the form **areaname⊃areaname⊃**...; if you specify only one areaname, it needn't be enclosed.

areaname is in the form **fileid** [**cn** [**pn**]], where **cn** is the component number which corresponds to the start of the page area, and **pn** is the file's passnumber.

flags is a two-element Boolean vector. The first element indicates whether the base node is to be materialized; 1 indicates that it is, 0 indicates that it is not. If you omit **flags**, the base node is materialized.

The second element indicates whether the page table is to be defined as a side effect, or if Δ*lpagefile* is to return the page table as a result; 1 indicates that the function is to return a result. If you omit **flags**, the page table is defined as a global variable in the workspace, and the function has no result.

Some examples:

Δ*lpagefile '501 inventory'*

This selects the (presumably only) page area in page file *501 inventory.*

Δ*lpagefile '501 inventory 1100'*

This selects the page area in *501 inventory* which begins at component 1100.

Δ*lpagefile '501 inventory 1100 88981'*

This selects the page area in *501 inventory* which begins at component 1100 and is accessed with the passnumber 88981.

Δ*lpagefile '501 inventory 1100'⊃'501 inventory 100'*

This selects two page areas in *501 inventory*, giving preference to the area that begins at component 1100.

Δ*lpagefile '501 inventoryd'⊃'501 inventory'*

This selects page areas in two page files, giving preference to the area in *501 inventoryd.*

∆*lpagein* nodelist

Defines in the workspace the objects contained in the specified nodes. If the objects (or objects with the same names) are already in the workspace, they are not paged in again. The use of ∆*lpagein* may result in the paging out of some objects, to make room in the workspace. See *Chapter 9: Generating End Environments* for details on page-out control. As well, ∆*lpagein* may have the side-effect of assigning stop and trace controls to paged functions, displaying page stop and trace messages, displaying paging activity messages, appending paging activity messages to file, or maintaining object reference counts. (See also the description of ∆*lpageset*.)

∆*lpageout* nodelist

Expunges from the workspace the objects contained in the specified nodes. Suspended or pendent functions are not expunged. Paged-out objects are not written to the page area where they originated; any changes made to the objects are lost.

By using object reference counts, ∆*lpageout* recognizes subtrees that are common to different paged-in objects. Paging out a root that is superordinate to a subtree required by another root does not expunge the common subtree.

currentvalue←[newvalue] ∆*lpageset* attribute

Sets or displays the paging control attributes. If **attribute** is empty, the list of keywords is displayed. If **attribute** is a simple character vector, it specifies one attribute to be set or displayed; if it is a vector of enclosed character vectors, it specifies several attributes. **newvalue** may be a scalar, or a vector with the same number of elements as **attribute.** If you omit **newvalue,** the current values are displayed without change. The valid attributes are as follows.

Attribute	**Default**	**Description**
threshold	(ws size)÷3	Page-out threshold
margin	(ws size)÷2	Page-out storage margin
refcounts	*on*	Object reference count control
debug	*off*	Display paging activity
audit	*0*	Write paging activity to file; value is tie number

For more information about these parameters, see *Chapter 9: Generating End Environments.*

currentvalue←[¯*1* | *0* | *1*] *∆lpagestop* '*?*' | **nodelist**

Indicates whether to set stop control on a node when it is about to be paged in. Because this operation occurs before a node is paged in, setting a page stop on a node is not quite the same as setting a function stop on the root of the node. If you specify a nodelist of *?*, the list of nodes with stop control is returned. If you omit the left argument, the stop control on the specified nodes is returned. In the left argument, ¯*1* indicates that the specified nodes are to have stop control removed; 0 indicates that the specified nodes are to be the only nodes with stop control set; 1 indicates that the specified nodes are to be added to the nodes with stop control set.

currentvalue←[¯*1* | *0* | *1*] *∆lpagetrace* '*?*' | **nodelist**

Indicates whether to set trace control on the specified nodes. If you specify a nodelist of *?*, the list of nodes with trace control is returned. If you omit the left argument, the trace control on the specified nodes is returned. In the left argument, ¯*1* indicates that the specified nodes are to have trace control removed; 0 indicates that the specified nodes are to be the only nodes with trace control set; 1 indicates the specified nodes are to be added to the nodes with trace control set.

∆lprestart

Reties paging files upon recovery from a crash. This function should be incorporated into an application's restart sequence before processing is restarted.

∆lpcmprs

Is a subfunction used by *∆lpagein* and *∆lpageout*. This function should not be called directly by your application.

[**linenumber←**] *∆lpwsfull 0* | *1*

Pages out nodes according to the general page-out controls, until the storage margin is reached or exceeded. If the right argument is 0, the program returns the number of the function line from which it was called; this is typically used in a *wsfull* trap to expedite recovery. With a right argument of 1, the program has no result.

APPENDIX D: STOP AND TRACE CONTROL

Stop and trace controls can be set on a function whether or not it is in the workspace. Once the function is paged in, the control works in the same way as APL stop and trace.

To display, set, or release LOGOS stop control on a function, use the *∆lfnstop* function. To display all the functions with stop control set (not just the functions currently in the workspace), use:

∆lfnstop '?'

You may display the stop control for specific functions as follows:

∆lfnstop 'input output'

If you specify a left argument to *∆lfnstop*, it becomes the new stop vector for the functions on the right:

1 3 4 ∆lfnstop 'input'

This sets stop control for *input* to lines 1, 3, and 4. You may also set different controls for different functions:

1 3 4 ⊃ 1 2 3 ∆lfnstop 'input output'

Here, the stop control for *input* is set as before, and the stop control for *output* is set to lines 1, 2, and 3.

To release stop control from *Input* and *output*, use:

0 *∆lfnstop 'input output'*

Function trace control works exactly the same way, using the utility *∆lfntrace*.

You may set stop or trace control on one or more *nodes*. When stop control is set on a node, the page-in program is suspended just before the node is actually dispersed into the active workspace. Notice how this differs from setting stop control on the root function of a node, where the control takes effect after page-in has occurred.

To display, set, or release stop control on a node, use the *∆lpagestop* function. To display all the nodes with stop control set, use:

∆lpagestop '?'

A left argument to *∆lpagestop* governs how stop control is to be changed: *1* sets stop control, *¯1* removes it, and 0 removes stop control from all nodes and then sets it on the node specified in the right argument. For example:

0 ∆lpagestop 'input output'

This removes stop controls from all nodes, and then enables it on the *input* and *output* nodes.

Page trace control works in exactly the same way, using the utility *∆lpagetrace.*

Monitoring Paging Activity

∆lpageset can be used to monitor the flow of pages into and out of your application. For example, to indicate that the names of the nodes being paged in or out are to be displayed, use:

'on' *∆lpageset 'debug'*

A left argument of *off* turns monitoring off.

In addition to or instead of displaying this information on your terminal, you may log it to a file for later perusal. When you log the paging diagnostic messages without local display, you avoid having these messages interspersed with your application's output. To write paging activity information to the file tied to 10, use:

'audit' *∆lpageset 10*

A right argument of 0 turns this behaviour off.

GLOSSARY

access	The privileges a user has to an object or an audit file. Also called permission. See also: *permission*
account number	See: *user number*
action template	A regular expression defining an action to be taken when a match to a locator template is found. See also: *locator template, regular expression*
active workspace	The active execution environment.
alias	A name that identifies a user enrolled in LOGOS. Any account can own any number of aliases. The alias under which your account was first enrolled in LOGOS is called your primary alias. Any alias other than your primary alias is a secondary alias. See also: *user number*
alias level directory	A special directory in the LOGOS file system immediately below the root which contains an entry for each user in LOGOS.
alternation	A pattern-matching construct that matches any one of a list of alternate patterns. The vertical bar denotes alternation. For example, *a* | *bc* | *d* matches *a* or *bc* or *d*.
Application Debugging Assistant	A tool that automates several steps in debugging applications constructed using LOGOS.
argument	A positional parameter passed to a LOGOS command. Also see: *modifier, parameter*
argument scope	A syntactic characteristic of every LOGOS command or script. Scope influences how a command identifies its arguments. A command can have short scope, long scope, or unprocessed scope. A command with short scope recognizes all unescaped blanks as argument delimiters and displays an error message if it encounters too many arguments. A command with long scope only recognizes blanks as argument delimiters until it has scanned the expected number of arguments. Additional blank-delimited fields at the end of the argument list are considered part of the last argument. A command with unprocessed scope takes only one argument and does not recognize any special characters other than the command separator.

attribute	A component of the object implied by the pathname. Attributes include source form, compilation directives, documentation, journal, note, and tag. See also: *compilation directives, documentation, journal, note, source form, tag*
audit file	A file that stores detailed tracking information about environments constructed by LOGOS. This information includes: • the full pathname and version of each object used • the names and types of the environments in which the object has been placed • the object's exact locations within those environments
audit record	A series of individual records inside an audit file, each describing a generation of an application. Each record contains a list of end environments built, the objects placed within them, and in some cases, their interrelationships.
auxiliary task	An S-task initiated and controlled through LOGOS.
auxiliary task commands	Commands that enable you to initiate an auxiliary task, communicate with it interactively or under program control, and inquire upon its status.
auxiliary task workspace	The workspace in which an auxiliary task executes commands passed to it.
base node	The first node in a paging file. There are several characteristics of the base node that distinguish it from other nodes. During generation, the base node is referred to by an asterisk (⋆) rather than by a name. The base node is automatically brought into the workspace when a paging file is opened via the *∆lpagefile* function. Finally, the contents of the base node are always present in the workspace of a paged application. See also: *inference, exclusion by, paging*
base page	See: *base node*
broadcast note	See: *note*
calling tree	A hierarchy that represents the relationships between a program and the other programs and data it references. This is a recursive definition; references can be direct (in the program itself) or indirect (through a referenced program) to an arbitrary depth.
calling tree analysis	A process by which some LOGOS commands (for example, *build*) determine the functions referenced by a specific root function. The analysis can be done to any depth.
capturing a workspace	See: *snapping a workspace*

change count	A counter that increases each time you save an object, despite whether the version number increases. You can display the change count of an object using the *list +summary +long* or *+date* modifiers with the *list* command.
change journal	See: *journal*
clipboard	Temporary storage used by some editor commands, such as *put,* to hold data that may be retrieved later by another command, such as *get.*
closing an object	Ending the editing of an object.
closure	The metacharacter ⋆ denotes closure. Used when searching for strings, the metacharacter allows you to match the phrase you are looking for any number of times, including none. See also: *positive closure, elision*
cluster	A SHARP APL package assembled by the *build* command from objects stored within LOGOS. The contents of a cluster can be explicitly enumerated in the argument to *build,* or calling tree analysis can be invoked and *build* will automatically include referenced objects in the cluster. Clusters can be stored in LOGOS paths or placed in workspaces, files, or paging files.
code tag	A special comment beginning with ⍝∇ and used to convey specific information about a function or variable to the compiler. For example: the ⍝∇⋄ code tag is used to indicate to the compiler that the line on which it appears is never to be joined to another line by the diamondize compilation directive.
command	There are two types of commands: LOGOS commands and editor commands. LOGOS commands are a sequence of letters or symbols that direct the actions of LOGOS. Editor commands are a sequence of letters or symbols that direct the actions of the LOGOS editor. Although some LOGOS and editor commands have the same name, they do not have the same command syntax.
command directory	A directory or list of directories that is searched whenever LOGOS does not recognize a command name. If a script with a matching name is found in the command directory, it is executed. The command directory is set and enquired upon with the *cmddir* command. See also: *script*
command output	See: *output*
command processor	The part of the LOGOS system that scans, validates, and executes commands entered by a user.
command prompt	See: *command separator*
command results	See: *output*
command separator	A character that can be used to separate LOGOS commands entered on the same line. The default character is the LOGOS command prompt, ∪. It can be changed using the *environment sepchar* parameter.

command syntax	The composition of a command. Commands have several parts, the arrangement of which is called the syntax. LOGOS commands and editor commands have a different syntax.
common structure	A form of hierarchical organization that groups objects based upon some common characteristic, such as related functionality or operation on common data structures. You might segregate general utilities, device handlers, data compression functions, and command routines. These directories are stored under a single directory unifying the entire system.
compilation	The process of transforming an object stored in LOGOS into an alternate form according to prescribed rules. This alternate form is usually more suitable for use in an end environment (it may be more compact, more efficient, etc.). For example, the compiler can be used to remove the comments from a function before it is placed into a workspace. Compilation is controlled through compilation directives. See also: *compilation directives, user-defined compilation directive*
compilation directives	A list of options that specify the transformations the compiler is to make to an object before the object is placed into an end environment. These can be stored as an object attribute [:*c*], provided through the *+compile* modifier supported by several commands, or included as part of a LOGOS session environment.
compiler	The part of the LOGOS system that carries out the process of compilation.
complementary indexing	References to versions of an object using negative numbers, ¯1, ¯2, etc. For example, *.public.util.vtom*[¯*1*] means the next to last version of the function *vtom*.
composite script	A script and a collection of objects referenced by the script that have been melded into a single object. This is accomplished by localizing the pathnames of the ancillary objects in the header of the script. When the script is invoked, the objects are materialized as local functions and variables. See: *script*
confirmation mode	A mode that issues a separate prompt for each object pending deletion. You can invoke it with the *delete +confirm* modifier.
context	The execution environment (workspace), or a LOGOS pathname (which must be a cluster) in which expressions subject to the *evaluate* modifier of a compilation directive are evaluated.
control permission	The privilege that enables a user to modify the access others have to an object. See also: *permission*
creator	The alias that saved the initial version of an object.
cyclic redundancy check (CRC)	A check used in the tracking table to determine whether an object was modified by you since fetched from LOGOS.

database

See: *file system*

datatype

See: *type*

debugging mode

A mode controlled by the *environment debug* parameter. When it is set off and an error is encountered while running a script, execution is abandoned and you return to the LOGOS prompt. When it is set on, execution is not abandoned, instead, debugging mode is enabled, and you are placed into immediate execution of a sort. You can:

- branch to any line of the script
- enter a naked branch arrow to terminate the script and return to the ᵘ prompt
- use →⎕*lc* to resume running the script
- execute LOGOS commands by preceding them with a right parenthesis
- use the)*si* command to examine the execution stack using the LOGOS *edit* command.
- edit the script to correct the error using the the LOGOS *edit* command

For example:

)*edit* **scriptname**

will update the script in the hierarchy and in the workspace.

decomment

A procedure that removes all comments from a function. It is invoked as a compilation directive.

del editor

The editor in SHARP APL in workspace *7 del*, which is based upon the VS APL Extended Editor and Full Screen Manager, enabled by VS APL's *xedit on* command. See the document *VS APL Extended Editor and Full Screen Manager* (IBM publication SH20-2341-1). See also: *editor*

demand paging

An operation triggered by the absence of an object whose name has been referenced in a running program. To use demand paging in LOGOS, you need:

- a global *value error* trap that executes the Δ*lpagein* function to resolve the error in the immediate environment
- a global trap for *wsfull* that executes →Δ*lpwsfull 0*.

The paging utilities are available in the public directory *.public.logos.paging*. See also: *page-in, paging, paging utilities, request paging*

diamondize	A procedure that merges each line of an object with the next, when possible, before saving the object in an end environment. It is invoked as a compilation directive.
directory	An object or a node in a LOGOS hierarchy that can have descendants.
display potential	The property of a command or script that determines whether its result displays. The result of a command or script displays only if display potential is on. Commands have default display potential settings. The *list* command, for example, displays its result by default. The display potential of a command can be forced on by using ⎕←*cmd*. It can be forced off by using *var←cmd* or *←cmd*.
distribution	The dispersion of changed objects into end environments. The *distribute* command will place all specified objects into their appropriate end environments, if an audit file was used in the original generation. *distribute* finds all end environments in the audit file that reference any of the objects specified, and updates them one by one. The actual update process depends on the type of end environment. A different kind of distribution is described under *export file*.
documentation	An object attribute that contains the description of the object as supplied by the object's owner.
editing stack	The list of objects that are currently open for editing.
editor	The LOGOS editor used to create and edit LOGOS objects. The LOGOS editor can be used in full screen or line mode, depending on your terminal type. It is based on the editor in SHARP APL workspace *7 del*. See: *del editor*
editor reference line	A line in the editor, highlighted when used in full screen mode, that functions as a starting point for several editor commands.
editor status line	A single line appearing at the top of the editor screen in full screen mode. It displays the name of the object being edited, an indication of whether or not you have modified it, and if it is an object from a LOGOS hierarchy, a list of its non-default attributes.
editor window	The editor window appears when you are using the editor in full screen mode. It displays output from editor commands and occupies a small area on the terminal screen below the display of the object being edited. The window does not always appear on the screen. When it is not required, the object display area expands to fill the rest of the screen. You can control the size of the window with the editor *window* command.
elision	The metacharacter - specifies a kind of closure and matches the phrase it follows 0 or 1 time.
end environment	A workspace, file, or paging file that is part of an application system generated by LOGOS.

entry command line	See: *entry expression*
entry expression	An expression stored in a LOGOS profile that is automatically executed every time the owner of the profile enters LOGOS. See also: *environment, exit expression, profile*
environment	The term *environment* is used to talk about four different things: • end environment • *environment* command • environment selection expression • context (for example, execution environment) **End environment**. A collection of values that influence the operation of LOGOS commands. They are set and enquired upon by the *environment* command. ***environment*** command. The command you use to set and enquire upon values determining the end environment. **Environment selection expression**. An expression which determines to what environments certain commands will apply. The environments selected may be workspaces, files, clusters. For example: */w 1234567 myws/p 7654321 pagefile∆1 450,reports/f myfile 20* would select 3 environments: the workspace *1234567 myws;* the LOGOS paging file *7654321 pagefile∆1*, paging area starting at component 450 and the node named reports within that area; component 20 of the file *myfile*. **Context**. When you use the *execute* compilation directive to evaluate objects enclosed within delimiters and flagged with the ⍝∇⍎ code tag, you may specify the context within which those objects are to be evaluated. This is specified as the pathname for a LOGOS-built cluster that contains the values for the objects being evaluated.

environment parameters	Parameters you can specify with the *environment* command to change aspects of your LOGOS environment. These include: • command separator • command directories • working directories • compilation directives • setting of debugging mode • auxiliary task identity • entry command line • exit command line • terminal type • status line detail • screen fields • screen attributes • keyword definitions • audit files • update environments
environment settings	The settings of certain aspects of your environment. See: *environment parameters*
environment stack	A means of saving and restoring environments to remove the burden of saving environment values before altering them. Allows you to stack multiple snapshots of the environment settings, and restore them at will.
error output	A type of result output produced by a LOGOS command or script. In a command, the error output describes error conditions arising from the arguments or modifiers, or values implied by them.

	In a script, you can generate error output for any condition you choose. Specifying that output as error output also causes a script's execution to be abandoned. See also: *message output, output, result output, script output, status output*
evaluated argument	A LOGOS command that is evaluated and whose result is passed as an argument to another LOGOS command. This is done by enclosing the argument command in parentheses. For example: *edit (locate ⎕replace)* performs a search for the string *⎕replace* in the current working directories and then passes any pathnames in which a match was found to the *edit* command.
execute permission	The privilege that allows a user to execute a function or script, but not to see it. See also: *permission*
execution environment	The workspaces in which a system is executed.
exit command line	See: *exit expression*
exit expression	An expression stored in a LOGOS profile that is executed every time the owner of the profile exits LOGOS. See also: *entry expression, environment, profile*
explicit page-in	Page in of a node explicitly done by a function's code by calling the *∆lpagein* function.
explicit page-out	Page out of a node explicitly done by a function's code by calling the *∆lpageout* function.
explicit selection	When selecting paging areas, you explicitly page in one node or select one paging area. This is useful when you have a repertoire of similar nodes, only one of which is to be selected.
export file	LOGOS files that can be transported to other machines and connected to other LOGOS file systems. They are created and built using the *export* command, which copies a list of pathnames to a specified export file. See also: *import file*
extended pathname	A pathname that includes explicit reference to a version of an object and/or to one of its attributes.
file	A storage medium external to an APL workspace. Files can contain one or more collections of defined functions, variables, or unnamed data objects.
file level directory	A special directory immediately beneath the alias level in which you must store other directories only.

file-resident	Components of a system that are stored in a file, brought into the workspace when needed, and then either written back to the file or expunged when no longer required.
file system	The part of the LOGOS system that stores LOGOS objects and their attributes.
file utilities	Functions that are used to assist in the creation and manipulation of SHARP APL files, particularly in scripts. They are found in the LOGOS public directory *.public.util.files*, and can be accessed by any user of LOGOS.
flat structure	The simplest form of hierarchical organization, which has all objects stored under a single directory. This reflects the flat format of a normal APL workspace.
full regular expression	Used to express search and replacement strings within the LOGOS editor, and for commands such as *locate* and *replace*. They consist of a locator template that specifies the string to be sought, and an optional action template that defines the action to be performed when a match to the locator template is found. See also: *action template, locator template, patterns, regular expression, ?⋆*
generated environment	The workspaces, files, and page files built by LOGOS in which resides all of the code necessary to execute an APL system.
generation	The process that constructs end environments (workspaces, files, and paging files) from objects stored in the LOGOS file system. See also: *end environment, paging file*
global permission	Identical privileges to everyone for everything in a particular directory. Set global permission by setting permission at the alias level. Permission is then inherited by all subordinate directories created under the alias level.
group	A named collection of related aliases. The primary benefit of groups is that they can be used to conveniently administer access privileges. For example, the aliases of all people working on the *invent* project might be enrolled in a group called *inventdev*. If *inventdev* has access to all the LOGOS paths pertaining to the project, the task of keeping access information up to date involves only the maintenance of the membership of the group. When group membership changes, new members have access to all the paths while former members no longer have access.
header	The first line of a function or script. It defines the syntax, arguments, modifiers, and locals as appropriate. See also: *script header*
hierarchy	A graded or ranked series of things. Both the LOGOS file system and the calling tree are hierarchies. See also: *calling tree, file system*
identifiers	Global or local names used in functions or scripts.

immediate execution	The state during which you can enter APL expressions and see the results immediately.
immediate execution prompt	The prompt that indicates that you are in immediate execution mode. In LOGOS, the immediate execution prompt is ⍎⍞.
implicit page-in	The process of paging-in a node of a LOGOS paging file, triggered indirectly by a value area trapped by a suitable event trap.
implicit page-out	The process of expunging a page from a workspace, triggered by a WSFULL event trap or a page-in activity.
implicit selection	When selecting paging areas, you can establish an implicit precedence of nodes by selecting more than one paging area, so that nodes in the latter areas are shadowed by nodes in the former. Implicit selection allows a system to fully customize its behaviour for the user.
import file	A file created by the *export* command that consists of a LOGOS hierarchy. It can be imported directly into LOGOS. See also: *export file*
inference, exclusion by	A property of the generation process that excludes objects from nodes if they represent the root of another node or if they appear in the base node. Inference by exclusion is controlled by the *build +inference* modifier. By default, inference by exclusion is turned on. See also: *base node, paging*
journal	A LOGOS object attribute in which LOGOS users can record information about the changes they have made to an object.
keep-in priority	A measure of how important it is that a node remain in the workspace. All other things being equal, the node with the highest keep-in priority will be paged out last. Keep-in priority is specified with the *build +keepin* modifier.
keyword	Used to store frequently used phrases. A keyword is set with the *keyword* command or the *environment keyword* parameter, and is referenced by preceding its name with a backslash (\). A keyword can be shortened to any abbreviation that is unique. The phrase or expression is substituted directly into the command in place of the **\keyword.**
limited regular expression	Used in pathnames to generate a sequence of names that match a particular pattern. The use of regular expressions saves typing, and also makes possible the selection of families of strings that would otherwise be awkward to express. The metacharacters *?*, ⋆, and | combine with partial pathnames to create these expressions. See also: *patterns, regular expression, ?⋆*
link	A LOGOS object that is a pointer to another object. Links can be used to create the illusion that an object resides in many places in a hierarchy, when in reality, there is only one central copy of the object. This provides the convenience of easy access, yet retains the advantages of single source maintenance.

local environment	Includes your working directories, command directories, separator character, status area control, and other dynamic properties of a session.
localize	To include an object name in the header of a function or script.
locator template	A text pattern that specifies a class of strings to be searched for. For example, the pattern *a?** matches all strings that begin with the letter *a*. See also: *action template, template*
LOF	See: *page-out strategy*
long scope	See: *argument scope*
LRP	See: *page-out strategy*
LRU	See: *page-out strategy*
message output	Text generated by a LOGOS command that is ancillary in nature. As an example, suppose the *display* command is used to display an object that has a broadcast note associated with it. The note is considered message output. See also: *error output, output, result output, status output*
metacharacter	See: *reserved characters*
modifier	A parameter that modifies the action of a command. Modifiers always begin with a plus sign (for example, +*summary*), and are specified after any appropriate arguments. See also: *argument, parameter*
module structure	A form of hierarchical organization that separates functional units of a system into individual directories. For example, if your system consists of subsystems, you might have a separate directory for each subsystem. These would be stored under a single directory unifying the entire system.
nested command	A command enclosed in parentheses whose result is interpolated into a command line.
node	Node has two definitions: • A collection of objects in a paging file that are treated as a unit for the purposes of materialization in or expulsion from the workspace. • A level in the LOGOS hierarchy. Non-terminal nodes have other nodes beneath them. Terminal nodes do not. See also: *non-terminal node, terminal node*
non-terminal node	A node in the LOGOS file system with subordinate nodes. See also: *node*

note	A LOGOS object attribute that is used to attach an informative message to an object. The message is displayed whenever the source or other attribute of the object is referenced.
object	A complete entity stored within the LOGOS file system. Each object has a type (function, variable, script, cluster, directory, link) and a number of attributes (compilation directives, documentation, journal, note, source, tag).
object attribute	See: *attribute*
object form	The compiled version of an object (for example, a function after all comments have been removed from it). See: *source form*
object header	See: *header*
object type	See: *type*
output	LOGOS commands and scripts can produce output. LOGOS command produce two kinds: • result output, which is essential information generated by a command • message output, which is ancillary information, such as a warning or an error Scripts can produce: • result output • message output • error output • status output • quadprime output Script output is controlled by the *output* command, which takes a textual message as its argument, and uses the modifiers *+error*, *+message*, *+quadprime*, *+result*, and *+status* to determine what to do with it. See also: *display potential, error output, message output, result output, status output*
overlay	The "threading" effect produced by the use of multiple working directories. Production versions of objects can be overlain by test versions to enable convenient testing of new or changed objects in an otherwise running production system.

page	See: *node*
page file	See: *paging file*
page-in	The movement of objects from a SHARP APL file into the workspace. LOGOS supports both demand and request paging. To use paging, a paging area containing the objects for page-in must be specified and built. See also: *demand paging, page-out, paging utilities, request paging*
page-out	The removal of objects from the workspace. Objects can be explicitly paged out by the *∆lpageout* utility (found in the public directory *.public.logos.paging*), or implicitly by the page-out mechanism when ⎕*wa* has fallen below a certain threshold. The contents of the base node are never paged out. See also: *demand paging, page-in, paging utilities, request paging*
page-out strategy	There are three page-out algorithms for selecting objects for page-out first. They are: • Largest objects first (LOF), which selects the largest objects in the workspace for page-out first. • Least recently paged (LRP) which selects the least recently paged objects for page-out first. • Least recently used (LRU) which selects the least recently paged objects for page-out first.
paged system	An application in which some or all of the functions are stored in a function file or a LOGOS paging file, and are brought into the workspace when required.
paging	An application architecture in which software is organized into functional groups or "nodes" that are brought into a workspace by an application as they are needed. Using this technique, extremely large applications can run in a smaller workspace, even though the size of the application code may be many times that of the workspace. See also: *base node, demand paging, inference (exclusion by), paging file, request paging*
paging area	A set of contiguous components in a LOGOS paging file containing any number of nodes. The paging area is referred to by a file name and a starting component number.
paging area header	Components at the beginning of the paging area containing information about the area (for example, time and date of last generation, alias of the user performing the generation, number of updates, etc.).

paging file	An end environment constructed by LOGOS consisting of a SHARP APL file organized into nodes. Each node is a LOGOS cluster containing functions and/or variables that are moved into the workspace when required. The LOGOS commands *build* and *filesave* will create paging files for you based on the calling tree of a root function you specify.
paging priority	See: *keep-in priority*
paging utilities	Functions used to assist in the running and debugging of applications that use LOGOS paging. They are found in the public directory *.public.logos.paging*, and can be accessed by any user of LOGOS. See also: *demand paging, page-in, page-out, request paging*
parameter	A collective term for a value (argument or modifier) passed to a LOGOS command or script. See also: *argument, modifier*
parent directory	The directory above any particular node in the LOGOS hierarchy. Many attributes of the parent directory (such as retention) are passed on to the objects or directories created below it.
parent node	See: *parent directory*
parent pathname comment	A special comment affixed to the end of the last line of a function by the LOGOS compiler. If you have selected the *p* compilation directive, functions fetched contain such a comment, distinguished by the prefix ⍝⋆. The parent pathname comment is used by the *snap* command to determine relationships between workspace objects and their counterparts stored in LOGOS.
password	A personalized key that can be attached to an alias/user number pair.
path	See: *pathname*
pathname	A name identifying an object in a LOGOS hierarchy. A pathname is formed by enumerating the directories lying along the path to the object, separating each by a dot (.). A rooted pathname always begins with a dot, indicating that the path begins at the very root of the hierarchy. A pathname that does not begin with a dot is called a relative pathname. Such pathnames are resolved relative to the current working directory or command directory. See also: *qualified pathname, relative pathname, rooted pathname, working directory*
patterns	Patterns are built of expressions that obey a small and simple set of rules, and are used to generate a series of names or strings that match them. See also: *full regular expression, limited regular expression, regular expression, ?⋆*

permission	The access privileges a given user has to an object or to an audit file. Also called access. Object permissions include control, execute, read, and write. Audit permissions include control, read, and write. See also: *control permission, execute permission, read permission, write permission*
positive closure	The metacharacter + specifies a kind of closure and matches the phrase it follows one or more times. See also: *closure*
primary alias	See: *alias*
primary user number	See: *user number*
primary working directory	See: *working directory*
priority	See: *keep-in priority*
profile	A collection of values associated with an alias and established whenever the alias begins a LOGOS session. Profiles are created using the *environment +profile* modifier. See also: *entry expression, environment, exit expression*
program file	A file used to store functions and constants that can be materialized in the active workspace. It can be built from objects stored in LOGOS or elsewhere.
program line labels	(*l0, l1, l2*).
public directory	See: *public path*
public library	See: *utility library*
public path	A path rooted under *.public*, the LOGOS public directory.
?*	LOGOS metacharacters, used in regular expressions to indicate match any character (?), and do so as many times as possible (*). See also: *full regular expression, limited regular expression, patterns*
qualified pathname	A pathname that is qualified by a suffix providing additional information. This suffix consists of a pair of brackets surrounding version, attribute, and type information. See also: *pathname*
rank	A characteristic of a LOGOS object.
read permission	The privilege that allows a user to fetch the source form of an object. See also: *permission, source form*
reference count	A count maintained for each paged-in object that indicates the number of paged-in nodes that the object belongs to. If a node that contains a particular object is paged-out, the object is nonetheless exempt from page-out until its reference count falls to 0.

reference line	See: *editor reference line*
reference type	Cluster, file, link, page file, script, workspace.
references	Workspace and file cross-references that LOGOS stores automatically for each path in the system, so you can determine where your objects are being used.
registration	Objects in the LOGOS file system can be registered out when you are making changes to them. Anyone else who accesses that object during that time will receive a warning that it is being modified by you. When you have finished with the changes, the object can be registered back in. Registration is controlled by the *register* command, with an argument of *in* or *out*.
registration potential	Registration potential causes an object to be automatically registered out when the *edit* command calls that object, and to be registered in when the object is closed. Set by the *register* command with an argument of *on* or *off*.
regular expression	A shorthand for generating a series of names or strings that match a particular pattern. See also: *full regular expression, limited regular expression, patterns, ?⋆*
relative pathname	A pathname that is not specified from the root of the LOGOS hierarchy. See also: *pathname, rooted pathname*
request paging	Based on some advance knowledge of the likely order of processing, and particularly applicable to closed prompting systems where the issuing of one command is statistically unrelated to, or biased against, subsequent issuing of the same command. The Δ*lpagein* function is used for request paging, and can be found in the LOGOS public directory *.public.logos.paging*. See also: *demand paging, page-out, paging, paging utilities*
reserved characters	Special characters, also called metacharacters, that are of two kinds: • Those which signify some special action on the part of the command processor and must be quoted if they are used in an argument to a command. • All of the characters in the first group, plus those that are part of the syntax of pathnames (for example, .), plus a few others that cannot be used as command separator characters (for example, ∇). There are different reserved characters in different contexts (the LOGOS command language, the editor, and regular expressions).
result output	The primary display generated by a LOGOS command or script. This excludes output such as error messages, broadcast notes, and so on. See also: *error output, message output, output, status output*

result potential	Each native LOGOS command has the potential for returning an explicit result. For some of these commands, such as *list,* the result potential is 'on' so that, by default, a result is displayed. The result may be suppressed by using the assignment arrow but not specifying a variable. For example: *←env +profile* For others, such as *delete,* the result potential is 'off' so no explicit result is returned unless specifically requested when the comment is used. For example ⎕←*delete .dick.util.vtom* For user written scripts, explicit result may be declared by the form of the script's header, or by using the *output* command with the *+result* modifier. In both cases, the result may be suppressed by using the technique mention above.
retention	A value associated with every LOGOS object that determines how many historical versions of the object are to be maintained. This value is set via the *retain* command. See also: *version*
retention count	See: *retention*
retract permission	Permission to retract the global shared variable for an active task without terminating the task. Controlled by the LOGOS *send* command.
root	The "top" of the LOGOS file system. The focal point of the file system through which all nodes can communicate with all other nodes.
rooted pathname	A pathname that begins at the top of the hierarchy and begins with a dot (.). See also: *pathname, relative pathname*
scope	See: *argument scope*
screen attributes	Attributes of the terminal screen, including colour, intensity, and highlight.
screen fields	Fields on the screen. You can display or establish the colour and highlighting of these fields during your LOGOS session using the *environment field* parameter.
script	A user-defined LOGOS command. Scripts can contain a mixture of APL expressions and LOGOS commands. See also: *command composite, script directory*
script debugging mode	See: *debugging mode*
script header	A header line resembling the header line of an APL user-defined function. A script's header is line 1 (unlike a function). See also: *header*
script output	See: *display potential, output*

secondary alias	See: *alias*
secondary user number	See: *user number*
separator character	See: *command separator*
shadowing	In APL, shadowing refers to a local concealing a more local value. In LOGOS, shadowing can also refer to page file shadowing and directory shadowing.
shell	An APL function that brings in a cluster from a file, localizes the contents of the cluster, and executes the root function of the cluster (or another expression you specify).
short scope	See: *argument scope*
signon lock	See: *password*
snapping a workspace	Examining the objects in a workspace and storing in LOGOS all objects that are new or have changed since they were last stored. Use the *snap* command.
software life cycle	The overall process of developing software. Life cycle phases are: specification, design, coding and testing, integration and testing, distribution, and maintenance.
source attribute	See: *source form*
source form	The definition of an object as it is stored in the LOGOS file system, unaltered by compilation directives. See: *object form*
static tree analysis	See: *calling tree analysis*
status line	Consists of zero, one, or two lines of status output at the top of the screen on 3270-type devices. The status area output includes the alias you are using, the current working directory, and information on the command that is running. HDS108 devices can have only one line of status information, but you can set the line that displays. The amount of status area is controlled with the *environment status* parameter.
status output	The output of a LOGOS command (or script) that is written to the status line on the terminal. See also: *error output, message output, output, result output*
stop control	For debugging a paged system. Stop control can be set on a function whether or not it is in the workspace. Once the function is paged in, this works the same as APL stop control.
storage margin	Used during implicit page-out to control when nodes are removed from the workspace. An attribute of your application that you can set with the utility function *Δlpageset*.

syntactic flags	A flag specified via *+flags=s* that qualifies the actions of the LOGOS *locate, replace,* and *compare* commands. The *locate* and *replace* commands take context into consideration when searching for a string, treating names and numbers as units that either match fully or not at all. The *compare* command disregards the spelling of names local to the function being compared. The comparison is based upon consistent use of a name set within each function, and the name sets across functions being compare need not be the same.
syntactic search qualifier	Qualifies the action of the editor *locate, change,* and *highlight* commands, so that it takes context into consideration when searching for a string. When searching syntactically, names and numbers are treated as units and either match fully or not at all.
syntax	See: *command syntax*
tag	A LOGOS object attribute that has no particular semantics ascribed to it by LOGOS. It can be thought of as a "user-defined attribute."
task	See: *auxiliary task*
task name	A name associated an auxiliary task, to differentiate it from other auxiliary tasks.
template	Expressions that specify patterns to be sought, and actions to be taken once found. See also: *action template, locator template*
terminal node	A node in the LOGOS hierarchy without any subordinate nodes. See also: *node*
terminal type	The type of terminal you are using with LOGOS. You can tell LOGOS the terminal type of your device using the *environment terminal* parameter.
trace control	For debugging a paged system. Can be set on a function whether or not it is in the workspace. One the function is paged in, the control works like APL trace control.
tracking table	A variable in your workspace that LOGOS uses to maintain a mapping between the objects in a workspace and their locations in the LOGOS file system. This table is used by the *snap, wssave, get,* and *build* commands.
tree analysis	See: *calling tree analysis*
trigger threshold	Used during implicit page-out. An attribute of your application that you can set with the utility function *∆lpageset.*

type	A characteristic of a LOGOS object. Types of objects are clusters, directories, functions, links, scripts, and variables. Pathname output of commands includes a designation of type in square brackets after the pathname. For example, [*f*] for function, [*c*] for cluster, and so on.
update generation	An incremental paging file generation invoked by using *filesave +update* modifier.
unprocessed scope	See: *argument scope*
UP version	A version of an editor command that causes the command to work in the opposite direction from the direction it usually works in. For example, the *up* version of the *add* command (*addup* or *upadd*) causes the editor to add a line above, rather than below the reference line.
used list	A list maintained with every LOGOS path that records where the associated object is used. The *references* command uses the information in used lists to relate paths to end environments and audit files.
user	Anyone enrolled in LOGOS.
user-defined command	See: *script*
user-defined compilation directive	A user-supplied function or cluster called by the LOGOS compiler to process an object's source. There are two varieties of user-defined directives: • *a* directives, which are executed before any of the built-in directives. • *z* directives, which are executed after all of the built-in directives (except for "lock").
user number	A SHARP APL signon number that identifies a LOGOS alias. A given alias can have several accounts associated with it. The main account associated with an alias is the primary user number for that alias. Other user numbers with the same alias are called secondary user numbers.
utility library	A set of directories containing APL functions and variables, and LOGOS scripts and clusters. The library provides a central location for objects that may be useful to many different programmers, and avoids re-invention of wheels. The utility directories have paths that begin with *.public*, and can be accessed by any user of LOGOS.
version	A numbered representation of an object in the LOGOS file system. An arbitrary number of versions can be retained for each object. See also: *retention*
version number qualification	See: *version qualifier*

version qualifier	A particular version specified in the pathname of an object. For example, in the path *chart*[*4*], the version qualifier [*4*] indicates version 4 of the object.
VS APL editor	The VS APL Extended Editor and Full Screen Manager. See the document *VS APL Extended Editor and Full Screen Manager* (IBM publication SH20-2341-1).
working directory	A directory or list of directories used to resolve pathnames that are not specified from the root (relative pathnames). The primary working directory is the first directory in the list. See also: *pathname*
write permission	The privilege that allows a user to modify the source form of an object. See also: *permission, source form*
WSDOC	IPSA workspace documentation facility. This is a package for generating descriptions of the contents of a workspace, including cross reference information, calling tree, etc. For more information, see the *WSDOC User's Manual*. WSDOC can be used to document LOGOS objects through the use of the LOGOS *wstofile* command.

INDEX