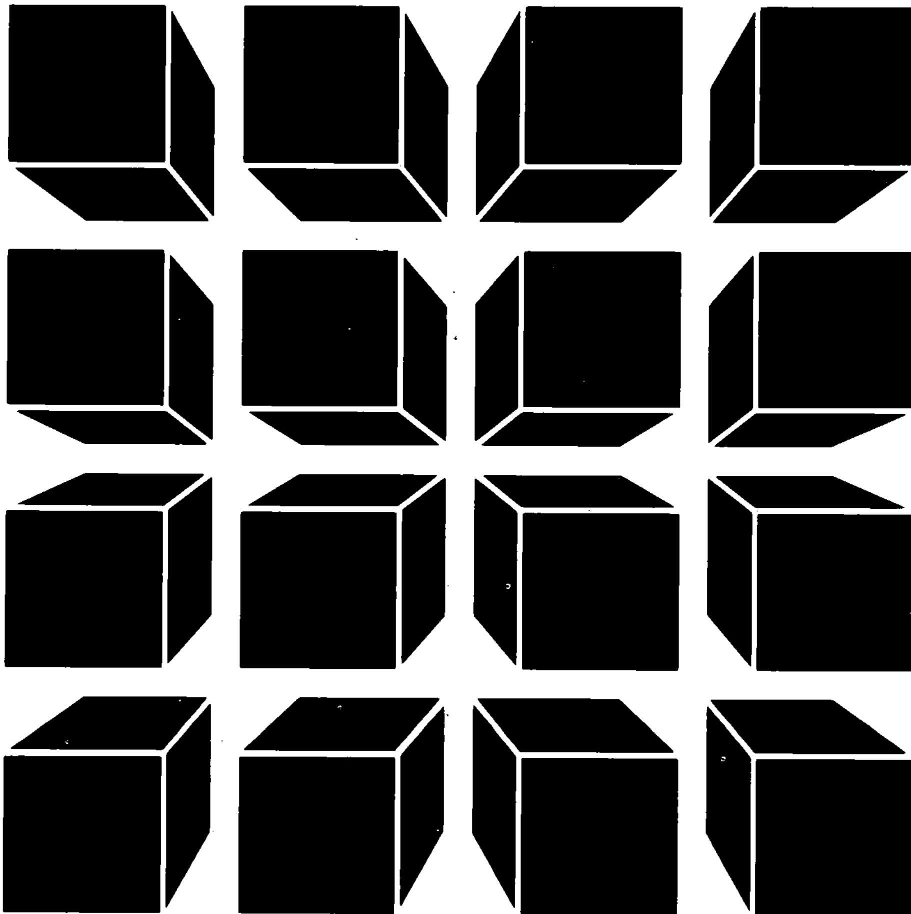
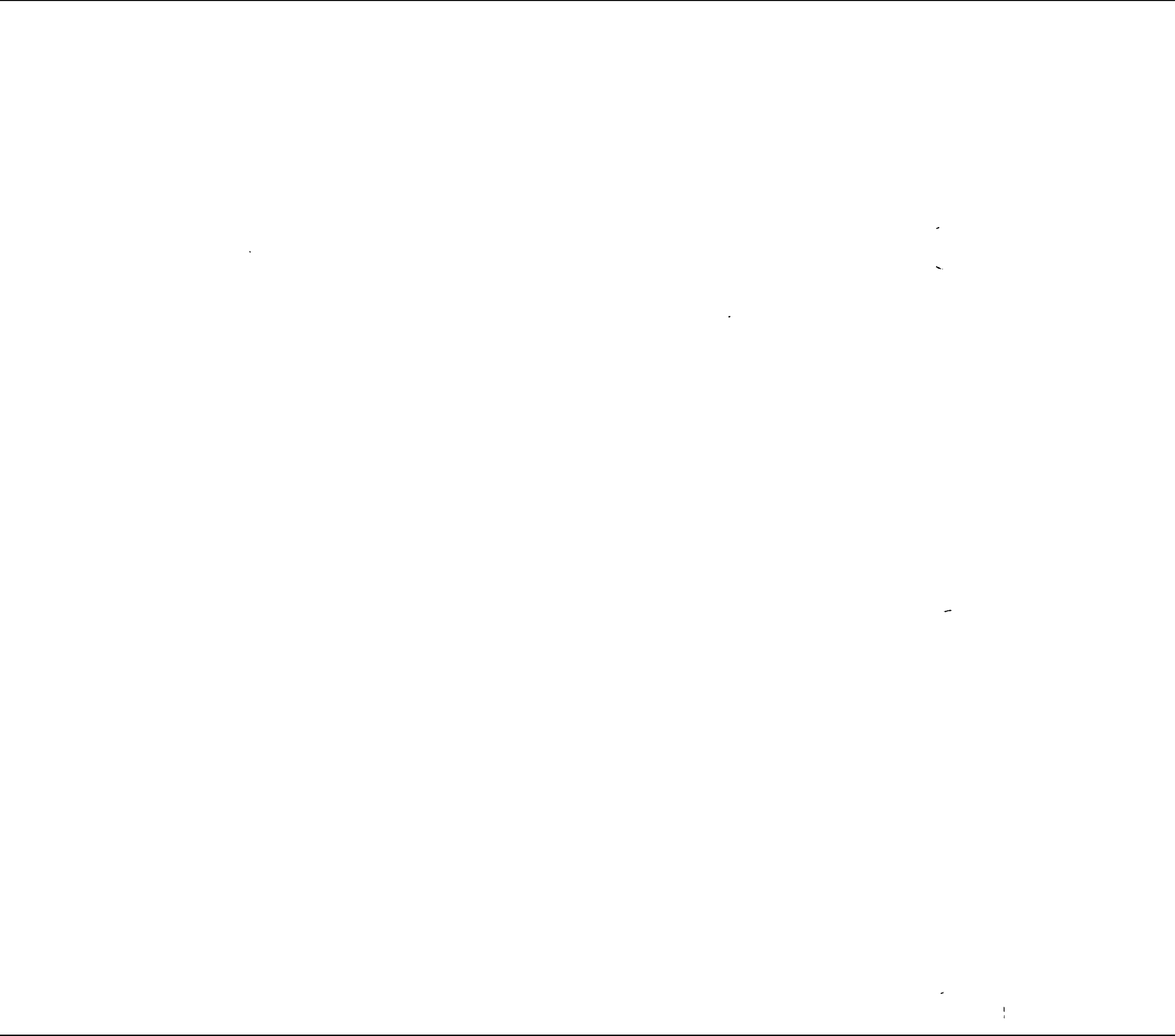
AT&T VIDEOTAPE LIBRARY

SHELL COMMAND LANGUAGE FOR PROGRAMMERS



WORKBOOK VOL I

AT&T



WORKBOOK VOLUME I

CONTENTS

INTRODUCTION i

VOLUME 1—Introduction to UNIX Shell Command Language

What You'll Be Able To Do After Volume 1 1-2
Shell Description - Command Interpreter 1-3
Compiled vs. Interpreted Languages 1-4
Steps Performed by /bin/login 1-5
/etc/passwd File 1-6
Set Up Default Environment 1-7
stty Command 1-10
Shell Reads Commands From a File 1-12
Shell Interprets The Command Line 1-13
Invoking Commands — Simple Execution 1-14
Sequential Execution 1-16
Command Groups — ( ) 1-17
Piplelines 1-18
Adding Execute Permission — chmod 1-20
The Path Variable 1-21
Redirection 1-22
here Document 1-23
here Document Example 1-24
Comments 1-25
Volume 1 Exercise 1-26
Volume 1 Exercise — Answers 1-27
Volume 1 Summary 1-28

VOLUME 2—The Shell Process

What You'll Be Able To Do After Volume 2 2-2
Process Definition 2-3
Creating New Processes 2-4
Foreground Processes 2-5
Background Processes 2-6
ps Command 2-7
wait Statement 2-8
kill Command 2-9

nohup Command 2-10
Invocation of Background Processes 2-11
Volume 2 Exercise 2-12
Volume 2 Exercise — Answers 2-13
Volume 2 Summary 2-14

VOLUME 3—Variables

What You'll Be Able To Do After Volume 3 3-2
Valid Variable Types 3-3
Declarations 3-4
Valid Variable Names — Rules 3-5
Reserved Names 3-6
Assignment Operator 3-7
String Terminators 3-8
echo Command 3-9
Variable Substitution 3-10
Positional Parameters 3-11
Positional Parameters — Examples 3-12
set Statement 3-13
Retaining Parameter Values 3-15
read Statement 3-16
read Statement — Example 3-17
export Statement 3-18
export Statement — Example 3-19
Scope of Variables 3-20
readonly Statement 3-21
Volume 3 Exercise 3-22
Volume 3 Exercise — Answers 3-26
Volume 3 Summary 3-32

VOLUME 4—Special Substitutions and Command Substitutions

What You'll Be Able To Do After Volume 4 4-2
Colon Operators 4-3
${var:-word} 4-4
${var:=word} 4-5
${var:=word} — Example 4-6
${var:?error} 4-7
${var:?error} — Example 4-8
${var:+error} 4-9

Omitting the : From Special Substitution Expressions 4-10
Command Substitutions — Purpose 4-11
Applications — Command Substitution — Example 4-12
Volume 4 Exercise 4-13
Volume 4 Exercise — Answers 4-16
Volume 4 Summary 4-19

VOLUME 5—Special Characters and Quoting

What You'll Be Able To Do After Volume 5 5-2
Special Characters 5-3
Single Quotes 5-6
Double Quotes 5-8
Double Quotes — Example 5-9
Backslash (\) Character 5-10
Backslash (\) Character — Example 5-11
Volume 5 Exercise 5-12
Volume 5 Exercise — Answers 5-13
Volume 5 Summary 5-14

VOLUME 6—Keyword Parameters and Preset Variables

What You'll Be Able To Do After Volume 6 6-2
How Keyword Parameters Are Used 6-3
Where Keyword Parameters Should Be Used 6-4
Keyword Parameters — Examples 6-5
Preset Shell Variables ${#} And #{?} 6-6
${?} — Examples 6-7
Preset Shell Variable ${$} 6-8
Preset Shell Variable ${!} 6-9
Preset Shell Variable ${*} 6-10
${*} — Examples 6-11
Preset Shell Variable${@} 6-12
${@} — Examples 6-13
Volume 6 Exercise 6-14
Volume 6 Exercise — Answers 6-15
Volume 6 Summary 6-17

VOLUME 7—Debugging Aids and Shell Efficiency

What You'll Be Able To Do After Volume 7 7-2
Explicit Invocation Of Shell Procedures 7-3
Verbose Trace 7-4
Verbose Trace — Example 7-5
Execution Trace.....................................7-6
Execution Trace — Example 7-9
Nonexecutable Shell 7-10
Nonexecutable Shell — Example.......................7-11
Unset Variable Exit 7-12
Unset Variable Exit — Example.......................7-13
Automatic Variable Export Shell 7-14
Automatic Variable Export Shell — Example...............7-15
Exit Immediate Shell 7-16
Exit Immediate Shell — Example 7-17
Shell Program Termination 7-18
Path Organization 7-19
Referencing Files...................................7-22
Built-in Statements 7-23
timex Command 7-25
Pipelines 7-27
Volume 7 Exercise...................................7-28
Volume 7 Exercise — Answers 7-29
Volume 7 Summary...................................7-30

VOLUME 8—Conditional Execution, Part 1

What You'll Be Able To Do After Volume 8 8-2
test Statement 8-3
test Statement — File Status Evaluation..................8-4
test Statement — File Status Evaluation — Example 8-5
test Statement — String Comparison 8-7
test Statement — String Comparison — Example 8-8
test Statement — String Size Testing 8-10
test Statement — String Size Testing — Example 8-12
test Statement — Numeric Comparison 8-13
test Statement — Numeric Comparison — Example 8-15
if Statement..8-17
if Statement — Example 8-19
exit Statement......................................8-22

exit Statement — Example 8-23
test Statement — The Test Condition 8-24
test Statement — Test Condition — Example 8-25
if-else Statement 8-28
if-else Statement — Exercises 8-30
Volume 8 Exercise 8-33
Volume 8 Exercise — Answers 8-35
Volume 8 Summary 8-38

WORKBOOK VOLUME II

CONTENTS

VOLUME 9—Conditional Execution, Part 2

What You'll Be Able To Do After Volume 9 9-2
if-elif-else Statement 9-3
if-elif-else Statement — Example 9-4
case Statement 9-6
case Statement — Patterns 9-8
case Statement — Example 9-10
tput 9-12
tput Capabilities 9-15
tput — Examples 9-19
sed Command 9-21
sed Command — Example 9-23
Volume 9 Exercise 9-24
Volume 9 Exercise — Answers 9-27
Volume 9 Summary 9-32

VOLUME 10—Looping Statement, Part 1

What You'll Be Able To Do After Volume 10 10-2
while Statement 10-3
while Statement — Logic 10-4
while Statement — Example 10-5
shift Statement 10-6
shift Statement — Example 10-7
getopt Command 10-10
getopt Command — Example 10-13

continue Statement 10-18
continue Statement — Example 10-20
break Statement 10-22
break Statement — Example 10-24
Volume 10 Exercise 10-26
Volume 10 Exercise — Answers 10-27
Volume 10 Summary 10-29

VOLUME 11—Looping Statement, Part 2

What You'll Be Able To Do After Volume 11 11-2
for Statement 11-3
for Statement — Example 11-4
expr Command 11-6
expr Command — Example 11-10
until Statement 11-12
until Statement — Example 11-14
line Command 11-16
line Command — Example 11-18
Volume 11 Exercise 11-19
Volume 11 Exercise — Answers 11-22
Volume 11 Summary 11-24

VOLUME 12—Command Line Interpretation

What You'll Be Able To Do After Volume 12 12-2
The Shell Command Line 12-3
Read Command Line 12-4
Verbose Trace Print 12-5
Variable Substitutions 12-6
Command Substitution 12-7
Command Substitution — Example 12-9
I/O Redirection 12-10
I/O Redirection — File Descriptor Table 12-11
I/O Redirection — Example 12-14
IFS Processing 12-18
Metacharacter Expansion 12-19
Execution Trace 12-23
Environment Processing 12-24
Execution Command 12-26

Volume 12 Exercise 12-27
Volume 12 Exercise — Answers 12-28
Volume 12 Summary 12-29

VOLUME 13—Built-in Statements

What You'll Be Able To Do After Volume 13
null (:) Statements 13-3
break, continue, and cd Statements 13-9
eval Statement 13-10
eval Statement — Example 13-11
exec Statement 13-13
exit and export Statements 13-14
Functions 13-15
Functions — Example 1 13-19
Functions — Example 2 13-20
Hashing 13-22
hash Statement 13-23
Hashing — Statement 13-25
newgrp Statement 13-27
read and readonly Statements 13-29
type Statement 13-30
set Statement 13-32
unset and wait Statements 13-33
time, trap, ulimit, and umask Statements 13-34
shift and test Statements 13-35
Volume 13 Exercise 13-36
Volume 13 Exercise — Answers 13-39
Volume 13 Summary 13-42

VOLUME 14—Redirection

What You'll Be Able To Do After Volume 14 14-2
Redirection — Review 14-3
Writing To Standard Error 14-4
Accessing Other Files — Example 14-7
Redirection 14-8
Reading From Files — Example 14-9
Redirection — Example 14-11
Writing To Files 14-13
Opening Files 14-14

exec Statement — Example 1 14-16
exec Statement — Example 2 14-17
Volume 14 Exercise 14-19
Volume 14 Exercise — Answers 14-21
Volume 14 Summary 14-23

VOLUME 15—Pipelines and Signal Processing

What You'll Be Able To Do After Volume 15 15-2
Pipelines — Definition 15-3
pipe Operator 15-4
UNIX System pipe Files 15-6
Using Control Constructs As Filters 15-7
Piping To a while Loop — Example 15-9
Piping To Conditional Statements — Example 15-10
Piping To/From Constructs — Example 15-12
Piping With Groups 15-14
Signals 15-16
trap Statement 15-21
Ignoring vs. the Null Statement 15-24
Resetting Traps 15-26
Example Using Trap 15-28
trap Statement — Quoting Considerations 15-30
Volume 15 Exercise 15-32
Volume 15 Exercise — Answers 15-33
Volume 15 Summary 15-34

AT&T COMPUTER SYSTEMS EDUCATION PROGRAM INTRODUCTION TO STUDENT WORKBOOK

To the Student,

Welcome to the AT&T Videotape Library. We hope you will enjoy using the videotape lessons and this workbook. You will find the material both practical and easy to follow. You will be using what you learn in your daily work almost immediately.

If you haven't taken a class by videotape before, you are in for an exciting and fun experience, because this medium gives you control over the pace of your learning. Each videotape lesson is divided into segments which are clearly marked with colored panels, so you can find them easily while using fast forward or rewind. When you encounter new terms or concepts, you can immediately review and reinforce them by using the videotape player's rewind feature. If you find some material that is already familiar to you, simply use the player's fast forward button to jump ahead to the next segment. You can even stop the tape completely to take a break. The lesson will pick up right where you left off, and you won't miss a word.

This course, on The **Shell Command Language for Programmers**, will enable you to write, debug, and implement powerful and versatile programs using the UNIX® System shell and other utilities. The course examines the shell interpreter, the shell process, variables, redirection, and conditional constructs.

This workbook has been prepared as a support tool for this videotape course. Each chapter in the workbook follows one lesson. At any point in the lesson, you can stop the tape and use the workbook to review examples or descriptions. The workbook also contains summaries and reviews of what you learned in each videotape, and a job aid referring to new commands or concepts.

At the end of each videotape lesson, there are exercises that test your knowledge of the material you have just studied. If you have trouble with any of the exercises, it would be a good idea to go back and review that section of the lesson.

The videotapes, combined with this workbook, are designed to be complete and sufficient. However, if you have technical questions on the course contents that you cannot resolve after viewing the tape or reading this workbook, dial the **AT&T Telephone Support Line** at **1-800-247-1212, Extension 1001**, and one of our subject matter experts will return your call within twenty-four hours. The Telephone Support Line is available to you during the first six weeks following your purchase of the AT&T Videotape Library. If you are leasing the course, you have access to the Telephone Support Line for the duration of your lease.

Welcome to the world of the UNIX System, and the AT&T Videotape Library. Enjoy yourself.

VOLUME 1

Introduction to UNIX Shell Command Language

Copyright © 1987 AT&T

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 1

- Describe the difference between a compiled and interpreted language and categorize the shell as one of the two
- Describe the actions of the login procedure
- Describe the default environment as set up by the login procedure
- List the actions that the shell performs on a command line
- Execute commands using simple and sequential execution, command groups, and pipelines
- Add execute permission to a shell procedure
- Describe the function of the **PATH** variable
- Use redirection to change the destination of output of a command or change the source of input to a command
- Use a here document to supply standard input to a command from within a shell procedure
- Use comments in a shell procedure.

Copyright © 1987 AT&T

SHELL DESCRIPTION — COMMAND INTERPRETER

- WRITTEN BY S. BOURNE
- COMMAND INTERPRETER
  - PROMPTS USER FOR INPUT (**$**)
  - READS COMMANDS FROM A FILE
  - PERFORMS INTERPRETATIONS AND SUBSTITUTIONS
  - EXECUTES USING *UNIX* SYSTEM PRIMITIVES
- PROGRAMMING LANGUAGE
- */bin/sh*

Version 1.0.0
Copyright © 1987 AT&T

COMPILED vs. INTERPRETED LANGUAGES

INTERPRETED LANGUAGES

- ACTIONS
  - LINES ARE READ FROM SOURCE FILE
  - LINES INTERPRETED
  - LINES EXECUTED
- GENERALLY RUN SLOWER THAN COMPILED PROGRAMS
- RELATIVELY SHORT DEVELOPMENT CYCLE.

COMPILED LANGUAGES

- READ ONCE FROM SOURCE FILE BY COMPILER
- COMPILER PRODUCES A LOADABLE FILE (MEMORY IMAGE)
- PROGRAMS RUN FASTER THAN INTERPRETED
- RELATIVELY LONG DEVELOPMENT CYCLE.

Copyright © 1987 AT&T

STEPS PERFORMED BY */bin/login*

READ LOGIN NAME

The login name is passed from */etc/getty* or is read from the standard input.

READ PASSWORD

The password is read, but not echoed. This password will be properly encrypted and checked against the entry in the file */etc/passwd*.

READ DIAL-UP PASSWORD

On some systems, at the option of the system administrator, a dial-up (or external security) password may be asked for if dialing from an outside line. This password is accepted like the password, not echoing the input.

CHECK PASSWORDS AGAINST LOGIN

If the login name matches the password and the dial-up password (if required) is correct, the login procedure continues; otherwise, it generates the message *login incorrect* and prompts for *login* again.

Copyright © 1987 AT&T

/etc/passwd FILE

Many shell programs need to reference information about the users who are using the program. Much of this information is found in the file */etc/passwd*. The format of the */etc/passwd* file is as follows:

- Login name (alphabetic).
- Encrypted password.
- Numeric user id (used by the system).
- Numeric group id (*/etc/group* contains the mapping between this number and the group name).
- The fifth field is an information field. Usually the system administrators use this to record such things as telephone extension, department number, bin number, or whatever they would like.
- Initial working directory (*HOME*) — default destination of the **cd** statement.
- Shell path name (default */bin/sh*) — This is the name of the shell interpreter that is used in place of the standard shell. This might be the restricted shell, for example.

Copyright © 1987 AT&T

SET UP DEFAULT ENVIRONMENT

The environment consists of sets of values assigned to variables. The default values for these variables are changeable by the system administrators. The standard defaults are as follows:

HOME=your_login_directory

HOME is set to the full path name of your login (or **HOME**) directory. This is made the current directory when you login, and is the default directory when using the **cd** statement. This information is taken from the entry in the file */etc/passwd*. If this information is incorrect, then the system administrator must be contacted to have it changed.

PATH=:/bin:/usr/bin

This contains the command search path. It is set to a series of path names separated by colons (:). When you invoke a shell or compiled program, the shell uses this search path to find the file being invoked.

SHELL=last_field_of_passwd_entry

This entry may be set by the system administrator to the path name of a shell interpreter other than the standard Bourne shell. In some cases, this might be a restricted version of the shell named */bin/rsh*. The information is supplied by the entry in the */etc/passwd* file and if not present there, defaults to the standard Bourne shell.

Copyright © 1987 AT&T

SET UP DEFAULT ENVIRONMENT

MAIL=/usr/mail/your_login_name

If set, the shell (before issuing a prompt) checks the mail file to see if any new mail has arrived since the last prompt. If it has, then it prints a message "**you have mail.**" Although this is set by default, some system administrators have removed it as a default setting for efficiency reasons.

TZ=timezone_specification

This is set to a series of abbreviations of the form **ttthsss**, where **ttt** is the time zone abbreviation, **h** is the hours from Greenwich Mean Time (GMT), and **sss** is the Daylight Savings Time abbreviation.

Copyright © 1987 AT&T

.profile

This file contains additional information supplied by the user. The commands in this file are executed sequentially and are generally used to tailor the environment. Commands in this file are those the user wants executed at login time.

EXAMPLE

```
$ cat .profile<CR>
stty erase '^h' echoe
PATH=/bin:/usr/bin:/ustg/hutch/bin:/ustg/hutch/caiac/bin:.
CDPATH=.:/ustg/hutch:/ustg/hutch/shell:/ustg/hutch/shell/doc
TERM=4424
PS1="${LOGNAME}: "
export PATH CDPATH TERM PS1
umask 023
mesg n
```

This example contains several variable settings, a resetting of the erase character, and an **export** statement. The methods used for setting variables and the **export** statement are discussed in Volume 3.

Copyright © 1987 AT&T

stty COMMAND

The **stty**(1) command is used to enable options on a terminal. These options may include erase and kill character changes, tab settings, carriage return, line feed delays, etc.

ERASE CHARACTER

The erase character is by default a sharp sign (#). To change this to something else, such as the backspace key or control-h, execute the following command line:

$ stty erase '^h' echoe<CR>

This changes the erase character from the sharp sign (#) to the backspace or control-h. (They are the same.) The **echoe** (echo erase) argument causes the erase to be echoed by displaying a backspace, a space, and another backspace; thus clearing the mistyped character from the screen. If a control character is desired, precede it by a caret (^) and enclose it in single quotes.

KILL CHARACTER

The default line kill character is @. To change this, use the **stty** command as follows:

$ stty kill '^x'<CR>

This sets the line kill character to control-x and makes the @ character a nonspecial character. Normally, a kill is echoed (a new line is displayed); if this is not desired, then the **-echok** (echo kill) option may be selected. In the same way as changing the erase character, the control key is specified by a caret (^) followed by the control character.

Version 1.0.0
Copyright © 1987 AT&T

stty COMMAND

TAB SETTINGS

For terminals without physical tab stops, use the **stty** command to set the tabs. Enter the command as follows:

$ stty -tabs<CR>

This causes the expansion of a tab character to the appropriate number of spaces, thus simulating tabs on a terminal without them.

INTERRUPT CHARACTER (NORMALLY DELETE)

$ stty intr '^C'<CR>

This sets the interrupt character to be <ctrl>C. The default interrupt character is <delete>.

Copyright © 1987 AT&T

SHELL READS COMMANDS FROM A FILE

- TERMINAL SPECIAL FILE (*/dev/tty...*)
  - — PROMPT FOR INPUT ($)
  - — OPTIONALLY NOTIFY OF NEWLY RECEIVED MAIL
- REGULAR FILE IN A *UNIX* SYSTEM
  - — SHELL PROCEDURES
  - — SHELL SCRIPTS
  - — COMMAND FILES
  - — SHELL PROGRAMS

Version 1.0.0
Copyright © 1987 AT&T

SHELL INTERPRETS THE COMMAND LINE

- READS COMMAND LINE (LINE INITIALLY PARSED) (**SP HT**)
- VARIABLE SUBSTITUTION
- COMMAND SUBSTITUTION
- I/O REDIRECTION
- WORD INTERPRETATION (LINE REPARSED)
- FILE NAME GENERATION EXPANSION (METACHARACTERS)
- VARIABLE ASSIGNMENTS
- COMMAND LOCATION

Version 1.0.0
Copyright © 1987 AT&T

INVOKING COMMANDS — SIMPLE EXECUTION

FORMAT

$ cmd -option argument ...<CR>

The first word on the command line is the command name, followed by options and arguments. By convention, options start with a plus (+) or minus (-). In some cases, options that begin with a minus (-) may be combined after one minus sign. However, this is entirely dependent on the command being executed.

EXAMPLES

$pr -5tw80 file.1<CR>

This command line prints the contents of *file.1* without the header (normally the **-t** option) in five columns, using a screen width of 80 columns.

Version 1.0.0
Copyright © 1987 AT&T

INVOKING COMMANDS — SIMPLE EXECUTION

These options can be designated separately, but for convenience, the **pr**(1) command permits grouping of options. In this course, the two methods of processing arguments are discussed in tape 10.

```
$ls -la          #dash shows options follow
                 #options are 'l' and 'a'

$stty -tabs      #terminal lacks hardware tabs

$stty tabs       #terminal can handle <ctrl>I

$set -x          #turn on (enable) execution trace

$set +x          #turn off (disable) execution trace

$pr -3           #print in 3 columns

$pr +3           #skip to 3rd page
```

MORAL

No set meaning to '+' or '-' with options!

Copyright © 1987 AT&T

SEQUENTIAL EXECUTION

Sequential execution involves a series of commands on a single line, separated by semicolons (;). Each command on the line is executed, one at a time, from left to right, without checking if the preceding command executed successfully.

FORMAT

$ command1 arg1 ; command2 -option2 argument2<CR>

This causes the first command to execute and, when finished, the second command to execute. Sequential execution is useful when a series of unrelated commands needs to be executed at one time. If they are executed using sequential execution, the operator does not have to wait until one finishes to ask for the second.

A semicolon may follow the last command, but is not necessary.

EXAMPLE

$pr encyclopedia ; pr dictionary >dict.out<CR>

In this example, the encyclopedia is typeset first, then the dictionary. The dictionary is typeset whether or not the encyclopedia **pr**(1) is successful. The output of the encyclopedia **pr** prints on the terminal; the output of the dictionary **pr** is placed in the file named *dict.out*.

Copyright © 1987 AT&T

COMMAND GROUPS — ( )

Command groups are used to group commands for either redirecting the output of the entire group or for putting a group of sequential commands in the background.

EXAMPLE 1

$ (pr encyclopedia > pr.ency ; pr dictionary > pr.dict) &<CR>

In this example, the two **pr** commands are done in order, from left to right, one at a time, but the group is put in the background.

EXAMPLE 2

$ (pr encyclopedia ; pr dictionary) > pr.out & <CR>

In this example, both commands are run in the background and the output of both are put in the file named *pr.out*.

In loose terms, a process is a program being executed. A more complete description of a process is in Volume 2.

In the example in the visual, the command lines are broken up over two lines. After the first line is entered, the shell prompts with prompt >. The shell issues this prompt (the secondary prompt) whenever it detects that something has started that has not finished. In this example, a command group has started and not finished with a **)**.

Copyright © 1987 AT&T

PIPELINES

Pipelines are command lines that involve the pipe operator. The pipe operator causes the output of the command to the left of the operator to be used as the input to the command to the right.

DEFINITION

source | filter | filter | sink

The parts of a pipeline are referred to as follows:

- The first command is referred to as the source. The source of a pipeline generates output and writes the output to the standard output.
- Commands between pipe operators are called filters. Filters accept input from the standard input and generate output to the standard output. Many shell tools are written so they may be used as filters. In the *UNIX System V User's Manual*, a filter is described by a sentence such as:

  "If no input file is supplied, the **xxxxx** command will read from the standard input."

  You should not assume that any command that normally uses a file name as its input reads from the standard input if that name is omitted.

  Typical filters include the commands **grep**(1), **sort**(1), **sed**(1), and **cut**(1). These commands, if not supplied with a file name, take their input from the standard input.
- The last command in the pipeline is called the sink. The sink may or may not generate output, but it must accept input from the standard input.

Copyright © 1987 AT&T

PIPELINES

Pipelines are one of the unique features of the UNIX operating system and are useful when programming in shell. The implementation and other uses of pipelines are discussed in Volume 14.

Example:

```
$ grep jab /etc/passwd | cut -d':' -f5
```

This command line selects lines which contain the pattern *jab* from the file */etc/passwd* and passes them to the cut command, which then prints only the fifth (5) field as delimited by colons (:).

Copyright © 1987 AT&T

ADDING EXECUTE PERMISSION — chmod

- USED TO CHANGE PERMISSIONS

ABSOLUTE MODE

chmod mode file<CR>

$ **chmod 754 laserit**<CR>

SYMBOLIC MODE

chmod [**ugo**]_**op**_**permission file**<CR>

$ **chmod ug+x shell.prog**<CR>

DEFAULT FOR [**ugo**] IS EVERYONE

Version 1.0.0
Copyright © 1987 AT&T

THE PATH VARIABLE

The **PATH** shell variable is set to a series of directory path names separated by colons (:). A null directory path name anywhere in the **PATH** implies the current directory. Initially, the login procedure will set **PATH** to

:/bin:/usr/bin

The **PATH** variable is used by the shell to find command files. If **PATH** is left in its default condition, it causes the shell to look at the three places to find a command. The shell first looks in the current directory, then in the directory */bin*, then */usr/bin*. If it does not find any executable files in any of these directories, then the shell produces an error message

progname: not found

If you would like a specific instance of a shell or compiled program to be executed without the shell referencing the **PATH** variable setting, then you could supply the entire path name of the command or compiled file.

EXAMPLE

$ **/bin/sort unsorted** > **sorted**<CR>

sort is the name of the command found in the */bin* directory.

The **PATH** variable may be tailored by the user to contain other directories. This will cause the shell to search these additional directories for commands.

Version 1.0.0
Copyright © 1987 AT&T

REDIRECTION

Input and output may be redirected by using the <, >, and >> operators. The < operator is used to redirect input. The word that appears to the right of the redirection indicates from where the input should be redirected.

EXAMPLE 1

$ **update** < **transactions**<CR>

In this example, the command **update** requires transactions to be entered in from the terminal (standard input). By using the redirection operator, you are directing the shell to read from the file *transactions* instead of from the terminal.

Output may be redirected in one of two ways: by using the > or >> operators. If the single right angle bracket is used, the shell causes the output of the command to be placed in the designated file or replaces an existing file with that name.

EXAMPLE 2

$ **cb program.c** > **pretty.c**<CR>

This either creates a file named *pretty.c* and places the output of the **cb**(1) command in it or causes the **cb** program to overwrite the existing file named *pretty.c*. If the append operator (>>) is used, the shell causes the **cb** program to append to the end of an existing file named *pretty.c* or create the file if none existed.

Copyright © 1987 AT&T

here DOCUMENT

The here document is used to supply a command with input from within a shell procedure. Normally, this input comes from the terminal and is ended with a control-d (EOF). The here document allows one of these commands to be fed information within the procedure. An example of a command that expects input from the terminal (or standard input) is the **cat**(1) command. If the **cat** command is invoked without a file name, it expects input from the standard input.

FORMAT

command << **word**<CR>
input <CR>
input <CR>
word <CR>

word is used to "frame" the input stream to the command. It is important that the format matches the above format with respect to the placement of the << and the **word**. The closing **word** must appear on a line by itself and must be the only word on the line.

A "–" in front of the initial "framing word" allows both input and the closing "framing word" to be preceded by tabs. This is useful for readability within shell scripts.

Note: Leading Spaces are not allowed.

The framing word can be almost any string of characters, but is often the characters **EOF** or even a single **!**. If the initial "framing word" is preceded by a backslash (\) or surrounded by quotes, the input is not processed by the shell.

here DOCUMENT — EXAMPLE

```
$ cat menu<CR>
cat <<EOF
                Inventory Control System

        update - Update Inventory Master Files

        report - Inventory On-Hand Report

       reorder - Inventory Reorder Report
EOF

$ menu<CR>
                Inventory Control System

        update - Update Inventory Master Files

        report - Inventory On-Hand Report

       reorder - Inventory Reorder Report
$
```

This program causes the **cat** command to take its input from within the shell program. This result could have been achieved by printing the contents of an existing file, but by using the here document, only one file (the program file) is necessary.

Copyright © 1987 AT&T

COMMENTS

If a sharp sign (#) starts a word on a command line, then that word and the rest of the line are assumed to be a comment and are ignored by the shell.

EXAMPLE

```
$ cat la<CR>
################################################
#      This procedure will list the contents         #
#      of the current directory in 5 columns         #
################################################

ls | pr -5tw80

$ la<CR>
foilz    shell.c     test.sh unit.2  unit.4
outline skeleton    unit.1   unit.3

$
```

The above comments were enclosed completely in sharp signs only to make it look good. The only sharp signs necessary are those at the beginning of each line.

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 1 EXERCISE

1. What do you expect the output of this shell script will do?

```
$ cat mail.here
mail $LOGNAME <<!

To: unix/c group
Have you seen the latest
AT&T UNIX video tapes?
ha
!
```


Copyright © 1987 AT&T

VOLUME 1 EXERCISE - ANSWERS

1. The here document is used to supply the mail command with input from within the shell procedure **mail.here**.

```
$mail.here <CR>
You have mail
$
$mail <CR>
From ha Fri May  1 10:58 EDT 1987
To: unix/c group
Have you seen the latest
AT&T UNIX video tapes?
ha

?
```


Copyright © 1987 AT&T

VOLUME 1 SUMMARY

Command	Meaning	Example
stty	Enables certain terminal I/O options	$ stty erase '^h' echoe <RET>
ls	List files in current working directory	$ ls -l <RET>
chmod	Change permissions on file	$ chmod 755 file <RET> $ chmod u+x file <RET>

Term	Meaning	Example
here document	Used to supply a command with input from within a shell procedure.	cat << EOF mail - read and send UNIX mail pr - print files EOF

Copyright © 1987 AT&T

VOLUME 2

The Shell Process

Copyright © 1987 AT&T

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 2

- Define what a process is and how it is created
- Describe the attributes of a newly created shell process
- Describe the actions of the shell in processing shell procedures and compiled programs
- Invoke procedures as foreground and background processes.

Version 1.0.0
Copyright © 1987 AT&T

PROCESS DEFINITION

- EXECUTION OF PROCESS IMAGE
- PROCESS IMAGE
  - — PROGRAM
  - — ENVIRONMENT
  - — SET OF REGISTERS
  - — FILE DESCRIPTOR TABLE
  - — NAME OF CURRENT DIRECTORY
  - — OTHER INFORMATION
- SEVERAL PROCESSES MAY BE RUNNING AT THE SAME TIME

Copyright © 1987 AT&T

CREATING NEW PROCESSES

- CREATED BY **fork** SYSTEM PRIMITIVE
- NEW PROCESS IDENTICAL TO CALLING PROCESS
  - — ENVIRONMENT
  - — CURRENT DIRECTORY INFORMATION
  - — FILE DESCRIPTOR TABLE
- ASSIGNED UNIQUE PROCESS IDENTIFICATION (PID) NUMBER
- PID RANGE (0 TO 30,000)

Version 1.0.0
Copyright © 1987 AT&T

FOREGROUND PROCESSES

- A FOREGROUND (SYNCHRONOUS PROCESS) COMMAND ENTERED FROM THE TERMINAL CAUSES THE INTERACTIVE SHELL TO
  - INTERPRET THE COMMAND LINE
  - CREATE CHILD PROCESS AND HAVE CHILD EXECUTE COMMAND OR INTERPRET SHELL PROCEDURE
  - WAIT FOR CHILD PROCESS TO COMPLETE
  - PROMPT FOR NEXT COMMAND

Version 1.0.0
Copyright © 1987 AT&T

BACKGROUND PROCESSES

- CALLED ASYNCHRONOUS PROCESSES
- INVOKING SHELL DOES NOT WAIT FOR COMPLETION OF CHILD PROCESS
- BY DEFAULT, STANDARD INPUT IS SUPPLIED FROM */dev/null*
- WILL IGNORE INTERRUPT AND QUIT SIGNALS
- WILL NOT IGNORE HANGUP SIGNAL
- USUALLY HAVE INPUT AND OUTPUT REDIRECTED
- RUNS AT LOWER PRIORITY (NICE -4)

Copyright © 1987 AT&T

ps COMMAND

The **ps**(1) command is used to print out information about running processes. If executed without any options, it produces output as follows:

```
$ ps<CR>
    PID    TTY      TIME COMMAND
    291    tty12    0:06 sh
   7330    tty12    0:15 vi
   8803    tty12    0:00 sh
   8805    tty12    0:01 ps
```

PID
is the process identification number.

TTY
is the terminal special file number.

TIME
is the accumulated execution time for the process.

COMMAND
is the name of the command.

If the **-f** option is used, the **ps** command prints out a full listing of the status. The following is a sample of the output of the **ps** command when used with the **-f** option:

```
$ ps -f<CR>
UID      PID  PPID   C    STIME  TTY      TIME COMMAND
hutch    291     1   0 07:22:51  tty12    0:06 -sh
hutch   7330   291   0 10:49:45  tty12    0:18 vi unit.2
hutch   8828  7330   1 11:28:47  tty12    0:00 sh -c ps -f
hutch   8829  8828  55 11:28:53  tty12    0:01 ps -f
```

UID is the login name of the user who is executing the process.

PPID is the process identification number of the parent process.

C is processor scheduling information.

Copyright © 1987 AT&T

wait STATEMENT

- **wait** FOR CHILD BACKGROUND PROCESSES
- **wait** FOR SPECIFIC BACKGROUND PROCESS BY PROVIDING PROCESS IDENTIFICATION NUMBER AS ARGUMENT
- NO ARGUMENT CAUSES THE SHELL TO WAIT FOR ALL CHILD BACKGROUND PROCESSES
- WILL NOT WAIT FOR CHILD PROCESSES OF CHILD PROCESSES (GRANDCHILDREN)
- A **wait** FOR A PIPELINE WAITS FOR SINK

Version 1.0.0
Copyright © 1987 AT&T

kill COMMAND

- USED TO SIGNAL A PROCESS (DEFAULT ACTION TERMINATES RECEIVING PROCESS)
- ARGUMENT TO **kill** COMMAND IS PROCESS IDENTIFICATION NUMBER OF PROCESS TO BE SIGNALLED
- THE COMMAND LINE

  $ **kill 24173**<CR>

  SEND A SIGNAL TO PROCESS NUMBER 24173

  $ **kill 0**<CR>

  SIGNALS ALL PROCESSES OF THE CURRENT SHELL'S PROCESS GROUP
- IF USED WITH **-9** OPTION, UNCATCHABLE KILL

  $ **kill -9 4182**<CR>
  terminated
  $

Copyright © 1987 AT&T

nohup COMMAND

If the word **nohup** precedes any background command line, it causes the process that executes that command line to ignore the SIGHUP signal and ignore logoff (in addition to SIGINT and SIGQUIT).

If output is not redirected, then it is written into a file named *nohup.out* in the current directory. If the current directory does not give permission to create a file or if the file exists and write permission is denied, then the file is created in the login directory of the user.

EXAMPLE 1

$ **nohup mm long.file 2**>**errors** >**out&**<CR>

This command line runs the **mm** command in the background, ignoring the HANGUP, INTERRUPT, and QUIT signals. If the output or error output has not been redirected, that output is appended to a file named *nohup.out*. If the file did not exist, it is created in the current directory (if the user has permission to create files in the current directory).

EXAMPLE 2

$ **nohup mm long.file &**<CR>
Sending output to nohup.out

Since the standard output has not been redirected, the **nohup** sends this output to the file *nohup.out*.

Version 1.0.0
Copyright © 1987 AT&T

INVOCATION OF BACKGROUND PROCESSES

- INVOKED BY ENDING COMMAND LINE WITH AN AMPERSAND (**&**)
- SEVERAL BACKGROUND PROCESSES CAN BE RUNNING AT THE SAME TIME
- PROCESS IDENTIFICATION NUMBER RETURNED, USEFUL FOR BACKGROUND MANAGEMENT TOOLS
    - — **ps** PRINTS PROCESS STATUS
    - — **wait** WAITS FOR BACKGROUND PROCESS TO COMPLETE
    - — **kill** TERMINATES BACKGROUND PROCESS
    - — **nohup** IGNORES HANGUPS AND HANDLES OUTPUT

Copyright © 1987 AT&T

VOLUME 2 EXERCISE

1. Finish this process diagram where the activation of the **ps** command was started from within the **vi** editor (e.g., :!ps -f).

   $ vi unit.2<CR>

   ---

2. Which command is used to terminate a background process?

3. The nohup command is used to:

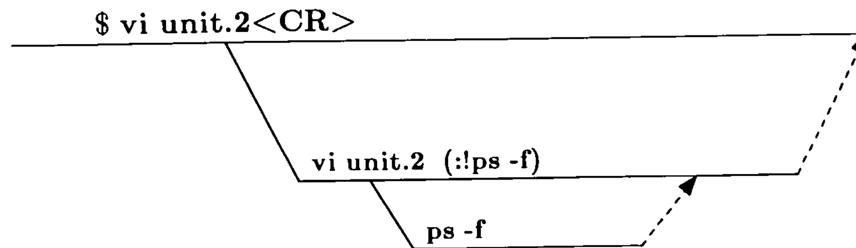
   1) kill a process
   2) hangup a process
   3) make a process immune of quit and interrupt
   4) none of the above.

4. What is the function of both ulimit and wait statements?

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 2 EXERCISE - ANSWERS

1.



2. $ kill PID or $ kill -9 PID (sure kill)

3. Answer is three 3)make a process immune of quit and interrupt.

4. ulimit: prints maximum file size in 512-byte blocks.

   wait: waits for all children background processes to terminate.

Copyright © 1987 AT&T

VOLUME 2 SUMMARY

Command	Meaning	Example
ulimit	If supplied with no arguments, prints maximum file size in blocks of 512 bytes.	$ ulimit <RET> 2048
ps	Prints information about running processes.	$ ps -f <RET>
wait	Wait for a background process to terminate.	$ wait PID <RET>
kill	Terminates background process.	$ kill -9 PID <RET>
nohup	Command line is executed ignoring disconnect and quit signals.	$ nohup mm docs& <RET> 3270 Sending output to nohup.out

Copyright © 1987 AT&T

VOLUME 3

Variables

Copyright © 1987 AT&T

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 3

- Describe the shell variable type and its characteristics.
- Formulate valid names for shell variables.
- Use the assignment operator.
- Use the **echo** statement to print messages to the standard output.
- Describe and use positional parameters.
- Describe how to reference the contents of a positional parameter.
- Use the **read** statement to assign values of variables.
- Combine the values of variables.
- Make variables accessible to other procedures.
- Describe the scope and duration of shell variables.
- Write-protect variables.

Version 1.0.0
Copyright © 1987 AT&T

VALID VARIABLE TYPES

- ONLY ONE TYPE — STRING VARIABLE
- MANIPULATION NOT LIMITED TO STRINGS
- CAN DO OTHER FUNCTIONS WITH STRING MANIPULATION TOOLS

EXAMPLE OF VALID VARIABLE NAMES

```
_Here
filename
BIN
Count_5
```

Version 1.0.0
Copyright © 1987 AT&T

DECLARATIONS

In languages where there may be several different types of variables, declarations are needed. These declarations may be implied by the name given to the variable (such as in BASIC or FORTRAN) or may be explicit (such as in Pascal or C or COBOL).

Declarations are used to inform the compiler of the properties of the variable that is going to be used and are a request to the compiler to save room for them (the declared variables). Declarations also tell the compiler or interpreter how to interpret the values in these variables.

For example, in C language when an integer is declared, the compiler reserves one machine word's worth of memory to store the value of the variable. It also knows how to interpret the contents of that word.

Since the shell only has one type of variable, it is not necessary to have declarations.

Copyright © 1987 AT&T

VALID VARIABLE NAMES — RULES

- NAME CAN BE ANY LENGTH
- ALL CHARACTERS SIGNIFICANT
- ALLOWABLE CHARACTERS
  - — ALPHABETIC
  - — NUMERIC
  - — UNDERSCORE
- MUST START WITH ALPHABETIC CHARACTER OR UNDERSCORE
- SPECIAL CHARACTERS NOT PERMITTED IN VARIABLE NAMES
- LOWER CASE LETTERS ARE CONSIDERED DIFFERENT FROM UPPER CASE LETTERS

Version 1.0.0
Copyright © 1987 AT&T

RESERVED NAMES

The following are names used by the shell and should not be used for any other purpose.

CDPATH

CDPATH is set to a colon-separated list of directory pathnames. This variable is used in conjunction with the **cd** statement. If this variable is set, and **cd** is invoked with an argument which does not begin with a "/" or a ".", the only directories searched for the directory you would like to change to are the directories listed in **CDPATH**. If the current directory is not listed in **CDPATH**, you **cannot** change directory to a directory that located directly "below" the current directory.

HOME

HOME is set to the path name of the login (or HOME) directory. This variable is set by the login procedure from the information taken from the user's */etc/passwd* entry.

The **HOME** variable is useful in making portable shell procedures. Instead of specifying an absolute path name, you could specify the file's relative position to the user's login directory.

IFS

IFS is used by the shell to separate a command line into words. The **IFS** variable is set by default to a space, tab, and newline by the login procedure.

LOGNAME

LOGNAME is set by the login procedure to the user's login name and is often used when a shell program needs to send a mail message to the user who is executing the program.

Copyright © 1987 AT&T

ASSIGNMENT OPERATOR

The assignment operator (=) is used to place a value in a shell variable. The assignment operator must not be surrounded by any spaces or tabs.

EXAMPLE 1

$ **directory=/ustg/hutch/doc**<CR>

EXAMPLE 2

$ **name=Hutchison**<CR>

Variables may be assigned a null value as follows:

$ **name=**<CR>

The assignment operator (=) is immediately followed by a newline.

The assignment operator causes the string that immediately follows the assignment operator to be assigned to the variable. Certain characters that may need to be included in the string may cause the termination of the string.

Copyright © 1987 AT&T

STRING TERMINATORS

- VALUE ASSIGNED TO VARIABLE MAY NOT CONTAIN:

  NEWLINE SPACE TAB ;) (| > < & $ ' " `

- IF REQUIRED, ENCLOSE STRING IN QUOTES

EXAMPLE 1

```
$ name="Hammond Aigs"<CR>
```

EXAMPLE 2

```
$ address="123 Disk Drive<CR>
> Grover's Mill, New Jersey"<CR>

$
```

Version 1.0.0
Copyright © 1987 AT&T

echo COMMAND

- ECHO IS BUILT IN TO THE SHELL
- WILL PRINT ITS ARGUMENTS ON STANDARD OUTPUT, SEPARATED BY SINGLE SPACES

EXAMPLE 1

```
$ echo I would like to be ambidextrous<CR>
I would like to be ambidextrous
```

EXAMPLE 2

```
$ echo *.c<CR>
ps.c name.c command.c graphic.c
```

- AFTER SHELL HAS DONE INTERPRETATIONS
- SPECIAL CHARACTERS

\n — Newline
\t — Tab
\b — Backspace
\c — No automatic newline

Version 1.0.0
Copyright © 1987 AT&T

VARIABLE SUBSTITUTION

- VALUE OF VARIABLE IS REFERENCED BY PLACING A DOLLAR SIGN ($) DIRECTLY BEFORE ITS NAME
- ENCLOSE NAME IN BRACES ({})

EXAMPLE

```
$ echo The value of name is ${name}<CR>
The value of name is Hammond Aigs

$
```

Version 1.0.0
Copyright © 1987 AT&T

POSITIONAL PARAMETERS

- USED TO PASS INFORMATION FROM THE COMMAND LINE TO SHELL PROCEDURES
- NAMES OF FIRST NINE POSITIONAL PARAMETERS ARE THE FIRST NINE DIGITS (1 THROUGH 9)
- OTHER POSITIONAL PARAMETERS CANNOT BE DIRECTLY REFERENCED
- MAY NOT APPEAR ON LEFT SIDE OF ASSIGNMENT STATEMENT
- ${**1**} IS VALUE OF FIRST POSITIONAL PARAMETER, ETC.
- ${**0**} IS NAME OF SHELL PROCEDURE

Version 1.0.0
Copyright © 1987 AT&T

POSITIONAL PARAMETERS — EXAMPLES

EXAMPLE 1

```
$ cat show.parms<CR>
echo the first  positional parameter is ${1}
echo the second positional parameter is ${2}
echo the third  positional parameter is ${3}
echo the fourth positional parameter is ${4}

$ show.parms this is a test<CR>
the first positional parameter is this
the second positional parameter is is
the third positional parameter is a
the fourth positional parameter is test
```

EXAMPLE 2

```
$ show.parms<CR>
the first positional parameter is
the second positional parameter is
the third positional parameter is
the fourth positional parameter is
```

Version 1.0.0
Copyright © 1987 AT&T

set STATEMENT

- **set** MAY BE USED TO CHANGE VALUES OF POSITIONAL PARAMETERS

EXAMPLE

```
$ cat reset<CR>
echo ${1} ${2} ${3} ${4}
set value1 value2
echo ${1} ${2} ${3} ${4}

$ reset This is an incomplete<CR>
This is an incomplete
value1 value2
```

- **set** WITH ARGUMENTS, WILL RESET ALL POSITIONAL PARAMETERS

Copyright © 1987 AT&T

set STATEMENT

- **set** STATEMENT WITHOUT ARGUMENTS PRINTS VARIABLE NAMES AND THEIR VALUES

EXAMPLE

```
$ set<CR>
CDPATH=.:/ustg/hutch:/ustg/hutch/shell
EDITOR=/usr/bin/vi
HOME=/ustg/hutch
IFS=

LOGNAME=hutch
MAIL=/usr/mail/hutch
PATH=/bin:/usr/bin:/usr/lbin:/std/bin:.
PS1=$
PS2=>
TERM=5420
TZ=CST6CDT
local=/ustg/hutch/shell/unit.1
```

Version 1.0.0
Copyright © 1987 AT&T

RETAINING PARAMETER VALUES

- VALUES OF POSITIONAL PARAMETERS MAY BE RETAINED AS FOLLOWS:

```
$ cat retainer<CR>
echo ${1} ${2} ${3} ${4}
set ${1} value1 value2 ${4}
echo ${1} ${2} ${3} ${4}

$ retainer This is an incomplete<CR>
This is an incomplete
This value1 value2 incomplete
```


Copyright © 1987 AT&T

read STATEMENT

FORMAT

$ **read variable**<CR>

$ **read var1 var2 ... varn**<CR>

- READS WORDS FROM STANDARD INPUT AND STORES IN VARIABLE
- IF MORE THAN ONE VARIABLE NAME
  - — FIRST WORD INTO FIRST VARIABLE
  - — SECOND WORD INTO SECOND VARIABLE, ETC.
  - — ANY EXTRA WORDS INTO LAST VARIABLE
  - — "WORDS" PARSED USING CHARACTERS IN **IFS** VARIABLE
  - — LEADING **IFS** CHARACTERS LOST UNLESS QUOTED
  - — ONLY QUOTING CHARACTER RECOGNIZED IS BACKSLASH
- READ CAN HAVE INPUT REDIRECTED

Copyright © 1987 AT&T

read STATEMENT — EXAMPLE

```
$ cat test.read<CR>
echo "Enter your name:"
read first middle last
echo ---------------------------------
echo first contains: ${first}
echo middle contains: ${middle}
echo last contains: ${last}
```

```
$ test.read<CR>
Enter your name:
Armond A. Legg is my name<CR>
------------------------------------
first contains: Armond
middle contains: A.
last contains: Legg is my name
```

Version 1.0.0
Copyright © 1987 AT&T

export STATEMENT

FORMAT

EXPORT VARIABLE_NAME . . .

- ALLOWS MULTIPLE SHELL PROCESSES TO SHARE VARIABLE NAMES AND VALUES
- SENT TO CHILD PROCESSES AND THEIR CHILDREN
- VALUES DIE WITH THE CHILD WHEN THE CHILD PROCESS DIES
- IF NO ARGUMENTS, PRINTS NAMES AND VALUES OF ALL VARIABLES EXPORTED BY THE CURRENT SHELL.

Version 1.0.0
Copyright © 1987 AT&T

export STATEMENT — EXAMPLE

```
$ cat exportation<CR>
name="Milly Ann Peare"
echo originally ${name}
export name
sub.export
echo finally ${name}
```

```
$ cat sub.export<CR>
echo received as ${name}
name="Polly Ester"
echo changed to ${name}
```

```
$ exportation<CR>
originally Milly Ann Peare
received as Milly Ann Peare
changed to Polly Ester
finally Milly Ann Peare
```


Copyright © 1987 AT&T

SCOPE OF VARIABLES

- SHELL VARIABLES ARE NOT GLOBAL

EXAMPLE

```
$ cat scope<CR>
echo This is ${0} running
variable='fatal error - try again'
echo :${variable}:
sub.scope

$ cat sub.scope<CR>
echo "This is ${0} running"
echo :${variable}:

$ scope<CR>
This is scope running
:fatal error - try again:
This is sub.scope running
::
```

- ONLY ACCESSIBLE BY CURRENT PROCESS UNLESS EXPORTED

Copyright © 1987 AT&T

readonly STATEMENT

- CAN WRITE-PROTECT A VARIABLE BY

  $ **readonly variable**<CR>

- WITHOUT ARGUMENTS, LISTS ALL **readonly** VARIABLE NAMES MADE READONLY BY THE CURRENT SHELL PROCESS

- ASSIGNMENT TO **readonly** VARIABLE CAUSES ERROR

EXAMPLE

```
$ cat protector<CR>
CAIAC=/std/caiac
readonly CAIAC
echo ${CAIAC}
CAIAC=/ustg/caiac
echo ${CAIAC}

$ protector<CR>
/std/caiac
protector: CAIAC: is read only
```

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 3 EXERCISE

1. An echo statement displays a menu on the terminal. Using echo statements, the menu should look something like this:

```
                UNIX SYSTEM V

          mail      Read or send news

      calendar      Reminder service

          text      Text processing tools

          prog      Programming

    file.maint      File subsystem maintenance

          acct      Accounting packages
```

   (The surrounding box is not necessary.)

   A. Name this procedure **menu** and place it in your current directory.

Copyright © 1987 AT&T

VOLUME 3 EXERCISE

2. Add a prompt message to the shell procedure you created in the last exercise. The message should look like this:

   Select one of the above:

   A. Now read the response into the variable named **option**.

   B. Print the contents of that variable as a verification. The verification should look like:

      The option you chose was "selection"

      "selection" is replaced by whatever was assigned to the variable named **option**.

Copyright © 1987 AT&T

VOLUME 3 EXERCISE

3. Create a shell program named **permissions** that will print the permissions and file names for all files in the current directory. This may be obtained using the **cut** command to "cut up" the output of the **ls -l** command. Print this output in three columns. Remember to discard the line containing the "total."

   A. Modify the program to accept the name of the directory to be listed as an argument.

4. Write a shell program named **whos_do** that takes the name of a command as an argument and prints all the logins that are executing that command. The output should be in five columns.

   EXAMPLE

   $ whos_do date

   ```
   cec21    cec18    jrc    cec3    tac
   cec34
   ```

   HINT: Use the **ps** command with appropriate options.

Copyright © 1987 AT&T

VOLUME 3 EXERCISES

5. Write a shell procedure which will display on the screen the following: The number of arguments entered are: Argl Arg2 Arg3. Name this procedure **disp**.

6.

```
$ cat rterm
TERM=5425
readonly TERM
echo $TERM
TERM=5620
echo $TERM
```

What do you expect the output to be when you run **rterm**?

Copyright © 1987 AT&T

VOLUME 3 EXERCISE - ANSWERS

1. $ **cat menu**<CR>

```
# program name: menu
#       author: programmer name
#    arguments: none
#      purpose: will print a UNIX (TM) System menu
#               on the terminal using  echo statements

echo '           UNIX SYSTEM V'
echo
echo '              mail     Read or send news'
echo
echo '          calendar     Reminder service'
echo
echo '              text     Text processing tools'
echo
echo '              prog     Programming'
echo
echo '        file.maint     File subsystem maintenance'
echo
echo '              acct     Accounting packages'
```


Copyright © 1987 AT&T

VOLUME 3 EXERCISE - ANSWERS

```
$ menu<CR>
        UNIX SYSTEM V

        mail    Read or send news

    calendar    Reminder service

        text    Text processing tools

        prog    Programming

  file.maint    File subsystem maintenance

        acct    Accounting packages

$
```


Copyright © 1987 AT&T

VOLUME 3 EXERCISE - ANSWERS

2. $ **cat menu**<CR>

```
# program name: menu
#       author: programmer name
#    arguments: none
#      purpose: will print a UNIX (TM) System menu on
#               the terminal using  echo statements.
#               Will then prompt the user for a
#               selection, read that selection,
#               and print it as a verification.

echo '          UNIX SYSTEM V'
echo
echo '           mail     Read or send news'
echo
echo '       calendar     Reminder service'
echo
echo '           text     Text processing tools'
echo
echo '           prog     Programming'
echo
echo '     file.maint     File subsystem maintenance'
echo
echo '     acct           Accounting packages'
echo "Select one of the above: \c"
read option
echo The option you chose was ${option}
```


Copyright © 1987 AT&T

VOLUME 3 EXERCISE - ANSWERS

```
$ menu<CR>
          UNIX SYSTEM V

            mail    Read or send news

        calendar    Reminder service

            text    Text processing tools

            prog    Programming

      file.maint    File subsystem maintenance

            acct    Accounting packages

Select one of the above: calendar
The option you chose was calendar

$
```


Copyright © 1987 AT&T

VOLUME 3 EXERCISE - ANSWERS

3. **$ cat permissions**<CR>

```
# program name: permissions
#       author: programmer name
#    arguments: directory to be listed (optional)
#      purpose: will print the names and permissions
#               of either the specified directory
#               or the current directory
ls -l ${1} | cut -c2-12,55-70 | tail +2 | pr -3tw80
```

$ permissions ${HOME}/unit1<CR>

```
rwxrwxr-x  d        rwxrwxrwx  logterm1  rwx------  rdir
rwxrwxr-x  dec      rwxr-xr-x  mailer    rwx------  r1
rwxrwxr-x  dir      rwxrwxr-x  pa        rwxrwxr-x  st
rwxrwxr-x  dirtree  rwxrwxrwx  print     rwx------  trp
rwxrwxr-x  doc      rw-rw-r--  print2    rwxr-xr-x  when
rwx------  1        rwxrwxr-x  pris      rwxr-xr-x  why
rwxrwxrwx  logger

$
```

4. **$ cat whos_do**<CR>

```
ps -ef | grep "${1}" | cut -c1-8 | pr -5atw80
```

$ whos_do date<CR>

```
cec32    ctr    new    cec3

$
```


Copyright © 1987 AT&T

VOLUME 3 EXERCISE - ANSWERS

5. $ **cat disp**<CR>
   echo The number of arguments entered are: ${1} ${2} ${3}"

   $

6. The output is:
   5425
   rterm: TERM: is read only

Copyright © 1987 AT&T

VOLUME 3 SUMMARY

Statement	Meaning	Example
echo	Prints its arguments on standard output	$ echo Hello! <RET>
set	Prints variable names and their values	$ set <RET>
read	Reads words from standard input and stores in variable	$ read var1 <RET>
export	Allows children processes to share variable names and values	$ export TERM <RET>
readonly	Write protect a variable	$ readonly var1 <RET>

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 4

Special Substitutions and Command Substitutions

Copyright © 1987 AT&T

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 4

- Use special substitution expressions to substitute default values and set values
- Generate error messages using special substitution expressions.
- Use command substitution expressions to set variables
- Use command substitution expressions to supply statements and commands with the output of a command as arguments.
- Use command substitution expressions to set variables
- Use command substitution expressions to supply statements and commands with the output of a command as arguments.

Version 1.0.0
Copyright © 1987 AT&T

COLON OPERATORS

Special substitution allows you to substitute default values for unset variables, assign values to unset variables, and generate error messages when unset variables are accessed.

NULL(:) STATEMENT

The **null** statement will do nothing and return an exit status of zero. The important thing that happens is that the complete command line is evaluated. This means that if the command line contains any expressions that involve side effects, those side effects will take place. Two types of side effects are special substitution and command substitution. Both of these will be discussed.

EXAMPLE

```
$ :    Your login directory is ${HOME}
$
```


Copyright © 1987 AT&T

${var:-word}

- IF **var** IS NOT SET OR NULL, THEN THE VALUE IS VALUE OF **word**
- IF **var** HAS A NON-NULL VALUE, THEN VALUE IS VALUE OF **var**

EXAMPLE 1

$ **echo ${CDPATH}**<CR>

$ **echo CDPATH is ${CDPATH:-undefined}**<CR>
CDPATH is undefined

$ **echo ${CDPATH}**<CR>

EXAMPLE 2

$ **echo ${unset:-"More than one word"}**<CR>
More than one word

$

Version 1.0.0
Copyright © 1987 AT&T

${var:=word}

- IF VARIABLE NAMED **var** HAS A NON-NULL VALUE, THE VALUE OF SPECIAL SUBSTITUTION EXPRESSION IS VALUE OF **var**
- IF VARIABLE NAMED **var** IS NULL OR NOT SET, THE VALUE OF SPECIAL SUBSTITUTION EXPRESSION IS **word**
- IN ADDITION, VARIABLE NAMED **var** IS ASSIGNED VALUE OF **word**. POSITIONAL PARAMETERS MAY NOT BE USED FOR "**var**" IN EXPRESSIONS OF THIS TYPE.

Copyright © 1987 AT&T

${var:=word} — EXAMPLE

```
$ cat use.defaults<CR>
selection=${1}
:  ${selection:=main}
echo Your selection is ${selection}
:  ${name:=VACANT}
echo Your name is ${name}

$ use.defaults programming<CR>
Your selection is programming
Your name is VACANT

$ name="Shanda Lear"<CR>

$ use.defaults docprep<CR>
Your selection is docprep
Your name is VACANT

$ export name<CR>

$ use.defaults<CR>
Your selection is main
Your name is Shanda Lear
```


Copyright © 1987 AT&T

${var:?error}

- IF VARIABLE NAMED **var** HAS A NON-NULL VALUE, THE VALUE OF SPECIAL SUBSTITUTION EXPRESSION IS VALUE OF **var**
- IF VARIABLE NAMED **var** IS NULL OR NOT SET, THE ERROR MESSAGE AS DEFINED BY **error** IS GENERATED IN THE FORM

```
var: error
```

AND CAUSES A NON-INTERACTIVE SHELL TO EXIT IMMEDIATELY

Version 1.0.0
Copyright © 1987 AT&T

${var:?error} — EXAMPLE

```
$ cat check.env<CR>
date
: ${TERM:?"should be set"}
who | wc -l
echo ${TERM}

$ echo ${TERM}<CR>
2621

$ check.env<CR>
Mon Sep 30 10:53:22 EST 1987
  21
2621

$ TERM=<CR>

$ check.env<CR>
Mon Sep 30 10:53:46 EST 1987
check.env: TERM: should be set

$
```

Version 1.0.0
Copyright © 1987 AT&T

${**var:+word**}

- IF VARIABLE NAMED **var** HAS A NON-NULL VALUE, THE VALUE OF SPECIAL SUBSTITUTION EXPRESSION IS VALUE OF **word**
- IF VARIABLE NAMED **var** IS NULL OR NOT SET, THE VALUE OF SPECIAL SUBSTITUTION EXPRESSION IS THAT OF THE NULL STRING
- THE VALUE OF **word** MAY BE A VARIABLE SUBSTITUTION EXPRESSION

EXAMPLE

```
$ echo ${BIN}<CR>
/ustg/hutch/bin
$ option=typing<CR>
$ echo ${BIN}/defs${option:+_${option}}<CR>
??
$ option=<CR>
$ echo ${BIN}/defs${option:+_${option}}<CR>
??
$ cat print.file<CR>
printer ${BIN:+"-b${BIN}"} ${FONT:+"-y${FONT}"} ${1}
```

Version 1.0.0
Copyright © 1987 AT&T

OMITTING THE : FROM SPECIAL SUBSTITUTION EXPRESSIONS

- THE SHELL CHECKS ONLY TO SEE IF VARIABLE HAS EVER BEEN SET
- IF VARIABLE IS CURRENTLY SET, SPECIAL SUBSTITUTIONS DO NOT TAKE PLACE

EXAMPLE

$ **echo ${option-main}** # **option was never set**<CR>
main

$ **option=hello**<CR>

$ **echo ${option-main}**<CR>
hello

$ **option=** # **option now has a null value**<CR>

$ **echo ${option-main}**<CR>

$ **echo ${option:-main}**<CR>
main
$

Version 1.0.0
Copyright © 1987 AT&T

COMMAND SUBSTITUTIONS — PURPOSE

- USED WHEN NECESSARY TO SUPPLY ARGUMENTS TO A COMMAND FROM THE OUTPUT OF ANOTHER COMMAND
- USED TO ASSIGN OUTPUT OF A COMMAND TO A VARIABLE

FORMAT

```
command argument `shell command`
```

- A COMMAND ENCLOSED WITHIN GRAVE ACCENTS (BACKWARD QUOTES)
- EXPRESSION IS REPLACED BY STANDARD OUTPUT OF COMMAND BETWEEN GRAVE ACCENTS
- MAY APPEAR ANYWHERE ON A COMMAND LINE
- DURING **IFS** PROCESSING, THE SHELL REPLACES **IFS** CHARACTERS WITH A SINGLE SPACE, CAUSING THE OUTPUT TO APPEAR AS ARGUMENTS

Copyright © 1987 AT&T

APPLICATIONS
COMMAND SUBSTITUTION — EXAMPLE

- EXAMPLE 1

```
$ echo The time is `date +%r`.
The time is 02:14:25 PM.
$
```

- EXAMPLE 2

```
$ users=`who |wc -l`

$ echo There are ${users} users on the system
There are 24 users on the system
$
```

- EXAMPLE 3

```
$ cat mailing.list<CR>
hutch
aigs
charles

$ mail `cat mailing.list` koffman<CR>
There is a group meeting
at the pool at 2:00 PM.
.<CR>

$
```

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 4 EXERCISE

1. Create a file containing two login names and name it **mail.list**. Make sure you include yourself in this list.

   A. Using command substitution, write a shell program that will send a mail message to the login names contained in **mail.list**. Name this program **ml**. Remember that a shell program may only contain command lines, not input to commands. If input is to be supplied from within a shell program, a here document must be used.

   B. Modify this program so it will take an argument and use it as a text file to be sent as the mail message.

   C. Modify this program so it accepts a second argument as the name of the mailing list file as a positional parameter. Use the file **mail.list** as the default mailing list file.

2. Write a shell procedure that will search files for a specified pattern. The files to be searched are all the regular files in the current directory and all of the regular files in any of its subdirectories. The pattern to be searched for should be specified as a positional parameter. Name this procedure **grep.down**.

   Note 1: The **find** command should be used with the **-type** option to find all regular files under the current directory.

   Note 2: The **grep** command should be used to search for the pattern.

VOLUME 4 EXERCISE

A. Modify this program to accept as the second positional parameter, the starting point from which to search. This would be a directory path name and would be supplied to the **find** command.

3. Write a shell procedure named **filect** that will print the number of files and subdirectories in the directory given as an argument. If there are no arguments, **filect** should count the number of files in the current directory.

   Example:

   $ **filect /bin**
   There are 91 files in /bin.

4. Write a shell program named **check.cdpath** that will print an error message if the variable **CDPATH** has no value or print the value of **CDPATH** if it has a value.

5. What is the output of the following statements.

   ```
   var=xyz
   echo ${var:+_$var}

   var=<CR>
   echo ${var:+_$var}
   ```

6. $ **cat removem<CR>**
   rm 'find ${1:-.} -type f -print'

   What is the ouput of this program if you typed

   $ **removem<CR>**

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 4 EXERCISE - ANSWERS

1. $ **cat ml<CR>**

```
#   ml - send mail messages to a group
#
#         Author: (programmer name)
#           Date: 7/7/87
#   Installation: Bell Labs
#       Keywords: none
#      Arguments: 1) the file_name of the message to be sent
#                 2) the mailing list file (optional)
#        Options: none
#
#  This program will...
#  send a message to login id's found in mail.list or any
#  file of login id's in position 2 and uses the message(s)
#  found in the file_name in position 1.

mail `cat "${2:-${HOME}/mail.list}"` < "${1}"
```

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 4 EXERCISE - ANSWERS

$ **cat mail.list**<CR>
hutch
hutch

$ **cat msg**<CR>
This is the message.

$ **ml msg**<CR>
you have mail

$ **mail**<CR>
From hutch Fri Jun 3 10:44 EDT 1987
This is the message.

? d
From hutch Fri Jun 3 10:44 EDT 1987
This is the message.

? d

Copyright © 1987 AT&T

VOLUME 4 EXERCISE - ANSWERS

2.
```
$ cat grep.down<CR>
# program name: grep.down
#         author: Sandy Beech
#      arguments: 1 - pattern
#                 2 - optionally the start point
#                     from which to search
#    exit status: that of the grep command
#       keywords: none
#        purpose: will search all of the files under
#                 the current directory for the
#                 specified pattern
grep "${1}" `find . -type f -print`

$ grep.down grep<CR>
./lab.01:who | grep ${1}
./lab.02:$ cat grep.down
./lab.02:# program name: grep.down
./lab.02:#  exit status: that of the grep command
./lab.02:grep "${1}" `find . -type f -print`
```

3.
```
$ cat filect
#program name: filect
#  programmer: student
#    arguments: 1 - directory name
echo There are `ls ${1} | wc -l` files in directory ${1:-`pwd`}.

$ filect ${HOME}
There are 14 files in directory /ustg/hutch.
```

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 4 EXERCISE - ANSWERS

4. $ **cat check.cdpath<CR>**
   echo ${CDPATH:?"is not set"}
   $ **echo ${CDPATH}<CR>**
   $ **check.cdpath<CR>**
   check.cdpath: CDPATH: is not set

   $ **CDPATH=.:${HOME} ; export CDPATH<CR>**
   $ **check.cdpath<CR>**
   .:/ustg/hutch

5. The value of the special substitution is _xyz.
   The value of the special substitution is the null string.

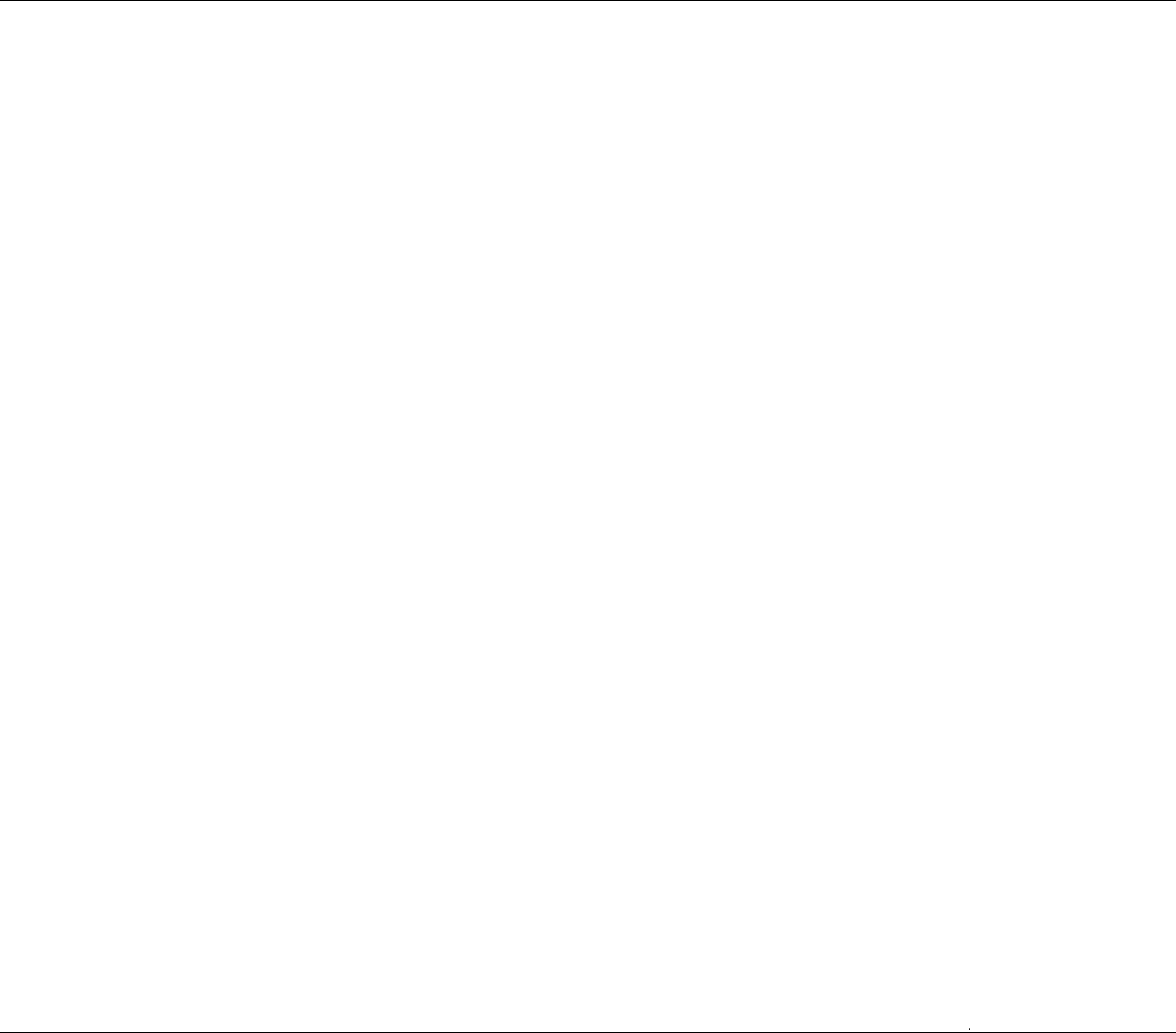
6. The **removem** program will remove all the regular files in your current directory.

Copyright © 1987 AT&T

VOLUME 4 SUMMARY

PARAMETER	VALUE OF SUBSTITUTION IF VARIABLE IS:		EFFECT
	SET TO NON-NULL	UNSET OR NULL	
${var}			Substitute value of var
${var:-*word*}	$var	*word*	Provides default value if none assigned.
${var:=*word*}	$var	*word*	Assigns default value if none assigned.
${var:?*error*}	$var	*error*	Aborts if no value assigned.
${var:+*word*}	*word*	$var	Allows value to be modified.

Copyright © 1987 AT&T



VOLUME 5

Special Characters and Quoting

Copyright © 1987 AT&T

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 5

- Use a backslash to remove the special meaning from a single character
- Use single quotes to remove the special meanings of all contained special characters
- Use double quotes to remove most of the special meanings of the contained characters.

Version 1.0.0
Copyright © 1987 AT&T

SPECIAL CHARACTERS

THE FOLLOWING CHARACTERS HAVE SPECIAL MEANINGS TO THE SHELL

- INVOCATION SPECIAL CHARACTERS
  - ; SEQUENTIAL EXECUTION
  - & BACKGROUND
  - (STARTS COMMAND GROUP
  -) ENDS COMMAND GROUP
- PIPELINES
  - | CREATES A PIPELINE
  - ^ CREATES A PIPELINE
- REDIRECTION
  - < REDIRECTS INPUT
  - > REDIRECTS OUTPUT
- SUBSTITUTION
  - $ VARIABLE SUBSTITUTION
  - ` COMMAND SUBSTITUTION

Copyright © 1987 AT&T

SPECIAL CHARACTERS

- FILE NAME GENERATION CHARACTERS

  * * MATCHES ANY NUMBER OF ANY CHARACTER
  * ? MATCHES ANY SINGLE CHARACTER IN A FILE NAME
  * [STARTS CHARACTER CLASS
  *] ENDS CHARACTER CLASS

- QUOTING CHARACTERS

  * \ REMOVES SPECIAL MEANING OF NEXT CHARACTER
  * ' REMOVES SPECIAL MEANING OF ALL CHARACTERS UP TO THE NEXT SINGLE QUOTE
  * " REMOVES SPECIAL MEANING OF ALL CHARACTERS UP TO THE NEXT DOUBLE QUOTE, EXCEPT $, ` AND \

Version 1.0.0
Copyright © 1987 AT&T

SPECIAL CHARACTERS

- INITIAL PARSING CHARACTERS

  <**NL**> ENDS COMMAND LINE

  <**SP**> SEPARATES WORDS ON COMMAND LINE

  <**HT**> SEPARATES WORDS ON COMMAND LINE

- COMMENT CHARACTER: (# BY DEFAULT)

- TO RETAIN LITERAL MEANINGS, MUST BE QUOTED

Copyright © 1987 AT&T

SINGLE QUOTES

- CHARACTERS ENCLOSED IN SINGLE QUOTES HAVE NO SPECIAL MEANING
- USED TO PROTECT PARTS OF A COMMAND LINE FROM SHELL PROCESSING

EXAMPLE

```
$ cat single.quotes<CR>
echo This \n line has \n several newlines
echo 'This \n line has \n several newlines'

$ single.quotes<CR>
This n line has n several newlines
This
    line has
    several newlines

$ cat single.example<CR>
login='`who | wc -l`'
echo '${login}'
echo ${login}

$ single.example<CR>
${login}
`who | wc -l`

$
```

Version 1.0.0
Copyright © 1987 AT&T

SINGLE QUOTES

EXAMPLE — grep COMMAND

```
$ cat outline<CR>
grep '\.H' ${1}

$ outline unit.9<CR>
.HU "Unit 9 - Quoting"
.HU "Overview"
.HU "Objectives"
.H 1 "special characters"
.H 1 "single quotes"
.H 2 "Example"
.H 2 "Example - the grep command"
   grep '\.H' ${1}
.H 2 "Example - the egrep command"
.H 1 "double quotes"
.H 1 "backslash"

$
```


Copyright © 1987 AT&T

DOUBLE QUOTES

- DOUBLE QUOTES TURN OFF THE SPECIAL MEANING OF MOST CHARACTERS
- CHARACTERS THAT RETAIN THEIR SPECIAL MEANING WHEN ENCLOSED IN DOUBLE QUOTES ARE

  $ INTRODUCES PARAMETER SUBSTITUTION

  ` INTRODUCES COMMAND SUBSTITUTION

  " ENDS QUOTED EXPRESSION

  \ WHEN FOLLOWED BY A $, `, ", OR ANOTHER \
- PARAMETER SUBSTITUTION AND COMMAND SUBSTITUTION ARE PERFORMED WITHIN A DOUBLE QUOTED EXPRESSION

Copyright © 1987 AT&T

DOUBLE QUOTES — EXAMPLE

```
$ cat double.quotes<CR>
echo The value of TERM is ${TERM}
echo "The value of TERM is ${TERM}"
echo The value of "*" is *

$ double.quotes<CR>
The value of TERM is 2645
The value of TERM is 2645
The value of * is unit.1 unit.2 unit.3
unit.4 unit.5 unit.6 unit.7 unit.8

$ me="`who am i`"

$ echo "${me}"
cec7        tty21        Mar 14 08:14
$
```

Version 1.0.0
Copyright © 1987 AT&T

BACKSLASH (\) CHARACTER

- MAY BE USED TO TURN OFF THE SPECIAL MEANING OF SINGLE CHARACTER THAT FOLLOWS IT
- HAS NO SPECIAL MEANING WHEN ENCLOSED IN SINGLE QUOTES
- WHEN ENCLOSED IN DOUBLE QUOTES, RETAINS ITS SPECIAL MEANING WHEN FOLLOWED BY

  $, `, ", \

Version 1.0.0
Copyright © 1987 AT&T

BACKSLASH (\) CHARACTER — EXAMPLE

```
$ cat backslash<CR>
echo This \* is an asterisk.
echo Enter your name: \\c
read name
echo ${name}

$ backslash<CR>
This * is an asterisk.
Enter your name: Armond Elig
Armond Elig
```


Copyright © 1987 AT&T

VOLUME 5 EXERCISE

1. Predict the output of the following command:
   $ **echo 'I can\'t do it'**<CR>

   >'

   Explain why the secondary prompt was issued by the shell.

Copyright © 1987 AT&T

VOLUME 5 EXERCISE - ANSWER

1. $ **echo 'I can\'t do it'**<CR>
   >'
   I can   do it

   $

   Once the first single quote was encountered by the shell, the backslash had no special meaning. Therefore the first single quote matched the second single quote, and the third single quote had no match. The shell wanted a matching single quote, so it prompted (>) for one. The **echo**(1) command received a literal backslash followed immediately by a "t" (\t). When **echo**(1) encountered a \t, it substituted a tab character.

Copyright © 1987 AT&T

VOLUME 5 SUMMARY

Quotation	Meaning
''	Single quotes turn off special meaning of all characters in them.
" "	Turn off special meaning of all characters except $, ‘, ", and \.
\	Turn off special meaning of single character which follows.

Copyright © 1987 AT&T

VOLUME 6

Keyword Parameters and Preset Variables

Copyright © 1987 AT&T

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 6

- Describe and use keyword parameters when invoking procedures
- Know when to use keyword parameters.
- Use the preset shell variables

  $**{#}, **${**?**}, **$**{**$**}, **$**{**!**}, **$**{*}, **$**{@}, and **$**{**-**}

  to find information about the shell environment.

Version 1.0.0
Copyright © 1987 AT&T

HOW KEYWORD PARAMETERS ARE USED

- APPEAR AS VARIABLE ASSIGNMENTS ON THE COMMAND LINE
- SHELL PLACES NAME-VALUE PAIR IN ENVIRONMENT OF CHILD PROCESS AND EXPORTS IT THERE
- MUST APPEAR BEFORE COMMAND NAME (UNLESS set -k IS USED)

FORMAT

$ **variable=value command -options arguments**<CR>

EXAMPLE

$ **dir=${LBHOME}/db dbname=polaris createdb**<CR>

Copyright © 1987 AT&T

WHERE KEYWORD PARAMETERS SHOULD BE USED

- NOT GENERALLY USED AS PART OF A USER INTERFACE
- APPEAR IN PROCEDURES CALLED BY OTHER PROCEDURES
- MAY BE USED FOR DEBUGGING

EXAMPLE

$ **TERM=4424 fancy.prog**<CR>

Version 1.0.0
Copyright © 1987 AT&T

KEYWORD PARAMETERS — EXAMPLES

```
$ cat laser.print<CR>
echo "Enter file_name: \c"
read file_name

echo "Enter font (vint): \c"
read font_style
: ${font_style:=vint}

echo "Enter mode (land): \c"
read print_mode
: ${print_mode:=land}

font=${font_style} mode=${print_mode} \
file=${file_name} laser

$ cat laser<CR>
: ${font:=vint} ${mode:=port}
: ${banner:=${LOGNAME}}
: ${copies:=1}
: ${user:=${LOGNAME}}
mmx -y${font} -p${mode} -j${banner} \
-c${copies} -u${user} ${file}

$
```


Copyright © 1987 AT&T

PRESET SHELL VARIABLES ${#} AND ${?}

${#}

- VALUE — NUMBER OF POSITIONAL PARAMETERS PASSED TO SHELL PROCEDURE
- USEFUL IN CHECKING IF PROPER NUMBER OF ARGUMENTS WERE SUPPLIED

${?}

- VALUE — EXIT STATUS OF LAST PROGRAM EXECUTED IN FOREGROUND
- GENERALLY 0 IF SUCCESSFUL
- BETWEEN 1 AND 255 IF UNSUCCESSFUL

Copyright © 1987 AT&T

${?} — EXAMPLES

EXAMPLE 1

$ **cat gonzo**<CR>
cat: cannot open gonzo

$ **echo ${?}**<CR>
2

$ **echo ${?}**<CR>
?

EXAMPLE 2

$ **grep 'not there' text.file**<CR>

$ **echo ${?}**<CR>
1

Copyright © 1987 AT&T

PRESET SHELL VARIABLE ${$}

- VALUE IS PROCESS IDENTIFICATION NUMBER OF PROCESS INTERPRETING COMMAND LINE
- EVERY PROCESS HAS A PROCESS IDENTIFICATION NUMBER
- USEFUL WHEN CREATING TEMPORARY FILES IN */tmp*

EXAMPLE

```
$ cat pretty.up<CR>
cb ${1} > /tmp/cb${$}
echo /tmp/cb${$}
mv /tmp/cb${$} ${1}

$ pretty.up program.c<CR>
/tmp/cb10964

$
```

Version 1.0.0
Copyright © 1987 AT&T

PRESET SHELL VARIABLE ${!}

- VALUE IS PROCESS IDENTIFICATION NUMBER OF LAST PROCESS PUT INTO BACKGROUND
- USEFUL FOR PARENT TO CHILD COMMUNICATION
- USEFUL WHEN TERMINATING A PROCESS WITH THE **kill** COMMAND

EXAMPLE

```
$ mm memo1&<CR>
1391

$ echo ${!}<CR>
1391

$ kill ${!}<CR>
mm: Terminated
```

Version 1.0.0
Copyright © 1987 AT&T

PRESET SHELL VARIABLE ${*}

- VALUE IS THE VALUE OF ALL POSITIONAL PARAMETERS
- IF ENCLOSED IN DOUBLE QUOTES, BECOMES ONE ARGUMENT

"${*}" is same as "${1} ${2} ${3} ... "

Copyright © 1987 AT&T

${*} — EXAMPLES

EXAMPLE 1

```
$ cat show.args<CR>
echo ${*}

$ show.args This is an incomplete<CR>
This is an incomplete
```

EXAMPLE 2

```
$ cat double.quote<CR>
cat ${*}
cat "${*}"

$ double.quote file.1 file.2<CR>
This is in file.1
This is in file.2
cat: cannot open file.1 file.2
```


Copyright © 1987 AT&T

PRESET SHELL VARIABLE ${@}

- VALUE IS THE VALUES OF ALL POSITIONAL PARAMETERS
- IF ENCLOSED IN DOUBLE QUOTES, APPEARS AS SEPARATE ARGUMENTS

```
"${@}" is same as "${1}" "${2}" "${3}" ...
```

Version 1.0.0
Copyright © 1987 AT&T

${@} — EXAMPLES

EXAMPLE 1

```
$ cat show.args<CR>
echo ${@}

$ show.args This is an incomplete<CR>
This is an incomplete
```

EXAMPLE 2

```
$ cat double.quote<CR>
cat ${@}
cat "${@}"

$ double.quote file.1 file.2<CR>
This is in file.1
This is in file.2
This is in file.1
This is in file.2
```


Copyright © 1987 AT&T

VOLUME 6 EXERCISE

1. Write a shell program that will achieve the same result as the shell procedure given below, but use keyword parameters instead of positional parameters. Make sure the parameters have been set. Name this procedure **kml**.

   ```
   mail `cat "${2:-${HOME}/mail.list}"` < "${1}"
   ```

2. Write a shell procedure named **mksent** that will print a five word sentence.

   i. The first two words of the sentence should be entered via keyword parameters from the command line.

   ii. The second two words should be entered via positional parameters.

   iii. The last word should be input via the read statement.

Copyright © 1987 AT&T

VOLUME 6 EXERCISE - ANSWERS

1. **$ cat kml**<**CR**>

```
#   kml - send mail messages to a group
#           Author: (programmer name)
#             Date: 7/7/87
#     Installation: Bell Labs
#         Keywords: list=mailing_list_name
#                   message=file_to mail
#        Arguments: 1) the file_name of the message
#                   2) the mailing list file (optional)
#          Options: none
#   This program will...
#     send a message to login id's found in mail.list or any
#     file of login id's in position 1 and uses the message(s)
#     found in the file_name in position 2.

: ${message:?"should be set"}
mail `cat "${list:-${HOME}/mail.list}"` <${message}
```

$ cat junk.file<CR>
"The stream editor should solve your problem," he said.

$ list=mail.list message=junk.file kml<CR>
you have mail

$ mail<**CR**>
From hutch Fri Jun 3 11:03 EDT 1987
"The stream editor should solve your problem," he said.

? d

$

Copyright © 1987 AT&T

VOLUME 6 EXERCISE - ANSWERS

2. **$ cat mksent**

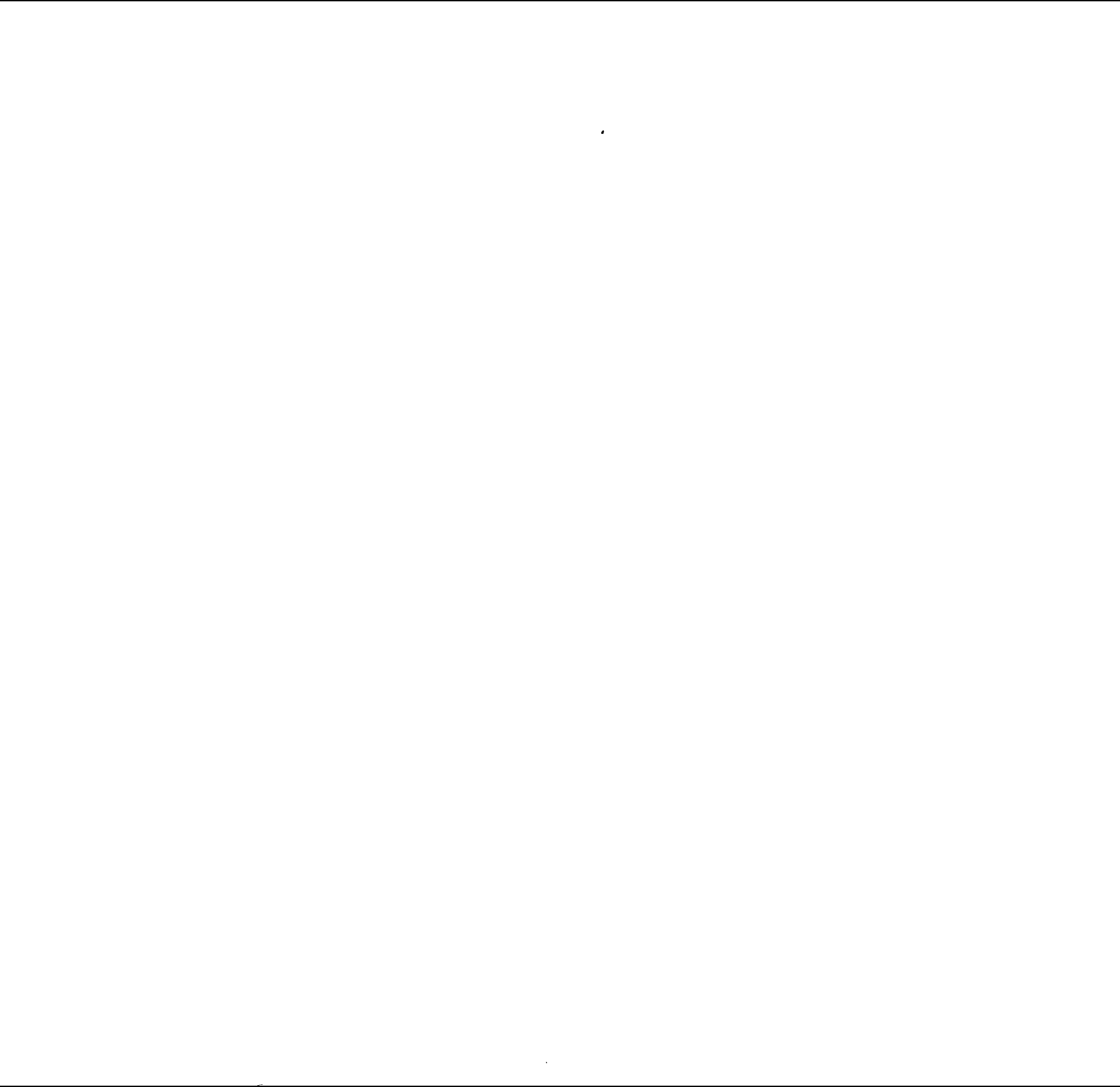
```
# program name: mksent
#    programmer: Sue Docode
#      arguments: 1 - third word
#                 2 - fourth word
#       keywords: word_1 - first word
#                 word_2 - second word
#
#   Prompt for last word.....

echo "Enter the final word: \c"
read last_word

echo ${word_1} ${word_2} ${1} ${2} ${last_word}

$ word_1=This word_2=is mksent my own
Enter the final word: sentence
This is my own sentence

$
```

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 6 SUMMARY

Preset Shell Variable	Meaning
${#}	Value of number of positional parameters passed to shell procedure
${?}	Value of the exit status of last program executed excluding background execution
${$}	Value of PID of process executing the command line
${!}	Value of PID of last process put in background
${*}	Value of all positional parameters. ($1, $2, etc.)
${@}	Value of all positional parameters. If enclosed in double quotes "${@}" same as "${1}" "${2}" ...
${-}	Value of current execution flag turned on for the shell procedure.

Copyright © 1987 AT&T



VOLUME 7

Debugging Aids and Shell Efficiency

Copyright © 1987 AT&T

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 7

- Invoke shell programs explicitly using the **sh**(1) command
- Use the verbose trace to watch the reading of a shell procedure by the shell
- Use the execution trace facility of the shell to watch the execution of a shell procedure
- Use a nonexecutable shell to harmlessly watch the execution of a shell procedure
- Use the unset variable exit shell to check the referencing of unset shell variables.
- Use the automatic variable export shell to automatically export shell variables.
- Recognize the **PATH** variable for efficiency
- Optimize referencing files in shell programs
- Know the built-in statements of the shell and why they are faster than nonbuilt-in commands
- Use the **timex**(1) command to benchmark shell programs
- Efficiently use pipelines to reduce data flow, thereby increasing efficiency of the pipeline

Version 1.0.0
Copyright © 1987 AT&T

EXPLICIT INVOCATION OF SHELL PROCEDURES

FORMAT

$ **sh file_name**<CR>

- EXECUTES **sh** COMMAND WITH *file_name* AS INPUT
- *file_name* MUST BE FULLY QUALIFIED PATH NAME OR RELATIVE PATH NAME OF SHELL PROCEDURE
- IF *file_name* IS OMITTED, WILL READ FROM STANDARD INPUT
- **PATH** IS NOT SEARCHED, WILL OVERRIDE BUILT-IN SHELL STATEMENTS
- FILE MODE DOES NOT NEED TO BE EXECUTABLE, BUT MUST BE READABLE

Copyright © 1987 AT&T

VERBOSE TRACE

The verbose trace feature of the shell permits you to watch the reading of a shell program. The shell, after a command line is read in, prints the command line (as read in) on the standard error output, permitting you to trace the execution of the program.

If the shell procedure is being explicitly invoked, give the **-v** option to the **sh** command as follows:

$ sh -v shell.procedure<CR>

This causes the verbose trace to be turned on for the duration of the execution of the shell program. It is also possible to turn on the verbose trace from within a shell procedure and then turn it off at some point. This is done using the **set** statement within the shell procedure as follows:

```
set -v
```

After the shell executes the **set** statement, the shell prints all lines as read until either the end of the program is reached or the statement

```
set +v
```

is encountered. This statement turns off the verbose trace, but shows up in the verbose trace (not executed until after the command line is printed).

The verbose trace option is not inherited by child shells. If the trace is turned on at the login shell level it only echoes the commands as they are typed in at the terminal, not in any shell programs which that shell process invoked. The verbose trace is a shell option. When new shell processes are started the options are not passed to the newly created shell. To trace a child shell they must be invoked explicitly or contain a **set -v** statement.

Copyright © 1987 AT&T

VERBOSE TRACE — EXAMPLE

```
$ cat traced<CR>
set -v
# This is a comment
date
echo ${TZ}
set +v        # Turn off verbose trace
date

$ traced<CR>
# This is a comment
date
Wed Apr 13 15:01:44 EST 1987
echo ${TZ}
EST5EDT
set +v        # Turn off verbose trace
Wed Apr 13 15:01:44 EST 1987

$
```

Version 1.0.0
Copyright © 1987 AT&T

EXECUTION TRACE

The execution trace facility of the shell permits you to watch the execution of a shell program. It causes the shell to print each command line immediately before execution, after the shell has done all of its processing, including parameter and command substitution and metacharacter expansions. A shell program can use the execution trace option two ways. The first involves the use of an explicitly invoked shell procedure as follows:

$ sh -x shell.procedure<CR>

If invoked this way, the shell causes all the command lines in the file *shell.procedure* to print just prior to their execution. Often, it is desirable to only trace part of a shell procedure. This is done using the **set** statement within the procedure as follows:

```
set -x
```

This causes the execution trace option to be turned on. It remains on until either the program ends or the following command line is encountered:

```
set +x
```

The use of these two command lines within a shell program permits you to trace only part of the program.

Version 1.0.0
Copyright © 1987 AT&T

EXECUTION TRACE

The shell prints the command lines on the standard error output and precedes those lines with a plus symbol (+). Execution trace and verbose trace may be combined by either invoking the command as follows:

```
sh -vx command.file
```

or by using the following command line within the shell program:

```
set -vx
```

Looping constructs and pipelines appear radically different in the two traces. These constructs are only read once, so the verbose trace only prints them once. Since the commands may be executed over and over (as with looping constructs) or the execution of the command may involve the separate execution of several commands (as with pipelines), the execution trace may appear to print the same command line more than once.

Copyright © 1987 AT&T

EXECUTION TRACE

FORMAT (ON COMMAND LINE)

$ **sh -x shell.procedure**<CR>

FORMAT (IN SHELL PROCEDURE)

set -x

- PRINTS COMMAND LINE AFTER SHELL HAS DONE ALL PROCESSING
- AFTER EXPANSIONS, SUBSTITUTIONS
- TRACED COMMANDS PRECEDED WITH "+ " (EXCEPT VARIABLE ASSIGNMENTS)
- MAY BE TURNED OFF WITHIN A SHELL PROGRAM BY

set +x

Version 1.0.0
Copyright © 1987 AT&T

EXECUTION TRACE — EXAMPLE

```
$ cat traced<CR>
set -x
# This is a comment
date
f=unit
echo ${f}
set +x       # Turn off execution trace
date

$ traced<CR>
+ date
Wed Apr 13 15:05:44 EST 1987
+ f=unit
+ echo unit
unit
+ set +x
Wed Apr 13 15:05:44 EST 1987

$
```

Version 1.0.0
Copyright © 1987 AT&T

NONEXECUTABLE SHELL

FORMAT (COMMAND LINE)

$ **sh -n command.file**<CR>

FORMAT (WITHIN SHELL PROCEDURE)

set -n

- READS COMMAND LINES BUT DOES NOT EXECUTE THEM
- USED IN CONJUNCTION WITH VERBOSE TRACE TO WATCH PROGRAMS HARMLESSLY
- **set +n** NOT AVAILABLE
- SHOULD NOT BE GIVEN TO LOGIN SHELL
- WILL REPORT CONTROL CONSTRUCT SYNTAX ERRORS

Version 1.0.0
Copyright © 1987 AT&T

NONEXECUTABLE SHELL — EXAMPLE

```
$ cat non.exec<CR>
set -v
set -n
variable=first
echo ${variable}
ls ¦ pr -5tw80

$ non.exec<CR>
set -n
variable=first
echo ${variable}
ls ¦ pr -5tw80

$
```

Version 1.0.0
Copyright © 1987 AT&T

UNSET VARIABLE EXIT

FORMAT (COMMAND LINE)

$ **sh -u command.file**<CR>

FORMAT (WITHIN SHELL PROCEDURE)

set -u

- CAUSES A NON-INTERACTIVE SHELL TO EXIT IMMEDIATELY IF UNSET VARIABLES ARE REFERENCED
- HELPFUL IN DEBUGGING — GLOBAL VARIABLE CHECK
- MAY BE TURNED OFF WITHIN A SHELL PROGRAM BY

  set +u

Version 1.0.0
Copyright © 1987 AT&T

UNSET VARIABLE EXIT — EXAMPLE

```
$ cat unset.var<CR>
set -v
set -u
name=first
cat ${nmae}.file
date
wc -l ${name}.file

$ unset.var<CR>
set -u
name=first
cat ${nmae}.first
unset.var: nmae: parameter not set

$
```


Copyright © 1987 AT&T

AUTOMATIC VARIABLE EXPORT SHELL

FORMAT (COMMAND LINE)

```
$ sh -a command.file<CR>
```

FORMAT (WITHIN A SHELL PROCEDURE)

```
set -a
```

- PERFORMS AUTOMATIC EXPORT OF SUBSEQUENTLY CREATED OR MODIFIED VARIABLES
- STILL CANNOT CARRY MODIFIED VALUES FROM CHILD TO PARENT
- OPTION IS INHERITED BY CHILD PROCESSES
- MAY BE TURNED OFF BY USING THE FOLLOWING WITHIN A SHELL PROGRAM:

```
set +a
```

Version 1.0.0
Copyright © 1987 AT&T

AUTOMATIC VARIABLE EXPORT SHELL - EXAMPLE

```
$ cat parent
var=new
child
var=change
child

$ cat child
echo "Value of var in child is ${var}"

$ parent
Value of var in child is
Value of var in child is

$ cat parent
set -a
var=new
child
var=change
child

$ parent
Value of var in child is new
Value of var in child is change
```

Version 1.0.0
Copyright © 1987 AT&T

EXIT IMMEDIATE SHELL

FORMAT (COMMAND LINE)

$ **sh -e command.file**<CR>

FORMAT (WITHIN A SHELL PROCEDURE)

set -e

- EXITS IMMEDIATELY IF ANY COMMAND EXITS WITH A NONZERO STATUS
- MAY BE TURNED OFF BY USING THE FOLLOWING WITHIN A SHELL PROGRAM:

set +e

Version 1.0.0
Copyright © 1987 AT&T

EXIT IMMEDIATE SHELL — EXAMPLE

$ **cat xit.im**<CR>

```
set -ex
date
cat not.there
date
```

$ **xit.im**<CR>

```
+ date
Mon Jun 13 10:03:28 EDT 1987
+ cat not.there
cat: cannot open not.there

$
```

Version 1.0.0
Copyright © 1987 AT&T

SHELL PROGRAM TERMINATION

- CAUSES A SHELL PROCEDURE TO TERMINATE
  - SYNTAX ERROR IN CONTROL CONSTRUCT
  - SIGNAL IS RECEIVED
  - FAILURE OF BUILT-IN STATEMENT (EXCEPT read AND test)
  - SYNTAX ERROR IN SPECIAL SUBSTITUTION
  - FAILURE OF ASSIGNMENT
  - ATTEMPTED EXECUTION OF A NON-EXISTING COMMAND
- WILL NOT CAUSE A SHELL PROCEDURE TO TERMINATE
  - I/O REDIRECTION FAILURE
  - COMMAND TERMINATES ABNORMALLY (CORE DUMP)
  - COMMAND TERMINATES NORMALLY BUT WITH NONZERO EXIT STATUS

Version 1.0.0
Copyright © 1987 AT&T

PATH ORGANIZATION

The **PATH** variable organization determines the speed that shell and other programs are found. The **PATH** variable is searched each time a nonbuilt-in command is executed. Poor organization of the **PATH** variable causes a directory or directories to be searched unnecessarily. A typical poor organization of the **PATH** variable is the default value assigned to **PATH**. It causes the current directory to be searched each time a command is executed. The question arises, "How many times is a program executed from the current directory?" "How many times are commands executed from */bin* or */usr/bin*?" The following setting of the **PATH** variable causes fewer directories to be searched if most of the commands are not in the current directory.

PATH=:/bin:/usr/bin:/ustg/hutch/shell #poor

PATH=/bin:/usr/bin:/ustg/hutch/shell:. #better

The time saved depends on the size of the current directory. If that directory is large, then the search time is large.

This also limits your ability to create programs in the current directory that have the same name as commands in either */bin* or */usr/bin*. If this practice cannot be avoided, put those commands in a small directory and that directory should be the first in the **PATH** variable. This causes only a small directory to be searched each time, and not a large current directory.

It may also be more efficient to use relative or absolute path names of command files. If the command name contains a slash, then the **PATH** variable is not searched and only one attempt is made to locate the command. This, however, is dangerous if the command is moved into another directory, for instance from */usr/bin* into */bin*. The shell program expecting it in */usr/bin* will no longer work.

Copyright © 1987 AT&T

PATH ORGANIZATION

Another practice that may result in command files being found with the minimum amount of **PATH** searching is to reset the variable **PATH** from within the shell program. This is possible because the shell program gets interpreted by a child process; and when that process finishes, the altered value of **PATH** dies with it.

Version 1.0.0
Copyright © 1987 AT&T

PATH ORGANIZATION

- WITHIN THE **PATH** VARIABLE, MOST COMMONLY USED DIRECTORIES SHOULD APPEAR FIRST
- RESET **PATH** WITHIN A SHELL PROGRAM TO A LIST OF DIRECTORY PATH NAMES USED IN THE PROGRAM
- HAVE CURRENT DIRECTORY SEARCHED AFTER THE STANDARD BINS

EXAMPLES

PATH=:/bin:/usr/bin:/ustg/hutch/shell #poor

PATH=/bin:/usr/bin:/ustg/hutch/shell:. #better

- USE RELATIVE OR FULL COMMAND PATH NAMES FOR COMMANDS IN A SHELL PROGRAM THAT ARE FREQUENTLY USED

Copyright © 1987 AT&T

REFERENCING FILES

When there is a need to reference many files in a certain directory, it is often better to change into that directory first. The operating system must do much work to interpret a long path name. For each name in the path name, the operating system reads in each of the directories specified and searches them for the next directory in the path. If a name is specified relative to the current directory, the operating system is able to reference it directly.

For example, the following are two methods for accessing all the files in the directory */ustg/hutch/shell/ex/exercises*. The first requires several long path names to be evaluated where the second will be able to access the files directly.

```
for file_name
in /ustg/hutch/shell/ex/exercises/*
do
  cat ${file_name}
done
```

```
cd /ustg/hutch/shell/ex/exercises
for file_name
in *
do
  cat ${file_name}
done
```

In this example, the second method is significantly faster and requires fewer system resources.

Copyright © 1987 AT&T

BUILT-IN STATEMENTS

Built-in statements are faster than those commands not built into the shell. There are three reasons why this is so:

1. A new process does not have to be created. This is probably the most significant reason, because the bulk of the time spent in executing a command is that of creating a new process. Memory has to be allocated, process creation tables need to be filled in, the shell interpreter needs to be copied, and then the file has to be executed.

2. Built-in statements do not cause the **PATH** to be searched. This is something you should be aware of, lest you create a shell program with the name **test**. In this case, when the program is executed, the shell executes the **test** statement and does not even look for a file with that name.

3. When a shell program is executed, the command file is opened, and the lines in the program are interpreted. In the case of a compiled program, the new program needs to be loaded into memory. Built-in statements only execute from the shell process they were called from.

Copyright © 1987 AT&T

BUILT-IN STATEMENTS

- SLOWNESS OF SHELL COMES FROM CALLING EXTERNAL PROGRAMS
- INCREASED USE OF BUILT-IN STATEMENTS IMPROVES EFFICIENCY
    - A NEW PROCESS IS NOT CREATED
    - **PATH** IS NOT SEARCHED
    - FILE DOES NOT HAVE TO BE EXECUTED
- EXAMPLE
  - USE **read** RATHER THAN **line**
  - USE **case** RATHER THAN **grep**

Copyright © 1987 AT&T

timex COMMAND

The **timex** command is used to benchmark the execution of a shell program. The **timex** command is invoked as follows:

$ timex command-name -options arguments<CR>

preceding the command name on the command line. The **timex** command prints the time the system took to execute the program in terms of:

— real: total elapsed time — this is the time (in seconds) from the invocation of the program until its completion.

— user: time executing the program — this is the time (in seconds) that the system took to execute routines in the user program, excluding the time that the operating system took to do its primitive operations (system calls).

— system: time spent executing routines in kernel address space — this is the time (in seconds) that the system took to perform the system primitives, such as memory allocation, certain I/O functions, etc.

The time is reported in hundredths of a second. There is another version of this program available named **time** that does the same thing but reports in tenths of a second.

Version 1.0.0
Copyright © 1987 AT&T

timex COMMAND

- USED TO TIME THE EXECUTION OF COMMANDS
- WILL REPORT
  - — REAL: TOTAL ELAPSED TIME
  - — USER: TIME EXECUTING THE PROGRAM
  - — SYSTEM: TIME EXECUTING ROUTINES IN KERNEL ADDRESS SPACE
- TIME IS REPORTED IN SECONDS, ACCURATE TO HUNDREDTHS OF A SECOND

EXAMPLE

```
$ timex who >/dev/null<CR>

real     0.61
user     0.24
sys      0.18


$
```

- **time** COMMAND IS SIMILAR, BUT REPORTS IN TENTHS OF A SECOND

Copyright © 1987 AT&T

PIPELINES

When formulating pipelines, it is often possible to rearrange the order of the filters to improve efficiency. The aim of doing this is to reduce the amount of information being processed. This may be done by putting reducing filters earlier in the pipeline than nonreducing filters. Reducing filters produce less output than they take in as input. An example of a reducing filter is the **grep** command, which only prints a subset of the input it receives. A nonreducing filter might be the **sort** command that produces the same volume of input as it receives.

An example of a pipeline that may be made more efficient is shown here along with the suggested improvement:

$ **who | sort | grep stu   # bad version**<CR>

$ **who | grep stu | sort   # better version**<CR>

The second example processes less information than the first. The second example needs to sort less information.

Copyright © 1987 AT&T

VOLUME 7 EXERCISE

1. To invoke a shell procedure explicitly, you need to use the ________ command.

2. When the __________ trace is invoked the shell prints the command lines on the standard error and precedes those lines with a + sign.

3. The___________ trace allows you to watch the reading of a shell procedure.

4. The command ____________ prints the time the system took to execute a shell procedure.

5. List two reasons why shell built-in statements are faster than nonbuilt-in commands.

Copyright © 1987 AT&T

VOLUME 7 EXERCISE - ANSWERS

1. To invoke a shell procedure explicitly, you need to use the **sh(1)** command.

   Example
   $ sh prog1 <CR>

2. When the **execution** trace is invoked the shell prints the command lines on the standard error and precedes those lines with a + sign.

3. The **verbose** trace allows you to watch the reading of a shell procedure.

   Example
   $ sh -v prog1 <CR>

4. The command **timex** prints the time the system took to execute a shell procedure.

5. The two reasons why shell built-in statements are faster than nonbuilt-in commands are:

   The first and most significant reason is that a child process does not have to be created.

   Second, built-in statements do not cause the **PATH** to be searched.

Copyright © 1987 AT&T

VOLUME 7 SUMMARY

Trace	Meaning
set -v	Verbose trace allows you to watch the shell procedure being read.
set -x	Execution trace allows you to watch the execution of the program.
set -n	Reads command lines but does not execute them.
set -u	Causes non-interactive shell to exit if unset variables are referenced.
set -a	Automatic export of subsequently created or modified variables.
set -e	Exit if any command exits with non-zero status.

Command	Meaning	Example
timex	Used to time the execution of a command	$ timex who <RET>

Copyright © 1987 AT&T

VOLUME 8

Conditional Execution, Part 1

Copyright © 1987 AT&T

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 8

- Use the **test** statement to check file status and comparison of strings.
- Use the **if** statement to conditionally execute a command line.
- Use the **exit** statement to terminate a procedure normally and abnormally.
- Use the **if-else** control construct to conditionally execute command lines.

Version 1.0.0
Copyright © 1987 AT&T

test STATEMENT

- CHECKS FILE STATUS
- COMPARES STRINGS
- COMPARES NUMERIC VALUES CONTAINED IN STRINGS
- RETURNS 0 IF TEST CONDITION IS TRUE
- RETURNS 1 IF TEST CONDITION IS FALSE
- ARGUMENTS TO **test** DESCRIBE CONDITION BEING TESTED
- USED WITH CONSTRUCTS

Copyright © 1987 AT&T

test STATEMENT — FILE STATUS EVALUATION

- TESTS REQUESTER'S PERMISSIONS AGAINST THOSE OF THE SPECIFIED FILE
- FILE NAME CAN COME FROM CAPABILITY OF THE SHELL THAT GENERATES CHARACTER STRINGS OR FROM THE COMMAND LINE
- FILE NAME CANNOT BE OMITTED
- IF FILE NAME IS GENERATED BY SUBSTITUTION REQUEST, ENCLOSE IN DOUBLE QUOTES
- FILE STATUS CONDITIONS

```
test -r    file_name  # exists and readable
test -w    file_name  # exists and writable
test -x    file_name  # exists and executable
test -s    file_name  # exists and nonzero size
test -f    file_name  # exists and ordinary
test -d    file_name  # exists and directory
```

Version 1.0.0
Copyright © 1987 AT&T

test STATEMENT — FILE STATUS EVALUATION

EXAMPLE

```
$ cat permissions<CR>
ls -ld ${*} | cut -c1-12,55-70

$ permissions unit.12<CR>
-rw-rw-r--  unit.12

$ test -x unit.12<CR>

$ echo ${?}<CR>
1

$ test -r unit.12<CR>

$ echo ${?}<CR>
0
```


Copyright © 1987 AT&T

test STATEMENT — FILE STATUS EVALUATION

EXAMPLE

```
$ permissions backup<CR>
drwxrwxr-x  backup

$ test -d backup<CR>

$ echo ${?}<CR>
0
```

- MAY COMBINE TESTS WITH THE LOGICAL OPERATORS **-a** (logical and) OR THE **-o** (logical or)
- ! WILL NEGATE CONDITION
- MAY USE PARENTHESES FOR GROUPING (NEED TO BE QUOTED)

```
$ test ! \( -d backup -a -x backup \)<CR>
```

Version 1.0.0
Copyright © 1987 AT&T

test STATEMENT — STRING COMPARISON

- TWO STRING COMPARISON TESTS
  - EQUALITY (=)
  - INEQUALITY (!=)
- OPERATORS MUST BE SURROUNDED WITH SPACES

FORMAT

```
test "string" = "string"

    or

test "string" != "string"
```


Copyright © 1987 AT&T

test STATEMENT — STRING COMPARISON

EXAMPLE

```
$ echo ${TERM}<CR>
2645

$ test "${TERM}" = "2645"<CR>
$ echo ${?}<CR>
0

$ test "${TERM}" = "2621"<CR>
$ echo ${?}<CR>
1
```


Copyright © 1987 AT&T

test STATEMENT — STRING COMPARISON

EXAMPLE

- ARGUMENTS TO **test** SHOULD USUALLY BE QUOTED
- IF NOT QUOTED, AND VALUE OF STRING IS NULL, STRING IS REMOVED FROM COMMAND LINE BY **IFS** PROCESSING

EXAMPLE

```
$ echo ${NOTSET}<CR>


$ test ${NOTSET} != "hello"<CR>
test: argument expected

$ test "${NOTSET}" != "hello"<CR>

$ echo ${?}<CR>
0
```


Copyright © 1987 AT&T

test STATEMENT — STRING SIZE TESTING

The **test** statement is also used to test the size of a character string. This is done by using the **-z** and **-n** options to the **test** statement to test for length of string zero and length of string nonzero, respectively.

FORMAT

```
test -z "string"

test -n "string"

test "string"
```

The tests for length are often used to see if a variable is set. This is either a shell variable or a positional parameter and is used in a conditional statement executing a command line or lines if the variable is set or unset. Examples are shown in the next lesson (the **if** statement).

If no options are specified to **test**, it assumes the **-n** option. Therefore, if **test** only sees one argument, it returns a true value if that argument has one or more characters in it.

Version 1.0.0
Copyright © 1987 AT&T

test STATEMENT — STRING SIZE TESTING

- OPERATORS:
  - — LENGTH OF STRING ZERO (**-z**)
  - — LENGTH OF STRING NONZERO (**-n**)
  - — IF NO OPTION, IMPLIES LENGTH OF STRING NONZERO

FORMAT

```
test -z "string"

test -n "string"

test "string"

test "-d file_name -a -f file_name"
```


Copyright © 1987 AT&T

test STATEMENT — STRING SIZE TESTING EXAMPLE

```
$ cat test.args<CR>
test -z "${1}"
echo ${?}

$ test.args hello there<CR>
1

$ test.args<CR>
0
```

Version 1.0.0
Copyright © 1987 AT&T

test STATEMENT — NUMERIC COMPARISON

- COMPARES INTEGERS AT BEGINNING OF STRING ONLY
- WHEN NONINTEGER IS ENCOUNTERED, NUMERIC VALUE IS DETERMINED

NUMERIC STRING CONVERSIONS

Character String	Arithmetic Value
"15"	15
"123 main st"	123
"132.43"	132
"4312786**"	4312786

Copyright © 1987 AT&T

test STATEMENT — NUMERIC COMPARISON

FORMAT

```
test "string1" -eq "string2"

test "string1" -ne "string2"

test "string1" -gt "string2"

test "string1" -ge "string2"

test "string1" -lt "string2"

test "string1" -le "string2"
```

- STRINGS ON EACH SIDE SHOULD CONTAIN NUMERIC INFORMATION
- OPERATORS MUST BE SURROUNDED WITH SPACES

Version 1.0.0
Copyright © 1987 AT&T

test STATEMENT — NUMERIC COMPARISON

EXAMPLE

```
$ test "`date +d`" -eq "1"<CR>
$ echo ${?}<CR>
1
```

It is important to surround all operators with spaces. If not, the shell **test** statement recognizes the rest of the command line as one argument and assumes that a length of string nonzero test is being performed on that string. Therefore, it is always true.

The **test** statement is frequently used as the test condition for an **if** statement. The shell **if** statement is discussed in Lesson 2.

Question: Are the following tests true or false?

```
$ test "123 Main St" -eq "123 Disk Drive"<CR>

$ test "123 Main St" = "123 Disk Drive"<CR>

$ test "123 Main St" -eq "0123 Disk Drive"<CR>
```

Question:

```
$ test "123 Main St" -eq "123 Disk Drive" <CR>
$ echo $?
0 True

$ test "123 Main St" = "123 Disk Drive" <CR>
$ echo $?
1 False

$ test "123 Main St" -eq "0123 Disk Drive" <CR>
$ echo $?
0 True
```


Copyright © 1987 AT&T

test STATEMENT — NUMERIC COMPARISON

EXAMPLE

EXAMPLE 1

```
$ test "`date +%d`" -eq "1"<CR>
$ echo ${?}<CR>
1
```

EXAMPLE 2

```
$ test "123 Main St" -eq "123 Disk Drive"<CR>
```

```
$ test "123 Main St" = "123 Disk Drive"<CR>
```

Version 1.0.0
Copyright © 1987 AT&T

if STATEMENT

Like most programming languages, the shell has several facilities to permit conditional execution. These facilities include the simple **if**, the **if-else**, and the **if-then-elif** constructs.

SYNTAX

```
if
   command
   command
   testing_command
then
   command
   command
   command
fi
```

All of the statements between the keyword **if** and the keyword **then** are unconditionally executed. If the exit status of the last of the commands in that list (the one that immediately precedes the keyword **then**) is true (has a value of 0), the commands between the keyword **then** and the keyword **fi** are executed. If the exit status of the command before the keyword **then** is false (not zero), execution resumes after the keyword **fi**.

Copyright © 1987 AT&T

if STATEMENT

- CONDITIONAL EXECUTION
- COMMANDS BETWEEN **then** AND **fi** ARE EXECUTED IF EXIT STATUS OF **testing_command** IS ZERO (0)

SYNTAX

```
if
    command
    command
    testing_command
then
    command
    command
    command
fi
```

Version 1.0.0
Copyright © 1987 AT&T

if STATEMENT — EXAMPLE

```
$ cat is.directory<CR>
if
   test -d ${1:?"File_name expected"}
then
   echo ${1} is a directory

fi
```

This example shows a typical application of the **if** statement where the conditional test is the **test** statement. The conditional test between the keywords **if** and **then** uses the special substitution expression for generating error messages if there are no positional parameters (arguments) supplied. In this case, if the positional parameter is not supplied, the diagnostic message

```
File_name expected
```

prints. If the argument is supplied and if the file is a directory (as tested by the **test** statement), the following message prints:

```
$ is.directory ${HOME}<CR>
/ustg/hutch is a directory
```


Copyright © 1987 AT&T

if STATEMENT -- EXAMPLE

An alternative to the **if** construct is the logical "and". The logical "and" has the format:

```
command_1 arg1 arg2 && command_2 arg1 arg2
```

The second command is executed if and only if the first command exits with a "true" exit status.

Version 1.0.0
Copyright © 1987 AT&T

if STATEMENT — EXAMPLE

$ **cat is.directory**<CR>

```
if
   test -d ${1:?"File_name expected"}
then
   echo ${1} is a directory
fi
```

- IF NO PARAMETER IS SUPPLIED, THE FOLLOWING MESSAGE IS PRINTED:

  File_name expected

- IF SUPPLIED FILE NAME IS A DIRECTORY, THE FOLLOWING MESSAGE IS PRINTED:

  $ **is.directory ${HOME}**<CR>
  /ustg/hutch is a directory

Copyright © 1987 AT&T

exit STATEMENT

- USED TO TERMINATE SHELL PROCEDURES
- EXIT STATUS SUPPLIED AS ARGUMENT
- 0 — SUCCESS
- 1 THROUGH 255 — ERROR CONDITION

Version 1.0.0
Copyright © 1987 AT&T

exit STATEMENT — EXAMPLE

```
$ cat is.directory<CR>
# command name: is.directory
#       exit status: 2 - no file_name specified
#                    0 - normal completion
#           purpose: check to see if the
#                    specified file_name
#                    is a directory
if
   test "${1}" = ""
then
   cat <<-!
      the ${0} command should be
      given the name of a file and
      will tell you if the file is
      a directory.
      !
   exit 2
fi
if
   test -d "${1}"
then
   echo "${1} is a directory"
fi
```


Copyright © 1987 AT&T

test STATEMENT — THE TEST CONDITION

- ANY COMMAND MAY BE USED IN THE CONDITIONAL PART OF AN **if** STATEMENT
- SEVERAL COMMANDS MAY APPEAR BETWEEN **if** AND **then**
- LAST COMMAND BEFORE THE KEYWORD **then** DECIDES IF COMMANDS BETWEEN **then** AND **fi** ARE EXECUTED

Version 1.0.0
Copyright © 1987 AT&T

test STATEMENT — TEST CONDITION

EXAMPLE

```
$ cat finder<CR>
#  program name : finder
#      arguments : 1 - pattern for search
#                  2 - file to search
#         author : Hammond Aigs
#   exit statuses : 0 - pattern found
#                  1 - pattern not found
#                  2 - arguments improperly
#                     specified
#        purpose : searches a file for a pattern
#                  but doesn't print matched lines

if
   test "${2}" = "" -o -r "${2}"
then
   echo usage ${0} pattern file
   exit 2
fi
if
   grep "${1}" "${2}" > /dev/null
then
   echo "${1} found in file ${2}"
   exit 0
fi
exit 1
```


Copyright © 1987 AT&T

EXAMPLE

This program may be used with the last echo removed in other shell programs that need to only check if a pattern was matched by any lines in the file without the matched lines printing. This program relies on the fact that the **grep** command will return a nonzero exit status if the pattern is not found and a zero exit status if the pattern is found.

Although this particular program could have been written using fewer lines of code, it serves as a good template for solving other kinds of similar problems.

Copyright © 1987 AT&T

test STATEMENT — TEST CONDITION — EXAMPLE

```
$ cat finder<CR>
#  program name : finder
                    .
                    .
                    .
if
   test "${2}" = ""
then
   echo usage ${0} pattern file
   exit 2
fi
if
   grep "${1}" "${2}" > /dev/null
then
   echo "${1} found in file ${2}"
   exit 0
fi
exit 1
```


Copyright © 1987 AT&T

if-else STATEMENT

The format of the **if-else** construct is similar to that of the **if** statement, but allows execution of statements if the test condition proves false.

FORMAT

```
if
   command
   testing command
then
   command
   command
else
   command
   command
fi
```

The words **if**, **then**, **else**, and **fi** are keywords to the **if-else** construct. Several commands may appear between any of the keywords; but if there are several commands between the keywords **if** and **then**, the exit status of the command that precedes **then** determines the control flow of the construct.

An alternative to the **if-then-else** is the logical "or". The logical "or" has the format:

```
command_1 arg1 arg2 || command_2 arg1 arg2
```

The second command is executed if and only if the first command exits with a "false" exit status.

Version 1.0.0
Copyright © 1987 AT&T

if-else STATEMENT

FORMAT

```
if
    testing command
then
    command
else
    command
fi
```

- IF THE LAST COMMAND BEFORE **then** HAS EXIT STATUS OF 0, DO COMMANDS BETWEEN **then** AND **else**
- IF THE LAST COMMAND BEFORE **then** DOES NOT HAVE EXIT STATUS OF 0, DO COMMANDS BETWEEN **else** AND **fi**

Version 1.0.0
Copyright © 1987 AT&T

if-else STATEMENT — EXAMPLE

```
$ cat telephone<CR>
if
   test ${#} -lt 1
then
   echo "Enter name: \c"
   read name
else
   name="${1}"
fi

if
   test "${name}" = ""
then
   echo "A name must be supplied"
   exit 3
else
   grep "${name}" ${HOME}/listings
fi
```

This example takes one of two routes, depending on whether or not an argument is supplied. The first **if** statement determines which route to take.

Copyright © 1987 AT&T

if-else STATEMENT -- EXAMPLE

If no argument is specified, the program prompts for a name as input. If an argument is supplied, the program assumes the argument is the pattern that is used to search the *phone-directory* file.

The second **if** statement first checks to see if the variable **name** has a null value. This could be introduced in one of two ways:

- As an explicitly null argument on the command line
- From a null response to the **read** statement.

If a null value is supplied, the error message is printed and the program exits with a status of 3.

Copyright © 1987 AT&T

if-else STATEMENT — EXAMPLE

```
$ cat ${HOME}/listings<CR>
981-2678    Robert Hutchison       hutch
981-6160    Dom Pante              pante
981-3607    Charlie de la Motte    chas
981-2677    John More              more
981-6509    Fred Eng               fredeng
981-6861    Annette Vaness         vaness

$ telephone<CR>
Enter name: Eng
981-6509    Fred Eng               fredeng

$ telephone hutch<CR>
981-2678    Robert Hutchison       hutch

$
```

In this example, the program **telephone** ran first using no argument. When the pattern **Eng** was entered as the response to the prompt, the program **grep**ped the file *${HOME}/listings* for the pattern **Eng**. The **grep** command then printed the matching line.

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 8 EXERCISE

1. Indicate whether the following tests are true or false. Assume that the file *file name* is a regular file and everyone has all permissions on that file.

```
test -z "hello"

test -n ""

test "hello"

test "-d file_name -a -f file_name"
```

   How many arguments are being supplied to the **test** statement in the last example?

2. If two shell processes are involved in running the shell procedure **is.directory**, which interpretations are done by:

   A. The parent shell process?

   B. The child shell process?

```
$ cat is.directory<CR>
if
   test -d ${1:?"File_name expected"}
then
   echo ${1} is a directory
fi

$ is.directory ${HOME}<CR>
/ustg/hutch is a directory
```


Copyright © 1987 AT&T

VOLUME 8 EXERCISE

3. Modify the shell program named **lookup** so that it prints a message similar to the example. If the user is not logged in, then exit with a non-zero exit status.

```
$ cat lookup <CR>
entry="`who | grep ${1}`"
if
   test "${entry}" = ""
then
   echo ${1} is not logged in
   exit
else
   echo ${1} is logged in on `echo "${entry}" | cut -c12-18`
fi
```

   EXAMPLE:

```
   $ lookup cec21
   cec21 is logged in on tty17.

   $ lookup cec12
   cec12 is not logged in.

   $
```

4. Write a shell program named **welcome** that will print "Good Morning" if it is before noon, "Good Afternoon" if it is before 6 PM, or "Good Evening".

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 8 EXERCISE - ANSWERS

1. False
   False
   True
   True
   One Argument

2. **$ cat is.directory**<CR>

```
if
   test -d ${1:?"File_name expected"}
then
   echo ${1} is a directory
fi
```

$ echo ${HOME}<CR>
/ustg/hutch

$ is.directory ${HOME}<CR>
/ustg/hutch is a directory

A. WHICH SUBSTITUTIONS ARE DONE BY THE PARENT SHELL PROCESS?

${HOME} IS EXPANDED BY THE PARENT PROCESS

B. WHAT IS DONE BY THE CHILD SHELL PROCESS?

${1} IS EXPANDED BY THE CHILD PROCESS

Copyright © 1987 AT&T

VOLUME 8 EXERCISE - ANSWERS

3.

```
$ cat lookup<CR>
#        lookup - check to see if someone is logged in
# one argument - login name
#   exit status 0 - user logged in
#                 1 - user not logged in
entry="`who | grep \^${1}`"
if
   test "${entry}" = ""
then
   echo ${1} is not logged in
   exit 1
else
   echo ${1} is logged in to `echo "${entry}" | cut -c12-18`
fi

$ lookup cec13<CR>
cec13 is logged in on tty01

$ lookup xyz<CR>
xyz is not logged in.
```

Version 1.0.0
Copyright © 1987 AT&T

VOLUME 8 EXERCISE - ANSWERS

4.

```
$ cat welcome
# welcome - print a welcome message
# no arguments
if
   test `date +%H` -lt 12
then
   echo Good Morning
else
   if
      test `date +%H` -lt 18
   then
      echo Good Afternoon
   else
      echo Good Evening
   fi
fi

$ welcome<CR>
Good Morning
```


Copyright © 1987 AT&T

VOLUME 8 SUMMARY

Statement	Meaning
test	
"string1" -eq "string2"	string1 is equal to string2
"string1" -ne "string2"	string1 is not equal to string2
"string1" -gt "string2"	string1 is greater than string2
"string1" -ge "string2"	string1 is greater or equal to string2
"string1" -lt "string2"	string1 is less than string2
"string1" -le "string2"	string1 is less than or equal to string2

Statement	Syntax

if-then	if
	  commands
	  testing_command
	then
	  commands
	fi

if-then-else	if
	  commands
	  testing_command
	then
	  commands
	else
	  commands
	fi

Copyright © 1987 AT&T