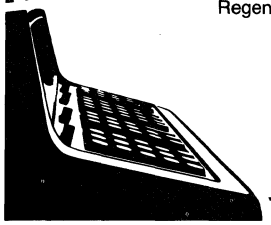
HP 9000 Series 200/500 Computers

HEWLETT PACKARD

HP-UX Concepts and Tutorials
Vol. 3: Shells and Miscellaneous Tools

HP-UX Concepts and Tutorials
Vol. 3: Shells and Miscellaneous Tools
for the HP 9000 Series 200/500 Computers

Manual Part No. 97089-90004



Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

July 1984...First Edition

Contents

The articles contained in *HP-UX Concepts and Tutorials* are provided to help you use the commands and utilities provided with HP-UX. The articles have several sources. Some were written at Hewlett-Packard specifically for the HP 9000 family of computers. Others were written at Bell Laboratories and may discuss options or command behavior that does not apply to your system.

HP-UX Concepts and Tutorials has four volumes:

- Volume 1: Text Processing and Formatting
- Volume 2: Program Development and Maintenance
- Volume 3: Shells and Miscellaneous Tools
- Volume 4: Data Communications

This is "Vol. 3: Shells and Miscellaneous Tools" and the articles it includes are:

Table of Contents

Bourne Shell Programming

The shell is perhaps one of the most versatile programs in your HP-UX system. The shell has two main roles. Its most common role is that of a *command interpreter* – reading input from your terminal, and interpreting it as a request for a particular program to be executed. The shell as a command interpreter is discussed in the supplied tutorial text by Jean Yates.

The second role of the shell is that of a *programming language*. The shell programming language is a structured programming language that is directly recognized and executed by the shell; it requires no compilation. It includes structured programming constructs like **if**, **case**, **for**, **while**, and **until**. In addition, all HP-UX commands, as well as any commands you may have written yourself, can be executed within a shell program. Other capabilities, such as parameters, parameter substitution, command substitution, and comments, are also supported.

This article describes the shell programming language in detail, including tutorial examples as appropriate.

Terminology

The terms *shell script* and *shell program* are often used interchangeably to refer to a program written in the shell programming language. In this article, however, these two terms are used differently. For the remainder of this article, *shell script* refers to a list of commands that is executed once, in the order the commands are listed. A shell script contains no conditional testing, no looping, no branching, and no structured programming constructs; it may or may not contain one or more parameters. A *shell program* refers to those procedures that do not qualify as shell scripts. The term *procedure* refers to both shell scripts and shell programs. It is used when describing concepts that are common to both.

Creating a Simple Shell Script

A *simple shell script* is one in which no parameters occur. A simple shell script is useful when the following two conditions are true:

you anticipate having to perform one or more tasks several times over a period of time, and

the tasks you must perform, the commands needed to perform those tasks, and the arguments to the commands *never change*.

A simple shell script is not well adapted to change, because you are required to edit the script if changes are necessary. However, a simple shell script can save you a lot of time and typing, as well as relieve you of certain mundane chores, if you have a well-defined set of objectives for your script.

Example

Suppose you want to monitor the user activity on a multi-user system. After considering the problem, you decide that your script should give you the following information:

1. the current date and time;
2. the names of the users currently logged in;
3. a list of all the processes on the system;
4. the number of disc blocks being used by each user.

You can see that each numbered item is satisfied by a single command:

1. **date**
2. **who**
3. **ps –e**
4. **du /users**

To make the output more readable, you decide to include the *echo* command, also. You now have a complete list of all the commands you need to perform the tasks at hand. All that remains is to put them all together in your script.

Creating the Script

Any procedure is simply an ASCII text file. Thus, you may use the editor of your choice to create your script. The editor *ed* is used in this example.

You decide that **status** is a good name for the script, since it gives information about the status of the system at a particular date and time. In the following editing session, your input is shown in **bold**:

```
$ ed status
?status
a
echo "Current date and time: `date`\n"
echo "\nUsers logged in:\n"
who
echo "\nCurrent processes:\n"
ps -e
echo "\nUser disc usage:\n"
du /users
echo "\n* * * * * * * * * * *\n"
.
w
204
q
$
```

You now have a file called **status** containing the commands you want to run.

Running the Script

You can execute **status** by typing

```
$ sh status
```

which creates a new shell to run the script. A more common way, however, is to mark **status** as executable, so that it may be executed just like other commands. To do this, use the *chmod* command, and type

```
$ chmod 755 status
```

which sets the mode of **status** such that everybody may read and execute **status**, but only you can modify it. (Note that you are free to assign whatever mode you want, as long as you give yourself execute permission. You should also give yourself read and write permission, because you cannot read or modify your script without them!) You may now execute **status** by typing

```
$ status
```

which causes the shell to execute the commands contained in **status**. The output appears on your terminal.

You have already saved yourself a lot of typing with **status**, but there is still more you can do. Suppose you want to collect this information in a file called **logfile** for later inspection. You can do this by typing

```
$ status >>logfile
```

The append redirection >> is used instead of > so that successive invocations of **status** do not overwrite the old contents of **logfile**. In this way, you can keep a history of user activity that is updated as often as **status** is executed.

The only problem with this is that you are required to manually execute **status** every time you want to update **logfile**. However, there is a command, called *cron*, that executes commands on a scheduled basis, according to entries in the file */usr/lib/crontab*. Thus, you can schedule **status** to be run as often as once a minute (note that **crontab** may be protected such that no one except the super-user may edit it – ask your super-user for details). Your only task is to check **logfile** periodically to get the information you need, and remove information that is no longer useful.

Parameters

Procedures almost always make use of at least one parameter. A *parameter* is a string of one or more characters that is made to stand for another string of characters. Parameters are similar in many respects to variables in other programming languages.

Types of Parameters

Parameters can be divided into three types: *common parameters*, *positional parameters*, and *special parameters*.

Common Parameters

A common parameter is a name consisting of a series of letters, digits, and/or underscores. The first character must be a letter or an underscore. Common parameters are completely user-defineable. Values are assigned to common parameters by writing

```
parameter = value
```

where *value* is the value to be assigned to *parameter*. The values of common parameters are always character strings. Some examples of valid assignment statements are:

```
dir = /usr/lib/gates
user = fred
null =
cwdir = `pwd`
```

In these examples, the string "/usr/lib/gates" is assigned to **dir**, "fred" is assigned to **user**, and the null string is assigned to **null**. In the last example, the name of your current working directory is assigned to **cwdir** using *command substitution*. Whenever a command is enclosed in grave accents (``), the command is replaced by its output when the command is executed. Command substitution is explained in the supplied tutorial text by Jean Yates.

Positional Parameters

Whenever you give the shell a command to execute, the shell automatically sets the values of ten positional parameters named **$0**, **$1**, **$2**, ... **$9**. Each positional parameter contains the value of one argument specified on a command line. For example, if you type

```
$ cc prog1.c prog2.c prog3.c
```

then the shell sets **$0** equal to "cc", **$1** equal to "prog1.c", **$2** equal to "prog2.c", and **$3** equal to "prog3.c". **$4** through **$9** are set equal to the null string. Thus, **$0** always contains the name of the invoked command, and **$1** through **$9** contain the values of the command's arguments, if any, in the order they are specified. The null string is assigned to all positional parameters not given a value on the command line.

If more than ten arguments (including the command name) are specified on the command line, only the first ten are assigned to positional parameters. The remaining arguments are saved, but are not accessible until the *shift* command is used. The *shift* command shifts the value of **$2** to **$1**, **$3** to **$2**, **$4** to **$3**, etc. Thus, the remaining arguments are shifted into **$9** one at a time, until eventually all the remaining arguments are addressable through positional parameters.

These parameters are available for use in procedures, and enable you to write procedures that accept arguments. Their values cannot be changed except with the *set* and *shift* commands (documented in *Special Commands* under *sh*(1), in the HP-UX Reference manual).

Special Parameters

Several parameters are "special", either because they are used elsewhere by the HP-UX system, or because they are automatically set by the shell. Parameters that are automatically set by the shell are:

the number of non-null positional parameters in decimal; can be greater than 10;

– the flags supplied to the shell on invocation, or by the *set* command; flags are stored in – as a character string, with each character in the string specifying a shell flag that has been set;

? the decimal value returned by the last synchronously executed (i.e. not executed in the background) command;

$ the process ID of the shell running the current procedure;

! the process number of the last background (asynchronous) command invoked.

The values of these parameters cannot be changed.

Parameters that are used by the system are **HOME**, **PATH**, **TERM**, **SHELL**, **EXINIT**, **MAIL**, **TZ**, **PS1**, **PS2**, and **IFS**. Some of these parameters (**TERM**, **PATH**, and possibly **MAIL**, **SHELL**, and **EXINIT**) are environment parameters, which are used to set up the environment in which your processes run. These parameters should not be changed indiscriminately. For example, **PATH** tells the shell which directories it should search to find the programs you want to execute. If you redefine **PATH** with some unrelated value, the shell no longer knows where to look. Thus, as a safety precaution, all of the above parameters should be avoided for general use in procedures, whether they are part of the environment or not.

How to Use Parameters

Once a parameter has been assigned a value, you may obtain its value by preceding the parameter with a dollar sign ($). For example,

```
dirname = /users/bill/dir1
cd $dirname
```

This example assigns the string "/users/bill/dir1" to **dirname**, and then uses **dirname** to change the current working directory. The *cd* command, after substituting the value of **dirname**, is equivalent to

```
cd /users/bill/dir1
```

The dollar sign signals the shell that the following characters specify a parameter, and that the parameter's value is to be substituted in its place. If you had omitted the dollar sign in the previous example, the shell would assume that you want to change your current working directory to a directory called "dirname".

Whenever a parameter is preceded by a dollar sign, the parameter is said to be *dereferenced.* All parameters must be dereferenced to obtain their values (positional parameters always appear with a dollar sign, and thus are always dereferenced).

One or more characters can be added to the end of a parameter value by enclosing the parameter name in braces, and explicitly typing the added character(s). For example,

```
dirname = /users/bill/dir
cd ${dirname}1
        .
        .
        .
cd ${dirname}2
        .
        .
        .
```

This example assigns "/users/bill/dir" to **dirname**, and then appends a single character to **dirname** in subsequent *cd* commands. The two *cd* commands are equivalent to

```
cd /users/bill/dir1
```

and

```
cd /users/bill/dir2
```

respectively. Note that the appended character does not affect the value of **dirname**. The following example shows how characters can be added to the beginning and the end of a parameter value:

```
fn = prog1
dirname = /users/bill
mv $fn $dirname/bill.${fn}.R2
```

Although somewhat difficult to read, this example shows a commonly used method of building file names from parameters. Using the given values of **fn** and **dirname**, the *mv* command is equivalent to

```
mv prog1 /users/bill/bill.prog1.R2
```

which moves **prog1** from the current working directory to the directory */users/bill*, and renames it **bill.prog1.R2**. Note that the braces are necessary only when a parameter name is *followed* by one or more characters that are not to be interpreted as part of the parameter name. (Note also that the braces are *not* necessary if the parameter name is immediately followed by a slash (/), as shown by **dirname** in the previous example.)

Parameter Substitution

There are constructs that enable you to substitute other values in place of parameter values, depending on whether or not the parameter is set or null. The four constructs are:

${*parameter***:–***word***}**
If *parameter* is set and non-null, then dereference its value. Otherwise, substitute *word*, where *word* can be any valid parameter value.

${*parameter***: = *word***}**
If *parameter* is not set or is null, then set it to *word*. The value of *parameter* is then dereferenced. Positional parameters may not be set in this way.

${*parameter***:?***word***}**
If *parameter* is set and is non-null, then dereference its value. Otherwise, print *word* and exit from the current process. If *word* is omitted, then the message "parameter null or not set" is printed.

${*parameter***: + *word***}**
If *parameter* is set and is non-null, then substitute *word*. Otherwise, substitute a null value.

Here are some examples:

```
echo "The directory being processed is ${1:-`pwd`}."
```

This statement could exist in a procedure that performs a specific task on each file in a particular directory. The first argument to the procedure (**$1**) is the name of the directory to process. If no directory is specified, the current working directory is used. This echo statement reports which directory is being processed. If **$1** is set, its value is printed; otherwise, command substitution is used to print the name of the current working directory.

```
pr ${whichfiles: = *} >/dev/lp
```

This statement could be used to print one or more files on the line printer */dev/lp*. The parameter **whichfiles** could be either a single file name, or a pattern of special characters, specifying the file(s) to print. If **whichfiles** is set, its value is dereferenced, and the specified files are printed. If **whichfiles** is not set or is null, it is set equal to an asterisk (*). Its value is then dereferenced, causing all the files in the current working directory to be printed.

```
cd ${arg3:?"Arg3 not set"}
```

This statement could appear in a procedure to inform the user when a necessary argument (**arg3**, in this case) has not been set. In this example, **arg3** is the name of a directory which must be specified. If **arg3** is set, the *cd* command is executed; otherwise, the message "arg3: Arg3 not set" is printed, and the process is terminated. The initial "arg3:" in the message is added by the shell as an additional identifier. (Note that, if your terminal is set to echo all eight bits of an ASCII character, you might get garbage output on your terminal.)

```
${dirname: + `echo cd $dirname`}
```

In this example, if **dirname** is set, then the command "cd $dirname" is substituted. Otherwise, no action is taken. The *echo* command used inside command substitution marks is necessary for the following reasons:

The shell expects a *single word* of information following the + in this example. Thus, the *cd* command and its argument must be enclosed in double quotes to force the shell to treat it as a single word.

Enclosing "cd $dirname" in double quotes is not enough, however, because a subtle error is generated. For example, suppose you typed the following lines in a shell program:

```
dirname = /users/bill
${dirname: + "cd $dirname"}
```

The shell sees that "cd $dirname" is to be treated as a single word, and looks for a command named "cd /users/bill" (instead of a command named "cd" with an argument of "/users/bill")! Expressed in this way, the shell cannot distinguish two arguments; it only sees one argument and, obviously, an error is generated.

The solution to this is to allow the *echo* command to pass two distinct arguments to the shell, but still make the entire construct look like a single word. This is easily done, because a command substitution construct is always treated as a single word by the shell. Thus, the double quotes surrounding "cd $dirname" are not necessary, and the construct `echo cd $dirname` causes this example to execute correctly.

The braces are necessary in all four of the previously described constructs. If the colon is omitted in any of the constructs, the shell simply checks to see if *parameter* is set or not, and no other action is taken.

Quoting

There are four characters used to quote other characters in procedures. These characters are the backslash (\), the single quote (´), the double quote ("), and the grave accent (`).

The Backslash

The special meaning of a character can be stripped away by preceding that character with a backslash. Whenever a character is preceded by a backslash, the character is said to be *quoted*, and it is interpreted literally. For example,

```
echo prog*.c  \*list\*  lib\?.3?
```

The first argument tells *echo* to print all files in the current directory whose names begin with "prog", followed by any number of characters, followed by ".c". The second argument tells *echo* to print "*list*", since both asterisks are quoted, and are thus interpreted literally. The third argument tells *echo* to print all files in the current directory whose names are "lib?.3" followed by any single character. The first question mark is literal; the second stands for any single character.

The backslash is the most powerful quoting character, in that it can quote all special characters, including itself. It is also the most limited in scope, since it can quote only one character at a time. The following list shows all the characters that are special to the shell, all of which are quotable with a backslash:

```
? * [ ] \ $ ´ " ` | & ; ( ) < > { } new-line
```

Note that, if a new-line is quoted by a backslash, the new-line is ignored completely.

If you are ever in doubt about whether or not a character needs quoting, it is safe to precede the character with a backslash; if the character has no special meaning, the backslash is ignored.

The Single Quote

The single quote quotes all the special characters except the single quote itself. It has the added advantage of enabling you to quote several characters at once. For example,

```
echo ´prog*.c *list* lib\?.3?´
```

prints the exact characters listed between the single quotes. Even the backslash is treated literally. This means that a string like

```
echo ´Can\´t find file´
```

does not work as expected, because the backslash loses its quoting ability when enclosed between single quotes. Thus, there is no way to put a single quote between single quotes without inadvertently confusing the shell.

The previous example produces a subtle error that deserves more explanation. If you type

```
$ echo ´Can\´t find file´
```

to the shell, the shell first examines the command line looking for syntax errors. It sees first the string "echo", followed by the quoted string "can\", followed by the characters "t find file", followed by the beginning of another quoted string. But wait a minute! Where's the rest of the second quoted string? The shell needs more input, so it types

```
$ echo ´Can\´t find file´
>
```

back at you. The > is the shell's default *secondary prompt*, which the shell uses to signal that more information is needed. Suppose you then type

```
$ echo ´Can\´t find file´
> text´
```

just to complete the command line so *echo* will run. Well, this satisfies the shell's syntax rules, so the shell prepares to execute the following command line:

```
echo Can\t find file(new-line)text
```

(Note that, although the single quotes have disappeared, their effect can be seen in that the backslash in the first quoted string has been interpreted literally.) Where did the new-line come from? It was the first character of the second quoted string! The command now runs "successfully", and you get

```
$ echo ´Can\´t find file´
> text´
Can        find file
text
$
```

on your screen. Hold on! Where's the "\t"? And where did all the spaces come from? This time it's not the shell's fault. The *echo* command has a few tricks of its own in the form of *escape sequences*. Escape sequences are character pairs consisting of a backslash and another character (for a complete list of *echo*'s escape sequences, refer to *echo*(1) in the HP-UX Reference manual). Each escape sequence is interpreted to mean something else, and \t causes *echo* to output a tab.

There are two lessons to learn from this. First, do not try to embed a single quote inside a string quoted by single quotes. You will invariably confuse the shell (and yourself, when you try to figure out what went wrong)! Second, beware of escape sequences in arguments to *echo*. It can be very difficult to figure out why literal characters disappear when using *echo* to print them on your screen.

Since $ and ` are quoted within single quotes, parameter and command substitution cannot be performed. For example,

```
echo '${cmdname}: current working directory is `pwd`.'
```

still echoes the exact characters shown between the single quotes.

The single quote also forces the shell to interpret several words as a single word (a *word* is a string of one or more characters, delimited by one or more spaces, tabs, and/or new-lines). Thus, many words can be enclosed in single quotes and assigned to a parameter. For example,

```
errmesg='Cannot find specified file.'
```

assigns the string "Cannot find specified file." to **errmesg**. The space characters embedded in the string no longer delimit words. Instead, the shell treats the entire string as a single word. If **errmesg** is later dereferenced, the string will look exactly as it did when it was assigned to **errmesg**.

The Double Quote

The double quote quotes all special characters except \, $, ", and `. Since the backslash is not quoted within double quotes, it may be used to quote these four characters. In the following example,

```
echo "The computer responds \"Not found\" and exits."
```

the backslash is used to quote the double quote character. Thus, a double quote may be included in a string enclosed in double quotes. Note that the backslash itself must also be quoted to be interpreted literally within double quotes.

The infamous example given in the last section can be executed with no surprises using double quotes:

```
echo "Can't find file."
```

This time, all characters show up as expected on your screen.

Since $ and ` are not quoted, parameter and command substitution are permitted. For example,

```
echo "$dirname processed at `date`."
```

prints out the name of the directory currently being processed, and the date and time at which it was processed. Note that braces are not required around **dirname**, since it is separated from the next word by a space. The backslash can be used to quote $ and `, to prevent parameter and command substitution from occurring.

The double quote also enables you to assign several words to a parameter. For example,

```
descr = "print date and time"
cmd = date
cmddescr = "$cmd – $descr"
```

This example assigns the short description of *date* to **descr**, using double quotes to force the shell to interpret the four words as a single word. The string "date" is assigned to **cmd**. Finally, the two parameters are dereferenced as shown, and assigned to **cmddescr**. Thus, **cmddescr** now contains a string similar to that found under the NAME heading in the HP-UX Reference manual.

The Grave Accent

The grave accent is used to signal the shell that a command substitution is to be performed. No special characters are quoted within grave accents. Whatever characters you type between grave accents are interpreted exactly as if they were typed after the shell prompt. For example,

```
echo "File contents:\n`cat *UX*[1–5]`"
```

This example outputs the heading "File contents:", followed by a new-line (\n). Then, the *cat* command is executed to print out the contents of all files in the current working directory whose names contain the characters "UX", and end with a single digit in the range 1 through 5. Note that, even though the command substitution is enclosed in double quotes, the characters *, [, and] are treated as unquoted in the command substitution.

The backslash may be used to quote characters within a command substitution.

Using Parameters in Shell Scripts

Adding parameters to shell scripts makes them more flexible and adaptable to your changing needs. Positional parameters are especially useful, in that they enable you to pass arguments to your shell scripts.

Examples

Example 1 - The Compile Shell Script

Compile is a short shell script that accepts one argument. It is useful when you have several C programs that you want to compile, one at a time. Each *a.out* file that is produced is renamed such that it has the same name as its corresponding source file, with the ".c" suffix removed. **Compile** contains the following lines:

```
cc $1
fn=`basename $1 .c`
mv a.out $fn
```

The *basename* command strips away all but the last component of a path name, and optionally removes a specified suffix (see *basename*(1) in the HP-UX Reference manual). Thus, if you invoke **compile** by typing

```
$ compile /users/fred/programs/prog1.c
```

the shell sets **$1** equal to "/users/fred/programs/prog1.c", and the commands in **compile** become

```
cc /users/fred/programs/prog1.c
fn=`basename /users/fred/programs/prog1.c .c`
mv a.out prog1
```

The *basename* command removed "/users/fred/programs/" and ".c" from the string contained in **$1**, causing **fn** to be set equal to "prog1". Thus, no matter what directory the source file is located in, you always end up with an executable file in your current working directory with a name similar to that of its source file.

Note that you can also invoke **compile** as

```
$ compile prog1.c
```

with the same results. The *basename* command removes parts of a path name only if those parts exist. Thus, the only part that is removed from "prog1.c" is ".c". **Compile** can therefore be used to compile C programs no matter where the source files reside.

Example 2 - The Modfile Shell Script

The **modfile** shell script copies a file from one directory into another, edits it according to a script of *ed* commands, and records the fact that the file has been copied in a bookkeeping file. It accepts three arguments: the source directory, the destination directory, and the name of the file to be copied, respectively. **Modfile** contains the following lines:

```
log = /users/kb/logfile
edsc = /users/kb/tools/edscript
cp $1/$3 $2
ed - $2/$3 <$edsc
echo "$3 copied and edited." >>$log
```

Absolute path names are used for **log** and **edsc** so that **modfile** does not depend on your current working directory. Thus, you can type

```
$ modfile /users/fred . file1
```

which copies **file1** from */users/fred* into your current working directory, or

```
$ modfile ../Cprogs /users/bill prog4.c
```

which copies **prog4.c** from the directory **Cprogs** in your parent directory into */users/bill*. Thus, **modfile** enables you to copy a file from any directory into any other directory, no matter what your current working directory is, provided the modes of the specified directories permit you to copy files.

The – option silences *ed* so that no character counts appear on your screen. The file **edscript** contains a list of one or more *ed* commands that *ed* is to apply to each file that is copied. Finally, the output from the *echo* command is redirected to the file */users/kb/logfile*, so that a record of the action taken is saved as a convenient reminder.

Note that **modfile** is written so that changes can be made without having to actually change the code. The directory names and the file to be copied are all parameters, and the file **edscript** can be edited to change the way *ed* modifies the copied file.

Example 3 - Comments and Here Documents

A *comment* is introduced by a pound sign (#), and continues until the next new-line. All characters between the pound sign and the new-line are ignored by the shell. Comments can be added to the **modfile** shell script as follows:

```
# Initialize parameters
#
log=/users/kb/logfile
edsc=/users/kb/tools/edscript
#
# Copy file
#
cp $1/$3 $2
#
# Modify copied file
#
ed - $2/$1 <$edsc
#
# Write record
#
echo "$1 copied and edited." >>$log
```

It is good programming practice to provide comments in your shell scripts where necessary. They not only help others understand your scripts, but they can also help refresh your memory if you revisit a script that you have set aside for awhile.

A *here document* is a type of I/O redirection that enables you to include input to a certain program inside the shell script itself. A here document has the following form:

command [*args* ...] <<[-] *word*
.
.
.
word

The << redirection tells the shell that the input for *command* is to be taken from the following here document. *Word* consists of one or more characters, and signals the beginning and the end of the here document. All lines between the beginning and ending *word* are given to *command* as its standard input. If any character of *word* is quoted, then all characters within the here document are quoted. Otherwise, parameter and command substitution take place, the characters \, $, and ` are special, and the first character of *word* must be quoted if it is used within the here document. If - is appended to <<, then all leading tabs are removed from the here document and from *word*.

The *ed* command in **modfile**

```
ed – $2/$3 <$edsc
```

can be rewritten to use a here document, as shown:

```
ed – $2/$3 <<–\%
        /for/c
        while(i != limit) {

        g/exit/d
%
```

In this example, % defines the beginning and end of a here document containing four lines of input for *ed*. % is preceded by a –, to strip away all leading tabs in the document, and by \, to ensure that all characters in the document are quoted. The here document is indented for clarity.

Using a here document in **modfile** requires that you edit **modfile** if you want to change the way *ed* modifies the copied file. However, by including the here document, you can eliminate the file **edscript**. Also, it is more convenient to debug a procedure if a command's input is readily accessible. Here documents become more valuable as the size and complexity of the procedure grows.

Command Separators

The characters **;**, **&**, **&&**, and || can be used to separate one or more commands or pipelines, causing sequential, asynchronous (background), or conditional execution of the commands or pipelines.

The Semicolon

The semicolon (;) causes sequential execution of each command or pipeline specified. It is equivalent to a new-line. For example,

```
cd /users/kb; mv ../fred/file1 .; ls
```

causes the shell to execute the *cd* command, then the *mv* command, and finally the *ls* command. *Sequential execution* means that the shell waits for each command to finish before executing the next one. Thus, only one additional process exists at any given time. Note that sequential execution is the normal mode of execution for the shell, so a semicolon is not needed after the *ls* command to ensure that it executes sequentially.

The Ampersand

The ampersand (&) causes asynchronous execution of each command or pipeline specified. For example,

```
cc prog1.c prog2.c & sort –d –o file1 file1 & wc textfile &
```

causes the shell to create a new process for each command listed. *Asynchronous execution* (sometimes referred to as executing a command in the background) means that the shell does not wait for termination of the first command before executing the next. Thus, depending on how long each command executes, three processes can exist at the same time in the previous example. The first process is compiling **prog1.c** and **prog2.c**, the second is sorting **file1**, and the third is counting the number of lines, words, and characters in **textfile**. Note that the ampersand following the *wc* command must be specified, or the *wc* command is executed sequentially.

The shell reports the process numbers of each process created by a & separator. Thus, if you execute the above example, three numbers are printed on your screen which identify the three processes created. These are provided for your convenience, should you decide to terminate these processes prematurely with the *kill* command.

Mixing ; and & Separators

Sequential execution works well with commands that have short execution times, and asynchronous execution works well with commands that have long execution times. It is helpful to be able to choose the type of execution based on the execution time of a command. The semicolon and ampersand separators can be intermixed on a line, so you can avoid having to wait for lengthy commands. For example,

```
cc prog1.c prog2.c prog3.c & cd /users/bill; ls –l
```

Here, the shell creates a new process and executes *cc* in that process. Then, without waiting for *cc* to finish, the shell sequentially executes *cd* and *ls*, waiting for the *cd* command to finish before executing the *ls* command.

Note that a syntax error is generated if a semicolon and an ampersand appear adjacent to each other.

The Double Ampersand

The double ampersand (&&) causes the next command or pipeline in the sequence to be executed only if the previous command or pipeline executes successfully. For example,

```
test –d /users/kb/tools && cd /users/kb/tools
```

first checks to make sure that */users/kb/tools* exists. If so, the current working directory is changed to */users/kb/tools*. If not, no further action is taken.

The Double Vertical Bar

The double vertical bar (||) causes the next command or pipeline in the sequence to be executed only if the previous command or pipeline was unsuccessful. For example,

```
test –d /users/kb/tools || mkdir /users/kb/tools
```

first checks to see if the directory */users/kb/tools* exists. If so, no further action is taken. If not, the directory is created using *mkdir*.

Mixing && and || Separators

The && and || separators can also be intermixed on a line. For example,

```
test –d /usr/tmp && rm /usr/tmp/* || echo "Permission denied"
```

which first checks to see if the directory */usr/tmp* exists. If so, all files in */usr/tmp* are removed. If *rm* fails, the message "Permission denied" is printed. If */usr/tmp* does not exist, no further action is taken.

Mixing ;, &, &&, and || Separators

All four command separators can be intermixed on a line, but the interpretation of the actual execution sequence becomes more complex. For example,

```
test –d /tools && cd /tools; test –z "$fn" || sort –o $fn $fn &
```

The shell uses ; and & to terminate a command sequence. Thus, this example contains two command sequences. The first command sequence is

```
test –d /tools && cd /tools;
```

which first checks to make sure that the directory */tools* exists. If it does, it becomes the current working directory; if not, no further action is taken. Since sequential execution is required by the semicolon, the shell executes this command sequence first, waiting until the *test* and *cd* commands have finished before executing the second command sequence. The second command sequence is

```
test –z "$fn" || sort –o $fn $fn &
```

which first checks to see if the value of **fn** has a zero length. If not, the contents of the file specified by **fn** is sorted; otherwise, no further action is taken. Note that the terminating & places this entire command sequence in the background, not just the *sort* command.

All four command separators are rarely combined in a single command line as shown above. Other constructs in the shell programming language provide the same functions in a much more readable format. Also, the time required to design and debug lengthy sequences of commands is prohibitive. If space is a consideration, however, there is no more compact way of expressing a particular command sequence.

Command Grouping

The left and right parentheses ( ) and the left and right braces { } can be used to force several commands to be grouped together in a single unit.

Grouping With Parentheses

All commands enclosed in parentheses are passed to a new shell process to be executed. For purposes of discussion, *new process* refers to the process that is created to execute the parenthesized commands, and *calling process* refers to the process that reads the parenthesized commands, creates the new process, and passes the commands to the new process. Both processes have their own shell. For instance, in this example,

```
(who;ls)
```

the calling process creates a new process to which the *who* and *ls* commands are passed. The new process executes *who* and *ls* sequentially. The calling process waits for the new process to signal that the commands have been executed. When the signal is received, the new process dies, and the calling process reads the next command.

The & separator can be used to cause asynchronous execution in one or both of the processes. For example,

```
(cd $HOME; ed – newfile <script; rm script) &
```

The & in this example is seen only by the calling process. The new process sees

```
cd $HOME; ed – newfile <script; rm script
```

and executes each command sequentially. The calling process treats the entire parenthesized sequence as a command that is to be run asynchronously. Thus, the process number of the new process is reported, and the calling process proceeds to the next command, without waiting for the new process to signal that the job is completed. (Note that the *cd* command affects only the current working directory in the new process; the calling process's current working directory is unchanged.)

If the sequence is typed as follows,

```
(cd $HOME; ed – newfile <script &)
```

then only the new process is aware of the & separator. The calling process simply sees a command that is to be run sequentially, and waits for a signal from the new process before continuing. The new process sees

```
cd $HOME; ed – newfile <script &
```

Thus, *cd* is executed sequentially, and a separate process is created to execute *ed*. The new process reports the process number of the process that is executing ed, signals the calling process that the job is completed, and dies, even though *ed* is still executing asynchronously. The calling process then reads the next command.

Parentheses can be nested, with the result that more than one new process is created. For example,

```
test –f $fn && (ed – $fn <ed1 && (rm ed1; sort –o $fn $fn &)) &
```

The calling process sees

```
test –f $fn && ( . . . ) &
```

which tells it to execute the sequence asynchronously. Thus, the calling process creates another process (process A) to execute the sequence, reports the process number of that process, and reads the next command. As far as the calling process is concerned, the specified command sequence has been executed.

Meanwhile, process A sees

```
test –f $fn && ( . . . )
```

which is everything that the calling process saw, except that the final & is missing. Thus, process A begins sequential execution of its command sequence. It first executes the *test* command, and, if unsuccessful, signals the calling process and dies. Nothing more is done. If successful, however, process A creates a new process (process B) to execute everything within the first level of parentheses. Because there is no & separator, process A waits for a signal from process B that the job is done. When process B's signal is received, process A in turn signals the calling process that the job is done. Note that process A's signal is ignored by the calling process, regardless of whether or not process B is created, since the calling process has already proceeded on to the next command.

Process B comes to life and sees

```
ed – $fn <ed1 && ( . . . )
```

Thus, process B begins sequential execution of its command sequence. The *ed* command is executed and, if unsuccessful, process B signals process A that the job is done, and dies. If successful, process B creates a new process (process C) to execute everything between the second level of parentheses. Process B then waits for a signal from process C that the job is completed. When process C's signal is received, process B sends a signal to process A, which in turn signals the calling process.

Process C sees the following command sequence:

```
rm ed1; sort –o $fn $fn &
```

Process C first executes the *rm* command. When *rm* has terminated, process C creates another process (process D) to execute the *sort* command, reports process D's process number, signals process B that the job is done, and dies.

Finally, process D sees the following command:

```
sort –o $fn $fn
```

Process D sequentially executes the *sort* command. When *sort* has terminated, process D signals process C that the job is done, and dies. Process C, however, has already died, so process D's signal is ignored. In fact, process D probably begins its task after processes C, B, and A have already died!

Command sequences like the previous example are seldom, if ever, used. Designing a command sequence that performs exactly like you want it to perform takes a great deal of time. There are other constructs available that perform the same functions and are much easier to read and debug. However, command sequences like the previous example are ideally suited to those programmers who want their procedures to be as concise as possible.

Grouping With Braces

Braces are useful for grouping two or more commands together for the purpose of redirecting their combined input or output. Braces do not in themselves cause the creation of new processes. For example, in the following procedure,

```
{
date
ls
pr *
} >dircontents
```

date, *ls*, and *pr* are executed sequentially, and their output is collected in the file **dircontents**. Note that the braces do not create a new process to execute the commands. The braces are simply used to cause the I/O redirection to apply to all three commands.

The ;, &, &&, and || separators can be used within braces, and braces can be nested. They are most commonly used as shown in the previous example, with each brace appearing on a line by itself, and any number of valid commands and/or control-flow constructs appearing between them.

Control-Flow Constructs

With the introduction of control-flow constructs, procedures cease to be shell scripts, and become shell programs. Control-flow constructs are perhaps the most useful elements of the shell programming language, for they enable you to create powerful shell programs that incorporate conditional testing, branching, and looping.

The FOR Construct

The **for** construct enables you to execute a set of commands once for every new value assigned to a parameter. The **for** construct has the following syntax:

for *name* [**in** *wordlist*]
do *command-list*
done

Name is any valid parameter name. *Wordlist* contains a list of one or more values that are to be assigned to *name*. One at a time, a value from *wordlist* is assigned to *name*, and the list of commands in *command-list* is executed. Execution terminates when there are no more values left in *wordlist*. If the **in** clause is omitted, then *name* is assigned the value of each positional parameter that is set, and execution terminates when all positional parameters have been used.

Examples

```
for i in *.c
do
        cc $i
        mv a.out `basename $i .c`
done
```

This example is a variation of the **compile** shell script discussed earlier. The parameter **i** is assigned the name of each file in your current working directory that ends in ".c". That file is then compiled, and the resulting *a.out* file is renamed such that its name is the same as its corresponding source file with the ".c" removed. Execution ends when **i** has been assigned the names of all C source files in your current working directory.

```
for dir in /dev /usr /users /lib /etc /bin /tmp
do
        num = `ls $dir | wc –w`
        echo "$num files in $dir"
done
```

This example assigns each directory name to **dir**. The contents of each directory are listed, and the number of files is counted with *wc* and assigned to **num**. The number of files in each directory is then printed.

```
for i
do
        sort –d –o ${i}.srt $i
done
```

Since the **in** clause is missing, **i** is assigned the value of each positional parameter that is set on the command line. The result is that each file that is specified on the command line is sorted. The sorted version is placed in a file having the same name as the unsorted file, with a ".srt" appended to it.

Note that the *wordlist* part of the **for** construct can be almost anything. Some examples are

```
for i in $1
```

which enables you to specify *wordlist* from the command line, or

```
for file in $dir/[a–f]?[1–4].c
```

which causes *wordlist* to include all files in the directory specified by **dir** that begin with a lowercase letter in the range a through f, followed by any single character, followed by a digit in the range 1 through 4, followed by ".c".

Note that **do** or **done** is recognized only when following a new-line or semicolon.

The CASE Construct

The **case** construct enables you to execute a specific set of commands, depending on the value of a parameter. The **case** construct has the following syntax:

case $*name* **in**
        pattern1 [| *pattern2* ...]) *command-list1* **;;**
                .
                .
                .
esac

Name is a dereferenced parameter name. The *patterns* are strings of one or more literal characters and/or the special characters *, ?, [,], and \. The *command-list* is a list of one or more commands to be executed if one of the associated patterns matches the value of *name*. The last command in *command-list* must be terminated with a double semicolon (;;).

Examples

```
case $fn in
        *.c)    cc $fn
                mv a.out `basename $fn .c` ;;
        *.f)    fc $fn
                mv a.out `basename $fn .f` ;;
        *.p)    pc $fn
                mv a.out `basename $fn .p` ;;
        *)      echo "$fn: not a source file."
                exit 1 ;;
esac
```

This example compares the value of **fn** with each pattern listed, in the order in which the patterns are listed. If **fn** is a file name ending in ".c", then the commands associated with the "*.c" pattern are executed, and so on. The final pattern consisting of a single asterisk acts as a default condition. If none of the other patterns are matched, the commands associated with the asterisk are executed, since the asterisk matches anything. It is important that the asterisk be listed last, because any patterns following the asterisk are *never* matched. Note that the **case** construct terminates after a match is made.

```
for i
do case $i in
        -[dD]) echo "Please specify directory."
                read dir ;;
        -b|-r) rflag=y ;;
        *)      echo "$i: unknown option."
                exit 1 ;;
esac
done
```

This example illustrates how a **case** construct may be included within a **for** construct. This combination is very common, and is most often used to process options from the command line. This particular example accepts **-d**, **-D**, **-b**, and **-r** as valid options, and flags any others as invalid. The **case** construct is executed once for every positional parameter set on the command line. The first pattern matches **-d** or **-D**, both of which require the user to enter a directory name from his terminal (the *read* command is described under *Special Commands* in *sh*(1), in the HP-UX Reference manual). The second pattern matches either **-b** or **-r**, both of which set **rflag** equal to "y" (note that the second pattern could be written as "-[br]"). Finally, any other option prompts an error message, and the process is terminated.

The IF Construct

The **if** construct enables you to execute certain commands, depending on the result of one or more conditional tests. The **if** construct has the following syntax:

if *command-list1*
then *command-list2*
elif *command-list3*
then *command-list4*
.
.
.
else *command-listn*
fi

Only the **if**, **then**, and terminating **fi** are necessary; all **elif** sections and the **else** section are optional. Each *command-list* is a list of one or more commands. The list associated with **if** is executed first. If the last command of the list is successful, the list associated with the first **then** is executed, and the construct is terminated. If the list associated with **if** is unsuccessful, the list associated with the next **elif** is executed. If that **elif**'s list is successful, the list associated with the next **then** is executed, and so on. If all **elif** lists are unsuccessful, the list following **else** is executed, and the **if** construct terminates.

Examples

```
if [ –f –w "$fn" ]
then
        ed – $fn <script
fi
```

This example shows the **if** construct in its simplest form. The square brackets are the alternate syntax for the *test* command, and are equivalent to

```
test –f –w "$fn"
```

Thus, if the file specified by **fn** is both an ordinary file and writable, then it is edited according to the *ed* commands in **script**. Otherwise, no action is taken. Note that this **if** construct is equivalent to

```
test –f –w "$fn" && ed – $fn <script
```

```
if [ –f –r /users/kb/$fn ]
then
        diff /users/kb/$fn $fn >diffile
elif [ –f –r /users/bill/$fn ]
then
        diff /users/bill/$fn $fn >diffile
elif [ –f –r /users/fred/$fn ]
then
        diff /users/fred/$fn $fn >diffile
else
        echo "Can't find $fn for comparison."
fi
```

This example shows all the parts of an **if** construct. Here, the directories */users/kb*, */users/bill*, and */users/fred* are searched for the file specified by **fn**. If it is found, it is compared to a file with the same name in your current working directory. The output from *diff* is redirected into a file called **diffile**. The **else** clause functions as a default; if all other tests fail, the *echo* command is executed.

The WHILE Construct

The **while** construct repeatedly executes a list of commands and, if the last command in the list is successful, executes a second list of commands. The **while** construct has the following syntax:

while *command-list1*
do *command-list2*
done

Command-list1 is a list of one or more commands that is repeatedly executed. If the last command in this list executes successfully, the commands in *command-list2* are executed. The loop terminates when the last command in *command-list1* executes unsuccessfully.

Example

```
while [ –n "$1" ]
do
        sort –d –o $1 $1
        ed – $1 <edscript
        pr –f $1 >/dev/lp
        shift
done
```

This example operates on the positional parameter **$1**. The **while** loop continues as long as the value of **$1** has a non-zero length. First, the file name specified by **$1** is sorted. Then, it is edited according to the *ed* commands in **edscript**, and printed on the system's line printer. The *shift* command moves the value of **$2** to **$1**, **$3** to **$2**, **$4** to **$3**, and so on. Thus, different files are sorted, edited, and printed each time through the loop, even though the parameter name stays the same. The loop terminates when a null value is shifted in for **$1**.

The UNTIL Construct

The **until** construct repeatedly executes a list of commands and, if the last command in the list is unsuccessful, executes a second list of commands. The **until** construct terminates when the last command in the first command list executes successfully. Thus, the **while** and **until** constructs differ only in the condition required to terminate the loop. The **until** construct has the following syntax:

until *command-list1*
do *command-list2*
done

Command-list1 is a list of one or more commands that is repeatedly executed. If the last command in *command-list1* executes unsuccessfully, the commands in *command-list2* are executed. The **until** construct terminates when the last command in *command-list1* executes successfully.

Example

```
until who | grep fred >/dev/null
do
        sleep 300
done
write fred <fredletter
```

This example checks to see if Fred is logged in. If not, the process "sleeps" for five minutes, and checks again. This continues until Fred finally logs in, at which time the **until** construct terminates, and the message in **fredletter** is sent to Fred via the *write* command. Note that the output from *grep* is redirected to */dev/null*, which essentially discards the data into the system's "bit bucket". A shell program like this can be executed in a background process to ensure that a particular user gets an important message as soon as he logs in.

An Example Shell Program

The following is a shell program that is used to print files on the system's line printer, */dev/lp*.

```
rd=n
range=*
dir=`pwd`
days=
#
# Parse options.
#
while [ -n "$1" ]
do case $1 in
        -c)     rd=y            # raw dump
                shift ;;
        -d)     shift           # directory name
                dir=$1
                shift ;;
        -f)     shift           # last-modified time
                days=$1
                shift ;;
        -r)     shift           # files to print
                range=$1
                shift ;;
        *)      echo "$1: unrecognized option."
                exit 1 ;;
esac
done
#
# Move to specified directory.
#
cd $dir
#
# If -f specified, move all affected files.
#
if [ -n "$days" ]
then
        mkdir ../temp
        find . -mtime $days -exec mv {} ../temp \;
fi
#
# Begin printing.
#
for i in $range
```

```
do case $rd in
        n)      pr –f –r $i >/dev/lp ;;
        y)      cat $i >/dev/lp
                echo "\n\n" >/dev/lp ;;
esac
done
#
# Printing is done. Clean-up time.
#
test –n "$days" && (mv ../temp/* .; rmdir ../temp)
exit 0
```

This shell program, called **print**, accepts four options:

the **–c** option, which specifies that the contents of the files are to be printed with no formatting (i.e. a "raw dump"). The *cat* command is used for this. The **–c** option requires no argument. If the **–c** option is not specified, then the contents of the files are printed out with a heading and page numbers. The *pr* command is used for this.

the **–d** option, which implies that the argument to follow specifies the name of the directory containing the files to be printed. If the **–d** option is not specified, the user's current working directory is used.

the **–f** option, which implies that the argument to follow specifies an argument for the *find* command. If **–f** is specified, a temporary directory called **temp** is created in the parent directory, and the *find* command is used to move all files of a certain modification date to **temp**, thus excluding them from the printing. The **–f** argument can have the following three forms:

+n exclude those files modified more than *n* days ago;

–n exclude those files modified less than *n* days ago;

n exclude those files modified exactly *n* days ago.

This argument is combined with the **–mtime** option of the *find* command (see *find*(1) in the HP-UX Reference manual). If the **–f** option is not specified, no files are excluded on the basis of modification date.

the **–r** option, which implies that the argument to follow specifies a string of literal and/or special characters. The string is used in the **in** clause of a **for** construct to print a subset of the files in the directory. If the **–r** option is not specified, an asterisk is used, causing all files to be printed.

The following examples show some of the valid ways to invoke **print**:

```
print –c
```

causes all the files in the current working directory to be printed in "raw" form using *cat*.

```
print –d /users/bill/Cprogs –r \[a–f\]\* –f +3 –c
```

does several things. First, the current working directory is changed to */users/bill/Cprogs*. Then, the directory */users/bill/temp* is created, and all files in **Cprogs** modified more than 3 days ago are moved to **temp**, thus excluding them from the printing. Finally, all files in **Cprogs** that begin with a lower-case letter in the range a through f are printed in "raw" form using *cat*. Note that the special characters in **[a–f]*** must be quoted to prohibit the shell from expanding the pattern, and replacing it with the files that match it in the current working directory. When the printing is done, all the files in **temp** are moved back to **Cprogs**, and **temp** is removed.

```
print –r thesis –d /users/bill/school
```

prints the single file **thesis** in the directory */users/bill/school*. The *pr* command is used to produce a formatted printing.

Using the **case** construct to parse options enables you to specify them in any order on the command line. The only requirements are that the options be immediately followed by their implied arguments, and that all options and arguments be delimited by spaces.

For More Information

The *sh*(1) entry in the HP-UX Reference manual functions as a comprehensive, though somewhat cryptic, reference for the shell programming language. Some topics are not covered in this chapter because of infrequent use, or because one example is sufficient to give guidance in several areas. Most of the omitted topics are related to the shell's special commands, which are discussed under *Special Commands* in the *sh*(1) entry. You should read this section thoroughly to familiarize yourself with the many commands that are built directly into the shell.

Many HP-UX commands are actually shell programs. The HP-UX Reference manual specifies which commands are shell programs under the appropriate manual entries. The following is a list of some of them:

```
/etc/rc
/etc/whodo
/etc/mkdev
```

It is helpful to examine the contents of these commands to see how the shell programming language is used. Since these files contain ASCII data, you can print them out using *cat*, provided your system administrator has assigned permissions to these files that enable you to do so.

Table of Contents

The C Shell (csh)

The C Shell (csh)

Introduction

The C Shell is an HP-UX command language interpreter and a high-level programming language. It is used to translate command lines into actions, such as running programs, moving between directories, and controlling the flow of information between programs. It also has the following features:

- a command history buffer and associated history substitution facility. Recently executed commands may be modified and re-executed with ease.
- an aliasing facility. Useful statements can be referenced with a short alias.
- an extensive, C-like command and control capability.

For further information about HP-UX shells, read "Bourne Shell Programming" in the *HP-UX Concepts and Tutorials*.

This document uses the following conventions:

- All examples assume the *C Shell* prompt has been changed to show the current command event number by entering the following *set* command in either *$HOME/.cshrc* or *$HOME/.login*:

  ```
  set prompt = "[\!] % "
  ```

  Your resulting prompt appears as:

  ```
  [23] %
  ```

- `Dot-matrix` font is used to show what you should see on your screen. For example, to activate the C shell type `csh`. Terminating command sequences with ENTER or RESULT is assumed.
- **Bold** font is used to emphasize important information and key words.
- *Italic* font is used to indicate commands and to refer to words illustrated in commands.

The System You Received

Your HP-UX system has both the Bourne Shell and the C Shell as command interpreters. When your system is shipped to you, the Bourne shell is the default shell when you login.

The default shell prompt for the Bourne Shell is the dollar sign ($) symbol. When C Shell is made the active shell, the prompt becomes the percent (%) symbol. The prompts for both shells can be changed to any symbols you want, but more about that later.

1 This software and documentation is based in part on the fourth Berkeley Software distribution under license from the regents of the University of California. We acknowledge the following individuals and institutions for their role in its development: William Joy.

Shell Startup and Termination

Running csh From the Bourne Shell

The name of the C Shell program is *csh*; therefore, to run the C Shell from the Bourne Shell, just type in:

```
csh
```

Your prompt changes to the C Shell prompt, **%** (provided you have not redefined the prompt).

Making the C Shell Your Login Shell

To make the C Shell your default login shell instead of the Bourne Shell, type in:

```
chsh login_name /bin/csh
```

The argument `login_name` is your login name.

The command *chsh* changes your login shell by modifying your entry in */etc/passwd* (see your *HP-UX System Administrator Manual*). When you change shells, the new shell is your default login shell until you use *chsh* again. *Chsh* changes your login shell, but **not** your current working shell. You must log out and then log in again in order for the new shell to be executed.

The full path name of the C Shell program is */bin/csh*, while the Bourne Shell program is */bin/sh*. If you want to make the Bourne shell your login shell, just type in:

```
chsh login_name /bin/sh
```

If no shell path name is specified on the *chsh* command line, the login shell is set to */bin/sh*.

Terminating a C Shell

The main control for how you terminate a C shell is the value of the shell boolean variable **ignoreeof**. To see the condition of this variable, execute *set* without arguments. This lists all shell variables and their current values. Boolean variables are listed if they are set. For example:

```
[25] % set
argv    () autologout     15
cwd /users/login_name
history         15
home /users/login_name
ignoreeof                 <-------- ignoreeof is set for this example
noclobber prompt [!] %
shell /bin/csh
status          0
term hp2622
path (/bin /usr/bin /lbin /usr/lbin /etc/users/login_name . )
[26] %
```

If **ignoreeof** is set, you must use either `exit` or `logout` to terminate the C Shell. If it is not set, you must use CTRL-D.

Returning to a Parent Shell

If you executed *csh* from a Bourne Shell or another C Shell, when you terminate the current C Shell process you return to the parent shell process. If **ignoreeof** is set, return to the parent process by typing:

```
exit
```

If the parent shell process was the Bourne Shell, you should now see your Bourne Shell prompt. If you use CTRL-D to exit the C Shell, the error message:

```
Use "exit" to leave csh.
```

is printed.

If you executed *csh* from the Bourne Shell without setting **ignoreeof**, use CTRL-D to exit the shell.

Logging Off Your System

If the C Shell is your login shell and the shell variable **ignoreeof** is not set, type in:

CTRL-D

to log off your system.

If **ignoreeof** is set, use either:

```
exit
```

or:

```
logout
```

to log off the system. In this case, CTRL-D does **not** log you off your system. If you do enter a CTRL-D, your system responds with:

```
Use "logout" to logout.
```

What Happens When *csh* Is Executed

There are two shell script files that *csh* looks for when it is executed:

.cshrc	Whenever a C Shell process is started, whether as your login shell or from another shell, this shell script file is executed, if it exists in your home (login) directory.
.login	If you log into your system and the C shell is your login shell, after executing *.cshrc* a shell script called *.login* is also executed, if it exists in your home directory.

Neither of these files are required by the C Shell; however, they provide a convient method of customizing your shell environment.

Setting Environment and Shell Variables

There are two kinds of variables that you can set in the *.cshrc* and *.login* files:

environment variables	variables that are global to your login shell process and any processes spawned by the shell process. These are represented by uppercase letters.
shell variables	variables that are local to a shell process and are not inherited by spawned processes. These are represented by lowercase letters.

Normally, environment variables are set with the *setenv* command and shell variables are set with the *set* command. However, three of the most commonly used environment variables USER, TERM, and PATH are automatically imported to and exported from three corresponding shell variables: user, term, and path. Thus, if you execute:

```
set path=(/bin /usr/bin)
```

the value of the environment variable PATH also becomes "/bin:/usr/bin" (Note the difference in the syntax of the two variables.)

The *set* and *setenv* commands can either be used interactively at a terminal or they can be placed in either the *.cshrc* or *.login* files.

The .cshrc Shell Script File

If this shell script file exists in your home directory, all C Shells started during your session execute the commands contained in this file. The C Shell uses the information contained in this shell script file to set variables and parameters that are local to the shell process.

Since every C Shell created executes this file, it is customary to use it to set shell variables by placing *set* command lines in it. If you do not have a *.cshrc* file, HP-UX spawns a C Shell process with default values for these variables.

To see what shell variables are currently set, execute *set*:

```
[25] % set
argv    ()
autologout      15
cwd /users/login_name
history         15
home /users/login_name
ignoreeof
noclobber
prompt [!] %
shell /bin/csh
status          0
term hp2622
path (/bin /usr/bin /lbin /usr/lbin /etc/users/login_name .)
[26] %
```

Some of the commands you might want to put in this file, and their meanings are shown below. Use your login name for the variable *login_name* shown.

Command	Meaning
set ignoreeof	Traps (CTRL)-(D)'s to avoid accidental system log off. Use the logout command.
set prompt = "[\!] %"	This command causes your C Shell prompt to be the current event number in square brackets followed by a percent sign. This is very helpful when using the command history buffer.
set history = 15	Sequentially keeps a buffer of your last (15 in this case) events.
set savehist = 15	This command saves the last (15 in this case) events when you log off your system. When you log back onto your system, the event history is restored.
set noclobber	This command stops the C Shell from overwriting and destroying the information in an existing file.

Note that you can suppress the execution of *.cshrc* by using *csh*'s **-f** option:

```
csh  -f
```

The .login Shell Script File

If the C Shell is your login shell and a shell script file **.login** exists in your home directory, the file is executed whenever you login (after *.cshrc*). It is customary to set environment variables in this file by including *setenv* command lines. Some of the commands you might want to put in this file and their meanings are shown below. The variable $LOGNAME refers to your login name.

Command	Meaning
setenv TERM hp2622	Sets the system variable TERM to recognize the HP 2622 as your terminal.
setenv TZ MST7MDT	This command sets the time zone variable. The example changes from Mountain Standard Time to Mountain Daylight Time.
setenv PATH /bin:/usr/bin:/lbin:/usr/lbin:/etc:/users/$LOGNAME	This command sets the the search pattern the system uses for finding commands.
set mail=/usr/mail/$LOGNAME	Required to set notification of mail for HP-UX.
/bin/mail -e	Message to notify you that you have mail.
if ($status = = 0) then echo "You have mail"	
alias h history	Make the character h an alias for your command history file.
alias bye logout	For some, bye is easier to remember than logout as a session termination order.
cat /etc/motd	Get the message of the day from the system.
tabs -Thp	Set tabs on HP terminals.
news -n	Get the names of the news items from the system.
news more	*More* the news.

What Happens When You Log Out

If the C Shell is your login shell and the variable **ignoreeof** is set, the system looks for a shell script file called ***.logout*** in your home directory to execute whenever you log out. You log out of your system by typing in:

```
logoff
```

or

```
exit
```

These commands read your shell script file *.logout* and execute any commands found there. Commands typically found in this shell script file are shown below, along with their meaning.

Command	Meaning
echo ' ' echo '** You are logged out now.' echo ' '	Print logout message to your standard output device.
date	Prints your log out date and time.
sync	Put all information stored in all buffers onto the system disc.

The C Shell Command History

The C Shell always maintains a Command History Buffer capable of holding your last command. By setting the **history** shell variable to some integer value, say 20, the history buffer can hold many (in this case 20) commands. These saved commands, sometimes called *events*, can be accessed in many useful ways. Since these commands can be quite complex, we use the term *event* to refer to commands stored in the Command History Buffer from now on. A buffer size of 10 to 20 is about right for most situations.

You can take advantage of this history buffer by using C Shell's history substitution facility. This facility allows you to use words from previous command events as portions of new commands, repeat command events, repeat arguments of a previous command in the current command event, and fix spelling mistakes in a previous event.

History substitutions begin with an exclamation point (!). They can begin anywhere in an event; however, they can not be nested.

To see how this all works, enter the following lines in either your *.cshrc* or *.login* files in your home directory.

```
set history = 15
set savehist = 15
set prompt = "[\!] % "
```

These statements:

- create a fifteen-event Command History Buffer;
- save the last 15 events in your command history buffer when you log off the system and restore them the next time you log on the system;
- cause your C Shell prompt to display the event number of each event.

All of the capabilities that you are about to see work without this special prompt, but history substitutions are easier to handle if your prompt indicates the event number of each event executed.

To see what is in your history buffer, type in the command `history` without arguments. Your display may appear as shown below:

```
[6] % history
      1  ls -als
      2  cat junk
      3  pr memo > /dev/lpr&
      4  mail jd < memo
      5  vi .cshrc
      6  history
[7] %
```

Re-executing Events

You can re-execute a previous event by using the history substitution facility to reference the event in your history buffer. An event can be referenced by:

- its event number;
- its location relative to the current event;
- the text of the event.

As a special case, the immediately previous event can be referenced by double exclamation points (!!). (Actually, the first exclamation point activates the history substitution facility and the second references the previous event.)

Referencing by Event Number

One way to re-execute an event stored in the history buffer is to reference its event number. For example:

```
[7] % !2
cat junk

This is the contents of the file junk.
[8] %
```

re-executes event number *2*. Notice that the event to be re-executed is echoed on the terminal before it is actually executed, allowing you to verify that you are referencing the correct event.

Referencing by Relative Location

Another way to re-execute an event is to reference its postion in the history buffer relative to the current event. For example:

```
[8] % !-4
mail jd < memo
[9] %
```

executes event four (8 – 4 = 4), in this case sending a memo to user jd again.

Referencing by Event Text

You can re-execute an event by entering the first few characters of the event's command line. If you have previously executed *history*, you can see what the current history buffer contains using:

```
[9] % !h
```

The history substitution facility searches backward through the buffer until it finds an event whose command line begins with the letter "h". When it finds the event with the *history* command line, it re-executes it.

```
[9] % !h
     1  ls -als
     2  cat junk
     3  pr memo > /dev/lpr&
     4  mail jd < memo
     5  vi .cshrc
     6  history
     7  vi memo
     8  mail jd < memo
     9  history
[10] %
```

Reuse of Command Arguments

The history substitution facility allows you to use parts of previous commands as building blocks of new commands. Each command argument in a command event is numbered. To reference a command argument, specify the event with one of the methods described above in "Re-executing Events" and then use a colon (:) followed by the argument's position number.

The first argument, usually the command, is argument number zero (0). The second argument is argument number one (1), etc. The last argument is given the special reference of the dollar sign ($). The second argument, usually the first argument after a command word is given the special reference of the caret (^). To see how this works, begin with the example shown below.

```
[10] % nroff -man csh.1 | col -l > /dev/lp &
```

To see what the last argument in this event is, type in:

```
[11] % !10:$
&
[12] %
```

The last argument in event 10 is the ampersand (&). The history substitution facility extends the normal meaning of 'argument' to include important metacharacters. The argument specified by a caret (^) is `-man`. To verify this, type in:

```
[12] % echo !10:^
echo -man
-man
[13] %
```

The referenced argument can be made part of another command. A range of event arguments can also be specified by using a dash (-) to separate the range endpoints. For example:

```
[13] % echo !10:3-$
echo | col -l > /dev/lp &
[1] 18634 18635
[14] %
```

Note that the example generated a new C Shell with the job number `[1]` and two process IDs `18634 18635`. This new shell is called a *background process.* The arguments `col -l` are printed on the line printer (/dev/lp). Jobs and job numbers are discussed later in this tutorial.

If you want to reuse all of the arguments of an event that follow an initial command, you can use an asterisk (*):

```
[14] % mkdir /users/bill /users/pete /users/mary
[15] % rmdir !14:*
```

Modifying Previous Events

As you use the C Shell, you will find that re-executing a previous event with minor modifications will save you typing. To modify and then re-execute a previous event, you should form a command line in the following manner:

1. Begin by referencing the previous event by either its event number, its location relative to the current event, or by text contained in the event's command line. See the previous section "Re-executing Commands". This reference always begins with a "!".
2. Optionally, you can specify particular words on the chosen event's command line. See the previous section "Reuse of Command Arguments". This specification is usually separated from the event reference (step 1) with a ":".
3. Finally, specify how you want the event changed using a modifier from the table below. If you skipped step 2, the modifier applies to the entire event. If you specified particular words in the event in step 2, the modifier applies to those words. The modifiers are always prefixed with a ":". You can use several modifiers in sequence, separating each of them with a ":".

The following is a list of modifiers that you can use to modify your re-executed events and their command arguments (step 3).

Modifier	Definition	Effect
s/old/new	substitute	Substitute *old* for *new*. Any character may be used as the delimiters between the substitution strings. An ampersand (&) in the new string is replaced with the whole old string. Note that this only effects the first occurrence of *old* on an event's command line. Use the "gs" combination if you want the effect to be global.
g	global	Use in combination with another modifier to make the effect of the modifier global for an event's entire command line. (e.g. *gs/old/new* replaces all occurances of *old* with *new*). Note that only one substitution can can made per argument in an event. For example, the effect of *gs/joe/mary* on the path name `/users/joe/joe_file` would be to make the following modification: `/users/mary/joe_file.`
h	head	Use only the directory path name of a specified argument in an event by by removing its final path name component (i.e. use only the path name's head).
p	print	Print the event specified, but do not execute it. This is useful if you just want to verify what a particular event was. For example: `[10] % !3:p` prints event number 3 on your terminal without executing it.
q	quote	Quote the modifications so that no further modifications can take place.
r	root	Remove the file name extension. If a file name's tail ends with a "." followed by one or more characters, the "." and the characters that follow it are dropped. (i.e. remove *.o* from *file.o* leaving *file*).
t	tail	Remove all elements of a path name except the last element (i.e. the path name's tail).
&	repeat	Do the previous substitution again. The history substitution facility keeps track of the last substitution you performed with the s modifier, thus allowing you to easily perform the same change on different events that you want to re-execute.

For example, suppose you enter the following command:

```
[14] % car /users/jack/documents/memo
car: Command not found.
[15] %
```

The *cat* command in event 14 was misspelled. To fix this, type:

```
[15] % !14:s/car/cat
cat /users/jack/documents/memo

This is a test.
[16] %
```

This executes the command correctly, without retyping the whole path name of the file that you want to look at. To look at a file called "list" in the same directory you can now enter:

```
[16] % !15:s/memo/list
cat /users/jack/documents/list

apples
oranges
bananas
pineapples
strawberries
plums
[17] %
```

Now, suppose that you want to move to the directory containing the files that you just looked at. You can do this with:

```
[17] % cd !!:^:h
```

This is quite a complex command, but typing is still saved. The double exclamation marks reference the immediately previous event, the caret (^) argument specifier selects the second word on the event's command line and the h modifier indicates that only the head of the specified word is used ("/users/jack/documents").

To return to your home directory, type in:

```
[18] % cd
[19] %
```

An Example

To see how this all comes together, the following example shows how you can use the C shell to modify, compile, and execute a C program called "bug.c".

```
[22] % cat bug.c          Prompt set to show current comand number

main()
{
    printf("hello);
}

[23] % cc !!:$            Compile file named in last event.
cc bug.c

"bug.c",line 4: newline in string or char constant
"bug.c",line 5: syntax error
```

```
[24] % ed !!:$                    Edit file named in last event.
ed bug.c

29
4s/);/"&/p
    printf("hello");
w
30
q

[24] % !c                         Do last event that began with
cc bug.c                          small c character.

[25] % a.out

hello [26] % !e                   Not right, run ed again.
ed bug.c

30
4s/lo/lo\\n/p
    printf("hello\n");
w
32
q

[26] % !c -o bug                  Do the last c event and append
cc bug.c -o bug                   the -o option and word "bug".

[27] % size a.out bug             Execute size.

a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b

[28] % ls -l !!:*                 Prefix last event's arguments
ls -l a.out bug                   with an ls -l command.

-rwxr-xr-x 1 derald    3932 Feb 29 09:00 a.out
-rwxr-xr-x 1 derald    3932 Feb 29 09:01 bug

[29] % bug                        Execute bug.

hello

[30] % num bug.c | spp
spp: Command not found.

[31] % !!:$:s/spp/ssp             Correct spelling in last event
num bug.c | ssp                   from "spp" to "ssp".

  1 main()
  3 {
  4     printf("hello\n");
  5 }

[32] % !! > /dev/lp               Execute last executable event
num bug.c | ssp > /dev/lp         (!!) and send to line printer.

[33] %
```

Alias

The C Shell allows you to customize commands with an alias facility. This facility allows you to make standard commands do non-standard functions and allows you to define new commands. The alias facility is similar to a macro facility in that when an alias is detected, it is replaced by the alias definition.

To see what aliases exist, enter *alias* without arguments. For example:

```
[41] % alias
cd      cd !* ; ls
h       history
print   pr !* | col -l > /dev/lp
w       who ; echo "You are ......." ; who am i
dir     (ls -als)
```

You can create the above aliases either interactively or by placing alias commands in a shell script.

Alias Substitution

After a command line is scanned, it is parsed into distinct command arguments The first word of each command, left-to-right, is checked to see if it has an alias. If it does, the alias string replaces the aliased word. The process begins again. The substituted alias string is marked to avoid looping and does not modify the rest of the command word's arguments.

Alias uses the same substitution scheme as the history facility uses. A single exclamation point represents the current event and is preceded by a backslash so that the shell does **not** interpret it, but passes it on to alias. History modifiers also work in alias statements.

Restrictions Using Alias

There are two basic restrictions that you must adhere to when you use the alias facility:

- You cannot alias the *alias* command. If you do, an error message is generated.
- To prevent the formation of an alias loop, the C Shell allows a particular alias string to appear only once in another alias definition. Also, the command that is being aliased can appear only once in its own alias definition. For example:

  ```
  [32] % alias ls ls
  ```

  works, but:

  ```
  [33] % alias ls 'ls ; ls'
  ```

  doesn't. If you try to execute *ls* after it has been aliased with event 33 above, you see:

  ```
  [34] % ls
  Alias loop.
  [35] %
  ```

Aliasing Existing Commands

You can alias HP-UX commands so that they perform non-standard functions. Suppose you like to get a directory listing whenever you change directories. Do this by aliasing *cd* in the following way:

```
[42] % alias cd 'cd \!* ; ls'
```

Using a command statement in the alias of the command is acceptable.

We enclosed the entire alias definition in single quote characters to prevent most substitutions from occuring and the semicolon character from being interpreted as a metacharacter.

We place the backslaah (\) in front of the exclaimation point to prevent the exclaimation point from being interpreted as a history substitution. The result of the string `\!*` is that it substitutes the entire argument list to the pre-aliasing cd command.

The semicolon separates commands to be executed sequentially.

Creating Custom Commands

The C Shell's alias facility also allows you to create new commands. Suppose you want to get a long, alphabetical listing of your current working directory showing the size of each file. You could type in:

```
ls -als
```

each time, but you want to make up your own command

```
dir
```

and get the same results. To do this, type in:

```
alias dir ls -als
```

Unaliasing an Alias

The following aliases exist:

```
[41] % alias
cd        cd !* ; ls
h         history
print     pr !* | col -l > /dev/lp
w         who ; echo "You are ......." ; who am i
dir       (ls -als)
```

To *unalias* the change directory command (*cd*) type in:

```
[42] % unalias cd
[42] % alias
   h         history
   print     pr !* | col -l > /dev/lp
   w         who ; echo "You are ......." ; who am i
   dir       (ls -als)
```

Command Substitution

A command enclosed in single quote characters is replaced, just before filenames are expanded, by the output from that command. Thus, it is possible to use:

```
[43] % set pwd='pwd'
```

to save the current directory in the variable `pwd`. You can now print the value of the pwd variable with:

```
[44] % echo $pwd
/users/joe/documents
[45] %
```

Command substitution also provides a way of generating arguments for other commands. For example:

```
ex 'grep -1 TRACE*.c'
```

runs the editor `ex`, supplying as arguments those files whose names end in `.c` and beginning with the string `TRACE`.

Metacharacters in C Shell

The C Shell recognizes a number of characters as having special meaning. We say that these special characters have syntactic and semantic meaning to the shell. These special characters are called **metacharacters**.

Metacharacters effect the C Shell only as the characters are read into the shell. The C Shell displays an **&** as a prompt when reading. The metacharacters recognized by the shell are not recognized when it is running another program, such as *vi* or *mail*. Thus, don't worry about metacharacters in a letter you are sending via *mail* or when you are typing in text or data with *vi*.

Syntactic Metacharacters

;	separates commands to be executed sequentially.
|	separates commands in a pipeline. Commands in a pipeling execute sequentially with the output of one command being fed as input to the next command.
( )	isolates commands separated by ";" or pipelines so that the result appears as a simple command. This allows pipelines enclosed in parentheses to themselves be components of another pipeline. Parenthesized commands are always executed in a subshell.
&	indicates commands to be executed as a background process. For example, to print the file `letter` as a background process on the system printer `/dev/lp`, type in: `cat letter > /dev/lp &`
||	separates commands or pipelines indicating that the second is performed only if the first fails.
&&	separates commands or pipelines indicating that the second is performed only if the first succeeds.

Filename Metacharacters

If a file's name contains one of the following metacharacters, then the name is a candidate for file name substitution. There are two basic types of file name metacharacters. The first type indicates that the name is a pattern which the shell should replace with all of the file names that match it. The second type indicates that the name is an abbreviation which the shell should expand to the appropriate file name.

The metacharacters that indicate patterns are:

? expansion character matching any single character when specifying a filename. For example, to collect the files filea.o, fileb.o and filec.o in the file named total.o, type in:

```
cat file?.o > total.o
```

* expansion character matching any sequence of characters, including the empty sequence. To remove all files beginning with the word `old`, type in:

```
rm old*
```

[] expansion matching of any one of the characters enclosed or range of characters separated with a dash (-) listed within the brackets. For example, to list all the files with the same root name (file), type in:

```
ls file.[a-z]
```

This could produce:

```
file.o  file.p
```

The metacharacters that indicate abbreviations are:

{} abbreviates a set of words which have common parts. Curly brackets are seldom needed. For example, the files `list`, `last` and `lost` can be listed with:

```
ls l{aio}st
```

~ gives access to the path name from the root to the home directory of a user. The syntax is the tilde followed by the login name of the desired user. You must have *superuser* capability to use this metacharacter. If a ~ appears in the middle of a word or is followed by a character other than a letter or a /, it is not interpreted as a metacharacter and is left undisturbed.

The following character also has a special meaning in file names:

/ separates components of a file's path name. For example, `/bin/csh` is the path name to the file `csh`. The first slash in a path name or a lone slash references the system's root directory.

Quotation Metacharacters

\ prevents meta-meaning of the following single character. For example, typing:

```
ls *
```

prints a list of all of your files and directories in your current directory. Typing in:

```
ls \*
```

prints:

```
* not found
```

' ' prevents meta-meaning of a group of characters. For example, if you set a variable to a command string, the command string may contain metacharacters. When you reference the variable, the metacharacters could be processed. Using single quotes inhibits the processing of any metacharacters in the string.

" " prevents meta-meaning of a group of characters, but allows variable and command expansion. This is like using a single-quote, except that only the metacharacters are left unprocessed.

Input/Output Metacharacters

<*name* indicates redirected input from *name*. For example,

```
mail boss < memo &
```

sends the file `memo` to `boss`.

>*name* indicates redirected output to *name*. For example:

```
grep -vn file1 file1 > numbered.file1
```

puts a copy of `file1`, with each line numbered, in the new file `numbered.file1`. This metacharacter causes the target file to be overwritten (unless the variable **noclobber** is set).

>&*name* direct the diagnostic output along with the standard output into the file *name*.

>!*name* redirect output with over-write of target file. This is used when **noclobber** is set. You can also combine the effect of >& and >! by using >&!.

>>*name* redirect output by appending it to the end of *name*. If the file *name* does not exist and the variable **noclobber** is set, and error occurs.

>>&*name* append diagnostic output along with the standard output to the end of *name*.

>>!*name* Acts like >> except in the case where *name* does not exist and the **noclobber** variable is set. In such a situation, >>! creates *name* and no error occurs. You can also combine the effect of >>& and >>! by using >>&!.

<<*word* read the shell input up to a line which is identical to *word*. *Word* is not subjected to variable, file name or command substitution, and each input line is searched for *word* before any substitutions are performed on it. Files processed in this manner are commonly called **here documents**.

Note that if you do not want meta-substitutions performed on the lines in the shell script, enclose *word* in single quotes (<<'*word*').

| forms a pipeline between two processes. A pipeline causes the output of the process before the "|" to be the input of the process after the "|".

|& forms a pipeline between two processes that sends diagnostic output as well as standard output from the first process as input to the second process.

Expansion/Substitution Metacharacters

$ indicates variable substitution. For example,

```
set M1 = /usr/man/man3
cd $M1
```

The path name is assigned to the variable M1. To use the variable, preceed the variable name with a dollar sign.

Note that you could also execute `cd M1`. The C Shell looks for a directory called "M1" and when it cannot find it it looks for a variable with that name. When the variable is found, its value is used as an argument to *cd*.

! indicates history substitution. See the History discussion in this tutorial.

: precedes substitution modifiers. See the History discussion in this tutorial.

? used in special forms of history substitution indicating command substitution.

Other Metacharacters

# indicates shell comments and begins scratch file names. Must be the first character in a shell script to be executed by the C Shell.

% prefixes job name specifications. For example:

```
[56] % cc test.c >& test &
[1] 3265
[57] % kill %1
[58] %
```

Event 57 kills the background process with the job number 1.

Using Metacharacters as Normal Characters

Metacharacters pose a problem in that we cannot use them directly as parts of command arguments. Thus the command:

```
echo *
```

does not echo the character *. It either echoes a sorted list of file names in the current working directory, or prints the message `No match` if there are no files in the working directory.

To change metacharacters into normal characters, put them in single quotes. The command:

```
echo '*'
```

echoes an asterisk to your display.

There are three metacharacters that cannot be "escaped" with single quotes. They are:

- the exclamation mark (!)
- the backslash (\)
- the single-quote (')

The backslash must be used to cancel the special shell meaning of these metacharacters. Thus:

```
echo \'\!\\
```

prints:

```
'!\
```

These two mechanisms, the single-quote and the backslash, let you use any printable character in a shell command. They can be combined, as in:

```
echo \''*'
```

which prints:

```
'*
```

The backslash (\) escapes the first single-quote (') and the astrisk (*) is enclosed between single-quotes. The result is a single-quote and astrisk.

Built-in Shell Variables

The shell maintains a set of variables. Shell variables may be assigned values by the *set* command. Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell. The following variables are supported by HP-UX and the C Shell.

The $argv Variable

This variable contains the command line arguments from the calling shell. These are position numbered so that $argv[0] is the command, $argv[1] is the first argument that follows it, and so on.

The $autologout Variable

This variable is used to automatically log you off the system if you do not use the system for a specified amount of time. For example,

```
set autologout = 60
```

will automatically log you off the system if you do not use the system for an hour (60 minutes).

To disable **autologout**, set it to zero (0) time. For example:

```
set autologout = 0
```

or

```
unset autologout
```

The $cwd Variable

The **cwd** variable contains the pathname to your current working directory. This variable is automatically changed with each cd (Change Directory) command. At log-on, the default for this variable is the directory in the system variable $HOME.

The $home Variable

The **home** variable contains the path name to your home directory. The default value for this variable is specified in the system file */etc/passwd*. (See passwd(5))

The ignoreeof Boolean Variable

The variable **ignoreeof** is a boolean that indicates if CTRL-D is allowed to log you off the system. If *set*,

```
set ignoreeof
```

you log off the system with *logout*. If *unset*,

```
unset ignoreeof
```

then the CTRL-D will log you off. The default is *set*.

The $cdpath Variable

This variable allows you to specify alternate directories that the system will search to find the subdirectory arguments that you use with the *pushd*, *cd*, and *chdir* commands.

The noclobber Boolean Variable

Suppose you use the following command sequence to take the input from the keyboard and redirect it to a file called `newfile`.

```
cat > newfile
```

If `newfile` exists before this command sequence is executed, the old copy of `newfile` will be overwriten and destroy. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files, set the **noclobber** boolean with:

```
set noclobber
```

in your .login file. To see how this can work, type in:

```
% cat > newfile
This is a test message.
EOT
%set noclobber
cat > newfile
newfile: File Exists.
%
```

When you tried to *cat* to an existing file with **noclobber** set, the system tells you the `File Exists.` and aborts the command. To override **noclobber**, use the exclamation mark metacharacter. For example:

```
%cat > newfile
newfile: File Exists.
%cat >! newfile
This is an override test.
EOT
%
```

The notify Boolean Variable

If the **notify** variable is set, you are immediately notified when a background process is finished. If it is unset, you will be notified that a background process is finished with the next presentation of the C Shell prompt. Use the *set* command to set **notify**.

The $path Variable

One of the most important variables is the variable **path**. This variable contains a sequence of directory names the C Shell searches for commands. For example:

```
set path=(/bin /usr/bin /lbin /usr/bin /etc .)
```

or

```
setenv PATH=(/bin /usr/bin /lbin /usr/bin /etc .)
```

PATH is an environment variable and **path** is a C Shell variable that does the same job. Normally, environment variables are global to the shell and any processes it creates while shell variables are local just to the shell process in which they are set. However, in the case of **PATH** and **path** when you assign a value to one the system automatically changes the other. Thus, the effect of modifying either **PATH** or **path** is global.

When *csh* is first executed, a hash table of command locations is created. This table is created by looking through the directories specified in $PATH, except for the current working directory, in the order shown. If you write new commands and store them in other than your current working directory, the system doesn't know they are there. To tell the system about these new commands, use the *rehash* command.

The $prompt Variable

This variable is used to customize your C Shell prompt. For example,

```
% set prompt = "[\!] % "
[22] %
```

sets the prompt to indicate the command (event) number of the current command. This is very useful when using the C Shell's history facility.

The $shell Variable

When a command is invoked that is not a C Shell command (see "C Shell Commands" later is this tutorial) a new shell is spawned to execute it. This variable is used to indicate what kind of shell should be created in this situation. It can be assigned to either */bin/csh* or */bin/sh*. For example:

```
set shell = /bin/csh
```

spawns a C Shell, while

```
set shell = /bin/sh
```

will spawn a Bourne shell.

There are some commands that usually depend upon the value of $shell, such as *mailx* and *vi*. If you change the value of $shell and execute *mailx* or *vi*, they will be spawned with the new shell type. This may or may not be what you intended.

The $status Variable

This variable return 0 is the most recently executed command executed without error. A non-zero value means an error was detected.

Numeric Shell Variables

The *at* (@) command assigns a value to a numeric variable name, just as the set command assigns a string to a non-numeric variable name. Numeric values can be integer(octal or decimal), real(octal or decimal) or boolean. Values beginning with zero (0) are considered octal.

For example:

```
[22] % @ sum=(1 + 4)
[23] % echo $sum
5
[24] % @ sum = (01 + 012)
[25] % !23
echo $sum
13
[26] %
```

Arrays of numeric variables must be declared with the *set* command before they can be used. Elements of the array are specified by:

```
@ variable_name[index] C_operator C_expression
```

where the index is a numeric constant or numeric variable and the C_operator and C_expression is defined as shown below.

Numeric expressions evaluated by (@) are very similiar to those found in the C programming language. The syntax for this command is:

(@) variable_name C_operator C_expression

The variable_name can be

The C_operator can be:

operator	meaning
A equals B	A equals B
A +equals B	A equals A + B
A −equals B	A equals A − B
A *equals B	A equals A * B
A /equals B	A equals A / B
A %equals B	A equals A MOD B

The C_expression can be composed of constants, numeric variables and the following operators.

operator	meaning
()	Parentheses used to change order of evaluation.
~	Tilde used as unary one's complement
!	Exclamation point for negation
+	Plus for addition
–	Dash for subtraction
*	Asterisk for multiplication
/	Slash for division
%	Percent used for remainder
>>	Double Greater Than for right shift
<<	Double Less Than for left shift
>	Boolean Greater Than
<	Boolean Less Than
>=	Boolean Greater Than or Equal
<=	Boolean Less Than or Equal
!=	Boolean Not Equal
==	String comparison equal
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR
&&	Logical AND
||	Logical OR

File Evaluation

Expressions can also return a value based on the status of a file. If the specified file expression is *true*, the expression returns one (1). If *not true* then the expression returns a zero (0). It the file does **not** exist or is not accessible, the expression returns zero (0). The syntax for a file expression is:

```
-file_test filename
```

where file_test is selected from the following list.

file_test	meaning
d	Is filename a directory?
e	Does filename exist?
f	Is filename a plain file?
o	Do I own filename?
r	Do I have read access to filename?
w	Do I have write access to filename?
x	Can I execute filename?
z	If filename empty (zero bytes long)?

An Example

The following example evaluates a list of filenames and return their status as to if the filename specifies a directory and if it is a directory, the number of lines in it.

```
#
# This script finds directories and lists the number of files
# in them and their word count.
#
foreach dir ($argv)
  set num = 0
  if ( -d $dir) then
     echo "***** $dir is a directory."
  set lsfile = `ls $dir`
     echo " number of file in $dir is $#lsfile"

  foreach file ($lsfile)
     set string = `wc -l $dir/$file`
     @ sum += $string[1]
  end
    echo "   total number of lines in $dir directory is $sum"
  else
    echo " ==> $dir is not a directory."
  endif
end
```

Now execute the script called "find_dir".

```
[45] % find_dir src find_dir
*** src is a directory.
     total number of lines in src directory is 3948
==> find_dir is not a directory.
[46] %
```

C Shell Commands

The C Shell supports several "built-in" commands – commands that are normally executed within the current shell. If you invoke a command that is not a "C Shell Command", a sub-shell is created to handle its execution.

The alias Command

The *alias* command is used to assign new aliases and to show which aliases have been assigned. When executed without arguments, the currently defined aliases are printed. If it is given one argument, the alias of that argument is printed. For example:

```
alias ls
```

shows the current alias, if there is one, for the directory list command *ls*.

The echo Command

The *echo* command prints its arguments to the shell's standard output. Unless redirected, standard output is your CRT. It is often used in shell scripts to print information about what is happening while the script is executing. For example:

```
echo 'Your mail is sent. '
```

tells you that your mail is now sent.

The history Command

The *history* command shows the contents of the C Shell's history buffer. Numbers are assigned to command events that can be used to reference and re-execute previous events.

There is also a shell variable called **prompt**. By placing a '!' character in its value the shell will then substitute the number your current command line will have in the history buffer. This provides an easy way of keeping track of event numbers so that you can use them to reference and re-execute previous events. To set the **prompt** variable you can use:

```
set prompt='\!% '
```

Note that the '!' character had to be escaped here even within single quote characters.

The jobs Command

This command returns information about currently running jobs, including their job numbers, the command event that created it, and the status of the job ("Stopped" or "Running"). See the section "Jobs" later in this tutorial for more information.

The logout Command

The *logout* command can be used to terminate a login shell which has **ignoreeof** set.

The rehash Command

The *rehash* command causes the shell to recompute a hash table of command locations. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it. If you add a command and do not execute *rehash* the hashing algorithm tells the shell that the command wasn't in that directory when the hash table was computed.

The repeat Command

The *repeat* command can be used to repeat a command several times. For example, to make 5 copies of the file `one` in the file `five` you could execute:

```
repeat 5 cat one >> five
```

The set Command

The *set* command with no arguments shows the value of all variables currently defined. For example:

```
[26] % set
argv        ()
cwd         /usr/djp
history 15
home        /usr/djp
cohorts     (bill becky Keith steve mark John)
ignoreeof
noclobber
path        (. /usr/ucb /bin /usr/bin)
prompt      [!] %
shell       /bin/csh
status      0
term        hp
[27] %
```

To set variables to specific values, use the *set* command with the appropriate variable names and arguments. Each of the variables shown above were set initially with the *set* command.

For example, you can set a variable equal to a list of string values or a set of numeric values.

```
[22] % set cohorts = (bill becky Keith steve mark John)
[23] % echo $#cohorts
6
[24] % echo $?cohorts
1
[25] % echo $cohorts[3]
Keith
[26] % unset cohorts
[27] % echo $?cohorts
0
[28] % set nums = (1.234 2 -3.45 )
[29] % echo $nums[3]
-3.45
[30] %
```

The variable expansion sequence *$#* returns the number of elements in the variable array. The sequence *$?* returns a one (1) if the variable exists and a zero (0) if it does **not** exit.

The setenv Command

The *setenv* command is used to set environment variables whose values are global to the shell and any processes it creates.

```
setenv TERM hp2627
```

sets the value of the environment variable TERM to `hp2627`. See *environ(7)* in the *HP-UX Reference*.

The source Command

The *source* command is used to force an update of the current shell environment by causing it to read commands from a file instead of standard input. For example:

```
source .cshrc
```

can be used after editing your .cshrc file to change any variables that you modified. Note that the commands executed from the specified file are not placed in the history buffer, only "source command_file" is.

The time Command

The *time* command is used to find out how long particular commands take to execute. When *time* is followed by a command name argument, the command is executed and then *time* displays information about user, system, and real-execution times of the command. If no argument is specified with *time*, it provides time information about the current shell and any child processes it has created.

The unalias Command

The *unalias* command can be used to remove aliases that you have assigned in the current shell. For example, if the *alias* command was used to cause the change directory command (*cd*) to also print the working directory (*pwd*) each time it was called:

```
alias cd 'cd\!*;pwd'
```

then

```
unalias cd
```

cancels that assigned meaning and *cd* is again interpreted as the standard HP-UX command.

The unset Command

This command removes the assigned values of a variable previously given those values by the *set* command.

The unsetenv Command

The *unsetenv* command returns variables set with the *setenv* command to their default condition.

Jobs

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the shell consisting of these commands together as a unit. Single commands wihout pipes or semicolons create the simplest jobs. Usually every line typed to the shell creates a job. Some lines that create jobs (one per line) are:

```
sort < data
ls -s|sort -n|head -5
mail harold
```

If the metacharacter '&' is typed at the end of the commands, then the job is started as a background job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

runs the *du* program, which reports on the disk usage of your current working directory (as well as any directories below it), puts the output into the file 'usage' and returns immediately with a prompt for the next command without waiting for *du* to finish. The *du* program executes in the background until it finished, and you can type and execute more commands in the mean time. When a background job terminates, a message is typed by the shell just before the next prompt telling you that the job has completed. In the following example the *du* job finishes sometime during the execution of the *mail* command and its completion is reported after the mail job is finished and before you receive your next shell prompt.

```
[30] %du > usage &
[1] 503
[31] % mail bill
How do you know when a background job is finished?
EOT
[1] - Done          du > usage
[32] %
```

If the job did not terminate normally the `Done` message might say something else, like `Killed`. If you want the terminations of background jobs to be reported at the time they occur, possibly interrupting the output of other foreground jobs, you can set the **notify** variable. In the previous example this would mean that the `Done` message might be displayed on your CRT while you are sending the message to Bill. However, it would not become a part of the message.

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments and the process numbers of all commands in the job as well as the working directory where the job was started. Each job in the table is either running in the foreground with the shell waiting for it to terminate, or it is running in the background. Only one job can be running in the foreground at one time, but several jobs can be running in the background at once. As each job is started, it is assigned a small identifying number called the job number which can be used later to refer to the job in the commands described below. A job keeps the same job number until it terminates, at which time the number can be re-used by another job.

When a job is started in the background using '&', its number and the process numbers of all its (top level) commands, are displayed by the shell before prompting you for another command. For example:

```
[40] % ls -s | sort -n > usage &
[2] 2034 2035
[41] %
```

executes pipes its output to the standard input of *sort* which puts its output into the file 'usage'. Since the '&' was at the end of the line, *ls* and *sort* were started together as a background job. After starting the job, the shell prints the job number in brackets ([2]) followed by the process number of each program started in the job. Then the shell immediately prompts for a new command.

To check to see what jobs are currently active, use the *job* command. For example:

```
[42] % jobs -l
```

gives you a list of the current jobs and their job numbers, the commands that they are executed, the process IDs of those commands. It also indicates a status of either "Running" or "Stopped" for each job.

C Shell Scripts

Shell scripts are files containing commands that the shell executes as a group. The files *.login*, *.cshrc* and *.logout* are all shell scripts.

When Not to Use a Script

It is important to note what shell scripts are **not** useful for. There are many excellent commands and program libraries provided with HP-UX. Before you write a script, check your *HP-UX Reference* to see if the solution to your problem is already provided for you.

Running a Script

A C Shell command script may be executed by typing in:

```
csh script_one arg_1 arg_2 ...
```

where `script_one` is the name of the shell script file to execute, and `arg_1 arg_2 ...` represents an optional list of arguments that may be required by the script. The shell places these arguments in the shell variable *argv* as *argv[1]*, *argv[2]*, etc. Instead of using the argv variable to reference these arguments in the shell script, you may use **$n**, where **n** is the position of the argument on the command line. In this example, *$0* is set to *script_one*. The C Shell sequentially reads commands from *script_one*.

If you want to execute shell scripts directly (not having to begin the command line with *csh*) you must:

1. Change the mode of the shell script file so that it is executable:

   ```
   chmod +x script_one
   ```

2. If you want the shell script executed by the C Shell you must make the first character of the first line a #.

   If the file does not begin with a #, the Bourne Shell will execute it.

Script Execution

The C Shell parses each shell script line into command arguments. Each distinct command is then identified. Next, **variable substitution** is performed. Keyed by the Dollar Sign Character (**$**), this substitution replaces the names of variables with their values. Thus:

```
echo $sum1
```

when placed in a command script would cause the current value of the variable `sum1` to be echoed to the output of the shell script. `sum1` must have a value at this time, or an error will result.

To discover if a variable has a value currently assigned to it, use the notation

```
$?sum1
```

The Question Mark (?) causes the expression to return a one (1) if the variable has a currently assigned value and a zero (0) if not. This notation is the only way to access a variable that does not have a value assigned to it without generating an error.

To discover the number of component variables assigned to a variable, use the notation

```
$#sum1
```

The Sharp Sign (#) notation causes the number of component variables assigned to be variable to be returned. For example,

```
set sum1=(a b c)
echo $?sum1
1
echo $#sum1
3
unset sum1
echo $?sum1
0
echo $#sum1
Undefined variable: sum1
%
```

It is possible to access the individual components of a variable which has several values. Thus

```
echo $sum1[1]
```

echoes the first component variable of sum1. In the example above `a` is echoed. Similarly

```
$sum1[$#sum1]
```

would return the component variable `c`.

```
$argv[1-2]
```

would return both `a` and `b`. Other notations useful in shell scripts include:

```
$n
```

a shorthand equivalent of

```
$argv[n]
```

which will return the nth component variable of sum1. Another is:

```
$*
```

which is a shorthand for

```
$argv
```

The difference between `$n` and `$argv[n]` should be noted. `$argv[n]` will yield an error if `n` is not in the range `1` through `$#argv` while `$n` will never yield an out of range subscript error. This is for compatibility with the way other shells handled parameters.

A way to avoid an error when `$argv[n]` is out of range, is to use a subrange of the form `n-m`. If there are less than `n` component variables for the given variable then an empty vector is returned. A range of the form `m-n` also returns an empty vector without giving an error when `m` exceeds the number of elements of the given variable, provided the subscript `n` is in range.

The form

```
$$
```

expands to the process number of the current shell. Since each process is unique, the process number can be used to generate unique temporary file names.

The form

```
$<
```

is replaced by the next line of input read from the shell's standard input, and **not** the script being processed. This is useful for writing shell scripts that are interactive. For example,

```
echo "yes or no?"
set a=($<)
```

would write the prompt `yes or no?` to the shells standard output device and then read the answer from the shells standard input device into the variable `a`.

Shell Script Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '|' implement the boolean and/or operations. The special operators '=~' and '!~' are similar to '==' and '!=' except that the string on the right side can have pattern matching metacharacters (like *.? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

```
-? filename
```

where `?` is replaced by a number of characters. For example the expression primitive

```
-r filename
```

returns the value TRUE if the file `filename` exists and is readable. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length. See *test(1)* in your *HP-UX Reference*, for specifications of these primitives.

It is possible to test whether `command` terminates normally by

```
{ command }
```

This notation returns a one (1) if `command` terminates normally with exit status 0, or a zero (0) if the command terminates abnormally or with exit status that is non-zero. If more detailed information about the execution status of a command is required, the command can be executed and the system variable **$status** examined in the next command. Since **$status** is set by every command, it is very transient.

For a complete list of expression components available for shell scripts, see *csh(1)* in the *HP-UX Reference*.

Shell Script Control Structures

The control structures allowed in C Shell are taken from the C programming language.

Comments (#)

Comment your script using the Sharp Sign (#).

The foreach Statement

The syntax for this statement is:

```
foreach index_variable ( loop_count_value_list )

Command_1
Command_2
.
.
.

end
```

All of the commands between the `foreach` line and its matching end line are executed for each value in `loop_count_value_list` . The variable `index_variable` is set to the successive values of `loop_count_value_list`.

Within this loop we may use the command `break` to stop executing the loop and the command `continue` to prematurely terminate one iteration and begin the next. After the `foreach` loop is done, the iteration variable, `index_counter`, has the value at the last element in `loop_count_value_list`.

The if-then-endif Statement

This statement has the following syntax:

```
if ( expression ) then
Command_1
Command_2
.
.
.

endif
```

The placement of the keywords here is not flexible due to the current implementation of the shell. That means the control structure has to be **exactly** as shown. In other words, `if` and `then` must be in the same line and `endif` must be in a separate line. For example:

```
if ( expression )                          # !!!!! Won't work
then
   Command_1
   Command_2
     .
     .
     .
endif
```

and

```
if ( expression ) then Command_1 endif # !!!!! Won't work
```

are not acceptable to the shell.

You can nest if-then-endif statements using the keyword `else`. For example:

```
If ( expression ) then
   Command_1
   Command_2
     .
     .
     .
else if ( expression ) then
   Command_A
   Command_B
     .
     .
     .
else
   Command_X
   Command_Y
     .
     .
     .endif
```

Note that one `endif` ends the whole structure.

The C Shell has another form of the `if` statement shown below.

```
if ( expression ) Command
```

which can be written

```
if ( expression ) \
       Command
```

If you only need one command executed, the `endif` statement can be omitted. In the second example, the non-printing newline character is escaped with the backslash (\) to allow the command to appear below the expression. This is to improve visual clarity.

The while Statement

The `while` structure is like that found in the C programming language. For example:

```
while ( expression )
   Command_1
   Command_2
      .
      .
      .
end
```

The switch Statement

The `switch` structure is like that found in the C programming language. For example:

```
switch ( word )

case str1:
          commands
             .
             .
             .
          breaksw
case str2:
          commands
             .
             .
             .

          breaksw
default:
          commands
             .
             .
             .

          breaksw
endsw
```

Note

C programmers should note that the `switch` statement uses `breaksw` to exit and not `break`. `While` and `foreach` loops allow `break`.

The goto Statement

The C Shell allows the `goto` statement with labels, just like C.

```
loop:
     Command_1
     Command_2
        .
        .
        .
     goto loop
```

Supplying Input to Commands

Commands executed in shell scripts receive by default the standard input of the shell which is running the script. This is different from how other shells run under HP-UX. This allows shell scripts executed by the C Shell to fully participate in pipelines, but requires extra notation for commands which use inline data.

Thus we need a way to supply inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file.

```
#
foreach i ($argv)
ed - $i << 'STOP'
1,$s/^[ ]*//
w
q
'STOP'
end
```

The notation `<< 'STOP'` means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line. The shell uses the word following the << as a pattern to search for to terminate the passing of text to a command (which in this case is *ed*). In the example above STOP is quoted with ' ' to ensure that no variable or command substitutions are performed on the text before it is passed to *ed*.

Catching Interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. To do this, start your program with

```
onintr label
```

where label is a program label marking the code that handles the interrupt condition. If an interrupt is received by the shell, the C Shell does an automatic

```
goto label
```

and executes the code located at `label`. If we wish to exit a program with a non-zero status, make

```
exit 1
```

a part of your interrupt handling code.

An Example Shell Script

```
#

foreach i ($argv)
if ($i \!~ *.c) then                          # is it a .c file?
echo $i is not .c
continue
else
echo $i is a .c program
endif

echo check file ~/backup/$i:t

if (\! -r ~/backup/$i:t) then                  # is file part of backup?

echo $i: not in backup ... not cp\'ed
continue
endif

echo compare two files $i and ~/backup/$i:t
cmp -s $i ~/backup/$i:t                        # has the file changed?

if ($status != 0) then
echo "new backup of $i"
cp $i ~/backup/$i:t
endif
end
```

This script backs up a list of C programs only if they have been previously backed up. The files are stored in your home directory and the subdirectory "backup". It makes use of the `foreach` statement to execute all the commands between the `foreach` statement and its matching `end`

Table of Contents

BC: An Arbitrary Precision Desk-Calcuator Language

BC: An Arbitrary Precision Desk-Calculator Language

BC is a language and a compiler for doing arbitrary precision arithmetic on your HP-UX system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers. These routines are based on a dynamic storage allocator. Overflow does not occur until all available internal memory is exahausted.

The BC language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution. A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

BC and BS are similiar in capabilities, with BS being a more complete language supporting strings and I/O, but limited to "ordinary" double-precision floating point numbers. BC is limited in operating on numeric data, but operates on arbitrary percision numbers and arbitrary bases. The selection of one or the other is primarily based on the need for large value or high percision calculations. If these are not needed, BS may be the better choice. There is no significant advantage of one over the other for activities such as balancing your checkbook, unless you are the federal government.

Some of the uses of this compiler are:

- to do computation with large integers
- to do computation accurate to many decimal places
- conversion of numbers from one base to another base.

There is a scaling provision that permits the use of decimal point notation. Provision is also made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base te equal eight.

The actual limit on the number of digits that can be handled depends on the amount of internal memory in the machine. Manipulation of numbers with many hundreds of digits is possible.

The syntax of BC is very similiar to the C programming language. This enables users who are familiar with C language to easily work with BC.

Running BC

To use *bc*, type in:

```
bc
```

Your prompt is no longer displayed, and the BC calculator is ready for use.

To exit *bc* and return to your shell, type in:

CTRL-D

Your shell's prompt is displayed showing that you are no longer using *bc*.

Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The following operators may be used:

operator	meaning
+	addition
-	subtraction
/	division
%	modulo (remaindering)
^	exponentiiation

Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the **unary** minus sign). The expression

```
7+ -3
```

is interpreted to mean that -3 is to be added to 7 for a result of

```
4
```

More complex expressions with several operators and with parentheses are interpreted using the following mathematical precedence hierarchy:

Precedence	Operator
Highest	Parentheses: (may be used to force any order of operations.)
	Functions, user-defined and machine- resident.
	Exponentiation (right to left)
	- (unary minus), + (unary plus)
	Multiplication and Division (left to right)
	Addition and Subtraction (left to right)
	All relational operators (=, <, >, ...)

The following expressions are equivalent:

```
a^b^c and a^(b^c)
a*b*c and (a*b)*c
a/b*c and (a/b)*c
```

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x.

When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)     execute assignment
x                 request current value of x
```

produce the printed result

```
13                current value of x
```

Bases

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number `10` is interpreted as octal, this statement will have no effect.

For those who deal in hexadecimal notation, the characters `A` through `F` are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values `10` through `15` respectively. The statement

```
ibase = A
```

will change you back to decimal input base no matter what the current input base is.

Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of *obase*, initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a **3-digit hexadecimal number**. Very large output bases are permitted.

An obase Gotcha

Remember that the output base of any number system is **10**. If you want your output to be in binary, and enter:

```
obase = 2
obase
10
12 + 10
10110
```

the `10` is correct and the obase is binary (2) as shown by the sum.

For example, large numbers can be output in groups of five digits by setting `obase` to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with a slash (\). Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that `ibase` and `obase` have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

Scaling

A third special internal quantity called `scale` is used to determine the scale of calculated quantities. We refer to the number of digits after the decimal point of a number as its `scale`. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

- For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result.
- For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity `scale` and always less than 100.
- The scale of a quotient is the contents of the internal quantity `scale`. The scale of a remainder is the sum of the scales of the quotient and the divisor.
- The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.
- The scale of a square root is set to the maximum of the scale of the argument and the contents of `scale`.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case, truncation is used when digits are discarded. No rounding is ever performed.

The contents of `scale` must be no greater than 99 and no less than 0. It is initially set to 0.

The internal quantities `scale`, `ibase`, and `obase` can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of `scale` by one, and the line

```
scale
```

causes the current value of `scale` to be printed.

The value of `scale` retains its meaning as a number of decimal digits to be retained in internal computation even when `ibase` or obase are not equal to `10`. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
```

or

```
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one `auto` statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
auto z
z = x*y
return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function

```
a
```

above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

Subscripted Variables (Arrays)

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript.

Only one-dimensional arrays are permitted.

The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use.

Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

Control Statements

The `if`, `while`, and `for` statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators:

operator	meaning
<	less than
>	greater than
<=	less than or equal
>=	greater than
==	equals
!=	not equal

Beware of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = will not do a comparison.

The `if` statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The `while` statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The `for` statement begins by executing `expression1`. Then the relation is tested and, if true, the statements in the range of the `for` are executed. Then `expression2` is executed. The relation is tested, and so on. The typical use of the `for` statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

will print

```
a
```

factorial if

```
a
```

is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n\-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to `x` and also increments `i` before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manual for their exact workings.

```
x=y=z  is the same as x=(y=z)
x += y  is the same as x = x+y
x -= y  is the same as x = -+y
x =* y  is the same as x = x*y
x =/ y  is the same as x = x/y
x =% y  is the same as x = x%y
x =^ y  is the same as x = x^y
```

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

In some of these constructions, spaces are significant. There is a real difference between x=-y and x= -y. The first replaces x by x-y and the second by -y.

Three Important Things

- To exit a BC program, type `quit` or CTRL-D.
- There is a comment convention identical to that of C. Comments begin with `/*` and end with `*/`.
- There is a library of math functions which may be obtained by typing at command level

```
bc -l
```

This command will load the following library functions:

- sin (named `s`)
- cos (named `c`)
- arctangent (named `a`)
- natural logarithm (named `l`)
- exponential (named `e`) and
- Bessel functions of integer order (named `j(n,x)`).

The library sets the scale to 20.If you type

```
bc file ...
```

`bc` will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

Notation

Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs, or comments. Newline characters or semicolons separate statements.

Comments

Comments are introduced by the characters `/*` and terminated by `*/`.

Identifiers

There are three kinds of identifiers:

- ordinary identifiers

  The characters `a` through `z` are used as ordinary identifiers.

- array identifiers

  Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers.

- function identifiers

  Function identifiers are followed by parentheses, possibly enclosing arguments.

All three types consist of single lower-case letters. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x[]** and a function named **x()** , all of which are separate and distinct.

Keywords

The following are reserved keywords:

- ibase
- if
- obase
- break
- scale
- define
- sqrt
- auto
- length
- return
- while
- quit
- for

Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits A through F are also recognized as digits with values 10 through 15 respectively.

Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

Named Expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

Simple Identifiers

Simple identifiers are named expressions. They have an initial value of zero.

Array Elements

Array elements are named expressions. They have an initial value of zero.

Scale, Ibase and Obase

The internal registers **scale**, **ibase** and **obase** are all named expressions.

- scale is the number of digits after the decimal point to be retained in arithmetic operations. Scale has an initial value of zero and a maximum possible value of 99.
- ibase and obase are the input and output number radix respectively. Both ibase and obase have initial values of 10.

Function Calls

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value.

As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

```
sqrt(expression)
```

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of scale, whichever is larger.

```
length(expression)
```

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

```
scale(expression)
```

The result is the scale of the expression. The scale of the result is zero.

Constants
Constants are primitive expressions.

Parentheses
An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

Unary operators
The unary operators bind right to left.

```
-(expression)
```

The result is the negative of the expression.

```
++(named-expression)
```

The named expression is incremented by one. The result is the value of the named expression after incrementing.

```
--(named-expression
```

The named expression is decremented by one. The result is the value of the named expression after decrementing.

```
(named-expression)++
```

The named expression is incremented by one. The result is the value of the named expression before incrementing.

```
(named-expression)--
```

The named expression is decremented by one. The result is the value of the named expression before decrementing.

Exponentiation Operator
The exponentiation operator binds right to left.

```
(expression)^(integer_expression)
```

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If **a** is the scale of the left expression and **b** is the absolute value of the right expression, then the scale of the result is:

min(axb,max(scale,a))

Multiplicative Operators
The operators *, /, % bind left to right.

*(expression) * (expression)*

The result is the product of the two expressions. If *a* and *b* are the scales of the two expressions, then the scale of the result is:

min(a + b,max(scale,a,b))
(expression) / (expression)

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

(expression) % (expression)

The % operator produces the remainder of the division of the two expressions. More precisely, a%b is a-a/b*b.

The scale of the result is the sum of the scale of the divisor and the value of scale.

Additive Operators

The additive operators bind left to right.

(expression) + (expression)

The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

(expression) - (expression)

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

Assignment Operators

The assignment operators bind right to left.

named_expression = expression

The above expression results in assigning the value of the expression on the right to the named expression on the left.

named_expression = + expression
named_expression = \- expression
named_expression = * *expression*
named_expression =/ expression
named_expression =% expression
named_expression =^ expression

The result of the above expressions is equivalent to `named expression = named expression OP expression`, where OP is the operator after the = sign.

Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, `while`, or inside a **for** statement.

expression < expression
expression > expression
expression <= expression
expression >= expression
expression == expression
expression != expression

Storage Classes

There are only two storage classes in BC, global and automatic (local).

- Only identifiers that are to be local to a function need be declared with the *auto* command.
- The arguments to a function are local to the function.
- All other identifiers are assumed to be global and available to all functions.
- All identifiers, global and local, have initial values of zero.

Identifiers declared as *auto* are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. *auto* arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in C. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

Quoted string statements

```
"any string"
```

This statement prints the string inside the quotes.

If Statements

if(relation)statement

The statement is executed if the relation is true.

While Statements
while(relation)statement

The statement is executed while the relation is true. The test occurs before each execution of the statement.

For Statements

```
for(expression; relation; expression)statement
```

The for statement is the same as

```
first-expression
        while(relation) {
             statement
             last-expression
        }
```

All three expressions must be present.

Break Statements
break

Break causes termination of a *for* or `while` statement.

Auto Statements

```
auto ordinary_identifier,array_identifier[]
```

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

Define statements
The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

Return Statements

```
return
```

return(expression)

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

Quit

The quit statement stops execution of a BC program and returns control to HP-UX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an *if*, *for*, or *while* statement.

Table of Contents

DC:
Interactive Desk Calculator

Dc is an arbitrary-precision arithmetic package implemented in the HP-UX operating system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily, *dc* operates on decimal integers, but you can optionally specify an input base, output base, and the number of fractional digits to be maintained.

The size of numbers that can be manipulated is limited only by available memory. HP-UX can handle number sizes varying from several hundred digits on the smallest systems to several thousand on the largest.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by *dc*. Some of the commands described here were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into *dc* are put on a push-down stack. *Dc* commands then take the top number or two off the stack, perform the desired operation, then push the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

Synoptic Description

This section describes the *dc* commands that are intended for use by people. The additional commands intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

`number`	The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A thru F which are treated as digits with values 10 thru 15 respectively. Negative numbers should be preceded by an underscore (_). Numbers can contain decimal points.
`+ - * % ^`	The top two values on the stack are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack, combined, then the result is pushed back on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.
s<x>	The top of the main stack is popped and stored into a register named <x>, where <x> may be any character. If *s* is uppercase (*L*), <x> is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.
l<*x*>	The value in register <*x*> is pushed onto the stack without being altered. If `l` is uppercase (`L`), register <*x*> is treated as a stack and its top value is popped onto the main stack. All registers start with empty value which is treated as a zero by the command *l* and is treated as an error by the command *L*
d	The top value on the stack is duplicated.
p	The top value on the stack is printed. The top value remains unchanged.
f	All values on the stack and in registers are printed.
x	Treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of *dc* commands.
[...]	Puts the bracketed character string onto the top of the stack.
q	Exits the program. If executing a string, the recursion level is popped by two. If *q* is capitalized (*Q*), the top value on the stack is popped and the string execution level is popped by that value.
`<x >x =x !<x !>x !=x`	The top two elements of the stack are popped and compared. Register <x> is executed if they obey the stated relation. Exclamation point is negation.

v Replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

! Interprets the rest of the line as an HP-UX command. Control returns to *dc* when the HP-UX command terminates.

c All values on the stack are popped; the stack becomes empty.

i The top value on the stack is popped and used as the number radix for further input. If *i* is uppercase (*I*), the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o The top value on the stack is popped and used as the number radix for further output. If *o* is capitalized (*O*), the value of the output base is pushed onto the stack.

k The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If *k* is uppercase (*K*), the value of the scale factor is pushed onto the stack.

z The value of the stack level is pushed onto the stack.

? A line of input is taken from the input source (usually the console) and executed.

Detailed Description

Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0 thru 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always -1 and all other digits are in the range 0 thru 99. The digit preceding the high order -1 digit is never a 99. The representation of -157 is $43,98,-1$. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

The Allocator

Dc uses a dynamic string storage allocator for all of its internal storage. All internal reading and writing of numbers is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and *dc* is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that resides next to it in memory and, if free, can be combined with it to make a string twice as long. This is an implementation of the "buddy system" of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in *dc*. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) used in the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called <scale> plays a part in the results of most arithmetic operations. <scale> is the bound on the number of decimal places retained in arithmetic computations. <scale> can be set to the number on the top of the stack truncated to an integer with the *k* command. *K* can be used to push the value of <scale> on the stack. <scale> must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of <scale> on the computations.

Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit-by-digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99, -1 by the digit -1. In any case, digits which are not in the range 0 thru 99 must be brought into that range, propagating any carries or borrows that result.

Multiplication

The scales are removed from the two operands and saved. The operands are both made positive, then multiplication is performed in a digit-by-digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register <scale> and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity <scale>. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity <scale> and the scale of the operand.

The method used to compute the square root of (Y) is Newton's method of successive approximations by the rule:

$$X_{(n+1)} = \frac{1}{2}\left[X_n + \frac{Y}{X_n}\right]$$

The initial guess is found by taking the integer square root of the top two digits.

Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

Input Conversion and Base

Numbers are converted to internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with an underscore (_). The hexadecimal digits A thru F correspond to the numbers 10 thru 15 regardless of input base. The *i* command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, using it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal- or hexadecimal-to-decimal conversions. The command *I* will push the value of the input base on the stack.

Output Commands

The command *p* causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command *f*. The *o* command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command *O* pushes the value of the output base on the stack.

Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-to-octal or decimal-to-hexadecimal conversions.

Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands *s* and *l*. The command *s*<x> pops the top of the stack and stores the result in register <x> where <x> can be any character. *l*<x> puts the contents of register <x> on the top of the stack. The *l* command has no effect on the contents of register <x>. The *s* command, however, is destructive.

Stack Commands

c clears the stack.

d pushes a duplicate of the number on the top of the stack on the stack.

z pushes the stack size on the stack.

X replaces the number on the top of the stack with its scale factor.

Z replaces the top of the stack with its length.

Subroutine Definitions and Calls

Enclosing a string in *[]* pushes the ASCII string on the stack. The *q* command quits or, in executing a string, pops the recursion levels by two.

Internal Registers – Programming *dc*

The load and store commands together with *[]* to store strings, *x* to execute and the testing commands `<`, `>`, `=`, `!<`, `!>`, and `!=` can be used to program *dc*. The *x* command assumes the top of the stack is a string of DC commands, and executes the string. Testing commands remove the top two elements on the stack, test them, then, if the relation holds, execute the register following the elements tested. For example, to print the numbers 0-9, use the following commands:

```
[lip1+ si li10>a]sa
0si lax
```

Push-Down Registers and Arrays

These commands involve push-down registers and arrays, and were designed for use by a compiler, not by people. In addition to the stack that commands work on, *dc* can be thought of as having individual stacks for each register. These registers are operated on by the commands *S* and *L*. *S*<x> pushes the top value of the main stack onto the stack for register <x>. *L*<x> pops the stack for register <x> and puts the result on the main stack. The commands *s* and *l* also work on registers, but not as push-down stacks. *l* doesn't affect the top of the register stack; *s* destroys what was there before.

The commands that work on arrays are colon (`:`) and semicolon (`;`). (`:`<x>) pops the stack and uses the value obtained as an index into the array <x>. The next element on the stack is stored at the indexed location in <x>. The index value must be greater than or equal to 0 and less than 2048. (`;`<x>) loads the main stack from the array <x>. The value on the top of the stack is popped and used as the index into the array <x>. The indexed value is then loaded from the array onto the stack.

Miscellaneous Commands

The command `!` interprets the rest of the line as an HP-UX command and passes it to HP-UX for execution. Another compiler command is *Q*. This command uses the top of the stack as the number of levels of recursion to skip.

Design Choices

The real reason for using dynamic storage allocation was that a general purpose program is useful for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seems to have no compelling advantage, but with a hardware limit of 127 and only 5% additional space required, debugging was made a great deal easier and decimal output was made much faster.

Stack-type arithmetic design permitted all *dc* commands from addition to subroutine execution to be implemented in essentially the same way, resulting in a considerable degree of logical separation of the final program into modules with very little communication required between modules.

Eliminating interaction between the scale and the bases provided an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. For example, if the value of <scale>. were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results.

The scheme implemented has the advantage that the value of the input and output bases are only used for input and output, respectively, and are ignored in all other operations. The scale value is not used during program operation, serving only to reasonably limit the number of decimal places resulting from arithmetic operations.

The design rationale for scaling arithmetic results was that no significant digits should be discarded if there was any indication that the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for <scale>. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a <scale> to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

Part No. 97089-90004
E 0784
Microfiche No. 97089-99004

Printed in U.S.A.
First Edition, July 1984