User's Guide for the Software Tools Subsystem Command Interpreter (The Shell)

T. Allen Akin
Terrell L. Countryman
Perry B. Flinn
Daniel H. Forsyth, Jr.
Jefferey S. Lee
Jeanette T. Myers
Arnold D. Robbins
Peter N. Wan

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

September, 1984

TABLE OF CONTENTS

Foreword

The Software Tools Subsystem is a set of program development tools based on the book Software Tools by Brian W. Kernighan and P. J. Plauger. It was originally developed for use on the Prime 400 computer in 1977 and 1978 in the form of several cooperating user programs. The present Subsystem, the ninth version, is a powerful tool that aids in the effective use of computing resources.

The command interpreter, also referred to as the "shell," is a vital part of the Subsystem. It is a program which accepts commands typed by the user on his terminal and converts them into more primitive directions to the computer itself. The user's instructions are expressed in a special medium called the "command language." The greatest part of this document is involved with describing the command language and giving examples of how it is used.

Three areas will be covered in the following pages. First, there is a tutorial on the use of the command language. New Subsystem users should read this chapter first. Some minimal knowledge of terminal usage is assumed; if you are unsure of yourself in this area, see Prime's published documentation and the Software Tools Subsystem Tutorial for help. Second, there is a summary of the syntax and semantics of the command language. Experienced users should find this chapter valuable as a reference. Finally, there is a selection of application notes. This chapter is a good source of useful techniques and samples of advanced usage. Experienced users and curious beginners should find it well worthwhile.

Tutorial

Commands

Input to the command interpreter consists of "commands". Commands, in turn, consist of a "command name", which is the name of an executable file. A command is executed simply by entering its name. For example,

```
] help
```

is a command that will describe how you can obtain online documentation.

Some commands may have arguments. Arguments are values supplied by you to the command. Arguments can be required or they may be optional in which case the system uses a default. In the above example when 'help' is invoked with no arguments the Subsystem assumes the command 'help help' (i.e. get me on-line documentation for the 'help' command). However, if you wanted on-line documentation for a specific command you would supply the command name as an argument, e.g.

```
] help lf
```

will describe the command that can be used to list information about files in a directory. Some commands may have options. Options are used to make the same command execute in slightly different ways. Options usually consist of one letter and are preceded by a dash. The command,

```
] help -f file
```

will list the names of commands and subroutines that may be associated with the keyword "file". The "-f" is an option and "file" is an argument. Commands, arguments and options are separated from each other by blanks.

Here is a final example:

```
] lf
adventure       ee              guide           m6800
shell           shell.doc       subsys          time_sheet
words           zunde
]
```

'Lf' is used to list the names of your files. Executed without any arguments, 'lf' prints the files in your current directory, but (like 'help') 'lf' may be used with or without arguments and options.

How the Command Interpreter Locates a Command

Recall that you can access files by their entrynames only if they are located in your current directory. Without help from the shell this would also be true for commands. That is, in order to execute 'help' you would need to have a copy of the 'help' command in your current directory or you would have to enter its full pathname so that the shell could locate it in another directory. Obviously, neither alternative is desirable. In reality, the shell uses a "variable" called "_search_rule" to find commands like "help" in other directories. Each user has his own search rule. (Refer to the section in this guide entitled "Shell Control Variables" for more information.) The search rule tells the shell in what locations to look for commands, and if there is more than one location possible, it specifies the order in which the locations will be searched.

Most new users are given the search rule that causes the command interpreter to look for commands in the following five locations in the order shown:

1. The shell's internal library for an internal command (e.g, 'stop', 'set')
2. The user's variables currently stored in memory
3. The user's current directory
4. The Subsystem library containing locally supported external commands, "=lbin=" (e.g. memo, moot)
5. The Subsystem library containing standard external commands, "=bin=" (e.g. 'lf', 'help')

This variable is explained in more detail in the "Application Notes" section of this guide.

Beware that this flexibility can get beginners (and some experienced users) into trouble. With the search rule above, the command interpreter will always look in your current directory for a command before it looks in one of the Subsystem command directories. Therefore, if you create a file having the same name as a command, the shell will try its best to execute the contents of that file.

Special Characters and Quoting

Some characters have special meaning to the command interpreter. For example, try typing this command:

```
] echo Alas, poor Yorick
Alas
poor:  not found
]
```

'Echo' is simply a command that types back its arguments. Obviously this example is not working as it should. The strange behavior is caused by the fact that the comma is used for dark

mysterious purposes elsewhere in the command language. (The comma actually represents a null I/O connection between nodes of a network. See the section on pipes and networks for more information.) In fact, all of the following characters are potential troublemakers:

, ; # @ > | { } [ ] ( ) _ blank

The way to handle this problem is to use quotes. You may use either single or double quotes, but be sure to match each with another of the same kind. Try this command now:

```
] echo "Alas, poor Yorick; I knew him well."
Alas, poor Yorick; I knew him well.
]
```

You can use quotes to enclose other quotes:

```
] echo 'Quoth the raven: "Nevermore!"  '
Quoth the raven: "Nevermore!"
]
```

A final word on quoting: Note that anything enclosed in quotes becomes a single argument. For example, the command

```
] echo "Can I use that in my book?"
```

has only one argument, but

```
] echo Can I use that in my book?
```

has seven.

Command Files

Suppose you have a task which must be done often enough that it is inconvenient to remember the necessary commands and type them in every time. For an example, let's say that you have to print the year-end financial reports for the last five years. If the "print" command is used to print files, your command might look like:

```
] print year74 year75 year76 year77 year78 year79
```

If you use a text editor to make a file named "reports" that contains this command, you can then print your reports by typing

```
] reports
```

No special command is required to perform the operations in this "command file;" simply typing its name is sufficient.

Any number of commands may be placed in a command file. It is possible to set up groups of commands to be repeated or

executed only if certain conditions occur. See the Applications Notes for examples.

It is one of the important features of the command interpreter that command files can be treated exactly like ordinary commands. As shown in later sections, they are actually programs written in the command language; in fact, they are often called "shell programs." Many Subsystem commands ('e', 'fos', and 'rfl', for example) are implemented in this manner.

Doing Repetitive Tasks --- Iteration

Some commands can accept only a single argument. One example of this is the 'fos' command. "Fos" stands for "format, overstrike, and spool." It is a shorthand command for printing "formatted" documents on the line printer. (A "formatted" document is one prepared with the help of a program called a "text formatter," which justifies right margins, indents paragraphs, etc. This document was prepared by the Software Tools text formatter 'fmt.') If you have several documents to be prepared, it is inconvenient to have to type the 'fos' command for each one. A special technique called "iteration" allows you to "factor out" the repeated text. For example,

```
] fos (file1 file2 file3)
```

is equivalent to

```
] fos file1
] fos file2
] fos file3
```

The arguments inside the parentheses form an "iteration group." There may be more than one iteration group in a command, but they must all contain the same number of arguments. This is because each new command line produced by iteration must have one argument from each group. As an illustration of this,

```
] (echo print fos) file(1 2 3)
```

is equivalent to

```
] echo file1
] print file2
] fos file3
```

Iteration is performed by simple text substitution; if there is no space between an argument and an iteration group in the original command, then there is none between the argument and group elements in the new commands. Thus,

```
file(1 2 3)
```

is equivalent to

```
file1
file2
file3
```

Iteration is most useful when combined with function calls, which will be discussed later.

I/O Redirection

Control of the sources and destinations of data is a very basic function of the command interpreter, yet one that deserves special attention. The concepts involved are not new, yet they are rarely employed to the extent that they have been used in the Subsystem. The best approach to learning these ideas is to experiment. Get on a terminal, enter the Subsystem, and try the examples given here until they seem to make sense. Above all, experiment freely; try anything that comes to mind. The Subsystem has been designed with the idea that users are intelligent human beings, and their freedom of expression is the most valuable of tools. Use your imagination; if it needs tweaking, take a look at the Application Notes in the last chapter.

Programs and commands in the Subsystem do not have to be written to read and write to specific files and devices. In fact most of them are written to read from "anything" and write to "anything." Only when the program is executed do you specify what "anything" is, which could be your terminal, a disk file, the line printer, or even another program. "Anything"s are more formally known as "standard input ports" and "standard output ports." Programs are said to "read from standard input" and "write on standard output." The key point here is that programs need not take into account how input data is made available or what happens to output data when they are finished with it; the command interpreter is in complete control of the standard ports.

A command we will use frequently in this section is 'copy'. 'Copy' does exactly what its name implies; it copies data from one place to another. In fact, it copies data from its first standard input port to its first standard output port.

The first point to remember is that by default, standard ports reference the terminal. Try 'copy' now:

```
] copy
```

After you have entered this command, type some random text followed by a newline. 'Copy' will type the same text back to you. (When you tire of this game, type a control-c; this causes an end-of-file signal to be sent to 'copy', which then returns to the command interpreter. Typing control-c to cause end-of-file is a convention observed by all Subsystem programs.) Since you did not say otherwise, standard input and standard output referred to the terminal; input data was taken from the terminal (as you typed it) and output data was placed on the terminal (printed by 'copy').

Obviously, 'copy' would not be of much use if this was all it could do. Fortunately, the command interpreter can change the sources and destinations of data, thus making 'copy' less trivial.

I/O Redirection to Disk Files or Devices

Standard ports may be altered so as to refer to disk files by use of a "funnel." The greater-than sign (>) is used to represent a funnel. Conventionally, the ">" points in the direction of data flow. For example, if you wished to copy the contents of file "ee" to file "old_ee", you could type

```
] ee>  copy  >old_ee
```

The greater-than sign must always be immediately next to its associated filename; no intervening blanks are allowed. At least one blank must separate the '>' from any command name or arguments. This restriction is necessary to insure that the command language can be interpreted unambiguously.

The construct "ee>" is read "from ee"; ">old_ee" is read "toward old_ee." Thus, the command above can be read "from ee copy toward old_ee," or, "copy from ee toward old_ee." The process of changing the file assignment of a standard port by use of a funnel is called "I/O redirection," or simply "redirection."

It is not necessary to redirect both standard input and standard output; either may be redirected independently of the other. For example,

```
] ee>  copy
```

can be used to print the contents of file "ee" on the terminal. (Remember that standard output, since it was not specifically redirected, refers to the terminal.) Not surprisingly, the last variation of 'copy',

```
] copy  >old_ee
```

is also useful. This command causes input to be taken from the terminal (until an end-of-file is generated by typing a control-c) and placed on the file "old_ee". This is a quick way of creating a small file of text without using a text editor.

It is important to realize that all Subsystem programs behave uniformly with regard to redirection. It is as correct to redirect the output of, say, 'lf'

```
] lf  >file_list
```

as it is to redirect the output of 'copy'.

Recall that special pathnames which begin with "/dev" may refer to peripheral devices. For example, by redirecting output to "/dev/lps" you can print a file on the line printer.

```
] cat myfile >/dev/lps
```

Although the discussion has been limited to one input port and one output port up to this point, more of each type are available. In the current implementation, there are a total of six; three for input and three for output. The highest-numbered output port is generally used for error messages, and is often called "ERROUT"; you can "capture" error messages by redirecting this output port. For example, if any errors are detected by 'lf' in this command

```
] lf  3>errors
```

then the resulting error messages will be placed on the file "errors".

Final words on redirection: there are two special-purpose redirection operators left. They are both represented by the double funnel ">>". The first operator is called "append:"

```
] lf  >>list
```

causes a list of files to be placed at the end of (appended to) the file named "list". The second operator is called "from command input." It is represented as just ">>" with no file name, and causes standard input to refer to the current source of commands. It is useful for running programs like the text editor from "scripts" of instructions placed in a command file. See the Application Notes for examples.

I/O Redirection to other Commands

The last section discussed I/O redirection --- the process of making standard ports refer to disk files or devices, rather than just to the terminal. This section will take that idea one step further. Frequently, the output of one program is placed on a file, only to be picked up again later and used by another program. The command interpreter simplifies this process by eliminating the intermediate file. The connection between programs that is so formed is called a "pipe," and a linear array of programs communicating through pipes is called a "pipeline."

Suppose that you maintain a large directory, containing drafts of various manuals. Each draft is in a file with a name of the form "MANxxxx.rr", where "xxxx" is the number of the manual and "rr" is the revision number. You are asked to produce a list of the numbers of all manuals at the first revision stage. The following command will do the job:

```
] lf -c | find .01
```

"lf -c" lists the names of all files in the current directory, in a single column. The "pipe connection" (vertical bar) causes this listing to be passed to the 'find' command, which selects those lines containing the string ".01" and prints them. Thus,

the pipeline above will print all filenames matching the conventional form of a first-revision manual name.

The ability to build special purpose commands cheaply and quickly from available tools using pipes is one of the most valuable features of the command interpreter. With practice, surprisingly difficult problems can be solved with ease. For further examples of pipelines, see the Applications Notes.

Combinations of programs connected with pipes need not be linear. Since multiple standard ports are available, programs can be and often are connected in non-linear networks. (Some networks cannot be executed if the programs in the network are not executed concurrently. The command interpreter detects such networks, and prints a warning message if they cannot be performed.) Further information on networks can be found in both the reference and applications chapters of this guide.

I/O Redirection for a Group of Commands

It is sometimes necessary to change the standard port environment of many commands at one time, for reasons of convenience or efficiency. The "compound node" (a set of networks surrounded by curly braces) can be used in these situations.

As an example of the first case, suppose that you wish to generate a list of manual names (see the last example) in either the first or the second stage of revision. One way to do this is to generate the list for the first revision stage, place it on a file using a funnel, then generate a list for the second revision stage and place it on the end of the same file using an "append" redirector. A compound node might simplify the procedure thusly:

```
] { lf -c | find .01;  lf -c | find .02 }  >list
```

The first network finds all manuals at the first revision stage, and the second finds all those at the second stage. The networks will execute left-to-right, with the output of each being placed on the file "list," thus generating the desired listing. With iteration, the command can be collapsed even farther:

```
] { lf -c | find .0(1 2) }  >list
```

This combination of iteration and compound nodes is often useful.

Efficiency becomes a consideration in cases where successive long streams of data are to be copied onto a file; if the "append" redirector is used each time, the file must be reopened and repositioned several times. Using a compound node, the output file need be opened only once:

```
] { (file1 file2 file3)> copy }  >all_files
```

This complex example copies the contents of files "file1," "file2," and "file3" into the file named "all_files."

I/O Redirection to a Command Argument

As mentioned before, some commands may have arguments. The standard output of a command (or a series of commands) can be used as an argument(s) by using the "function call" mechanism. For example, recall the situation illustrated in the section on pipes and networks; suppose it is necessary to actually print the manuals whose names were found. This is how the task could be done:

```
] print [lf -c | find .01]
```

The function call is composed of the pipeline "lf -c | find .01" and the square brackets enclosing it. The output of the pipeline within the brackets is passed to 'print' as a set of arguments, which it accesses in the usual manner. Specifically, all the lines of output from the pipeline are combined into one set of arguments, with spaces provided where multiple lines have been collapsed into one line.

'Print' accepts multiple arguments; however, suppose it was necessary to use a program like 'fos', that accepts only one argument. Iteration can be combined with a function call to do the job:

```
] fos ([lf -c | find .01])
```

This command formats and prints all manuals in the current directory with revision numbers "01".

Function calls are frequently used in command files, particularly for accessing arguments passed to them. Since the sequence "lf -c | find pattern" occurs very frequently, it is a good candidate for replacement with a command file; it is only necessary to pass the pattern to be matched from the argument list of the command file to the 'find' command with a function call. The following command file, called 'files', will illustrate the process:

```
lf -c | find [arg 1]
```

"arg 1" retrieves the first command file argument. The function call then passes that argument to 'find' through its argument list. 'Files' may then be used anywhere the original network was appropriate:

```
] files .01
] print [files .01]
] fos ([files .01])
```

Variables

It has been claimed that the command language is a programming language in its own right. One facet of this

language that has not been discussed thus far is the use of its variables. The command interpreter allows the user to create variables, with scope, and assign values to them or reference the values stored in them.

Certain special variables are used by the command interpreter in its everyday operation. These variables have names that begin with the underline (_). One of these is '_prompt', which is the prompt string the command interpreter prints when requesting a command. If you object to "]" as a prompt, you can change it with the "set" command:

```
] set _prompt = "OK, "
OK, set _prompt = "% "
% set _prompt = "] "
]
```

You may create and use variables of your own. To create a variable in the current scope (level of command file execution), use the "declare" command:

```
] declare i j k sum
```

Values are assigned to variables with the 'set' command. The command interpreter checks the current scope and all surrounding scopes for the variable to be set; if found, it is changed, otherwise it is declared in the current scope and assigned the specified value.

Variables behave like small programs that print their current values. Thus the value of a variable can be obtained by simply typing its name, or it can be used in a command line by enclosing it in brackets to form a function call. The following command file (which also illustrates the use of 'if', 'eval', and 'goto') will count from 1 to the number given as its first argument:

```
declare i
set i = 1
:loop
   if [eval i ">" [arg 1]]
      goto exit
   fi
   i
   set i = [eval i + 1]
   goto loop
:exit
```

Note the use of the "eval" function, which treats its arguments as an arithmetic expression and returns the expression's value. This is required to insure that the string "i + 1" is interpreted as an expression rather than as a character string. Also note that 'fi' terminates the 'if' command.

When setting a variable to a string containing unprintable characters, you may use a special mnemonic form to prevent having

to type the literal characters. For example

```
set crlf = "<cr><lf>"
```

sets the variable 'crlf' to a literal carriage return followed by a linefeed. There are times when this is not desirable, so to prevent the interpretation of the string, simply escape the start the start on the mnemonic with the Subsystem escape character (an '@'). To set set the variable 'crlf' to the literal string "<cr><lf>" you would type

```
set crlf = "@<cr>@<lf>"
```

The quotes in these two cases are necessary, otherwise the shell would try to interpret the '>' as an I/O redirector. If the string between the "<>" characters is not a legal ASCII mnemonic, no substitution will be made and the string will be passed unchanged.

Interrupts, Quits and Error Handling Mechanisms

Normally, if you interrupt a program, it will terminate and the next thing you will see is the Subsystem's prompt for your next command. However, by defining the shell control variable "_quit_action" in your "=varsdir=/.vars" file, the fault handler will, upon detection of the interrupt, prompt you as to whether to abort the current program, continue, or call Primos. For program errors, the fault handler will always ask whether you want to abort the program, continue, or call Primos (regardless of whether "_quit_action" is defined or not). The Application Notes discuss how to go about creating shell variables (which are kept in "=varsdir=/.vars" for storage between login sessions).

Conclusion

This concludes the tutorial chapter of this document. Despite the fact that a good deal of material has been presented, much detail has been omitted. The next chapter is a complete summary of the capabilities of the command interpreter. It is written in a rather technical style, and is recommended for reference rather than self-teaching. The last chapter is a set of examples that may prove helpful. As always, the best approach is simply to sit down at a terminal and try out whatever you wish to do. Should you have difficulty, further tutorials are available, and the 'help' command can be consulted for quick reference.

Summary of Syntax and Semantics

This section is the definitive document for the syntax and corresponding semantics of the Software Tools Subsystem Command Interpreter. It is composed of several sub-sections, each covering some major area of command syntax, with discussions of the semantic consequences of employing particular constructs. It is not intended as a tutorial, nor is it intended to supply multitudinous examples; the other sections of this document are provided to fill those needs.

Commands

<command> ::= [<net> { ; <net> }] <newline>

The "command" is the basic unit of communication between the command interpreter and the user. It consists of any number of networks (described below) separated by semicolons and terminated by a newline. The networks are executed one at a time, left-to-right; should an error occur at any point in the parse or execution of a network, the remainder of the <command> is ignored. The null command is legal, and causes no action.

The command interpreter reads commands for interpretation from the "command source." This is initially the user's terminal, although execution of a command file may change the assignment. Whenever the command source is the terminal, and the command interpreter is ready for input, it prompts the user with the string contained in the shell variable '_prompt'. Since this variable may be altered by the user, the prompt string is selectable on a per-user basis.

Networks

<net> ::= <node>
{ <node separator> { <node separator> } <node> }

<node separator> ::= , | <pipe connection>

<pipe connection> ::= [<port>] '|' [<node number>] [.<port>]

<port> ::= <integer>

<node number> ::= <integer> | $ | <label>

A <net> generates a block of (possibly concurrent) processes that are bound to one another by channels for the flow of data. Typically, each <node> corresponds to a single process. (<Node>s are described in more detail below.) There is no predefined "execution order" of the processes composing a <net>; the command interpreter will select any order it sees fit in order to satisfy the required input/output relations. In particular, the user is specifically enjoined <u>not</u> to assume a left-to-right serial

execution, since some <net>s cannot be executed in this manner.

Input/output relations between <node>s are specified with the <node separator> construct. The following discussion may be useful in visualizing the data flows in a <net>, and clarifing the function of the components of the <node separator>.

The entire <net> may be represented as a directed graph with one vertex for each <node> (typically, equivalent to each process) in the net. Each vertex may have up to n arcs terminating at it (representing "input data streams"), and m arcs originating from it (representing "output data streams"). An arc between two vertices indicates a flow of data from one <node> to another, and is physically implemented by a pipe.

Each of the n possible input points on a <node> is assigned an identifier consisting of a unique integer in the range 1 to n. These identifiers are referred to as the "port numbers" for the "standard input ports" of the given <node>. Similarly, each of the m possible output points on a <node> is assigned a unique integer in the range 1 to m, referred to as the port numbers for the "standard output ports" of the given <node>.

Lastly, the <node>s themselves are numbered, starting at 1 and increasing by 1 from the left end of the <net> to the right.

Clearly, in order to specify any possible input/output connection between any two <node>s, it is sufficient to specify:

- The number of the "source" <node>.
- The number of the "destination" <node>.
- The port number of the standard output port on the source <node> that is to be the source of the data.
- The port number of the standard input port on the destination <node> that is to receive the data.

The syntax for <node separator> includes the specifications for the last three of these items. The source <node> is understood to be the node that immediately precedes the <node separator> under consideration. The special <node separator> "," is used to separate <node>s that do not participate in data sharing; it specifies a null connection. Thus, the <node separator> provides a means of establishing any possible connection between two <node>s of a given <net>.

The full flexibility of the <node separator> is rarely needed or desirable. In order to make effective use of the capabilities provided, suitable defaults have been designed into the syntax. The semantics associated with the defaults are as follows:

- If the output port number (the one to the left of the vertical bar) is omitted, the next unassigned output port (in increasing numerical order) is implied. This default action takes place only after the entire <net> has been examined, and all non-defaulted output ports for the given node have been assigned. Thus, if the first <node separator> after a <node> has a defaulted output port number, port 1 will be assigned if and only if no other <node separator> attached to that <node> references output port 1. It is an error for two <node separators> to reference the same output port.

- If the destination <node> number is omitted, then the next node in the <net> (scanning from left to right) is implied. Occasionally a null <node> is generated at the end of a <net> because of the necessity for resolving such references.

- If the destination <node>'s input port number is omitted, then the next unassigned input port (in increasing numerical order) is implied. As with the defaulted output port, this action takes place only after the entire <net> has been examined. The comments under (1) above also apply to defaulted input ports.

In addition to the defaults, specifying input/output connections between widely separated <node>s is aided by alternative means of giving <node> numbers. The last <node> in a <net> may be referred to by the <node number> $, and any <node> may be referred to by an alphanumeric <label>. (<Node> labelling is discussed in the section on <node> syntax, below.) If the first <node> of a <net> is labelled, the <net> may serve as a target for the 'goto' command; see the Applications Notes for examples.

As will be seen in the next section, further syntax is necessary to completely specify the input/output environment of a <node>; the reader should remember that <node separator>s control only those flows of data between processes.

A few examples of the syntax presented above may help to clarify some of the semantics. Since the syntax of <node> has not yet been discussed, <node>s will be represented by the string "node" followed by a digit, for uniqueness and as a key to <node number>s.

A simple linear <net> of three <node>s without defaults:

```
node1  1|2.1  node2  1|3.1  node3
```

(Data flows from output port 1 of node1 to input port 1 of node2 and output port 1 of node2 to input port 1 of node3.)

The same <net>, with defaults:

```
node1 | node2 | node3
```

(Note that the spaces around the vertical bars are mandatory, so that the lexical analysis routines of the command interpreter can parse the elements of the command unambiguously.)

A simple cycle:

```
node1 |1.2
```

(Data flows from output port 1 of node1 to input port 2 of node1. Other data flows are unspecified at this level.)

A branching <net> with overridden defaults:

```
node1  |$  node2  |.1  node3
```

(Data flows from output port 1 of node1 to input port 2(!) of node3 and output port 1 of node2 to input port 1 of node3.)

Nodes

```
<node> ::=  {:<label>}  [ <simple node> | <compound node> ]

<simple node> ::= { <i/o redirector> }
                  <command name>
                  { <i/o redirector> | <argument> }

<compound node> ::= { <i/o redirector> }
                    '{' <net> { <net separator> <net> } '}'
                    { <i/o redirector> }

<i/o redirector> ::= <file name> '>'  [ <port> ]    |
                     [ <port> ]  '>'  <file name>   |
                     [ <port> ]  '>>' <file name>   |
                                 '>>' [ <port> ]

<net separator> ::=  ;

<command name> ::= <file name>

<label> ::= <identifier>
```

The <node> is the basic executable element of the command language. It consists of zero or more labels (strings of letters, digits, and underscores, beginning with a letter), optionally followed by one of two additional structures. Although, strictly speaking, the syntax allows an empty node, in practice there must be either a label or one of the two additional structures present.

The first option is the <simple node>. It specifies the name of a command to be performed, any arguments that command may require, and any <i/o redirector>s that will affect the data environment of the command. (<I/o redirectors will be discussed below.) The execution of a simple node normally involves the creation of a single process, which performs some function, then

returns to the operating system.

The second option is the <compound node>. It specifies a <net> which is to be executed according to the usual rules of <net> evaluation (see the previous subsection), and any <i/o redirector>s that should affect the environment of the <net>. The <compound node> is provided for two reasons. One, it is occasionally useful to alter default port assignments for an entire <net> with <i/o redirector>s, rather than supplying <i/o redirector>s for each <node>. Two, use of compound nodes containing more than one <net> gives the user some control over the order of execution of his processes. These abilities are discussed in more detail below.

Since it is the more basic construct, consider the <simple node>. It consists of a <command name> with <argument>s, intermixed with <i/o redirector>s. The <command name> must be a filename, usually specifying the name of an object code file to be loaded. The command interpreter locates the command to be performed by use of a user-specified "search rule." The search rule resides in the shell variable "_search_rule", and consists of a series of comma-separated elements. Each element is either a template in which ampersands (&) are replaced by the <command name> or a flag instructing the command interpreter to search one of its internal tables. The flag "^int" indicates that the command interpreter's repertoire of "internal" commands is to be checked. (An internal command is implemented as a subroutine of the command interpreter, typically for speed or because of a need to access some private data base.) The flag "^var" causes a search of the user's "shell variables" (see below for further discussion of variables and functions). The following search rule will cause the command interpreter to search for a command among the internal commands, shell variables, and the directory "=bin=", in that order:

```
"^int,^var,=bin=/&"
```

The purpose of the search rule is to allow optimization of command location for speed, and to admit the possibility of restricting some users from accessing "privileged" commands. (For example, the search rule

```
"^var,//project/library/&"
```

would restrict a user to accessing his variables and those commands in the directory "//project/library". He could not alter this restriction, since he does not have access to the (internal) 'set' command; the "^int" flag is missing from his search rule.) In addition to restricting a user to commands in specific directories, the system administrator can also restrict a user from using certain internal commands (and allow use of all other internal commands). This is accomplished by adding "qualifiers" after the internal command flag in the search rule. The qualifiers are characters representing the class of commands to be excluded in the search for internal commands to be executed. Qualifiers follow the "^int" flag, separated from it by a slash.

The following table summarizes the qualifiers and which internal commands they exclude :

Qualifier	meaning
a	access to arguments in shell files ('arg', 'args', 'argsto', 'nargs', and 'quote')
b	access to debugging commands ('dump' and 'shtrace')
c	access to flow of control commands ('case', 'elif', 'else', 'esac', 'exit', 'fi', 'goto', 'if', 'label', 'out', 'repeat', 'then', 'until', and 'when')
d	ability to change directories (via 'cd')
h	access to environment information ('date', 'day', 'echo', 'eval', 'installation', 'line', 'login_name', and 'time')
m	access to string manipulation functions ('drop', 'index', 'substr', and 'take')
q	ability to exit the shell (via 'stop')
s	access to variable setting commands ('forget', 'set', and 'sh')
v	access to variable manipulating commands ('declare', 'declared', and 'vars')
x	access to commands which allow execution of Primos commands ('dbg', 'primos', 'vpsd', and 'x')

For instance, if the system administrator wanted to keep someone from executing the Primos Fortran compiler directly, then the following search rule would accomplish this :

```
"^int/qxv,^var,=bin=/&"
```

The "q" qualifier prevents exit from the shell (so that you can't run the Primos Fortran compiler directly), the "x" qualifier prevents you from accessing external commands from within the shell (i.e., via "x ftn prog"), and the "v" qualifier prevents you from using 'declare' to modify or create a search rule (the shell file 'fc', which is the Subsystem interface to the Primos Fortran compiler, declares its own search rule) which contains an unqualified "^int" flag. It should be noted, however, that this is not a fool-proof method of limiting a user's access to com-

mands; a better solution is to write a program which is run at login and which "supervises" the user's session. One way of overcoming such a restriction placed by the system administrator would be to execute a command within a function call, such as the following:

```
[declare _search_rule = "<normal search rule>"; _
        <unrestricted command>]
```

By redefining the search rule, the user is then allowed to execute any desired command, including a new invocation of the command interpreter.

<Argument>s to be passed to the program being readied for execution are gathered by the command interpreter and placed in an area of memory accessible to the library routine 'getarg'. They may be arbitrary strings, separated from the command name and from each other by blanks. Quoting may be necessary if an <argument> could be interpreted as some other element of the command syntax. Either single or double quotes may be used. The appearance of two strings adjacent to one another without blanks implies concatenation. Thus,

```
"quoted "string
```

is equivalent to

```
"quoted string"
```

or to

```
quoted' string'
```

Single quotes may appear within strings delimited by double quotes, and vice versa; this is the only way to include quotes within a string. Example:

```
"'quoted string'"
'"Alas, poor Yorick!"'
```

Arguments are generally unprocessed by the command interpreter, and so may contain any information useful to the program being invoked.

In the previous section, it was shown that streams of data from "standard ports" could be piped from program to program through the use of the <pipe connection> syntax. It is also possible to redirect these data streams to files, or to use files as sources of data. The construct that makes this possible is the <i/o redirector>. The <i/o redirector> is composed of filenames, port numbers (as described in the last section), and one or two occurrences of the "funnel" (>).

The two simplest forms take input from a file to a standard port or output from a standard port to a file. In the case of delivering output to a file, the file is automatically created if

it did not exist, and overwritten if it did. In the case of taking input from a file, the file is unmodified. Example:

```
documentation>1
```

causes the data on the file "documentation" to be passed to standard input port 1 of the node;

```
1>results
```

causes data written to standard output port 1 of the node to be placed on the file "results".

If no <i/o redirector> is present for a given port, then that port automatically refers to the user's terminal.

If port numbers are omitted, an assignment of defaults is made. The assignment rule is identical to that given above for <pipe connections>: the first available port after the entire <net> has been scanned is used. <I/O redirector>s are evaluated left-to-right, so leftmost defaulted redirectors are assigned to lower-numbered ports than those to their right. For example,

```
data> requests>  trans  2>summary 3>errors | sp
```

is the same as

```
data>1 requests>2 trans 2>summary 3>errors 1|2.1 sp
```

where all defaults have been elaborated. 'Trans' might be some sort of transaction processor, accepting data input and update requests, and producing a report (here printed off-line by being piped to a spooler program), a summary of transactions, and an error listing.

In addition to the <i/o redirector>s mentioned above, there are two lesser-used redirectors that are useful. The first appends output to a file, rather than overwriting the file. The syntax is identical to the other output redirector, with the exception that two funnels '>>' are used, rather than one. For example,

```
2>>stuff
```

causes the data written to output port 2 to be appended to the file "stuff". (Note the lack of spaces around the redirector; a redirector and its parameters are never separated from one another, but are always separated from surrounding arguments or other text. This restriction is necessary to insure unambiguous interpretation of the redirector.) The second redirector causes input to be taken from the current command source file. It is most useful in conjunction with command files. The syntax is similar to the input redirector mentioned above, but two funnels are used and no filename may be specified. As an example, the following segment of a command file uses the text editor to change all occurrences of "March" to "April" in a given file:

```
>> ed file
g/March/s//April/
w
q
```

When the editor is invoked, it will take input directly from the command file, and thus it will read the three commands placed there for it.

The "command source" and "append" redirectors are subject to the same resolution of defaults as the other redirectors and <pipe connection>s. Thus, in the example immediately above,

```
>> ed file
```

is equivalent to

```
>>1 ed file
```

Now that the syntax of <node> has been covered, just two further considerations remain. First, the nature of an executable program must be defined. Second, the problem of execution order must be clarified.

In the vast majority of cases, a <node> is executed by bringing an object program into memory and starting it. However, the <command name> may also specify an internal command, a shell variable, or a command file. Internal commands are executed within the command interpreter by the invocation of a subroutine. When a shell variable is used as a command, the net effect is to print the value of the variable on the first output port, followed by a newline. If the filename specified is a text file rather than an object file, the command interpreter "guesses" that the named file is a file of commands to be interpreted one at a time. In any case, command invocation is uniform, and any <i/o redirector> or <pipe connection> given will be honored. Thus, it is allowable to redirect the output of a command file just as if it were an object program, or copy a shell variable to the line printer by connecting it to the spooler through a pipe.

As mentioned in the section on <net>s, the execution order of nodes in a <net> is undefined. That is, they may be executed serially in any order, concurrently, or even simultaneously. The exact method is left to the implementor of the command interpreter. In any case, the flows of data described by <pipe connection>s and <i/o redirector>s are guaranteed to be present. There are times when it would be preferable to know the order in which a <net> will be evaluated; to help with this situation, <compound node>s may be used to effect serialization of control flow within a network. <Net>s separated by semicolons or newlines are guaranteed to be executed serially, left-to-right, otherwise the command interpreter would exhibit unpredictable behavior as the user typed in his commands. Suppose it is necessary to operate four programs; three may proceed concurrently to make full use of the multiprogramming capability

of the computer system, but the fourth must not be executed until the second of the three has terminated. For simplicity, we will assume there are no input/output connections between the programs. The following command line meets the requirements stated above:

```
program1, {program2; program4}, program3
```

(Recall that the comma represents a null i/o connection.) Suppose that we have a slightly different problem: the fourth program must run after all of the other three had run to completion. This, too, can be expressed concisely:

```
program1, program2, program3; program4
```

Thus, the user has fairly complete control over the execution order of his <net>s. (The use of commas and semicolons in the command language is analogous to their use for collateral and serial elaboration in Algol 68.)

This completes the discussion of the core of the command language. The remainder of the features present in the command interpreter are provided by a built-in preprocessor, which handles function calls, iteration, and comments. The next few sections deal with the preprocessor's capabilities.

Comments

Any good command language should provide some means for the user to comment his code, particularly in command files that may be used by others. The command interpreter has a simple comment convention: Any text between an unquoted sharp sign (#) and the next newline is ignored. A comment may appear at the beginning of a line, like this:

```
# command file to preprocess, compile, and link edit
```

Or after a command, like this:

```
file.r> rp   # Ratfor's output goes to the terminal
```

Or even after a label, for identification of a loop:

```
:loop     # beginning of daily cycle
```

As far as implications in other areas of command syntax, the comment is functionally equivalent to a newline.

Variables

<variable> ::= <identifier>

| <value> ::= { <printable char> | <unprintable char> }

<unprintable char> ::= '<' <ascii mnemonic> '>'

<set command> ::= set [<variable>] = [<value>]

<declare command> ::= declare { <variable [= <value>] }

<forget command> ::= forget <variable> { <variable> }

The command interpreter supports named string storage areas for miscellaneous user applications. These are called variables. Variables are identified by a name, consisting of letters of either case, digits, and underscores, not beginning with a digit. Variables have two attributes: value and scope. The value of a variable may be altered with the 'set' command, discussed below. The scope of a variable is fixed at the time of its creation; simply, variables declared during the time when the command interpreter is taking input from a command file are active as long as that file is being used as the command source. Variables with global scope (those created when the command interpreter is reading commands from the terminal) are saved as part of the user's profile, and so are available from terminal session to terminal session. Other variables disappear when the execution of the command file in which they were declared terminates.

Variables may be created with the 'declare' command. 'Declare' creates variables with the given names at the current lexical level (within the scope of the current command file). The newly-created variables are assigned a null value, unless an initialization string is provided.

Variables may be destroyed prematurely with the 'forget' command. The named variables are removed from the command interpreter's symbol table and storage assigned to them is released to the system. Note that variables created by operations within a command file are automatically released when that command file ceases to execute. Also note that the only way to destroy variables at the global lexical level is to use the 'forget' command.

The value of a variable may be changed with the 'set' command. The first argument to 'set' is the name of the variable to be changed. If absent, the value that would have been assigned is printed on 'set's first standard output. The last argument to 'set' is the value to be assigned to the variable. It is uninterpreted, that is, treated as an arbitrary string of text. If missing, 'set' reads one line from its first standard input, and assigns the resulting string. If the variable named in the first argument has not been declared at any lexical level, 'set' declares it at the current lexical level.

A variable may contain any legal ASCII character. To allow the user to enter unprintable characters that might be a problem to Primos or the shell, the commands that manipulate variables allow the use of ASCII mnemonics in the value of a shell variable. The following would set the "_kill_resp" variables to

two ASCII escape characters, a backspace, and the string "*del*":

```
set _kill_resp = "<esc><esc><bs>*del*"
```

To prevent the interpretation of the mnemonics (i.e. to enter a literal "<esc><esc><bs>*del*", in this case) the user simply uses the Subsystem escape character in front of the mnemonics:

```
set _kill_resp = "@<esc>@<esc>@<bs>*del*"
```

Variables are accessed by name, as with any command. (Note that the user's search rule must contain the flag "^var" before variables will be evaluated.) The command interpreter prints the value of the variable on the first standard output. This behavior makes variables useful in function calls (discussed below). In addition, the user may obtain the value of a variable for checking simply by typing its name as a command.

Iteration

<iteration> ::= '(' <element> { <element> } ')'

Iteration is used to generate multiple command lines each differing by one or more substrings. Several iteration elements (collectively, an "iteration group") are placed in parentheses; the command interpreter will then generate one command line for each element, with successive elements replacing the instance of iteration. Iteration takes place over the scope of one <net>; it will not extend over a <net separator>. (If iteration is applied to a <compound node>, it will, of course, apply to the entire <node>; not just to the first <net> within that <node>.)

Multiple iterations may be present on one command; each iteration group must have the same number of elements, since the command interpreter will pick one element from each group for each generated command line. (Cross-products over iteration groups are not implemented.)

An example of iteration:

```
] fos part(1 2 3)
```

is equivalent to

```
] fos part1; fos part2; fos part3
```

and

```
] cp (intro body summary) part(1 2 3)
```

is equivalent to

```
] cp intro part1; cp body part2; cp summary part3
```

Function Calls

<function call> ::= '[' <net> { <net separator> <net> } ']'

Occasionally it is useful to be able to pass the output of a program along as arguments to another program, rather than to an input port. The "function call" makes this possible. The output appearing on each of the first standard output ports of the <net>s within the function call is copied into the command line in place of the function call itself. Line separators (newlines) present in the <net>'s output are replaced by blanks. No quoting of <net> output is performed, thus blank-separated tokens will be passed as separate arguments. (If quoting is desired, the filter 'quote' can be used or the shell variable "_quote_opt" may be set to the string "YES" to cause automatic quotation.)

A <net> may of course be any network; all the syntax described in this document is applicable. In particular, the name of a variable may appear with the brackets; thus, the value of a variable may be substituted into the command line.

History Mechanism

<history_command> ::= <cmd_select> <arg_select> <substitution>

The shell provides a sort of dynamic macro replacement facility for commands that are entered from the terminal. This is called a command history mechanism. It allows the user to recall commands he has previously entered, extract portions of the command, edit the portions he has selected, and either execute what remains or incorporate it into another command, with a minimum of typing.

A history substitution contains three parts; command selection, argument selection, and editing. Command selection chooses what command will be used. Argument selection decides which arguments are to be extracted from the chosen command line, and the editing phase allows the result to be edited to change spelling or substitute a different word for portions of the line. To prevent any history substitution from taking place, the 'hist' command can turn off the history mechanism. It also controls the saving and restoration of the current history environment. For the rest of this discussion, the assumption will be that history is currently enabled.

History substitution is triggered by the '!' character. A history substitution is normally stopped by a blank or tab character, but a trailing '!' will stop the interpretation of any further characters. This is used when concatenating supplementary text to the result of a history substitution. To prevent this and any other interpretation of the special history characters, they may be escaped with the Subsystem escape character, '@'. When a history substitution is discovered, the mechanism modifies the command line, prints the resulting command line on the user's terminal, and then passes the command to the

rest of the shell for execution. History processing occurs before any other evaluation in the shell, such as function calls and iteration. However, the use of '_' to continue an input line is done even before the history mechanism sees what you have typed; if the '_' is the last character in your history command, and the last character on the line, follow it with a terminating '!'.

Command Selection.

<cmd_select> ::= '!' [<str> | '?' <str> '?' | <num>]

The first thing in a history substitution is command selection. This is used to retrieve a given command line for use, or further processing. In a history command selection '!<str>' will find the most recent command line that started with the characters in <str>. '!?<str>?' will find the most recent command line that contained <str> anywhere on the line. It also allows <str> to contain blanks or tabs whereas the first form does not. '!<num>' allows the user to specify the number of a command according to the output of the 'hist' command. As a convenience, '!' by itself will repeat the last command entered.

Argument Selection.

<arg_select> ::= '`' [<num>] ['-' <num>]

The next portion of a history substitution is an optional argument selection. This chooses which portions of the command are to be kept. History arguments are not exactly the same as the arguments the rest of the shell uses, since history expansion occurs before argument collection. Arguments in this context are blank or tab seperated words on the command line. Function calls, iterations, and quotations will be extracted as a single argument, even if they contain blanks or tabs. Arguments are numbered from zero, starting at the leftmost portion of the line. In an argument selection, '`<num>' specifies that only argument <num> is to be extracted and kept for further processing or use, and the rest of the command line is to be dropped. '`<num>-<num>' specifies that arguments from the first <num> to the last <num> are to be kept. In place of any <num>, '$' may be specified to obtain the last argument on the line. The form '`-<num>' is a shorthand for '`1-<num>' and '`<num>-' is a short form for '`<num>-$'.

Substitution.

<substitution> ::= { '^' <str> '^' <str> '^' ['g'] }

The last portion of a history substitution is also optional and is the editing phase. This allows the portions of the command line that remain to actually be modified like the substitution command in 'ed', although much more limited. In the history mechanism, <str> is not a regular expression, as in 'ed', but is taken as a simple string. The regular expression special characters are not recognized in the history mechanism. Each substitu-

tion happens only once on the line unless a 'g' is appended on the substitution, in which case the change occurs globally on the line. Substitutions may be strung together, so that more than one may be performed at a time.

Finally, after all history substitutions have been made, the Shell will echo the new command line to the terminal, and then execute it. See the Application Notes for a discussion of the 'hist' command.

Conclusion

This concludes the description of command syntax and semantics. The next, and final, chapter contains actual working examples of the full command syntax, along with suggested applications; it is highly recommended for those who wish to gain proficiency in the use of the command language.

Application Notes

This section consists mostly of examples of current usage of the command interpreter. Extensive knowledge of some Subsystem programs may be necessary for complete understanding of these examples, but basic principles should be clear without this knowledge.

Basic Functions

In this section, some basic applications of the command language will be discussed. These applications are intended to give the user a "feel" for the flow of the language, without being explicitly pedagogical.

One commonly occurring task is the location of lines in a file that match a certain pattern. The 'find' command performs this function:

```
] file> find pattern >lines_found
```

Since the lines to be checked against the pattern are frequently a list of file names, the following sequence occurs often:

```
] lf -c directory | find pattern
```

Consequently, a command file named 'files' is available to abbreviate the sequence:

```
] cat =bin=/files
lf -c [args 2] | find [arg 1]
```

('Cat' is used here only to print the contents of the command file.) The internal command 'arg' is used to fetch the first argument on the command line that invoked 'files'. Similarly, the internal command 'args' fetches the second through the last arguments on the command line. The command file gives the external appearance of a program 'files' such that

```
] files pattern
```

is equivalent to

```
] lf -c | find pattern
```

and

```
] files pattern directory
```

is equivalent to

```
] lf -c directory | find pattern
```

Once a list of file names is obtained, it is frequently processed

further, as in this command to print Ratfor source files on the line printer:

```
] pr [files .r$ | sort]
```

'Files' produces a list of file names with the ".r" suffix, which is then sorted by 'sort'. 'Pr' then prints all the named files on the line printer.

One problem arises when the pattern to be matched contains command language metacharacters. When the pattern is substituted into the network within 'files', and the command interpreter parses the command, trouble of some kind is sure to arise. There are two solutions: One, the filter 'quote' can be used to supply a layer of quotes around the pattern:

```
lf -c [args 2] | find [arg 1 | quote]
```

Two, the shell variable "_quote_opt", which controls automatic function quotation by the command interpreter, can be set to the string "YES":

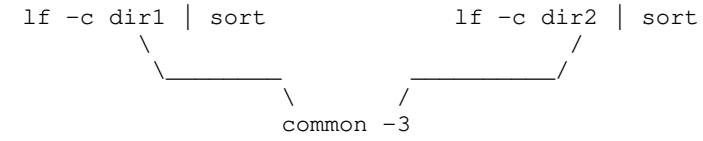
```
declare _quote_opt = YES
lf -c [args 2] | find [arg 1]
```

This latter solution works only because 'args' prints each argument on a separate line; the command interpreter always generates separate arguments from separate lines of function output. In practice, the first solution is favored, since the non-intuitive quoting is made more evident.

One common non-linear command structure is the so-called "Y" structure, where two streams of data join together to form a third (after some processing). This situation occurs because of the presence of dyadic operations (especially comparisons) in the tools available under the Subsystem. As an example, the following command compares the file names in two directories and lists those names that are present in both:

```
] lf -c dir1 | sort |$ lf -c dir2 | sort | common -3
```

Visualize the command in this way:



The two 'lf' and 'sort' pairs produce lists of file names that are compared by 'common', which produces a list of those names common to both input lists.

Command files tend to be used not only for oft-performed tasks but also to make life easier when typing long, complex commands. Quite often these long command lines make use of line

continuation -- a newline preceded immediately by an underscore is ignored. The following command file is used to create a keyword-in-context index from the heading lines of the Subsystem Reference Manual. Although it is not used frequently, it does a great deal of work and is illustrative of many of the features of the command interpreter.

```
# make_cmd.k --- build permuted index of commands
   files .d$ -f s1 _
      | change % "find %.hd -o 1" _
      | sh _
      | change '%.hd  *{[~ ]*} [ "]*{[~"]*}?*' '@1: @2' _
      | kwic -d =aux=/spelling/discard _
      | sort -d | unrot -w [width]  >cmd.k
```

First a few words on how Subsystem documentation is stored: The documentation for Subsystem commands resides in a subdirectory named "s1". The documentation for each command is in a separate file with the name "<command>.d". The heading line in each file can be identified by the characters ".hd" at the beginning of the line.

The entire command file consists of a single network. The 'files' command produces a list of the full path names (the -f option is passed on to 'lf') of the files in the subdirectory "s1" that have path names ending with the characters ".d". The next 'change' command generates a 'find' command for each documentation file to find the heading line. These command lines are passed back to the shell ('sh') for execution. The outputs of all of these 'find' commands, namely the heading lines from all the documentation files, are passed back on the first standard output of 'sh'. The second 'change' command uses tagged patterns to isolate the command name and its short description from the header line and to construct a suitable entry for the kwic index generator. Finally, 'kwic', 'sort', and 'unrot' produce the index on the file "cmd.k".

To this point, only serially-executed commands have been discussed, however sophisticated or parameterized. Control structures are necessary for more generally useful applications. The following command file, 'ssr', shows a useful technique for parameter-setting commands. Like many APL system commands, 'ssr' without arguments prints the value it controls (in this case, the user's command search rule), while 'ssr' with an argument sets the search rule to the argument given, then prints the value for verification. 'Ssr' looks like this:

```
# ssr --- set user's search rule and print it

   if [nargs]
      set _search_rule = [arg 1 | quote]
   fi

   _search_rule
```

The 'if' command conditionally executes other commands. It

requires one argument, which is interpreted as "true" if it is present, not null, and non-zero. If the argument is true, all the commands from the 'if' to the next unmatched 'elif', 'else' or 'fi' command are executed. If the argument is false, all the commands from the next unmatched 'else' command (if one is present) to the next unmatched 'fi' command are executed. In 'ssr' above, the argument to 'if' is a function call invoking 'nargs', a command that returns the number of arguments passed to the command file that is currently active. If 'nargs' is zero, then no arguments were specified, and 'ssr' does not set the user's search rule. If 'nargs' is nonzero, then 'ssr' fetches the first argument, quotes it to prevent the command interpreter from evaluating special characters, and assigns it to the user's search rule variable '_search_rule'.

'If' is useful for simple conditional execution, but it is often necessary to select one among several alternative actions instead of just one from two. The 'case' command is available to perform this function. One example of 'case' is the command file 'e', which is used to invoke either the screen editor or the line editor depending on which terminal is being used (as well as remembering the name of the file last edited):

```
# e --- invoke the editor best suited to a terminal
#       (this is not the current version of 'e' in =bin=)

   if [nargs]
      set f = [arg 1 | quote]
   fi

   case [line]
      when 10
         se -t consul [se_params] [f]
      when 11
         se -t b200   [se_params] [f]
      when 15
         se -t b150   [se_params] [f]
      when 17
         se -t gt40   [se_params] [f]
      when 18
         se -t b200   [se_params] [f]
      when 25
         se -t b150   [se_params] [f]
      out
         ed [f]
   esac
```

The first 'if' command sets the remembered file name (stored in the shell variable 'f') in the same fashion that 'ssr' was used to set the search rule (above). The 'case' command then selects from the terminals it recognizes and invokes the proper text editor. The argument of 'case' is compared with the arguments of successive 'when' commands until a match occurs, in which case the group of commands after the 'when' is executed; if no match occurs, then the commands after the 'out' command will be executed. (If no 'out' command is present, and no match occurs,

then no action is taken as a result of the 'case'.) The 'esac' command marks the end of the control structure. In 'e', the 'case' command selects either 'se' (the screen editor) or 'ed' (the line editor), and invokes each with the proper arguments (in the case of 'se', identifying the terminal type and specifying any user-dependent personal parameters).

The 'goto' command may be used to set up a loop within a command file. For example, the following command file will count from 1 to 10:

```
# bogus command file to show computers can count

   declare i = 1

   :loop
      i
      set i = [eval i + 1]
      if [eval i <= 10]
         goto loop
      fi
```

The 'repeat' command is used to set up loops but, unlike the 'goto' command, will also work from the terminal. The following loop will do exactly what the previous command file did, but will also work when entered from a terminal:

```
# not quite as bogus a loop to show computer counting

   declare i = 1

   repeat
      i
      set i = [eval i + 1]
   until [eval i '>' 10]
```

History Examples

Command history provides a quick way of re-executing a command without retyping the entire command line. The following example shows how a user can run the previous command again by only typing a '!':

```
] time
11:59:04
] !
time
11:59:08
```

Another advantage is the ability to fix a mistyped command. For example, to list the contents of the directory "stuff.u" where the ".u" was omitted in the 'lf' command and then correc-

ted.

```
] lf stuff
stuff: not found
] !!.u
lf stuff.u
bogus        gorf         snert
```

Two '!'s are used because text must be entered right next to the history substitution. Any other time, the trailing '!' is not needed.

The 'hist' command, without any arguments, will print a list of the current history and their command numbers.

```
] hist
  1: pmac gorf.s; ld gorf.b -o snert
  2: se gorf.s
  3: pmac gorf.s; ld gorf.b -o gorf
  4: gorf
  5: se gorf.s
```

At this point it is time to execute the 'pmac' and 'ld' statements, again. There are several ways to do this. One is to give the specific command number (as printed by 'hist'):

```
] !3
pmac gorf.s; ld gorf.b -o gorf
```

or let the history do more of the work for us by telling it to look for the command starting with 'pmac':

```
] !pmac
pmac gorf.s; ld gorf.b -o gorf
```

or if that is not the correct command, entering a unique string that appears anywhere on the command line:

```
] !?-o sn
pmac gorf.s; ld gorf.b -o snert
```

Notice that the trailing '?' wasn't needed. This is because it would have occured at the end of the line. None of the delimiting characters need to be entered at the end of the line because the command substitution will place them there for you at the end of a line. Also notice that the shell will <u>always</u> echo the command produced by the history mechanism to the terminal, so that you can know for sure exactly what the shell is doing.

Argument selection allows the user to retrieve certain arguments from the selected command line. After a command line is selected (as in the previous examples) then argument selection takes place. For example, given the command line

```
] echo 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
```

to retrieve only arguments 3 to 7 one can type:

```
] echo 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
] echo !`3-7
echo 3 4 5 6 7
3 4 5 6 7
```

or to grab the first item on the line,

```
] echo 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
] echo !`0
echo echo
echo
```

because argument zero (the command name) is the first item on the line.

The history mechanism does not know about command <nodes>. E.g., a '|', and the command name after it, are treated as just plain arguments. Numbering starts at zero, and each successive blank separated "item" is considered another argument. In the case of a function call, iteration, or quoted string, blanks and tabs are insignificant until all the brackets, parentheses, and quotes match up. In this manner, an entire function call, iteration group, or string counts as a single argument, whether or not it contains spaces.

```
] echo (gorf.s snert.r)
gorf.s snert.r
] cat -h !`1
cat -h (gorf.s snert.r)
==================== gorf.s ====================
 SEG
 DYNT BURF$
 END
==================== snert.r ====================
   call print(STDOUT, "burf*n"s)
   stop
   end
```

or for a more complicated example

```
] echo [echo berf] (blert blort) "final word"
berf blert final word
berf blort final word
] echo !`3 !`1 !`2
echo "final word" [echo berf] (blert blort)
final word berf blert
final word berf blort
```

The last portion of a history replacement is substitution. This allows previously selected portions of the command line to be placed through a set of substitutions similar to the 'change'

command or the substitute command in the editor. To change the "blert" in the previous example to "bonzo", you would type

```
] echo [echo berf] (blert blort) "final word"
berf blert final word
berf blort final word
] !^blert^bonzo^
echo [echo berf] (bonzo blort) "final word"
berf bonzo final word
berf blort final word
```

The operations can be combined. For instance to move arguments around, and make substitutions

```
] echo one two three
one two three
] echo !`3 !`1^one^1^ !`2
echo three 1 two
three 1 two
```

There can be more than one substitution per command line, and the given changes can be made globally.

```
] echo aa bb cc dd ee
aa bb cc dd ee
] !^a^z
echo za bb cc dd ee
za bb cc dd ee
] !?aa?^b^y^g
echo aa yy cc dd ee
aa yy cc dd ee
] !?a bb?^a^z^g^b^y^g^ee^ve^^d^w
echo zz yy cc wd ve
zz yy cc wd ve
```

The first substitution simply changes the first "a" to a "z". The second one recalls the most recent command with an "aa" in it and changes the first "b" to a "y". The last one looks for the most recent command that contains an "a bb" string (the first line) and then substitutes a "z" for all occurences of an "a", a "y" for all occurences of a "b", a "ve" for the first "ee", and a "w" for the first "d". Notice that for the last substitution, the trailing '^' was not necessary.

History processing takes place across the entire input line, even inside quoted strings. To get one of the literal history characters (!^`), you must escape it with the Subsystem escape character, '@'.

Finally, the 'hist' command is available to control the use of the history mechanism. 'Hist on' turns on history processing. By default, it is off. 'Hist off' turns history processing off. 'Hist save <file>' will save the current list of remembered commands into <file>, or into =histfile= if <file> is not specified. 'Hist restore <file>' will retrieve a saved history session from <file>, or from =histfile= if <file> is not specified. It is

recommended that you put a 'hist restore' into your '_hello' variable or the file it executes (if you want to save your shell sessions across logins). If history processing is not turned on when you do a 'hist restore', the shell will automatically turn it on for you, and then restore your saved command history. If history is turned on, whenever you issue a 'stop' command (like =bin=/bye does), the shell will automatically do a 'hist save' for you. This will also happen if you type an EOF at the shell (usually control-c), unless you also have "_nottyeof" set (see below).

Shell Control Variables

Many special shell variables are used to control the operation of the command interpreter. You can define or change any shell variable with 'set' and can delete it with 'forget'. The current value of a shell variable can be examined by entering its name. The values of all your shell variables can be examined with the 'vars' command. Certain shell variables are read into the SWT common block at Subsystem initialization to control the terminal input routines. If these variables are changed, the shell will modify the Subsystem common to reflect the change immediately. The variables that could accept control characters as values may be entered using the ASCII mnemonics supported by the shell variable commands (see the heading "variables" in the reference section of this manual). The following table identifies these variables and gives a short explanation of the function of each.

Variable	Function
_ci_name	This variable is used to select a command interpreter to be executed when the user enters the Subsystem. It should be set to the full pathname of the command interpreter desired. This variable is only checked on entrance to the Subsystem, so if this is changed, the user should exit the Subsystem (say with 'stop') and then reenter (using the 'swt' command). The default value is "=bin=/sh".
_eof	This variable may be set to a single character which will be used to signal the end of file from a terminal. The Subsystem input routines will recognize an instance of this character anywhere on the input line and send the appropriate signal to the input routine. The default value is the ASCII character ETX (control-c).
_erase	This variable may be set to a single character to be used as the "erase," or character delete, control character for Subsystem terminal input processing.

_escape This variable may be set to a single character to be used as the "escape" control character for Subsystem terminal input processing. Note that this will not not change the standard Subsystem escape character, it remains an '@'. (See the help on 'tcook$' for the gory details.)

_hello This variable, if present, is used as the source of a command to be executed whenever the user enters the Subsystem. It is frequently used to implement memo systems, supply system status information, and print pleasing messages-of-the-day.

_kill This variable may be set to a single character to be used as the "kill," or line delete, control character for Subsystem terminal input processing.

_kill_resp This variable may be set to any string which will appear on the user's terminal when the kill character is entered. If this variable is not present "\\" is the kill response.

_mail_check This variable determines how often mail is checked during the login session. If not declared, the user is not notified of incoming mail while he is logged in. If the variable is set to an integer value, the shell will check for changes in his mailbox status after that many seconds has elapsed, just before his prompt string is printed. The user is notified by the message, "You have new mail". If the variable is declared but not set, or set to an illegal value, the default is to check every 60 seconds.

_newline This variable may be set to a single character which will be interpreted as the end-of-line. Whenever this character is encountered, a carriage return and linefeed will be echoed to the terminal. If it is not set, then the ASCII character LF is the default.

_nottyeof An EOF character typed at command level 1 will normally terminate the Subsystem and place the user face to face with the Primos operating system. Most commands accept input from the terminal if an alternate file is not specified and if the user's keyboard happens to bounce, the user is bounced into Primos. If this variable is declared, an EOF typed at command level 1 will not terminate the shell but will type the message "use 'stop' to exit the subsystem" and return to command level.

_pause_gossip This variable controls the paging of gossip messages. If this variable is set, the gossip will pause at the last page, otherwise it simply returns to command level without allowing any paging commands.

_prompt This variable contains the prompt string printed by the command interpreter before any command read from the user's terminal. The default value is a right bracket (]).

_prt_dest This variable contains the location where all files spooled by this user are to be printed. If this variable is not present, files will be printed at the system-defined default printer.

_prt_form This variable contains the form to be used for files spooled by this user (e.g. "narrow"). If this variable is not present, files will be printed on the system-defined default form.

_quit_action If this variable is present, whenever the fault handler detects a break, it will prompt you as to whether you want to continue, terminate the program or call Primos. Otherwise, a break will return you to the Subsystem.

_quote_opt This variable, if set to the value "YES", causes automatic quotation of each line of program output used in a function call. It is mainly provided for compatibility with an older version of the command interpreter, which performed the quoting automatically. The program 'quote' may be used to explicitly force quotation.

_retype This variable may be set to a single character to be used as the "retype" control character for Subsystem terminal input processing.

_search_rule This variable contains a sequence of comma-separated elements that control the procedure used by the command interpreter to locate the object code for a command. Each element is either (1) the flag "^int", meaning the command interpreter's table of internal commands, (2) the flag "^var", meaning the user's shell variables, or (3) a template containing the character ampersand (&), meaning a particular directory or file in a directory. In the last case, the command name specified by the user is substituted into the template at the point of the ampersand, hopefully providing a full pathname that locates the object code needed.

| _vth_gossip This causes any gossip that is received to be paged using the screen oriented paging mechanism.

Shell Command Statistics

If the public or private template "=statistics=" is defined with the value "yes", the shell will record every command issued by the user in the directory defined by the system template "=statsdir=". If you set your private template "=statistics=" to "yes" then your commands will be recorded in the directory defined by your "=statsdir=" template. The files in the directory "=statsdir=" are named "sh<pid>"; command statistics for a given process are stored in the file with the corresponding process id. Here is an example of the file:

```
122680 171812 16 system  1 F //bin/x
122680 171816 16 system  1 F //bin/lf
122680 171822 16 system  1 F //bin/template
(date) (time)    (user)  | | (command)
                 (pid) (level) (F - command found)
```

The date begins in the first column. The (level) is the depth of nesting of shell files at which the command is requested; 1 is the terminal level.

Symbiotic Commands

There are several commands that, in effect, live symbiotically with the Shell. In the following sections, some of the more useful of these will be reviewed. For further information, consult the Software Tools Subsystem Reference Manual.

Argument Fetching. Four internal commands are frequently used in shell programs to fetch arguments given on the command line. 'Arg' fetches a single argument, 'args' fetches several, 'argsto' fetchs a specified group, and 'nargs' returns the number of available arguments.

arg <position> [<level>]

'Arg' prints on its first standard output the argument which appeared in the <position>th position in the command line invoking the shell program containing 'arg'. Position zero refers to the command name, position one to the first argument, etc. If an illegal position is specified, 'arg' prints nothing. The optional second argument, <level>, specifies the number of lexic levels to ascend in order to reach the desired argument list. The entry of any command file or function call constitutes a new lexic level; thus, an 'arg' command used in a function call to fetch an argument to the command file

containing the function call needs a <level> of 1 (to escape the lexic level in which the function is evaluated). In fact, this is the most common use of 'arg', so the default value for <level> is 1. The following three commands, when placed in a command file, would cause that command file's first argument to be printed three times on standard output one:

```
echo [arg 1]
echo [arg 1 1]
arg 1 0
```

args <first> [<last> [<level>]]

'Args' prints on its first standard output the arguments specified on the command file <level> lexic levels above the current level. <First> is the position on the command line of the first argument to be printed; <last> is the position of the last argument to be printed. If <last> is omitted, the final argument on the command line is assumed. <Level> has the same meaning as for 'arg' above.

argsto <delim> [<number> [<start> [<level>]]]

'Argsto' prints a group of arguments delimited by arguments consisting of <delim>. <Number> is an integer that controls which group of arguments is printed. If <number> is 0 or omitted, arguments up to the first occurrence of <delim> are printed; if <number> is 1, arguments between the first occurrence of <delim> and the second occurrence of <delim> are printed, and so on. <Start> is an integer indicating the argument at which the scan is to begin; if <start> is omitted (or is 1), the scan begins at the first argument. <Level> has the same meaning as for 'arg' above.

nargs [<level>]

'Nargs' prints on its first standard output the number of arguments passed to the command file <level> lexic levels above the current level. <Level> has the same meaning as for 'arg' above.

Shell Tracing. The 'shtrace' command is useful for tracing the operation of the shell. Although primarily intended for debugging the command interpreter itself, it also finds use in monitoring and debugging shell files. To turn the trace on, enter

```
shtrace on
```

To turn the trace off, enter

```
shtrace
```

Many other options are available. Consult the Software Tools Subsystem Reference Manual for details.

Shell Variable Utilities. The following commands (in addition to 'declare', 'set', and 'forget' discussed earlier) have been found useful in dealing with shell variables. Further information can, as usual, be found in the Software Tools Subsystem Reference Manual.

vars
: 'Vars' lists the names (and optionally the values) of the user's shell variables. 'Vars' can also save and restore the user's variables from arbitrary files. Various options control the listing format, the number of lexic levels scanned, and whether or not shell control variables are listed. The most common form is probably

```
vars -alv
```

: which lists all variables at all lexic levels along with their values.

Program Interface

The shell provides a set of routines which allows the user of the standard shared libraries to create shell variables, retrieve their values, and change them as well. You may also execute shell commands from within a program. This facility is not available when using the non-shared libraries, and even using the shared libraries it is somewhat restrictive until Prime supports EPF runfiles. Further information on these routines can be found in the Software Tools Subsystem Reference Manual.

shell
: 'Shell' is the subroutine which starts another level of the SWT shell. It is used to execute commands read from an open input file. It is analagous to the 'sh' command.

subsys
: 'Subsys' is used to execute a single command from within a program. It combines all the operations needed to execute a string with 'shell' without the user having to perform the operations. It is a convenience for the user.

svdel
: 'Svdel' accepts the name of a shell variable and

deletes it at the current shell level. It takes care of updating the SWT common block in the case of a special shell variable (see "Shell Control Variables", above). It is analagous to the command 'forget'.

svdump
'Svdump' prints a representation of the internal shell variable common block. It scans all levels of the variables dumping the chains and the hash tables. It is analagous to the 'dump sv' command.

svget
'Svget' simply retrieves the value of a given shell variable. Since "executing" a variable from the command level prints the value of the variable, the action of 'svget' is closest to the execution of a variable.

svlevl
'Svlevl' returns the current lexic level of the shell. This is useful in cooporation with 'svscan' (described below) to retrieve the value of all currently declared variables. This routine has no command equivalent.

svmake
'Svmake' creates a given shell variable at the current lexic level of the shell. It returns the lexic level of the shell. If the variable already exists at the current level, then 'svmake' will have no effect. Any special variables (see "Shell Control Variables", above) that are changed will cause a change in the SWT common block to reflect the value of the variable. 'Svmake' is analagous to the 'declare' command.

svput
'Svput' sets the value of a given shell variable in the most recent lexic level that it appears. If the variable does not exist in any scope of the shell, it is created in the current level. 'Svput' also makes modifications to the SWT common block if any special variables are changed. 'Svput' is analagous to the 'set' command.

svrest
'Svrest' reads a file written by 'svsave' (see below) and attempts to merge those variables with those at the current lexic level. 'Svrest' is analagous to the 'vars -r' command.

svsave
'Svsave' attempts to save the shell variables at lexic level number 1 (the top level) in the given file. 'Svsave' is analagous to the 'vars -s' com-

mand.

svscan
'Svscan' provides a way for the user to obtain the value of all shell variables at any or all lexic levels. It operates in a method similar to 'tscan$'. There is no command associated with 'svscan'.

Conclusion

This concludes the Application Notes section of the guide. Hopefully it has presented some ideas that will make the use of the command interpreter more productive and enjoyable.

Messages from the Shell

Listed here are messages with obscure meanings that are produced by the Shell; several indicate dire internal problems that should not occur during normal operation. In the interest of saving paper, self-explanatory messages are not included.

<command>: not found
The list of elements in the search rule was exhausted, but the command had not been located.

<command>: too many ci files
The nesting depth of command files has been exceeded. This is usually caused by an infinitely recursive call on a command file. The maximum nesting depth (currently 10) is a compile time option of the shell and may be increased at the expense of additional table space.

continue?
This message occurs after each network when the "single_step" shell trace option is set. A line beginning with anything other than an upper or lower case letter "n" will cause the shell to execute the next network. A response beginning with "n" will cause the shell to return to command level.

illegal destination node spec
The destination node specifier must be a defined label or a number between 1 and the number of nodes in the network.

illegal port number
A port number must be a number between 1 and the maximum number of standard ports defined (currently 3).

missing command name
Although an empty net is allowable, redirectors must not be specified without a command name.

missing pathname in redirector
A greater-than sign was encountered without a pathname on either side.

net is not serially executable
Because multiple processes per user are not supported, each node of a net must be executed serially. Therefore, nets which have pipe connections that form a complete cycle cannot be executed.

overflow (save_state): <level>
The nesting depth of command files has been exceeded. This is usually caused by an infinitely recursive call on a command file. The maximum nesting depth (currently 10) is a compile time option of the shell

and may be increased at the expense of additional table space.

pipe destination not found
The destination node of a pipe is not in the range of the current net.

state save stack overflow
The nesting depth of command files has been exceeded. This is usually caused by an infinitely recursive call on a command file. The maximum nesting depth (currently 10) is a compile time option of the shell and may be increased at the expense of additional table space.

unbalanced iteration groups
Because of the semantics of iteration, each iteration group in the same net must contain the same number of arguments.

unexpected EOF on variable save file
End of file has been encountered on the shell variable save file when a value has been expected. The shell variables have been corrupted. To recover what might be left, exit the Subsystem with a <break> or control-P and consult your system administrator.

whitespace required around pipe connector
A pipe connector and its associated port numbers and destination label must be surrounded by spaces.

whitespace required around i/o redirector
An i/o redirector and its associated i/o redirector must be surrounded by spaces.