CN/US/142

August 12th 1991

Intermediate Bourne Shell Programming

Michel Goossens/CN

CERN Geneva, Switzerland

Abstract

This document, whose material was originally part of the CERN Apollo Users Guide, assumes that the reader has already a basic knowledge of shell programming. If this is not the case it is suggested to first read a basic Unix textbook such as "**A Practical Guide to the Unix System**" by Mark M. Sobell, available from the Computer Documentation Office, Bat. 513, with the CERN stores card, price 38CHF.

Comments by Judy Richards and the technical help of Roger Woolnough are gratefully acknowledged.

Please address your comments via electronic mail to `GOOSSENS@CERNVM`

This document is available as a PostScript file `bournint.ps` on all central systems in the PostScript documentation directories.

Table of Contents

List of Tables

Chapter 1: Intermediate Bourne Shell Programming

1.1 Introduction

The shell is both a command language and a programming language that provides an interface to the Unix operating system. The **Bourne shell** is available on all Unix systems, and if portability is important, users should use this shell.

For interactive use many users prefer to use the **C** shell, which has control syntax similar to the C language. Most systems now also provide the **Korn** shell, which is backward compatible with the Bourne shell and offers many of the extensions in functionality introduced in the C shell. It is the basis of the proposed POSIX standard shell (POSIX 103.2).

The remainder of this document discusses an introduction to the functionality of the Bourne shell within the framework of the BSD environment. It should be noted that the average user would in any case see very little difference between the behaviour of the commands in BSD or Sys5.

1.2 Input-output redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing, for example,

```
$ ls -l >myfile
```

The notation `>myfile` is interpreted by the shell and is not passed as an argument to `ls`. If `myfile` does not exist then the shell creates it; otherwise the original contents of `myfile` are replaced with the output from `ls`. Output is **appended** to a file, which is created if it does not already exist, using the notation

```
$ ls -l >> myfile
```

The standard input of a command may be taken from a file instead of the terminal as in the following example, where the command `wc` reads its standard input redirected from `myfile`) and prints the number of characters, words and lines found.

```
$ wc < myfile
      28     245    1723
```

1.3 Pipelines and filters

The standard output of one command may be connected to the standard input of another by writing the **pipe** operator, indicated by |, as in,

```
$ ls -l | wc
      28     245    1723
```

Two commands connected in this way constitute a pipeline and the overall effect is the same as

```
$ ls -l > myfile; wc < myfile
      28     245    1723
```

except that no file is used. Instead the two processes are connected by a pipe and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting `wc` when there is nothing to read and halting `ls` when the pipe is full.

A **filter** is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, `grep`, selects from its input those lines that contain some specified string. For example,

```
$ ls -l | grep 'Feb 22'
-rw-r--r--   1 goossens     72946 Feb 22 18:42 unixintr.tex
-rw-r--r--   1 goossens      1208 Feb 22 17:01 unixmain.tex
```

prints those lines, if any, of the output from `ls` that contain the string `Feb 22`, in this case all files modified on the 22nd of February. Another useful filter is `sort`.

For example, the following will print an alphabetically sorted list of logged in users:

```
$ who | sort
```

A pipeline may consist of more than two commands, for example,

```
$ ls | grep 'Feb 22' | wc -l
      2
```

prints the number of filenames in the current directory containing the string `Feb 22`.

1.4 Filename generation

Many commands accept arguments which are filenames. For example,

```
$ ls -l main.c
```

prints information relating to the file `main.c`.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
$ ls -l *.c
```

generates, as arguments to `ls`, all filenames in the current directory that end in `.c`. The character `*` is a **pattern** that matches any string including the null string. In general patterns are specified as described in Table 1.1.

Table 1.1: Filename generation patterns

Pattern	Description
`*`	Matches any string of characters including the null string, but not a **leading dot**.
`?`	Matches any single character.
`[...]`	Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

Example:

`[a-z]*` Match all names in the **current** directory starting with one of the lowercase letters `a` through `z`.

`/usr/fred/test/?` Matches all names in directory `/usr/fred/test` consisting of a single character. If no filename is found which matches the pattern then the latter is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
$ echo /usr/fred/*/core
```

finds and prints the names of all core files in subdirectories of `/usr/fred`. (`echo` is a standard Unix command that prints its arguments, separated by blanks). This last feature can be expensive, requiring a scan of all subdirectories of `/usr/fred`.

There is one exception to the general rules given for patterns. The character **.** at the start of a filename must be explicitly matched.

```
$ echo *
myfile_1 myfile_2 myfile_3 my_dir_1 my_dir_2
```

will therefore echo all filenames in the current directory not beginning with a dot, while

```
$ echo .*
 . ..
```

will echo all those filenames beginning with a dot. This avoids inadvertent matching of the names **.** and **..** which mean **the current directory** and **the parent directory** respectively. (Notice that `ls` suppresses information for the files **.** and **..** unless you specify the `-a` option.)

1.5 Quoting

Characters that have a special meaning to the shell, such as `< > * ? | &`, are called **metacharacters**. A complete list of metacharacters is given in Table 1.2 Any character preceded by a `\`is quoted and loses its special meaning, if any. The `\`is elided. At the end of a line, the `\`acts as a **continuation** character. For example:

```
$ echo *                # files in current directory
myfile_1 myfile_2 myfile_3 my_dir_1 my_dir_2
$ echo \*               # should echo a *
*
$ echo \\               # should echo a \
\
$ echo first line of a long string \    # first line
> second line of a long string \        # continuation lines
> and last line         # continuation lines
first line of a long string second line of a long string and last line
```

Although \ is convenient for quoting single characters, it can become clumsy and error prone when more than one character needs quoting. A string of characters may be quoted by enclosing the string between **single quotes**, e.g.

```
$ echo xx'****'xx
$ xx****xx
```

The quoted string may not contain a single quote but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using **double quotes** is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to later.

1.6 Prompts

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is `$`. It may be changed by saying, for example,

```
$ PS1="yesdear: "
yesdear: "PS1=$ "
$
```

This sets the prompt to be the string `yesdear:` and back again to the default prompt `$`. If a `newline` is typed and further input is needed then the shell will issue the prompt > . Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt will return the shell to read another command. This continuation prompt may be changed by saying, for example,

```
$ PS2=more
```

1.7 Shell procedures (also called shell scripts)

The shell may be used to read and execute commands contained in a file. For example,

```
sh myfile [ args ... ]
```

calls the shell to read commands from `myfile`. Such a file is called a **command procedure**, **shell procedure** or **shell script**. Arguments may be supplied with the call and are referred to in `myfile` using the positional parameters `$1`, `$2 ...`, e.g.

```
$ cat wg            # Look at contents of file  wg
who | grep $1
$ sh wg fred        # Execute file  wg
fred                 display      Apr 22 11:07
```

```
$ who | grep fred # Type the commands at the terminal
fred               display      Apr 22 11:07
```

Unix files have three independent attributes, read (`r`), write (`w`), and execute (`x`). The Unix command `chmod` may be used to make a file executable. For example,

```
$ chmod +x wg      # Give execute status to  wg
$ sh wg fred       # Execute file  wg
fred                display      Apr 22 11:10
$ wg fred          # Execute file  wg directly
fred                display      Apr 22 11:10
```

This allows shell procedures and programs to be used interchangeably. In either case a **new process** is created to run the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as `$#`. The name of the file being executed is available as `$0`.

A special shell parameter `$*` is used to substitute for all positional parameters except `$0`. A typical use of this in a shell script is to provide some **default** arguments, as in,

```
$ nroff -T450 -ms $*
```

which simply prepends some arguments to those already given.

1.8 Control flow – for

A frequent use of shell procedures is to loop through the arguments (`$1`, `$2...`) executing commands once for each argument. An example of such a procedure is `pass`, which searches the file `/etc/passwd` that contains lines of the form

```
  ...
fred:86zFEDAi.2sVfB:2741:78:User Fred/EP://node1/user/fred:/bin/ksh
  ...
bert:BU.BeSjH.VmHpj:1956:69:User Bert/DD://node2/user/bert:/bin/sh
  ...
```

The text of `pass` is

```
for i
do grep $i /etc/passwd; done
```

The commands

```
$ chmod +x pass
$ pass fred
```

allow the printing of those lines in `/etc/passwd` which contain the string `fred`.

```
$ pass fred bert
```

prints those lines containing `fred` followed by those for `bert`.

The `for` loop notation is recognized by the shell and has the general form

```
for name  in w1 w2 ...
  do command-list
done
```

A command-list is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like `do` and `done` are only recognized **following a newline or semicolon**. `name` is a shell variable that is set to the words `w1 w2 ...` in turn each time the command-list following `do` is executed. If `in w1 w2 ...` is omitted then the loop is executed once for each positional parameter; that is, `in $*` is assumed.

Another example of the use of the `for` loop is the `create` command whose text is

```
for i do >$i; done
```

The command

```
$ create alpha beta
```

ensures that two empty files `alpha` and `beta` exist and are empty. The notation `>file` may be used on its own to create or clear the contents of a file. Notice also that a `semicolon ;` (or newline) is required before `done`.

1.9 Control flow – case

A multiple way branch is provided for by the `case` notation. For example,

```
case $# in
  1) cat >>$1 ;;
  2) cat >>$2 <$1 ;;
  *) echo 'usage: append [ from ] to' ;;
esac
```

is an `append` command. When called with one argument as

```
$ append file
```

`$#` results in the string `1` and the standard input is copied onto the end of file using the `cat` command. On the contrary if called with two arguments

```
$ append file1 file2
```

the contents of `file1` is appended onto `file2`. If the number of arguments supplied to `append` is other than 1 or 2 then a message is printed indicating proper usage.

The general form of the `case` command is

```
 case word  in
  pattern) command-list;;
  ...
 esac
```

The shell attempts to match `word` with each pattern, in the order in which the patterns appear. If a match is found the associated command-list is executed and execution of the `case` is complete. Since `*` is the pattern that matches any string it can be used for the default case.

A word of caution No check is made to ensure that only one pattern matches the case argument. The **first match found defines the set of commands to be executed**.

In the example below the commands following the second `*` will never be executed.

```
case $# in
  *) ... ;;
  *) ... ;;
esac
```

Another example of the use of the `case` construction is to distinguish between different forms of an argument. The following example is a fragment of a `cc` command.

```
for i
  do case $i in
    -[ocs])  ... ;;
    -*) echo 'unknown flag $i' ;;
    *.c) /lib/c0 $i ... ;;
    *)   echo 'unexpected argument $i' ;;
  esac
done
```

To allow the **same** commands to be associated with more than one pattern the `case` command provides for **alternative** patterns separated by a `|`.

For example, the following commands are equivalent:

```
case $i in              case$i in
  -x|-y)   ...            -[xy])   ...
esac                    esac
```

The usual quoting conventions apply so that the following will match the character `?`.

```
case $i in
  \?)     ...
```

1.10 Here documents

The shell procedure `pass` on page 4 uses the file `/etc/passwd` to supply the data for `grep`. An alternative is to include this data within the shell procedure as a **here document**, as in,

```
for i
  do grep $i <<!
   ...
  fred:86zFEDAi.2sVfB:2741:78:User Fred/EP://node1/user/fred:/bin/ksh
   ...
  bert:BU.BeSjH.VmHpj:1956:69:User Bert/DD://node2/user/bert:/bin/sh
   ...
!
done
```

In this example the shell takes the lines between `<<!` and `!` as the standard input for `grep`. The string `!` is arbitrary, the document being terminated by a **line** that consists of the string following <<.

Parameters are **substituted** in the document **before** it is made available to `grep` as illustrated by the following procedure called `edg`.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The following call to `edg` or the explicit commands are thus equivalent:

```
$ edg string1 string2 myfile          ed myfile <<%
                                             g/string1/s//string2/g
                                             w
                                             %
```

Both methods change all occurrences of `string1` in file `myfile` to `string2`. **Substitution** can be **prevented** by **quoting** using \, as for the special character `$` (meaning the end-of-file) for `ed`.

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

(This version of `edg` is equivalent to the first except that `ed` will print a `?` if there are no occurrences of the string `$1`. **Substitution within a here document may be prevented entirely by quoting the terminating string**, for example,

```
grep $i <<\#
...
#
```

The document is presented **without modification** to `grep`. If parameter substitution is not required in a here document this latter form is more efficient.

1.11 Shell variables

The shell provides **string-valued variables**. Variable names begin with a letter and consist of letters, digits and underscores. Variables may be given values by writing, e.g.

```
$ user=fred box=m000 acct=mh0000
```

which assigns values to the variables `user`, `box` and`acct`. A variable may be set to the null string by saying, e.g.

```
$ null=
```

The **value of a variable** is substituted by preceding its name with `$`, e.g.

```
$ echo $user
fred
```

Variables may be used interactively to provide abbreviations for frequently used strings, e.g.

```
$ b=/usr/fred/bin
$ mv pgm $b
```

will move the file `pgm` from the current directory to the directory `/usr/fred/bin`. A more general notation is available for parameter (or variable) substitution, as in,

```
$ echo ${user}
```

which is equivalent to

```
$ echo $user
```

and is used when the parameter name is followed by a letter or digit, e.g.

```
$ tmp=/tmp/ps
$ ps a >${tmp}a
```

will direct the output of `ps` to the file `/tmp/psa`, whereas,

```
$ ps a >$tmpa
```

would cause the value of the variable `tmpa` to be substituted.

Except for the variable `$?`, which is set **after executing** of each command, the general shell variables introduced below are **initially** set by the shell. See table 1.2 on page 8 for more details.

Table 1.2: shell variables

Variable	Description
`$@`	The positional parameters are expanded into separate quoted strings `"$1" "$2",...`
`$*`	The positional parameters are expanded into a single quoted strings `"$1 $2,..."`.
`$#`	The number of **positional parameters** (in decimal).
`$0`	Name of the command (script) being executed.
`$-`	The **current shell flags**, such as `-x` and `-v`.
`$?`	The **exit status** (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned.
`$$`	The **process number of this shell** (in decimal). Since process numbers are unique this string is frequently used to generate unique temporary filenames, e.g. `ps a >/tmp/ps$$`
`$!`	The **process number** of the last process run in the **background** (in decimal).
`$MAIL`	Name of the file where mail is stored. If this variable is set, the shell informs the user of the arrival of mail in the specified file. Usually this file is in `/usr/mail/name`, where `name` is the user's login name.
`$HOME`	Is the default argument (home directory) for the cd command. It is set in `/etc/passwd` and can be changed by the user. `echo $HOME          # display HOME directory` `/usr/fred/bin` `cd                   # go to HOME directory` `pwd                  # display current directory` `/usr/fred/bin`
`$PATH`	A list of directories that contain commands (the search path). Each time a command is executed by the shell a list of directories is searched for an executable file. If `$PATH` is not set then the current directory, `/bin`, and `/usr/bin` are searched by default. Otherwise `$PATH` consists of directory names separated by colons (`:`), e.g. `PATH=:/usr/fred/bin:/bin:/usr/bin` specifies that the current directory (the null string before the first `:`), `/usr/fred/bin`, `/bin` and `/usr/bin` are to be searched in that order. In this way individual users can have their own **private** commands that are accessible independently of the current directory. If the user wants to **add** directory `/usr/newdir` to his search list, then the `PATH` variable should be redefined as follows: `PATH=$PATH:/usr/newdir` **This directory search is not used for command names starting with /**
`$PS1`	The primary shell prompt string, by default, `$ `.
`$PS2`	The shell prompt when further input is needed, by default, `> `.
`$IFS`	The set of characters used by blank interpretation (see Page 15)

1.12 The test command

The `test` command, although not part of the shell, is intended for use by shell programs, e.g.

```
test -f myfile
```

returns zero exit status if `myfile` exists and non-zero exit status otherwise. In general `test` evaluates a predicate and returns the result as its exit status. Some of the more frequently used `test` arguments are given here (do a `man test` for more details):

Table 1.3: Structure of the test command

Command Syntax	**Description**
`test s`	true if the argument `s` is not the null string
`test -f myfile`	true if `myfile` exists
`test -r myfile`	true if `myfile` is readable
`test -w myfile`	true if `myfile` is writable
`test -d myfile`	true if `myfile` is a directory

Arithmetic is not provided by the shell but is available indirectly using the **expr** command

The **test** command is used so often in shell programming that an alternative format is recognised. This format improves readability, especially with **if** statements.

`test expression` is equivalent to **`[ expression ]`**

1.13 Control flow – while

The actions of the `for` loop and the `case` branch are determined by **data available to the shell**. A `while` or `until` loop and an `if then else` branch are also provided whose actions are determined by the **exit status returned by commands**. A `while` loop has the general form

```
while command-list1
  do command-list2
done
```

The value tested by the `while` command is the **exit status of the last simple command** following `while`. Each time round the loop `command-list1` is executed; if a zero exit status is returned then `command-list2` is executed; otherwise, the loop terminates. For example, the following are equivalent

```
while test $1          for i
  do ...                 do ...
    shift              done
done
```

where `shift` is a shell command which renames the positional parameters `$2, $3, ...` as `$1, $2, ...` and loses `$1`.

Another kind of loop is to wait **until** some external event occurs and then run some commands. In an `until` loop the termination condition is reversed. For example,

```
until test -f file
  do sleep 300; done
commands
```

will loop until `file` exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually create the file.)

1.14 Control flow – if

Also available is a general conditional branch of the form,

```
if  command-list
then    command-list
else     command-list
fi
```

which **tests the value returned by the last simple command following** `if`.

The `if` command may be used in conjunction with the `test` command to test for the existence of `myfile` as in

```
if test -f myfile
then     # process myfile
else     # do something else
fi
```

A multiple test `if` command can be written in two equivalent forms

```
if ...                                   if ...
then    ...                                 then  ...
else    if ...                              elif  ...
           then ...                            then   ...
                ......                              ......
        fi
fi                                       elif
```

The **dw** command shown in Fugure 1.1 recursively scans a directory tree and lists the files found. At the beginning the `PATH` variable is set explicitly to ensure that the right commands are executed (e.g the user can have commands called `dw` or `test` in the directories traversed, and the script should be guarded against these being executed).

Figure 1.1: Example of a shell script using the command test

```
PATH=/bin:/user/bin/:$HOME/bin   # explicit command search path

cd $1             # go the start directory
ls -l             # list files

for i in *        # for all files in present directory
  do     if test -d $i          # is it a directory file
         then dw $i             # recursively call dw again
         fi
  done
```

The following notations are all equivalent:

```
if test -d $i          if [ -d $i ]           test -d $i && dw $i
  then dw $i             then  dw $i
fi                     fi
```

Note that the following two constructs are complementarity :

```
command_1 && command_2 # execute command_2 if command_1 is successful

command_1 || command_2 # execute command_2 if command_1 fails
```

1.15 Command grouping

Commands may be grouped in two ways.

```
{ command-list ; }          and          ( command-list )
```

The semicolon `;` is needed, since braces are reserved words and not special characters like parentheses, and reserved words are only recognised following a `newline` or `;`. In grouping using braces `command-list` is executed in the parent shell. The form with parentheses executes `command-list` as a separate shell process. For example,

```
(cd x; rm junk )
```

executes `rm junk` in the directory `x` without changing the current directory of the invoking shell, whereas

```
{ cd x; rm junk ; }
```

has the same effect but leaves the invoking shell in the directory `x`.

1.16 Debugging shell procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set -v
```

(v for **verbose**) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

```
sh -v proc ...
```

where `proc` is the name of the shell procedure. This flag may be used in conjunction with the `-n` flag which prevents execution of subsequent commands. (Note that saying `set -n` at a terminal will render the terminal useless until an `end-of-file` CTRL D is typed.)

An execution trace, i.e. printing each command as it is executed following parameter substitution, is obtained with the command:

```
set -x
```

The current setting of the shell flags is available as `$-` . Both flags are turned off by saying

```
set -
```

Chapter 2: More advanced features

2.1 Keyword parameters

Shell variables may be given values by **assignment** or when a shell procedure is **invoked**. An argument to a shell procedure of the form `name=value` that precedes the command name causes value to be assigned to name before execution of the procedure begins. The value of name in the invoking shell is not affected. For example,

```
user=fred command
```

will execute command with user set to fred. The -k flag causes arguments of the form `name=value` to be interpreted in this way anywhere in the argument list. Such names are sometimes called **keyword parameters**. If any arguments remain they are available as positional parameters `$1`, `$2`, etc.

The `set` command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

will set `$1` to the first filename in the current directory, `$2` to the next, and so on. The first argument, `-`, ensures correct treatment when the first filename begins with a `-`.

2.2 Parameter transmission

When a shell procedure is invoked both positional and keyword parameters may be supplied with the call. **Keyword parameters** are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be **exported**. For example,

```
export user box
```

marks the variables `user` and `box` for export. When a shell procedure is invoked **copies** are made of all exportable variables for use within the invoked procedure. **Modification of such variables within the procedure does not affect the values in the invoking shell.** It is generally true of a shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared **readonly**. The form of this command is the same as that of the export command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

2.3 Parameter substitution

A parameter is a sequence of letters, digits, or underscores (a name), a digit, or any of the following characters: `* @ # ? - $ !`. The character `$` (dollar) is used to introduce substitutable parameters. Positional parameters may be assigned values by set. Variables may be set by writing

```
name=value [ name=value ] ...
```

In table 2.1 the shell does not evaluate `word` unless it is to be used as the substituted string, e.g.

```
$ echo ${d-`pwd`}
```

only executes `pwd` if variable `d` is not set.

Table 2.1: Parameter substitution

Construct	**Explanation**	**Example**
`${parameter}`	Substitute the value, if any, of `parameter`. Braces are required only when `parameter` is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If `parameter` is a digit, it is a positional parameter. If `parameter` is `*` or `@` then all the positional parameters, starting with `$1`, are substituted separated by spaces. `$0` is set from argument zero when the shell is invoked.	The following echoes the value of d when it is set or null otherwise `$ echo $d` The following sequence is identical to the previous one. `$ echo ${d}`
`${parameter-word}`	If `parameter` is set, substitute its value; otherwise, substitute `word`.	`$ echo ${d-.}` echoes the value of variable `d` if it is set and `.` otherwise. `$ echo ${d-'*'}` echoes `*` if variable `d` is not set. `$ echo ${d-$1}` echoes the value of `d` if it is set and the value (if any) of `$1` otherwise.
`${parameter=word}`	If `parameter` is not set, set it to `word`; then, substitute the value of `parameter`. Positional parameters may not be assigned in this manner.	`$ echo ${d=.}` substitutes the same string as `$ echo ${d-.}` If `d` were not previously set then it is set to the string `.`.
`${parameter?word}`	If `parameter` is set, substitute its value; otherwise, print `word` and exit from the shell. If `word` is omitted, a standard message is printed.	A script where some variable must be set could start with `: ${user?} ${acct?}` Since `colon (:)` is a command built into the shell doing nothing once its arguments have been evaluated, if any of the variables `user` or `acct` are not set then the shell will abandon execution of the procedure.
`${parameter+word}`	If `parameter` is set, substitute `word`; otherwise, substitute nothing.	

2.4 Command substitution

The standard output from a command can be substituted in a similar way to parameters. The command `pwd` prints on its standard output the name of the current directory. For example, if the current directory is `/usr/fred/bin` then the command

```
$ d=`pwd`
```

is equivalent to

```
$ d=/usr/fred/bin
```

The entire string between grave accents (`` `...` ``) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a `` ` `` must be escaped using a \, e.g.

```
$ ls `echo "$1"`
```

is equivalent to

```
$ ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including here documents) and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is `basename` which removes a specified suffix from a string. For example,

```
$ basename main.c .c
```

will print the string `main`. Its use is illustrated by the following fragment from a `cc` command.

```
case $A in
  ...
  *.c)   B=`basename $A .c`
  ...
esac
```

that sets `B` to the part of `$A` with the suffix `.c` stripped.

Some composite examples

In the following example variable `i` is set to the names of the files in the current directory in time order, most recent first.

```
for i in `ls -t`; do ...
```

Information can be extracted from the shell by using the `set` command, which sets positional parameters:

```
set `date`; echo $6 $2 $3, $4
1989 Oct 16, 15:39:59
```

Operations can be performed on a shell variable by using the shell command `expr`, which evaluates arguments as an expression. For instance the following example shows how to increment a shell variable.

```
$ a=4         # Set value of variable a
$ echo $a     # Show present value of variable a
4
$ a=`expr $a + 1`              # Increment value of variable a
$ echo $a     # Show present value of variable a
5
```

2.5 Evaluation and quoting

The shell is a macro processor that provides **parameter substitution**, **command substitution** and **filename generation** for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in Table 2.6. Before a command is executed the following substitutions occur.

Parameter substitution

`$user` is an example of parameter substitution. Only **one evaluation** occurs so that if, for example the value of the variable `X` is the string `$y` then

```
echo $X
```

will echo `$y`.

Command substitution

`` `pwd` `` is an example of command substitution. In can be used, e,g, in a profile to set the `HOME` variable as

```
HOME=`pwd`
```

to make the profile independent from the login directory.

Blank interpretation

Following the above substitutions the resulting characters are broken into non-blank words (blank interpretation). For this purpose **blanks** are the characters of the string `$IFS`. By default, this string consists of `blank, tab` and `newline`. The `null`string is not regarded as a word unless it is quoted, e.g.

```
echo ''
```

will pass on the null string as the first argument to `echo`, whereas

```
echo $null
```

will call `echo` with no arguments if the variable `null` is not set or set to the null string.

File name generation

Each word is finally scanned for the file pattern characters `*, ?` and `[...]` and an alphabetical list of filenames is generated to replace the word. Each such filename is a separate argument.

The four evaluations just described also occur in the list of words associated with a `for` loop. Only substitution occurs in the word used for a `case` branch.

As well as the quoting mechanisms described earlier using \ and `'...'` a third quoting mechanism is provided using double quotes. Within **double quotes** parameter and command substitution occurs but filename generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \ .

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters `$ ` " \`

Example

```
$ echo "$x"          Uses value of variable  x as single argument
$ echo "$*"          Uses all positional parameters as single argument
$ echo "$1 $2 ..."   Equivalent to statement above
$ echo "$@"          Uses positional pameters  unevaluated
$ echo "$1" "$2" ... Equivalent to statement above
```

Table 2.2 shows, for each quoting mechanism, the shell metacharacters that are evaluated.

Table 2.2: Shell quoting mechanism

Quoting character	**Metacharacter**					
	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n
t means "terminator", **y** means "interpreted" and **n** means "not interpreted"						

In cases where more than one evaluation of a string is required the built-in command `eval` may be used. For example, if the variable `X` has the value `$y`, and if `y` has the value `pqr` then

```
eval echo $X
```

will echo the string `pqr`.

In general the `eval` command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg='eval who|grep'
$wg fred
```

is equivalent to

```
who|grep fred
```

In this example, `eval` is required since **there is no interpretation of metacharacters, such as |, following substitution.**

2.6 Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by `getty`). A shell invoked with the -i flag is also interactive.

Execution of a command may fail for any of the following reasons.

- **Input output redirection may fail** (e.g. a file does not exist or cannot be created).
- **The command itself does not exist or cannot be executed**.
- **The command terminates abnormally** (e.g. a "bus error" or "memory fault').
- **The command terminates normally but returns a nonzero exit status**.

In all of these cases the shell will go on to execute the next command. Except for the last case an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

- **Syntax errors** (e.g., `if ... then ... done`)
- **A signal such as interrupt** (e.g. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal).
- **Failure of any of the built-in commands** (e.g. `cd`).

The shell flag -e causes the shell to terminate if any error is detected.

Table 2.3: Unix signals

Signal Number	Description
1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (from `kill (1)`)
Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14 and 15.	

2.7 Fault handling

Shell procedures normally terminate when an interrupt is received from the terminal. The `trap` command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received will execute the commands

```
rm /tmp/ps$$; exit
```

`exit` is another built-in command that terminates execution of a shell procedure. The `exit` is required; otherwise, after the `trap` has been taken, the shell will resume executing the procedure at the place where it was interrupted.

Unix signals can be handled in one of three ways.

1 They can be **ignored**, and the signal is never sent to the process.
2 They can be **caught**, and the process must decide what action to take when the signal is received.
3 They can **terminate** the process without it having to take any further action.

If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background then `trap` commands (and the signal) are ignored.

The use of `trap` is illustrated by the following fragment of a command, where the cleanup action is to remove a file `junk$$`:

```
trap 'rm -f junk$$; exit' 0 1 2 3 15
```

The `trap` command must appear before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0 in Unix it is used by the shell to indicate exit from a shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to `trap`. The following fragment is taken from the `nohup` command.

```
trap '' 1 2 3 15
```

which causes `hangup, interrupt, quit` and `kill` to be ignored both by the procedure and by invoked commands.

Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing:

```
trap
```

The procedure `scan` (Figure 2.1) is an example of the use of `trap` where execution resumes once the trap has been executed (i.e. no exit is specified). `scan` takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. the **:** command does nothing once its arguments have been evaluated, hence

```
trap : 2
```

causes interrupts to be effectively ignored while executing the `eval $x`. On the other hand, when waiting for input, executing the statement `read x`,

```
trap  exit 2
```

is in effect.

`read x` is a built-in command that reads one line from the standard input and places the result in the variable `x`. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

Figure 2.1: The scan command

```
d=`pwd`
for i in *
  do if test -d $d/$i
       then cd $d/$i
       while echo "$i:"
       trap exit 2
       read x
       do trap : 2; eval $x; done
     fi
done
```

2.8 Invoking the shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, then commands are read from the file `.profile`.

-c string Read the commands from `string`.

-s If the -s flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

-i If the -i flag is present or if the shell input and output are attached to a terminal (as told by `getty`) then this shell is interactive. In this case `terminate` is ignored (so that kill 0 does not kill an interactive shell) and `interrupt` is caught and ignored (so that wait is interruptable). In all cases `quit` is ignored by the shell.

The basic environment variables are initialized at login according to Table 2.4

Table 2.4: Environment variables set at login

Variable	Default value	Example of actual value
`HOME`	your-log-in-directory	`/user/goossens`
`LOGNAME`	your-log-in-name	`goossens`
`MAIL`	/usr/mail/your-log-in-name	`/usr/mail/goossens`
`PATH`	:/bin:/usr/bin	variable normally reset in `.profile`
`SHELL`	last-field-of-passwd-entry	`/bin/ksh`
`TERM`	your-terminal-type	`xterm`
`USER`	your-log-in-name	`goossens`

2.9 Command execution

To run a command (other than one built into the kernel) the shell first **creates a new process** using the system call `fork`. The **execution environment** for the command includes input, output and the states of signals, and is established in the **child process** before the command is executed. The built-in command `exec` is used in the rare cases when no `fork` is required and simply replaces the shell with a new command. For example, a simple version of the `nohup` command looks like

```
trap '' 1 2 3 15
exec "${@?}"
```

The `trap` turns off the signals specified so that they are ignored by subsequently created commands and `exec` replaces the shell by the command specified as arguments.

2.10 More on input-output redirection

Most forms of input output redirection have already been described. In the following `word` is only subject to parameter and command substitution. **No filename generation or blank interpretation takes place** so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is `*.c`. Input output specifications are evaluated left to right as they appear in the command. if `word` is the null string no redirection occurs.

Table 2.5: Syntax for command input-output redirection

Syntax	**Description**
`> word`	Use file `word` as standard output (file descriptor 1). If the file does not exist, create it; otherwise, truncate it to zero length.
`>> word`	The standard output is sent to file `word`. If the file exists then output is appended (by seeking to the end); otherwise the file is created.
`< word`	Use file `word` as standard input (file descriptor 0)
`<< word`	The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of `word` or `end-of-file`. If `word` is quoted then no interpretation of the document occurs. If `word` is not quoted then parameter and command substitution occur and \ quotes the characters `\`, `$`, `` ` `` and the first character of `word`. (see Page 6 for more details)
`>&digit`	The file descriptor `digit` is duplicated using the system call `dup` and the result is used as the standard output.
`<&digit`	The standard input is duplicated from file descriptor `digit`.
`<&-`	The standard input is closed.
`>&-`	The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>myfile
```

runs a command with message output (file descriptor 2) directed to `myfile`.

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

Within a shell procedure the standard input, output and error output can be redirected using the `exec` command e.g. to redirect standard output to `stdout`. and error output to `errout` one uses:

```
exec >stdout 2>errout
```

2.11 A Bourne shell grammar

Table 2.6: Definition of a Bourne shell grammar

<table>
<tr><th>Definition Term</th><th>Elements</th></tr>
<tr><td>item:</td><td>word
input-output
name = value</td></tr>
<tr><td>simple-command:</td><td>item
simple-command item</td></tr>
<tr><td>command:</td><td>simple-command
(command-list)
{ command-list }
for name do command-list done
for name in word ... do command-list done
while command-list do command-list done
until command-list do command-list done
case word in case-part ... esac
if command-list then command-list else-part fi</td></tr>
<tr><td>pipeline:</td><td>command
pipeline | command</td></tr>
<tr><td>andor:</td><td>pipeline
andor && pipeline
andor || pipeline</td></tr>
<tr><td>command-list:</td><td>andor
command-list ;
command-list &
command-list ; andor
command-list & andor</td></tr>
<tr><td>input-output:</td><td>> file
< file
>> word
<< word</td></tr>
<tr><td>file:</td><td>word
& digit
& -</td></tr>
<tr><td>case-part:</td><td>pattern) command-list ;;</td></tr>
<tr><td>pattern:</td><td>word
pattern | word</td></tr>
<tr><td>else-part:</td><td>elif command-list then command-list else-part
else command-list
empty</td></tr>
<tr><td>empty:</td><td></td></tr>
<tr><td>word:</td><td>Sequence of non-blank characters</td></tr>
<tr><td>name:</td><td>Sequence of letters, digits or underscores starting with a letter</td></tr>
<tr><td>digit:</td><td>0 1 2 3 4 5 6 7 8 9</td></tr>
</table>

Index