KORN SHELL FEATURES:

The Korn shell, developed by David Korn at the AT&T Labs, is a dual-purpose utility. It can be used interactively as an interpreter that reads, interprets, and executes user commands. It can also be used as a programming language to write shell scripts.

Korn Shell Sessions: When we use the Korn shell interactively, we execute commands at the shell prompt.

Standard Streams: We defined three Standard Streams – standard input (0), standard output (1), and standard error (2) – available in all shells.

Redirection: The standard streams can be redirected from and to files. If we don't use redirection, standard output and standard error both go to the monitor.

Pipes: The pipe operator temporarily saves the output from one command in a buffer that is being used at the same time as the input to the next command.

tee command: The tee command copies standard input to standard output and at the same time copies it to one or more files. If the stream is coming from another command, such as who, it can be piped to the tee command.

Combining Commands: We can combine commands in four ways: sequenced commands, grouped commands, chained commands, and conditional commands.

Command Line Editing: The Korn shell supports command-line editing.

Quotes: There are three quote types that Korn shell supports: backslash, double quotes and single quotes.

Command Substitution: Command Substitution is used to convert a command's output to a string that can be stored in another string or a variable. Although the Korn shell supports two constructs for command substitution ['command' and $ (command)].

Job Control: Job control is used to control how and where a job is executed in the foreground or background.

Aliases:

An alias is a means of creating a customized command by assigning a name or acronym to a command. If the name we use is one of the standard shell commands, such as dir, then the alias replaces the shell command. In the Korn shell, an alias is created by using the **alias** command. It's format is: **alias name=command-definition**

Where alias is the command keyword, name is the name of the alias name being created, and command-definition is the code for the customized command.

Listing Aliases:

The Korn shell provides a method to list all aliases and to list a specific alias. Both use the alias command. To list all aliases, we use the alias command with no arguments. To list a specific command, we use the alias command with one argument, the name of the alias command.

Removing Aliases:

Aliases are removed by using the **unalias** command. It has one argument, a list of aliases to be removed. When it is used with the all option (-a), it deletes all aliases.

TWO SPECIAL FILES:

There are two special files in UNIX that can be used by any shell.

Trash File (/dev/null):

The trash file is a special file that is used for deleting data. Found under the device (dev) directory, it has a very special characteristic: Its contents are always emptied immediately after receiving data. In other words, no matter how much or how often data are written to it, they are immediately deleted. Physically there is only one trash file in the system: It is owned by the superuser.

Because it is a file, it can be used as both a source and destination. However, when used as a source, the result is always end of the file because it is always empty. While the following two commands are syntactically correct, the first has no effect because the string “Trash me”, when sent to the trash file, is immediately deleted. The second has no effect because the file is always empty, which means that there is nothing to display.

$ **print “Trash me” > /dev/null**

$ **cat /dev/null**

Terminal File (/dev/tty):

Although each terminal in UNIX is a named file, such as /dev/tty13 and /dev/tty31, there is only one logical file, /dev/tty. This file is found under the device directory; it represents the terminal of each user. This means that someone using terminal /dev/tty13 can refer to the terminal using either the full terminal name (/dev/tty13) or the generic system name (/dev/tty).

VARIABLES:

The Korn shell allows you to store the values in variables. A shell variable is a location in memory where values can be stored. In the Korn shell, all data are stored as strings. There are two broad classifications of variables: user-defined and predefined.

User-Defined Variables:

As implied by their name, user defined variables are created by the user. Although the user may choose any name, it should not be the same as one of the predefined variables. Each variable must have a name. The name of the variable must start with an alphabetic or underscore (_) character. It then can be followed by zero or more alphanumeric or underscore characters.

Predefined Variables:

Predefined variables are either shell variables or environmental variables. The shell variables are used to configure the shell. The environmental variables are used to configure the environment.

Storing Values in Variables:

There are several ways that we can store a value in a variable, but the easiest method is to use the assignment operator, =. The variable is coded first, on the left, followed by the assignment operator and then the value to be stored. There can be no spaces before and after the assignment operator; the variable, the operator, and the value must be coded in sequence immediately next to each other as ***varA=7*** Here varA is the variable that receives the data, and 7 is the value being stored in it.

Accessing Value of a Variable: To access the value of variable, the name of the variable must be preceded by a dollar sign as shown below:

$ **count=7**
$ **print *$count* is the number after 6 and before 8**
Result:
7 is the number after 6 and before 8

Null Variables:

If we access a variable that is not set (no value is stored in it), we receive what is called a null value (nothing). We can also explicitly store a null value in a variable by either assigning it a null string (“”) or by assigning it nothing.

Unsetting a Variable:

We can clear a variable by assigning a null value to it. Although this method works, it is better to use the **unset** command.

$ **x=1**
$ **print "(x contains:" $x")"**
(x contains: 1)

$ **unset x**
$ **print "(x contains:" $x")"**
(x contains:)

Storing Filenames:

We can also store a filename in a variable. We can even use the wildcards. However, we should be aware of how wildcards are handled by the shell. The shell stores the file name including the wildcard in the variable without expanding it. When the value is used, the expansion takes place.

Example:

$ **ls**
File1 File2 File3.bak
$ **filename="File*"**
$ **print "Filename contains: $filename"** # show contents
Filename contains: File*
$ **print $filename**
File1 File2 File3.bak

$ **filename="File?"**
$ **print $filename**
File1 File2

Storing File Contents:

We can also store the contents of a file in a variable for processing, such as parsing words. Two steps are required to store the file:

1. Create a copy of the file on standard output using the **cat** utility.
2. Using command substitution, convert the standard output contents to a string.

The string can now be stored in a variable. The entire process is done in one command line.

Example:

$ **cat storeAsVar.txt**
This is a file
used to show
the result of storing a file in a variable

$ **x=$(cat storeAsVar.txt)**
$ **print $x**
This is a file used to show the result of storing a file in a variable

Storing Commands in a Variable:

We can also store a command in a variable. For example, the list command can be stored in a variable. We can then use the variable at the command prompt to execute its contents. Storing commands in a variable works only with simple commands. If the command is complex (for example, piping the results of the list command to **more**) a command variable will not work.

Read-Only Variables:

Most programming languages provide a way for a programmer to define a named constant. A named constant defines a value that cannot be changed. Although the Korn shell does not have named constants, we can create the same effect by creating a variable, assigning it a value, and then fixing its value with the **readonly** command. The command format is:

readonly variable-list

Example:

$ **cHello=Hello**
$ **cBye="Good Bye"**
$ **readonly cHello cBye**
$ **cHello=Howdy**
cHello: is read only
$ **cBye=TaTa**
cBye: is read only
$ **print cHello " . . . " $cBye**
Hello . . . Good Bye

INPUT AND OUTPUT:

INPUT:

Reading data from a terminal or a file is done using the **read** command. The **read** command reads a line and stores the words in variables. It must be terminated by a return, and the input line must immediately follow the command. The **read** command format is shown below:

read options variable$_1$ . . . variable$_n$

Options:

-r: ignore newline
-u: stream descriptor

Read Word by Word:

When the read command is executed, the shell reads a line from the standard input and stores it in variables word by word. Words are characters separated by spaces or tabs. The first word is stored in the first variable; the second is stored in the second variable, and so forth. Another way of saying this is that the read command parses the input string (line) into words.

If there are more words than there are variables, all the extra words are placed in the last variable. If there are fewer words then there are variables; the unmatched variables are set to a null value. Any value in them before the read is lost.

Reading Line by Line:

The design for handling extra words provides an easy technique for storing a whole line in one variable. We simply use the read command, giving it only one variable. When executed, the whole line is in the variable.

Reading from a File:

The Korn shell allows scripts to read from a user file. This is done with the stream descriptor option (-u). A stream descriptor is a numeric designator for a file. We have seen that the standard streams are numbered 0, 1, and 2 for standard input, standard output and standard error respectively.

OUTPUT:

The output statement in the Korn shell is the print command. Although the Korn shell also supports the echo command (inherited from the Bourne shell), we use print because it is

faster and there is the possibility that echo may become depreciated in a future version of Korn shell. The format of the print command is shown below:

print options argument$_1$ . . . argument$_n$

Options:

-n : no new line

ENVIRONMENT VARIABLES:

The environmental variables control the user environment. The following table lists the environmental variables. In Korn shell, environmental variables are in uppercase.

VARIABLE	EXPLANATION
CDPATH	Contains the search path for cd command when the directory argument is a relative path name.
COLUMNS	Defines the width, in characters, of your terminal. The default is 80.
EDITOR	Pathname of the command-line editor.
ENV	Pathname of the environment file.
HISTFILE	Pathname for the history file.
HISTSIZE	Maximum number of saved commands in the history file.
HOME	Pathname for the home directory.
LINES	Defines the height, in lines, of your terminal display. The default is 24.
LOGNAME	Contains the user's login name from the /etc/passwd file
MAIL	Absolute pathname for the user's mailbox.
MAILCHECK	Interval between tests for new mail. The default is 600 seconds.
OLDPWD	Absolute pathname of the working directory before the last cd command.
PATH	Searches path for commands.
PS1	Primary prompt, such as $ and %.
PS2	Secondary prompt. Used when complete command not entered on first line. the default is >.

STARTUP SCRIPTS:

Each shell uses one or more scripts to initialize the environment when a session is started. The Korn shell uses three startup files. They are 1) System profile file 2) Personal profile file and 3) Environment file.

SYSTEM PROFILE FILE:

The system-level profile file is a one which is stored in the /etc directory. Maintained by the system administrator, it contains general commands and variable settings that are applied to every user of the system at login time. The system profile file is generally quite large and contains many advanced commands.

The system profile is a read-only file; its permissions are set so that only system administrator can change it.

PERSONAL PROFILE FILE:

The personal profile, *~/.profile*, contains commands that are used to customize the startup shell. It is an optional file that is run immediately after the system profile file. Although it is a user file, it is often created by the system administrator to customize a new user's shell. If you make changes to it, we highly recommend that you make a backup copy first so that it may be restored easily if necessary.

ENVIRONMENT FILE:

The Korn shell allows users to create a command file containing commands that they want to be executed to personalize their environment. It is most useful when the Korn shell is started as a child of a non-Korn login shell. Because we can use any name for it, the absolute pathname of the environment file must be stored in the ENV variable. The shell then locates it by looking at the ENV variable.

COMMAND HISTORY:

The Korn shell provides an extensive command history capability consisting of a combination of commands, environmental variables and files. A major feature of the design is the ability to recall a command and reexecute it without typing it again.

HISTORY FILE:

Every command that we type is kept in a history file stored in our home directory. By default, the filename is ~/ .sh_history. It can be renamed, provided that we store its pathname in the HISTFILE environmental variable.

The size of the file (i.e. the number of commands that it can store) is 128 unless changed. The HISTSIZE variable can be used to change it when we need to make it larger or smaller.

HISTORY COMMAND:

The formal command for listing, editing, and executing commands from the history file is the fc command. However, the Korn shell contains a preset alias, history, that is easier to use and more flexible. Executed without any options, the history command lists the last 16 commands.

REDO COMMAND (r):

Any command in the history file can be reexecuted using the redo command (r).

SUBSTITUTION IN REDO COMMAND:

When we redo a command, we can change part of the command.

COMMAND EXECUTION PROCESS

To understand the behavior of the shell, it helps to understand how Korn executes a command. Command execution is carried out in six sequential steps:

EXECUTION STEPS:

The six execution steps are recursive. This means that when the shell performs the third step, command substitution, the six steps are followed for the command inside the dollar parentheses.

Command Parsing: The shell first parses the command into words. In this step, it uses whitespace as delimiters between the words. It also replaces sequences of two or more spaces or tabs with a single space.

Variable Evaluation: After completely parsing the command, the shell looks for variable names (unquoted words beginning with a dollar sign). When a variable name is found, its value replaces the variable name.

Command Substitution: The shell then looks for a command substitution. If found, the command is executed and its output string replaces the command, the dollar sign, and the parenthesis.

Redirection: At this point, the shell checks the command for redirected files. Each redirected file is verified by opening it.

Wildcard Expansion: When filenames contain wildcards, the shell expands and replaces them with their matching filenames. This step creates a file list.

Path Determination: In this last step, the shell uses the PATH variable to locate the directory containing the command code. The command is now ready for execution.

KORN SHELL PROGRAMMING:

Basic Script Concepts:

A shell script is a text file that contains executable commands. Although we can execute virtually any command at the shell prompt, long sets of commands that are going to be executed more than once should be executed using a script file.

Script Components:

Every script has three parts: the interpreter designator line, comments and shell commands.

Interpreter Designator Line: One of the UNIX shells runs the script, reading it and calling the command specified by each line in turn. The first line of the script is the designator line; it tells UNIX the path to the appropriate shell interpreter. The designator line begins with a pound sign and a bang (**#!**). If the designator line is omitted, UNIX will use the interpreter for the current shell, which may not be correct.

Comments: Comments are documentation we add in a script to help us understand it. The interpreter doesn't use them at all; it simply skips over them.

Comments are identified with the pound sign token (#). The Korn shell supports only line comments. This means that we can only comment one line at a time, c comment cannot extend beyond the end of the line.

Commands: The most important part of a script is its commands. We can use any of the commands available in UNIX. However, they will not be executed until we execute the script; they are not executed immediately as they are when we use them interactively. When the script is executed, each command is executed in order from the first to the last.

Command Separators → Shell use two tokens to separate commands; semicolons and newlines.

Blank Lines → Command separators can be repeated. When the script detects multiple separators, it considers them just one. This means that we can insert multiple blank lines in a script to make it more readable.

Combined Commands → We can combine commands in a script just as we did in the interactive sessions. This means that we can chain commands using pipes, group commands or conditional commands.

MAKING SCRIPTS EXECUTABLE:

We can make a script executable only by the user (ourselves), our group, or everybody. Because we have to test a new script, we always give ourselves execute permission. Whether or not we want others to execute it depends on many factors.

EXECUTING THE SCRIPT:

After the script has been made executable, it is a command and can be executed just like any other command. There are two methods of executing it; as an independent command or as an argument to a subshell command.

Independent Command:

We do not need to be in the Korn shell to execute a Korn shell script as long as the interpreter designator line is included as the first line of the script. When it is, UNIX uses the appropriate interpreter as called out by the designator line.

To execute the script as an independent command, we simply use its name as in the following example:

$ **script_name**

Child Shell Execution:

To ensure that the script is properly executes, we can create a child shell and execute it in the new shell. This is done by specifying the shell before the script name as in the following example:

$ **ksh script_name**

EXPRESSIONS:

Expressions are a sequence of operators and operands that reduces to a single value. The operators can be either mathematical operator, such as add and subtract, that compute a value; relational operators, such as greater than and less than, that determine a relationship between two values and return true or false; file test operators that report status of a file; or logical operators that combine logical values and return true or false. We use mathematical expressions to compute a value and other expressions to make decisions.

Mathematical expressions

Mathematical expressions in the Korn shell use integer operands and mathematical operators to compute a value.

Mathematical operators

Mathematical operators are used to compute a numeric value. The Korn shell supports the standards add, subtract, multiply and divide operators plus a special operator for modulus.

let command:

The Korn shell uses either the expr command or the let command to evaluate expressions and store the result in another variable. The expr command is inherited from the Bourne shell; the let command is new.

Example:

$ **let y=x+16**

In this example, note that we don't use a dollar sign with the variables. The **let** command doesn't need the dollar sign; its syntax expects variables or constants.

The Korn shell has an alternate operator, a set of double parentheses, that may be used instead of let command.

Example:

$ **((y = x + 16))**

Relational expressions:

It compares two values and returns a logical value such as true or false. The logical value depends on the values being compared and the operator being used.

Relational operators:

The relational operators are listed in table given below:

Numeric Interpretation	Meaning	String Interpretation
>	Greater than	
>=	Greater than or equal	
<	Less than	
<=	Less than or equal	
==	Equal	=

Numeric Interpretation	Meaning	String Interpretation
!=	Not equal	!=
	String length not zero	-n
	String length zero	-z

The sting equal and not equal logical operators support patterns for the second (right) operand. The patterns supported are listed in the table below:

Pattern	Interpretation
string	Must exactly match the first operand.
?	Matches zero or one single character.
[. . .]	Matches one single character in the set.
*	Repeats pattern zero or more times.
?(pat1|pat2|. . .)	Matches zero or one of any of the patterns.

Relational Test Command:

In the Korn shell we can use wither the test command inherited from the Bourne shell or one of the two test operators, ((. . .)) or [[. . .]].

Which operator is used depends on the data. Integer data require the double parenthesis as shown in the example: i.e. **((x < y))**

For string expressions, the Korn shell requires the double bracket operator. Although the integer operator parentheses do not require the variable dollar sign, the double brackets operator does. The next example demonstrates this format:

[[$x != $y]]

File Expressions:

File expressions use file operators and test command to check the status of a file. A file's status includes characteristics such as open, readable, writable, or executable.

File Operators:

There are several operators that can be used in a file test command to check a files status. They are particularly useful in shell scripts when we need to know the type or status of file. The following table lists the file operators and what file attributes they test.

Operator	Explanation
-r file	True if file exists and is readable
-l file	True if file exists and is a symbolic link.

Operator	Explanation
-w file	True if file exists and is writable.
-x file	True if file exists and is executable.
-f file	True if file exists and is a regular file.
-d file	True if file exists and is a directory.
-s file	True if file exists and has a size greater than zero.
file1 –nt file2	True if file1 is newer than file2.
file1 –ot file2	True if file1 is older than file2.

Test File Command:

Although we could use the test command inherited from the Bourne shell, in the Korn shell we recommend the Korn shell double bracket operator to test the status of the file.

Logical Expressions:

Logical expressions evaluate to either true or false. They use a set of three logical operators: not (!), and (&&), or (||).

DECISION MAKING & REPETITION:

DECISION MAKING:

The Korn shell has two different statements that allow us to select between two or more alternatives. The first, the if-then-else statement, examines the data and chooses between two alternatives. For this reason this is sometimes referred to as a two-way selection. The second, the case statement, selects one of several paths by matching patterns to different strings.

if-then-else

Every language has some variation of the if-then-else statement. The only difference between them is what keywords are required by the syntax. For example, the C language does not use then. In fact, it is an error to use it. In all languages however, something is tested. Most typically data values are tested.

In the Korn shell, the exit value from a command is used as the test. The shell evaluates the exit status from the command following `fi`. When the exit status is 0, the `then` set of commands is executed. When the exit status is 1, the `else` set of commands is executed.

Case syntax: The case statement contains the string that is evaluated. It ends with an end case token, which is `esac` (case spelled backward). Between the start and end case statements is the **pattern list.**

For every pattern that needs to be tested, a separate patter is defined in the pattern list. The pattern ends with a closing parenthesis. Associated with each pattern is one or more commands. The commands follow the normal rules for commands with the addition that the last command must end in two semicolons. The last action in the pattern list is usually the wildcard asterisk, making it the default if none of the other cases match.

REPETITION:

The real power of computers is their ability to repeat an operation or a series of operations many times. This repetition, known as looping, is one of the basic programming concepts.

A loop is an action or a series of actions repeated under the control of loop criteria written by the programmer. Each loop tests the criteria. If the criteria tests valid, the loop continues; if it tests invalid, the loop terminates.

Command-Controlled and List-Controlled Loops:

Loops in Korn shell can be grouped into two general categories: command-controlled loops and list-controlled loops.

Command-Controlled loops:

In a command controlled loop, the execution of a command determines whether the loop body executes or not. There are two command-controlled loops in the Korn shell : the *while* loop and the *until* loop.

Syntax for while:

```
while command
do
        action
done
```

The until loop works just like while loop, except that it loops as long as the exit status of the command is false. In this sense, it is the complement of the while loop. The syntax of until loop is:

```
until command
do
        action
done
```

List-Controlled loops:

In a list-controlled loop, there is a control list. The number of elements in the list controls the number of iterations. If the control list contains five elements, the body of the loop is executed five times; if the control list contains ten elements, the body of the loop is executed ten times.

The **for-in** loop is the first list-controlled loop in the Korn shell. The list can be any type of string; for example it can be words, lines, sentences, or a file.

for-in loop Syntax:

for variable in list
do
    action
done

Example:

```
for I in 1 2 3 4 5
do
        print $I hello
done
```

The **select** loop is the second Korn shell list-controlled loop. The select loop is a special loop designed to create menus. A menu is a list of options displayed on the monitor. The user selects one of the menu options, which is then processed by the script. The format of the select loop is similar to for-in loop. It begins with the keyword select followed by a variable and a list of strings:

$ select variable in list

Example:

```
select choice in month year quit
do
        case $choice in
                month) cal; ;
                year)   yr-$(date "+%Y")
                        cal $yr; ;
                quit)   print "Hope you found your date"
                        exit; ;
                *)      print "sorry, I don't understand your answer"
        esac
done
```

SPECIAL PARAMETERS AND VARIABLES

The Korn shell provides both special parameters and special variables for our use.

SPECIAL PARAMETERS:

Besides having positional parameters numbered 1 to 9, the Korn shell script can have four other special parameters: one that contains the script filename, one that contains the number of arguments entered by the user, and two that combine all other parameters.

Script Name ($0):

The script name parameter ($0) holds the name of the script. This is often useful when a script calls other script. The script name parameter can be passed to the called script so that it knows who called it. As another use, when a script needs to issue an error message, it can include its name as part of the message. Having the script name in the message clearly identifies which script had a problem.

Number of Arguments ($#):

A second special parameter holds the number of arguments passed to the script. Scripts can use this parameter, referred to as $#, programmatically in several ways.

All Parameters ($* and $@):

Two special parameters combine the nine positional parameters in to one string. They can be used with or without quotes.

SPECIAL VARIABLES:

Internal Field Separator (IFS):

The IFS variable holds the tokens used by the shell commands to parse the string into substrings such as words. The default tokens are the three white space tokens: the space, tab and newline.

One common use of the internal field separators parses a read string into separate words. It receives a login id as an argument and then searches the password file (/etc/passwd) for the matching id. When it finds, it prints the login id and the user name.

Special Parameter and Variable Summary:

Parameter or Variable	Description
$#	Number of arguments to a script

Parameter or Variable	Description
$0	Script Name
$*	All parameters
$@	All parameters
$?	Exit status variable
IFS	Internal field separator

CHANGING POSITIONAL PARAMETERS:

The positional parameters can be changed within the script only by using the set command; values cannot be directly assigned to them. This means that to assign values to the script positional parameters, we must use set. The set command parses an input string and places each separate part of the string into a different positional parameter (up to nine).

If the IFS is set to the default, set parses to words. We can set the IFS to any desired token and use set to parse the data accordingly.

Shift Command:

One very useful command in shell scripting is the shift command. The shift command moves the values in the parameters toward the beginning of the parameter list. To understand the shift command, think of the parameters as a list rather than as individual variables. When we shift, we move each parameter to the left in the list. If we shift three positions, therefore, the fourth parameter will be in the first position, the fifth will be in the second position, and so forth until all parameters have been moved three positions to the left. The parameters at the end of the set become null.

ARGUMENT VALIDATION:

Good programs are designed to be fail-safe. This means that anything that can be validated should be confirmed before it is used. We discuss various techniques used to validate user-supplied arguments:

Number of Arguments Validation:

The first code in a script that contains parameters should validate the number of arguments. Some scripts use a fixed number of arguments; other scripts use a variable number of arguments.

Even when the number of arguments is variable, there is usually a minimum number that is required. Both fixed-and variable- numbered arguments are validated by using the number of arguments parameter ($#).

Minimum Number of Arguments:

When a script expects a variable number of arguments and there is a minimum number required, we should verify that the minimum number has been entered.

Type of Argument Validation:

After the exact or minimum number of arguments is validated, the script should verify that each arguments type is correct. While all arguments are passed as strings, the string contents can be a number, a filename, or any other verifiable type.

Numeric Validation:

The value of numeric parameters is virtually unlimited; some scripts simply need number. Scripts that extract a range of lines from a file are of this nature. Other scripts may require that the number be in a range.

File Type Validation:

If an argument is an input file, we can verify that the file exists and that it has read permission. If the file is an output file, there is no need to verify it because UNIX will create it if it doesn't exist.

DEBUGGING SCRIPTS:

Whenever we write a script we test it. Often multiple tests are necessary. Sometimes the tests don't deliver the expected results. In these cases, we need to debug the script. There are two Korn shell options that we can use to help debug script: the verbosity (`verbose`) option and the execute trace (`xtrace`) option.

The *verbose* option prints each statement that is syntactically correct and displays an error message if it is wrong. Script output, if any is generated.

The xtrace option prints each command, preceded by a plus (+) sign, before it is executed. It also replaces the value of each variable accessed in the statement. For example, in the statement y=$x, the $x is replaced with actual variable value at the time the statement is executed.

SCRIPT EXAMPLES:

Cat → cat command purpose is explained with script Copy → cp command purpose is explained with script. ***Refer Pg. No.-601 UNIX and Shell Programming by Behrouz A. Forouzan, Richard F. Gilberg***