Hyperpolyglot

Unix Shells: Bash, Fish, Ksh, Tcsh, Zsh

Grammar

	bash	fish	ksh	tcsh	zsh
simple command	`ls`	`ls`	`ls`	`ls`	`ls`
simple command with argument	`echo hi`	`echo hi`	`echo hi`	`echo hi`	`echo hi`
simple command with redirect	`ls > /tmp/ls.out`	`ls > /tmp/ls.out`	`ls > /tmp/ls.out`	`ls > /tmp/ls.out`	`ls > /tmp/ls.out`
simple command with environment variable	`EDITOR=vi git commit`	`env EDITOR=vi git commit`	`EDITOR=vi git commit`	`env EDITOR=vi git commit`	`EDITOR=vi git commit`
pipeline	`ls | wc`	`ls | wc`	`ls | wc`	`ls | wc`	`ls | wc`
sublist separators	`&& ||`	*none*	`&& ||`	`&& ||`	`&& ||`
list terminators	`; &`	`; &`	`; &`	`; &`	`; &`
group command	`{ ls; ls;} | wc`	`begin; ls; ls; end | wc`	`{ ls; ls;} | wc`	*none*	`{ ls; ls;} | wc`
subshell	`(ls; ls) | wc`	`fish -c 'ls; ls' | wc`	`(ls; ls) | wc`	`(ls; ls) | wc`	`(ls; ls) | wc`

Shells read input up to an unquoted newline and then execute it. An unquoted backslash followed by a newline are discarded and cause the shell to wait for more input. The backslash and newline are discarded before the shell tokenizes the string, so long lines can be split anywhere outside of single quotes, even in the middle of command names and variable names.

In the shell grammar, *lists* contain *sublists*, which contain *pipelines*, which contain *simple commands*.

Subshells and *grouping* can be used to put a list in a pipeline. Subshells and groups can have newlines, but the shell defers execution until the end of the subshell or group is reached.

The section on execution control describes structures which do not fit into the simple grammar and execution model outlined here. The shell will not execute any of the control structures until the end keyword is reached. As a result, the control structure can contain multiple statements separated by newlines. Execution control structures cannot be put into pipelines.

simple command

In its simplest form a line in a shell script is a word denoting a command. The shell looks successively for a user-defined function, built-in function, and external command in the search path matching the word. The first one found is run. If no matching function or external command is found the shell emits a warning and sets its status variable to a nonzero value. It does not return the status value to its caller unless it has reached the end of its input, however.

`tcsh` lacks user defined functions but built-ins still take precedence over external commands.

simple command with argument

Commands can be followed by one or more words which are the arguments to the command. How a shell tokenizes the input into words is complicated in the general case, but in the common case the arguments are whitespace delimited.

simple command with redirect

The standard output, standard input, and standard error of the command can be redirected to files. This is described under redirection.

simple command with environment variable

A nonce environment variable can be set for the exclusive use of the command.

pipeline

Pipelines are a sequence of simple commands in which the standard output of each command is redirected to the standard input of its successor.

A pipeline is successful if the last command returns a zero status.

sublist separators

Sublist is a term from the `zsh` documentation describing one or more pipelines separated by the shortcut

operators `&&` and `||`. When `&&` is encountered, the shell stops executing the pipelines if the previous pipeline failed. When `||` is encountered, the shell stops executing if the previous pipeline succeeded. A sublist is successful if the last command to execute returns a zero status.

fish:

Fish has short-circuit operators; the following are equivalent to `ls && ls` and `ls || ls`:

```
$ ls ; and ls
$ ls ; or ls
```

list terminators

A list is a sequence of sublists separated by semicolons `;` or ampersands `&` and optionally terminated by a semicolon or ampersand.

If the separator or terminator is an ampersand, the previous sublist is run in the background. This permits the shell to execute the next sublist or the subsequent statement without waiting for the previous sublist to finish.

group command

A group command can be used to concatenate the stdout of multiple commands and pipe it to a subsequent command.

If the group has an input stream, it is consumed by the first command to read from stdin.

`bash` requires that the final command be terminated by a semicolon; `zsh` does not.

subshell

Like the group command, but the commands are executed in a subshell. Variable assignments or change of working directory are local to the subshell.

Quoting and Escaping

	bash	fish	ksh	tcsh	zsh
literal quotes	`'foo'`	*allows \' and \\ escapes:* `'foo'`	`'foo'`	`'foo'`	`'foo'`
interpolating quotes	`foo=7` `"foo is $foo"`	`set foo 7` `"foo is $foo"` *double quotes do not perform command substitution*	`foo=7` `"foo is $foo"`	`setenv foo 7` `"foo is $foo"`	`foo=7` `"foo is $foo"`
interpolating quotes escape sequences	`\$ \\ \` \"`	`\" \$ \\`	`\$ \\ \``	*none*	`\$ \\ \` \"`
quotes with backslash escapes	`$'foo\n'`	*none*	`$'foo'`	*none*	`$'foo'`
quoted backslash escapes	`\a \b \e \E \f \n` `\r \t \v` `\\ \' \" \`*`ooo`* `\`*`xhh`* `\`*`cctrl`*	*none*	`\a \b \e \E \f \n` `\r \t \v` `\\ \' \" \`*`ooo`* `\`*`xhh`* `\`*`cctrl`*	*none*	`\a \b \e \E \f \n` `\r \t \v` `\\ \' \" \`*`ooo`* `\`*`xhh`* `\`*`cctrl`*
unquoted backslash escapes	`\`*`space`*	`\a \b \e \f \n \r \t \v` `\`*`space`* `\$ \\ \* \? \~ \% \# \( \)` `\{` `\} \[ \] \< \> \^ \& \; \" \'` `\`*`xhh`* `\X`*`hh`* `\`*`ooo`* `\u`*`hhhh`* `\U`*`hhhhhhhh`* `\`*`cctrl`*	`\`*`space`*	`\`*`space`*	`\`*`space`*
command substitution	`$(ls)` `` `ls` ``	`(ls)`	`$(ls)` `` `ls` ``	`` `ls` ``	`$(ls)` `` `ls` ``
backtick escape sequences	`\$ \\ \` \`*`newline`*	*none*	`\$ \\ \` \`*`newline`*	`\$ \\` `\`*`newline`*	`\$ \\ \` \`*`newline`*

literal quotes

Literal quotes (aka single quotes) create a word with exactly the characters shown in the source code. For the shells other than `fish` there is no escaping mechanism and hence no way to put single quotes in the word.

Literal quotes can be used to put characters that the shell lexer uses to distinguish words inside a single word. For `bash` these characters are:

```
| & ; ( ) < > space tab
```

Literals quotes can also be used to prevent the parameter, brace, pathname, and tilde expansion as well as command substitution. For `bash` the special characters that trigger these expansions are:

```
$ { } * ? [ ] ` ~
```

interpolating quotes

Interpolating quotes (aka double quotes) perform parameter expansion and command substitution of both

the $() and ` ` variety. They do not perform brace, pathname, or tilde expansion. $ and ` are thus special characters but they can be escaped with a backslash as can the backslash itself, the double quote, and a newline.

interpolating quotes escape sequences

The escape sequences available in interpolating quotes.

quotes with backslash escapes

String literals which support C-style escapes.

quoted backslash escapes

The C-style string literal escapes.

unquoted backslash escapes

`fish` permits the use of C escapes outside of quotes.

command substitution

How to execute a command and get the output as shell text.

If the command output contains whitespace, the shell may parse the output into multiple words. Double quotes can be used to guarantee that the command output is treated as a single word by the shell:

```
"$(ls)"
"`ls`"
```

backtick escape sequences

Escape sequences that can be used inside backtick quotes.

Characters

	bash	fish	ksh	tcsh	zsh
word separating	| & ; ( ) < > SP HT LF	| & ; ( ) < > SP HT LF	| & ; ( ) < > SP HT LF	| & ; ( ) < > SP HT LF	| & ; ( ) < > SP HT LF
quoting and escaping	" ' \	" ' \	" ' \	" ' \	" ' \
shell expansion	*variable:* $ *brace:* { } *tilde:* ~ *command:* ` *pathname:* * ? [ ] *history:* ! ^	*variable:* $ *brace:* { } *tilde:* ~ *command:* ( ) *pathname:* * ?	*variable:* $ *brace:* { } *tilde:* ~ *command:* ` *pathname:* * ? [ ]	*variable:* $ *brace:* { } *tilde:* ~ *command:* ` *pathname:* * ? [ ] *history:* ! ^	*variable:* $ *brace:* { } *tilde:* ~ *command:* ` *pathname:* * ? [ ] *history:* ! ^
other special	# =	# [ ]	# = .	#	# =
bareword	A-Z a-z 0-9 _ - . , : + / @ %	A-Z a-z 0-9 _ - . , : + / @ % ! ^ =	A-Z a-z 0-9 _ - , : + / @ % ! ^	A-Z a-z 0-9 _ - . , : + / @ % =	A-Z a-z 0-9 _ - . , : + / @ %
variable name	A-Z a-z 0-9 _	A-Z a-z 0-9 _	A-Z a-z 0-9 _	A-Z a-z 0-9 _	A-Z a-z 0-9 _

word separating

The shell tokenizes its input into words. Characters which are not word separating and do not have any word separating characters between them are part of the same word.

quoting and escaping

For two characters to be in different words, the presence of a word separating character between them is *necessary* but not *sufficient*, because the separating character must not be quoted or escaped.

The following two lines both tokenize as a single word:

```
"lorem ipsum"
lorem" "ispum
```

shell expansion

The presence of shell expansion characters in a word causes the shell to perform a transformation on the word. The transformation may replace the word with more than one word.

In the following example, the word `*.c` will be replaced by multiple words if there is more than one file with a `.c` suffix in the working directory:

```
grep main *.c
```

Square brackets `[ ]` are used for both pathname expansion, where the brackets contain a list of characters, and array notation, where the brackets contain an index. We believe that in cases of ambiguity, the syntax is always treated as array notation. `fish` does not have this ambiguity because it does not use square brackets in pathname expansion.

zsh:

In `zsh` variable expansion will expand to a single word, even if the variable contains word separating characters. This behavior is different from the other shells.

A variable can be expanded to multiple words with the `${=VAR}` syntax, however.

```
$ function countem() { echo $#; }

$ foo='one two three'

$ countem $foo
1

$ countem ${=foo}
3
```

other special characters

comments:

The number sign `#` can be used to start a comment which ends at the end of the line. The `#` must be by itself or the first character in a word.

In `tcsh`, comments are not supported when the shell is interactive.

In `zsh`, comments are not supported by default when the shell is interactive. This can be changed by invoking `zsh` with the `-k` flag or by running:

```
set -o INTERACTIVE_COMMENTS
```

variable assignment:

The equals sign `=` is used for variable assignment in `bash`, `ksh`, and `zsh`. Given that spaces cannot be placed around the equals sign, it seems likely the tokenizer treats it like other bareword characters. Note that in a simple command, the command name is the first word which does not contain an equals sign.

namespaces:

`ksh` has namespaces. They can be used for variable names and function names:

```
$ bar=3

$ namespace foo { bar=4; }

$ echo $bar
3

$ namespace foo { echo $bar; }
4

$ echo ${.foo.bar}
4
```

bareword characters

A bareword is a word which is not quoted and does not contain escapes. The characters which are listed above are those which can appear anywhere in a bareword.

Some of the other characters can appear in barewords under certain circumstances. For example the tilde `~` can appear if it is not the first character.

variable name characters

Characters which can be used in variable names.

Note that a variable name cannot start with a digit. Also, `$_` is a special variable which contains the previous command.

Variables

	bash	fish	ksh	tcsh	zsh	external
global variables *set, get, list, unset, edit*	*var=val* *$var* set unset -v *var* *none*	set -g *var val* *$var* set -g set -e *var* vared *var*	*var=val* *$var* set unset -v *var* *none*	set *var=val* *$var* set unset *var* *none*	*var=val* *$var* set unset -v *var* vared *var*	
read-only variables *mark readonly, set and mark readonly, list readonly*	readonly *var* readonly *var=val* readonly -p	*none*	readonly *var* readonly *var=val* readonly -p	*none*	readonly *var* readonly *var=val* readonly -p	
exported variables *export, set and export, list exported, undo export*	export *var* export *var=val* export -p export -n *var*	set -gx *var* *$var* set -gx *var val* set -x set -gu *var* *$var*	export *var* export *var=val* export -p *none*	setenv *var* *$var* setenv *var val* printenv *none*	export *var* export *var=val* export -p *none*	*none* *none* printenv *none*

options *set, list, unset*	`set -o` *opt* `set -o` `set +o` *opt*	*none*	`set -o` *opt* `set -o` `set +o` *opt*	*none*	`set -o` *opt* `set -o` `set +o` *opt*	
other variable built-ins	`declare`			`@`	`declare` `functions` `setopt` `float` `integer` `unsetopt`	

global variables

How to set a global variable; how to get the value of a global variable; how to list all the global variables; how to unset a global variable; how to edit a variable.

Variables are global by default.

In `tcsh` if *var* is undefined then encountering $*var* throws an error. The other shells will treat $*var* as an empty string.

If there is a variable named `foo`, then

```
unset foo
```

will unset the variable. However, if there is no such variable but there is a function named `foo`, then the function will be unset. `unset -v` will only unset a variable.

read-only variables

How to mark a variable as read-only; how to simultaneously set and mark a variable as read-only; how to list the read-only variables.

An error results if an attempt is made to modify a read-only variable.

exported variables

How to export a variable; how to set and export a variable; how to list the exported variables.

Exported variables are passed to child processes forked by the shell. This can be prevented by launching the subprocess with `env -i`. Subshells created with parens ( ) have access non-exported variables.

The `tcsh` example for exporting a variable without setting it isn't the same as the corresponding examples from the other shells because in `tcsh` an error will result if the variable isn't already set.

options

Options are variables which are normally set via flags at the command line and affect shell behavior.

Variable Expansion

	bash	fish	ksh	tcsh	zsh	external
set variable value	*var*=*val*	`set -g` *var val*	*var*=*val*	`setenv` *var val*	*var*=*val*	
get variable value	$*var*	$*var*	$*var*	$*var*	$*var*	
concatenate variable and value	${*var*}*val*	{$*var*}*val*	${*var*}*val*	${*var*}*val*	${*var*}*val*	
coalesce	${*var*:-*val*}		${*var*:-*val*}		${*var*:-*val*}	
coalesce and assign if null	${*var*:=*val*}		${*var*:=*val*}		${*var*:=*val*}	
message to stderr and exit if null	${*var*:?*msg*}		${*var*:?*msg*}		${*var*:?*msg*}	
substring	*offset is zero based:* ${*var*:*offset*} ${*var*:*offset*:*len*}		*offset is zero based:* ${*var*:*offset*} ${*var*:*offset*:*len*}		*offset is zero based:* ${*var*:*offset*} ${*var*:*offset*:*len*}	*offset is one based; when input lacks newlines:* `awk '{print substr($0,` *offset*`,` *len*`)}'`
length	${#*var*}		${#*var*}	${%*var*}	${#*var*}	`wc -m`
remove prefix greedily	`foo=do.re.mi` `${foo##*.}`		`foo=do.re.mi` `${foo##*.}`		`foo=do.re.mi` `${foo##*.}`	`sed 's/^.*\.'`
remove prefix reluctantly	`foo=do.re.mi` `${foo#*.}`		`foo=do.re.mi` `${foo#*.}`		`foo=do.re.mi` `${foo#*.}`	`sed 's/^[^\.]*\.'`
remove suffix greedily	`foo=do.re.mi` `${foo%%.*}`		`foo=do.re.mi` `${foo%%.*}`		`foo=do.re.mi` `${foo%%.*}`	`sed 's/\..*$'`
remove suffix reluctantly	`foo=do.re.mi` `${foo%.*}`		`foo=do.re.mi` `${foo%.*}`		`foo=do.re.mi` `${foo%.*}`	`sed 's/\.[^\.]*$'`
single substitution	`foo='do re mi mi'` `${foo/mi/ma}`		`foo='do re mi mi'` `${foo/mi/ma}`		`foo='do re mi mi'` `${foo/mi/ma}`	`sed 's/mi/ma/'`
global substitution	`foo='do re mi mi'` `${foo//mi/ma}`		`foo='do re mi mi'` `${foo//mi/ma}`		`foo='do re mi mi'` `${foo//mi/ma}`	`sed 's/mi/ma/g'`

prefix substitution	`foo=txt.txt` `${foo/#txt/text}`		`foo=txt.txt` `${foo/#txt/text}`		`foo=txt.txt` `${foo/#txt/text}`	`sed 's/^txt/text/'`
suffix substitution	`foo=txt.txt` `${foo/%txt/html}`		`foo=txt.txt` `${foo/%txt/html}`		`foo=txt.txt` `${foo/%txt/html}`	`sed 's/txt$/html/'`
upper case	`foo=lorem` `${foo^^}`		*none*		`foo=lorem` `${foo:u}`	`tr '[:lower:]'` `'[:upper:]'`
upper case first letter	`foo=lorem` `${foo^}`		*none*		*none*	
lower case	`foo=LOREM` `${foo,,}`		*none*		`foo=LOREM` `${foo:l}`	`tr '[:upper:]'` `'[:lower:]'`
lower case first letter	`foo=LOREM` `${foo,}`		*none*		*none*	
absolute path					`foo=~` `${foo:a}`	
dirname					`foo=/etc/hosts` `${foo:h}`	`foo=/etc/hosts` `dirname $foo`
basename					`foo=/etc/hosts` `${foo:t}`	`foo=/etc/hosts` `basename $foo`
extension					`foo=index.html` `${foo:e}`	
root					`foo=index.html` `${foo:r}`	

Brace, Tilde, Command, and Pathname Expansion

	bash	fish	ksh	tcsh	zsh
brace expansion: list	`echo {foo,bar}`	`echo {foo,bar}`	`echo {foo,bar}`	`echo {foo,bar}`	`echo {foo,bar}`
brace expansion: sequence	`echo {1..10}`	*none*	`echo {1..10}`	*none*	`echo {1..10}`
brace expansion: character sequence	`echo {a..z}`	*none*	`echo {a..z}`	*none*	*none*
tilde expansion	`echo ~/bin`	`echo ~/bin`	`echo ~/bin`	`echo ~/bin`	`echo ~/bin`
command expansion: dollar parens	`echo $(ls)`	`echo (ls)`	`echo $(ls)`	*none*	`echo $(ls)`
command expansion: backticks	`echo \`ls\``	*none*	`echo \`ls\``	`echo \`ls\``	`echo \`ls\``
process substitution	`wc <(ls)`	`wc (ls | psub)`	`wc <(ls)`	*none*	`wc <(ls)`
path expansion: string	`echo /bin/c*`	`echo /bin/c*`	`echo /bin/c*`	`echo /bin/c*`	`echo /bin/c*`
path expansion: character	`echo /bin/c??`	`echo /bin/c??`	`echo /bin/c??`	`echo /bin/c??`	`echo /bin/c??`
path expansion: character set	`echo /bin/[cde]*`	*none*	`echo /bin/[cde]*`	`echo /bin/[cde]*`	`echo /bin/[cde]*`
path expansion: negated character set	`echo /bin/[^cde]*`	*none*	`echo /bin/[^cde]*`	`echo /bin/[^cde]*`	`echo /bin/[^cde]*`
path expansion: sequence of characters	`echo /bin/[a-f]*`	*none*	`echo /bin/[a-f]*`	`echo /bin/[a-f]*`	`echo /bin/[a-f]*`

Special Variables

in zsh terminology, special means read-only variables that cannot have their type changed

non-alphabetical variables					
	bash	fish	ksh	tcsh	zsh
name of shell or shell script	`$0`	`(status -f)`	`$0`	`$0`	`$0`
command line arguments	`$1, $2, …`	`$argv[1], $argv[2], …`	`$1, $2, …`	`$1, $2, …`	`$1, $2, …` `$argv[1], $argv[2], …`
number of command line args	`$#`	`(count $argv)`	`$#`	`$#`	`$#` `$#argv`
arguments $1, $2, …	`$*` `$@`	*none*	`$*` `$@`	`$*`	`$*` `$@`
"$1" "$2" "$3" …	`"$@"`	`$argv`	`"$@"`		`"$@"`
"$1*c*$2*c*$3 …" where *c* is first character of $IFS	`"$*"`	`"$argv"`	`"$*"`		`"$*"`
process id	`$$`	`%self`	`$$`	`$$`	`$$`
process id of last asynchronous command	`$!`	*none*	`$!`	`$!`	`$!`
exit status of last non-asynchronous command	`$?`	`$status`	`$?`	`$?`	`$?`
previous command executed	`$_`	*current command executing:* `$_`	`$_`	`$_`	`$_`
command line options	`$-`	*none*	`$-`	*none*	`$-`
read input	*none*	*none*	*none*	`$<`	*none*

$* and $@

These parameters behave differently in double quotes.

Normally you should use "$@" to pass all the parameters to a subcommand. The subcommand will receive the same number of parameters as the caller received.

"$*" can be used to collect the parameters in a string. The first character of $IFS is used as the join separator. This could be used to pass all of the parameters as a single parameter to the subcommand.

Outside of double quotes, $* and $@ have the same behavior. Their behavior varies from shell to shell, however. In `bash` if you use them to pass parameters to a subcommand, the subcommand will receive more parameters than the caller if any of the parameters contain whitespace.

In `zsh` $* and $@ behave like "$@".

set by shell					
	bash	fish	ksh	tcsh	zsh
shell version	BASH_VERSION		KSH_VERSION	tcsh	ZSH_VERSION
return value of last syscall					ERRNO
history		history			
current line number of script	LINENO		LINENO		LINENO
set by getopts	OPTARG OPTIND		OPTARG OPTIND		OPTARG OPTIND
operating system and machine type					OSTYPE MACHTYPE
shell parent pid	PPID		PPID		PPID
working directory and previous working directory	PWD OLDPWD	PWD *none*	PWD OLDPWD		PWD OLDPWD
random integer	RANDOM	*built-in function:* random	RANDOM		RANDOM
return value	REPLY		REPLY		REPLY
seconds since shell was invoked	SECONDS		SECONDS		SECONDS
incremented each time a subshell is called	SHLVL				SHLVL

read by shell					
	bash	fish	ksh	tcsh	zsh
browser		BROWSER			
cd search path	CDPATH	CDPATH	CDPATH	cdpath	CDPATH cdpath
terminal width and height			COLUMNS LINES		COLUMNS LINES
command history editor	FCEDIT EDITOR		FCEDIT EDITOR		FCEDIT EDITOR
shell startup file	ENV		ENV		ENV
function definition search path			FPATH		fpath FPATH
history file path	HISTFILE		HISTFILE		HISTFILE
size of history	HISTSIZE		HISTSIZE		HISTSIZE
home directory	HOME	HOME	HOME		HOME
input field separators	IFS		IFS		IFS
locale	LANG	LANG			LANG
null redirect command					NULLCMD READNULLCMD
command search path	PATH	PATH	PATH		PATH
prompt customization *main, secondary, select, trace*	PS1 PS2 PS4		PS1 PS2 PS3 PS4		PS1 PS2 PS3 PS4
right prompt customization					RPS1 RPS2
terminal type	TERM				TERM
timeout			TMOUT		TMOUT
system tmp directory			TMPDIR		
user		USER			

Arithmetic and Conditional Expressions

	bash	fish	ksh	tcsh	zsh
test command	[-e /etc] test -e /etc	[-e /etc] test -e /etc	[-e /etc] test -e /etc		[-e /etc] test -e /etc
true command	true	true	true		true
false command	false	false	false		false
conditional command	[[ ]]		[[ ]]		[[ ]]
conditional expression				( )	
arithmetic expansion	$((1 + 1))	math '1 + 1'	$((1 + 1))		$((1 + 1))
floating point expansion	*none*	math '1.1 + 1.1'	$((1.1 + 1.1))		$((1.1 + 1.1))
let expression	let "*var = expr*"		let "*var = expr*"		let "*var = expr*"

external expression	`expr 1 + 1` `expr 0 '<' 1`	`expr 1 + 1` `expr 0 '<' 1`	`expr 1 + 1` `expr 0 '<' 1`	`expr 1 + 1` `expr 0 '<' 1`	`expr 1 + 1` `expr 0 '<' 1`
arithmetic command	`(( ))`		`(( ))`		`(( ))`
eval	`while true; do` `  read -p '$ ' cmd` `  eval $cmd` `done`	`while true` `  read cmd` `  eval $cmd` `end`	`while true; do` `  read cmd?'$ '` `  eval $cmd` `done`	`while (1)` `  echo -n '% '` `  eval $<` `end`	`while true; do` `  read cmd\?'$ '` `  eval $cmd` `done`
				`filetest`	

Expressions are implemented as either command expressions which return an integer status like a command, or variable expressions which evaluate to a string. Command expressions return a status of 0 for true and a nonzero status for false. Only commands and command expressions can be used as the conditional in *if*, *while*, and *until* statements.

Expressions which support arithmetic only support integer arithmetic.

	`[ ]`	`[[ ]]`	`$(( ))`	`(( ))`	`( )`	`expr`	`math`
name	test command	conditional command	arithmetic expansion	arithmetic command	conditional expression	external expression	
used as	*command*	*command*	*argument*	*command*	`tcsh` *conditionals*	*command*	`fish` *expressions*
word splitting?	*yes*	*no*					
expansions							
true	*anything but* `''`	*anything but* `''`	`1`	`1`	`1`	*anything but* `'' or 0`	
falsehoods	`''`	`''`	`0`	`0`	`0 ''`	`0 ''`	
logical operators	`-a -o !`	`&& || !`	`&& || !`	`&& || !`	`&& || !`	`& |` *none*	
regex comparison operator	*none*	`=~`	*none*	*none*		*str* `:` *regex*	
string comparison operators	`= !=`	`== !=`	*none*	*none*	`== !=`	`= > >= < <= !=` *but comparison is numeric if operands are digits*	
arithmetic comparison operators	`-eq -ne -lt -gt -le -ge`	`-eq -ne -lt -gt -le -ge`	`== != < > <= >=`	`== != < > <= >=`	`== != < > <= >=`	`= > >= < <= !=`	
arithmetic operators	*none*	*none*	`+ - * / % **`	`+ - * / % **`	`+ - * / %`	`+ - * / %`	
grouping	`\( \)`		`2 * (3 + 4)`			*use cmd substitution, ie. for bash:* `expr 2 \* $(expr 3 + 4)`	
assignment	*none*	*none*	`$(( n = 7 ))` `echo $n`	`(( n = 7 ))` `echo $n`			
compound assignment	*none*	*none*	`+= -= *= /= %=` *and others*	`+= -= *= /= %=` *and others*			
comma and increment	*none*	*none*	`$(( n = 7, n++ ))` `echo $n`	`(( n = 7, n++ ))` `echo $n`			
bit operators	*none*	*none*	`<< >> & | ^ ~`	`<< >> & | ^ ~`	`<< >> & | ^ ~`		
file tests	`-e EXISTS?` `-d DIR?` `-f REGULAR_FILE?` `-(h|L) SYMLINK?` `-p NAMED_PIPE?` `-r READABLE?` `-s NOT_EMPTY?` `-w WRITABLE?` `-x EXECUTABLE?` `-S SOCKET?`						

name

The name of the expression.

test command

conditional command

conditional expression

arithmetic expansion

let expression

external expression

arithmetic command

An arithmetic command can be used to test whether an arithmetic expression is zero.

Supports the same type of expressions as `$(( ))`.

true command

A no-op command with an exit status of 0. One application is to create an infinite loop:

```
while true; do
  echo "Are we there yet?"
done
```

false command

A no-op command with an exit status of 1. One application is to comment out code:

```
if false; then
  start_thermonuclear_war
fi
```

eval

How to evaluate a string as a shell command.

Arrays

	bash	fish	ksh	tcsh	zsh
declare	`typeset -a var`	*none*	*none*	*none*	`typeset -a var`
list all arrays	`typeset -a`	*none*	*none*	*none*	`typeset -a`
literal	`a=(do re mi)`	`set a do re mi`	`a=(do re mi)`	`set a = (do re mi)`	`a=(do re mi)`
lookup	`${a[0]}`	`$a[1]`	`${a[0]}`	`${a[1]}`	`${a[1]}` `$a[1]`
negative index lookup	*returns last element:* `${a[-1]}`	*returns last element:* `$a[-1]`	*returns last element:* `${a[-1]}`	*none*	*returns last element:* `${a[-1]}`
slice	`${a[@]:2:3}` `${a[*]:2:3}`	`$a[(seq 2 3)]`	`${a[@]:1:2}` `${a[*]:1:2}`	`${a[2-3]}`	`$a[2,3]`
update	`a[0]=do` `a[1]=re` `a[2]=mi`	`set a[1] do` `set a[2] re` `set a[3] mi`	`a[0]=do` `a[1]=re` `a[2]=mi`	`set a[1] = do` `set a[2] = re` `set a[3] = mi`	`a[1]=do` `a[2]=re` `a[3]=mi`
out-of-bounds behavior	*lookup returns empty string* *update expands array; array can have gaps*	*error message and nonzero exit status* *update expands array; in-between slots get empty strings*	*lookup returns empty string* *update expands array; array can have gaps*	*lookup and update both produce error message and nonzero exit status*	*lookup returns empty string* *update expands array; in-between slots get empty strings*
size	*highest index:* `${#a[@]}` `${#a[*]}`	`count $a`	*highest index:* `${#a[@]}` `${#a[*]}`	`${#a}`	`${#a}` `${#a[@]}` `${#a[*]}`
list indices	*can contain gaps:* `${!a[@]}` `${!a[*]}`	`(seq (count $a))`	*can contain gaps:* `${!a[@]}` `${!a[*]}`	`` `seq ${#a}` ``	`$(seq ${#a})`
regular reference	*return first element*	*return all elements joined by space*	*return first element*	*return all elements joined by space*	*return all elements joined by space*
regular assignment	*assigns to 0-indexed slot*	*convert array to regular variable*	*assigns to 0-indexed slot*	*convert array to regular variable*	*convert array to regular variable*
delete element	`unset a[0]`	`set -e a[1]` *re is now at index 1*			`a[0]=()`
delete array	`unset a[@]` `unset a[*]`	`set -e a`			`unset -v a`
pass each element as argument	*cmd* `"${a[@]}"`	*cmd* `$a`	*cmd* `"${a[@]}"`		*cmd* `"${a[@]}"`
pass as single argument	*cmd* `"${a[*]}"`	*cmd* `"$a"`	*cmd* `"${a[*]}"`		*cmd* `"${a[*]}"`

Shell arrays are arrays of strings. In particular arrays cannot be nested.

Arrays with one element are for the most part indistinguishable from a variable containing a nonempty string. Empty arrays are for the most part indistinguishable from a variable containing an empty string.

In the case of `bash` or `zsh`, it is possible to tell whether the variable is an array by seeing whether it is listed in the output of `typeset -a`.

declare

`bash` and `zsh` allow one to declare an array. This creates an empty array. There doesn't appear to be any need to do this, however,

list all arrays

literal

`bash` and `zsh` us parens to delimit an array literal. Spaces separate the elements. If the elements themselves contain spaces, quotes or backslash escaping must be used.

lookup

update

out-of-bounds behavior

size

list indices

regular reference

regular assignment

delete value

Deleting elements from a `bash` array leaves gaps. Deleting elements from a `zsh` arrays causes higher indexed elements to move to lower index positions.

delete array

Associative Arrays

	bash	fish	ksh	tcsh	zsh
declare	`typeset -A var`	*none*	*none*	*none*	`typeset -A var`
list all associative arrays	`typeset -A`	*none*	*none*	*none*	`typeset -A`
assign value	`foo[bar]=baz`	*none*	*none*	*none*	`foo[bar]=baz`
lookup	`${foo[bar]}`	*none*	*none*	*none*	`${foo[bar]}`
list indices	`${!foo[@]}` `${!foo[*]}`	*none*	*none*	*none*	
delete value	`unset "foo[bar]"`	*none*	*none*	*none*	`unset "foo[bar]"`
delete array	`unset "var[@]"`	*none*	*none*	*none*	`unset -v foo`

Associative arrays were added to `bash` with version 4.0.

Functions

	bash	fish	ksh	tcsh	zsh
define with parens	`foo() {` `  echo foo` `}`	*none*	`foo() {` `  echo foo` `}`	*none*	`foo() {` `  echo foo` `}`
define with keyword	`function foo {` `  echo foo` `}`	`function foo` `  echo foo` `end`	`function foo {` `  echo foo` `}`	*none*	`function foo {` `  echo foo` `}`
define with doc string		`function foo -d 'echo foo'` `  echo foo` `end`			
edit function definition		`funced foo`			*in .zshrc:* `autoload -U zed` *^J when done:* `zed -f foo`
parameters	`$1, $2, …`	`$argv[1], $argv[2], …`	`$1, $2, …`	*none*	`$1, $2, …`
number of parameters	`$#`	`(count $argv)`	`$#`	*none*	`$#`
return	`false() {` `  return 1` `}`	`function false` `  return 1` `end`	`false() {` `  return 1` `}`	*none*	`false() {` `  return 1` `}`
return values	`{0, …, 255}`	`{0, …, 2**31 - 1}` *negative values result in return value of "-"*	`{0, …, 255}`	*none*	`{-2**31, …, 2**31 - 1}` *other integers converted to one of the above values by modular*

		*values above 2**31 - 1 cause error*			*arithmetic*
local variables	`foo() {` `  local bar=7` `}` *variables set without the local keyword are global*	`function foo` `  set -l bar 7` `end` *without the -l flag, the the variable will be global if already defined, otherwise local*	*none*	*none*	`foo() {` `  local bar=7` `}` *variables set without the local keyword are global*
list functions	`typeset -f | grep '()'`	`functions`		*none*	`typeset -f | grep '()'`
show function	`typeset -f` *func*	`functions` *func*	`typeset -f` *func*		`typeset -f` *func*
delete function	`unset -f` *func*	`functions -e` *func*	`unset -f` *func*	*none*	`unset -f` *func* `unfunction` *foo*

define with parens

How to define a function.

POSIX calls for parens in the declaration, but parameters are not declared inside the parens, nor are parens used when invoking the function. Functions are invoked with the same syntax used to invoke external commands. Defining a function hides a built-in or an external command with the same name, but the built-in or external command can still be invoked with the `builtin` or `command` modifiers.

define with keyword

How to define a function using the `function` keyword.

define function with doc string

edit function definition

parameters

The variables which hold the function parameters.

Outside of a function the variables $1, $2, ... refer to the command line arguments provided to the script.

$0 always refers the name of the script in a non-interactive shell.

number of parameters

The variable containing the number of function parameters which were provided.

Outside of a function $# refers to the number of command line arguments.

return

If a function does not have an explicit `return` statement then the return value is the exit status of the last command executed. If no command executed the return value is 0.

return values

Shell functions can only return integers. Some shells limit the return value to a single byte. This is all the information one can get from the exit status of an external process according to the POSIX standard.

If a shell function needs to return a different type of value, it can write it to a global variable. All variables are global by default. The value in one of the parameters can be used to determine the variable to which the return value will be written. Consider this implementation of `setenv`:

```
setenv() {
  eval $1=$2
}
```

local variables

How to declare and set a local variable.

Local variables are normally defined inside a function. `bash` throws an error when an attempt is made to define a local outside a function, but `dash` and `zsh` do not.

Local variables have lexical, not dynamic scope. If a function recurses, locals in the caller will not be visible in the callee.

list functions

How to list the user defined functions.

`typeset -f` without an argument will show all function definitions.

`bash` and `zsh` always the function definitions with the paren syntax, even if the function keyword syntax was used to define the function.

show function

How to show the definition of a function.

delete function

How to remove a user defined function.

Command Resolution

	bash	fish	ksh	tcsh	zsh
alias: *define, list, remove, define suffix alias*	`alias ll='ls -l'` `alias` `unalias ll` *none*	`alias ltr 'ls -ltr'` `functions` `functions -e ltr` *none*	`alias ll='ls -l'` `alias` `unalias ll` *none*	`alias ll ls -l` `alias` `unalias ll` *none*	`alias ll='ls -l'` `alias -L` `unalias ll` `alias -s txt=cat`
built-ins: *run, list, help, enable, disable*	`builtin` *cmd* `enable -a` `help` *cmd* `enable` *cmd* `enable -n` *cmd*	`builtin` *cmd* `builtin -n` *cmd* `--help` *none* *none*	`builtin` *cmd* *none* *none* *none* *none*	*none* `builtins` *none* *none* *none*	`builtin` *cmd* *none* *type command name; then M-h* `enable` *cmd* `disable` *cmd*
run external command	`command` *cmd*	`command` *cmd*	`command` *cmd*		`command` *cmd*
run with explicit environment	`env -i` *var=val … cmd args …*				
external command hashes: *list, set, delete from, clear, rebuild*	`hash` *none* `hash -d` *cmd* `hash -r` *none*	*does not cache command paths*	`alias -t` `alias -t` *cmd=path* *none* `alias -r` *none*	*none* *none* *none* `rehash` *none*	`hash` `hash` *cmd=path* `unhash` `hash -r` `hash -f`
command type	`type` *cmd*	`type` *cmd*	`type` *cmd*		`type` *cmd*
command path	`command -v` *cmd*		`whence` *cmd*	`command -v` *cmd* `which` *cmd*	`command -v` *cmd* `which` *cmd* `whence` *cmd*
command paths				`where` *cmd*	`where` *cmd* `which -a` *cmd*

alias

Alias expansion is done after history expansion and before all other expansion. A command can be expanded by multiple aliases. For example the following will echo "baz":

```
alias bar=echo "baz"
alias foo=bar
foo
```

On the other hand the shells seem smart enough about aliasing to not be put into an infinite loop. The following code causes an error "foo not found":

```
alias foo=bar
alias bar=foo
foo
```

Alias definitions are not registered until an entire line of input is read. The following code causes an error "lshome not found":

```
alias lshome='ls ~'; lshome
```

User defined functions can replace aliases in the shells which have them; i.e. all shells except `tcsh`.

The Korn shell has a feature called tracked aliases which are identical to the external command hashes of the other shells.

built-ins

run external command

When resolving commands, user-defined functions take precedence over external commands. If a user-defined function is hiding an external command, the `command` modifier can be used to run the latter.

run with explicit environment

How to run a command with an explicit environment. `env -i` clears the environment of exported variables and only provides the external command with the environment variables that are explicitly specified. If the `-i` option is not specified then the environment is not cleared, which in many cases is no different than if the command had been run directly without the `env` command. The `env` command without the `-i` option is used in shebang scripts to avoid hard-coding the path of the interpreter.

Multiple environment variables can be set with the env command:

```
env -i VAR1=VAL1 VAR2=VAL2 ... CMD
```

external command hashes

External command hashes are a mapping from command names to paths on the file system.

The Korn Shell calls external command hashes "tracked aliasaes", and `ksh` defines `hash` as an alias for `alias -t`.

command type

Determine what type a command is. The possible types are alias, shell function, shell builtin, or a path to an external command. If the command is not found an exit status of 1 is returned.

command path

Return the absolute path for an external command. For shell functions and shell builtins the name of the command is returned. For aliases the statement used to define the alias is returned. If the command is not found an exit status of 1 is returned.

command paths

Arguments and Options

	bash	fish	ksh	tcsh	zsh
execute command and exit	`$ bash -c 'echo foo'`	`$ fish -c 'echo foo'`	`$ ksh -c 'echo foo'`	`$ tcsh -c 'echo foo'`	`$ zsh -c 'echo foo'`
usage	`$ bash --help`	`$ fish --help`		`$ tcsh --help`	`$ zsh --help`
interactive shell	`$ bash -i`	`$ fish -i`	`$ ksh -i`	`$ tcsh -i`	`$ zsh -i`
login shell	`$ bash -l` `$ bash --login`	`$ fish -l` `$ fish --login`	`$ ksh -l`	`$ tcsh -l`	`$ zsh -l` `$ zsh --login`
make posix compliant	`$ bash --posix`				
restricted mode	`$ bash -r` `$ bash --restricted`		`$ ksh -r`		`$ zsh -r` `$ zsh --restricted`
show version	`$ bash --version`	`$ fish --version`		`$ tcsh --version`	`$ zsh --version`
shift positional parameters: *by one, by n*	`shift` `shift n`		`shift` `shift n`	`shift` *none*	`shift` `shift n`
set positional parameters	`set -- arg …`		`set -- arg …`		`set -- arg …`
getopts	`getopts opts var`		`getopts opts var`		`getopts opts var`

options can be set by the script using `set`. Also `set -o` (bash) and pipefail.

execute command and exit

Shell executes a single command which is provided on the command line and then exits.

usage

Shell provides list of options and exits.

interactive shell

An interactive shell is one that is not provided a script when invoked as an argument or is not invoked with the `-c` option. The `-i` option makes a script interactive regardless. Typically an interactive shell gets its input from and sends its output to a terminal. An interactive shell ignores SIGTERM and will handle but not exit when receiving a SIGINT. Interactive shells display a prompt and enable job control. In an interactive shell the octothorpe # causes a syntax error, unlike in non-interactive shells where it is treated as the start of a comment.

login shell

A login shell is a special type of interactive shell. It executes different startup files and will also execute any logout files. When it exits it sends a SIGHUP to all jobs. (is this true?) A login shell ignores the `suspend` built-in.

make posix compliant

Change the behavior of the shell to be more POSIX compliant.

restricted mode

Shell runs in restricted mode.

show version

Show version and exit.

shift positional parameters

Outside of a function `shift` operates on the command line arguments. Inside a function `shift` operates on the function arguments.

set positional parameters

How to set the positional parameters from within a script.

getopts

How to process command line options.

`getopts` operates on the positional parameters $1, $2, ...

The first argument to `getopts` is a word specifying the options. The options are single characters which cannot be ':' or '?'. The colon ':' indicates that the preceding letter is an option which takes an argument. If an option is encountered which is not in the option word, `getopts` sets the variable to '?'.

```
while getopts a:b:c:def OPT
do
    case $OPT in
        a) OPTA=$OPTARG ;;
        b) OPTB=$OPTARG ;;
        c) OPTC=$OPTARG ;;
        d) OPTD=1 ;;
        e) OPTE=1 ;;
        f) OPTF=1 ;;
    esac
done
```

Execution Control

	bash	fish	ksh	tcsh	zsh
negate exit status	! *cmd*	not *cmd*	! *cmd*		! *cmd*
no-op command	:		:	:	:
break	break	break	break	break	break
case	case *arg* in *pattern*) *cmd*;; … *) *cmd*;; esac	switch *arg* case *pattern* … *cmd* … … case '*' *cmd* … end	case *arg* in *pattern*) *cmd*;; … *) *cmd*;; esac	switch (*arg*) case *pattern*: *cmd* … breaksw … default: *cmd* … breaksw endsw	case *arg* in *pattern*) *cmd*;; … *) *cmd*;; esac
continue	continue	continue	continue	continue	continue
for	for *var* in *arg* … do *cmd* … done	for *var* in *arg* … *cmd* … end	for *var* in *arg* … do *cmd* … done	foreach *var* (*arg* …) *cmd* … end	for *var* in *arg* … do *cmd* … done
goto				goto *label*	
if	if *test* then *cmd* … elif *test* then *cmd* … else *cmd* … fi	if *test* *cmd* … else if *test* *cmd* … else *cmd* … end	if *test* then *cmd* … elif *test* then *cmd* … else *cmd* … fi	if (*expr*) then *cmd* … else if (*expr*) then *cmd* … else *cmd* … endif	if *test* then *cmd* … elif *test* then *cmd* … else *cmd* … fi
repeat				repeat *count cmd*	repeat *count* do *cmd* … done
select	select *var* in *arg* … do *cmd* … done		select *var* in *arg* … do *cmd* … done		select *var* in *arg* … do *cmd* … done
until	until *test* do *cmd* … done		until *test* do *cmd* … done		until *test* do *cmd* … done
while	while *test* do *cmd* … done	while *test* *cmd* … end	while *test* do *cmd* … done	while (*expr*) *cmd* … end	while *test* do *cmd* … done

negate exit status

How to run a command and logically negate the exit status. This can be useful if the command is run as the conditional of a `if` statement.

The `!` precommand modifier converts a zero exit status to 1 and a nonzero exit status to 0.

The `!` must be separated from the command by whitespace, or it will be interpreted by the shell as a history substitution.

no-op command

break

Exits the enclosing for, select, until, or while loop.

case

The syntax for a switch statement.

Default clauses, which are indicated by the * pattern in most shells, are optional.

continue

Go to the next iteration of the enclosing for, select, until, or while loop.

for

A loop for iterating over a list of arguments.

`zsh` has alternate syntax which uses parens instead of the `in` keyword:

```
for VAR (ARG ...)
do
  CMD
  ...
done
```

goto

`tcsh` supports the `goto` statement. The target the first line containing just the *label* followed by a colon. Here's an example:

```
#/bin/tcsh
goto foo
echo "goto doesn't work!"
exit -1
foo:
echo "goto works"
```

if

The if statement.

The *test* which is the argument of `if` or `elif` can be any simple command, pipeline, or list of commands. The *test* executes and if the exit status is zero the corresponding clause is also executed.

Often the *test* which is the argument of `if` or `elif` will be one of the test operators: `test`, `[ ]`, `[[ ]]`, or `(( ))`.

The `elif` and `else` clauses are optional.

tcsh:

The argument of `if` and `elif` clauses must be an expression inside parens. Unlike the other shells it cannot be an arbitrary command. One can think of expressions as being built-in to the `tcsh` shell language rather than being delegated to specialized (albeit built-in) commands such as `test` and `[ ]`.

Note that the `then` keyword must be on the same line as the conditional expression. This is different from the POSIX syntax where the `then` keyword is separated from the test command by a newline or semicolon.

The `else if` and `else` clauses are optional.

`tcsh` has the following syntax for conditionally executing a single command:

```
if (EXPR) CMD
```

repeat

Here are a couple of ways to do something 10 times if you aren't using `tcsh`. Neither technique is POSIX compliant, however:

```
for i in `seq 1 10`; do echo "la"; done

for i in {1..10}; do echo "la"; done
```

select

The select statement creates a numbered menu inside an infinite loop. Each time the user selects one of the numbers the corresponding command is executed. The user can use ^D or EOF to exit the loop.

On each iteration *var* is set to the value corresponding to the number the user chose. The `break` keyword can be used to give the user a numbered option for exiting the loop.

until

The remarks above on if conditions also apply to the until loop condition.

while

The remarks above on if conditions also apply to the while loop condition.

Redirection

	bash	fish	ksh	tcsh	zsh
stdin from file	`tr a-z A-Z < file`	`tr a-z A-Z < file`	`tr a-z A-Z < file`	`tr a-z A-Z < file`	`tr a-z A-Z < file`
stdout to file	`ls > file`	`ls > file`	`ls > file`	`ls > file`	`ls > file`
stderr to file	`ls /not_a_file 2> file`	`ls /not_a_file ^ file`	`ls /not_a_file 2> file`	*none*	`ls /not_a_file 2> file`
stdout and stderr to file	`ls > file 2>&1`	`ls > file ^&1`	`ls > file 2>&1`	`ls >& file`	`ls > file 2>&1`
append stdout to file	`ls >> file`	`ls >> file`	`ls >> file`	`ls >> file`	`ls >> file`
append stderr to file	`ls 2>> file`	`ls ^^ file`	`ls 2>> file`	*none*	`ls 2>> file`
append stdout and stderr to file	`ls >> /tmp/bash.out 2>&1`	`ls >> /tmp/bash.out ^&1`	`ls >> /tmp/bash.out 2>&1`	`ls >>& file`	`ls >> /tmp/zsh.out 2>&1`
stdout to pipe	`ls | wc`	`ls | wc`	`ls | wc`	`ls | wc`	`ls | wc`
sdout and stderr to pipe	`ls 2>&1 | wc`	`ls ^&1 | wc`	`ls 2>&1 | wc`	`ls |& wc`	`ls 2>&1 | wc`
stdin from here-document	`wc << EOF` `do` `re` `mi` `EOF`	*none*	`wc << EOF` `do` `re` `mi` `EOF`	`wc << EOF` `do` `re` `mi` `EOF`	`wc << EOF` `do` `re` `mi` `EOF`
stdin from here-string	`wc <<< "do re mi"`	*none*	`wc <<< "do re mi"`	*none*	`wc <<< "do re mi"`
tee stdout	`ls | tee file | wc`				`ls > file | wc`
stdout to two files	`ls | tee file1 | tee file2 > /dev/null`				`ls > file1 > file2`
turn on noclobber	`set -o noclobber`		`set -o noclobber`	`set noclobber`	`set -o noclobber`
clobber file anyways	`ls >! /tmp/exists.txt`		`ls >! /tmp/exists.txt`	`ls >! /tmp/exists.txt`	`ls >! /tmp/exists.txt`
turn off noclobber	`set +o noclobber`		`set +o noclobber`	`unset noclobber`	`set +o noclobber`

A gap in the above chart is how to redirect just stderr to a pipe. One would guess by analogy with `2>` and `2>>` that this might work:

```
$ ls 2| wc
```

However, none of the shells support it. The correct syntax is:

```
$ ls 3>&1 1>&2 2>&3 | wc
```

The `3>&1` is equivalent to the C system call `dup2(1, 3)`. This makes file descriptor 3 a copy of file descriptor 1.

The `1>&2` is equivalent to the C system call `dup2(2, 1)`. This changes what file descriptor 1 writes to, but does not change what file descriptor 3 writes to, even though file descriptor 3 was initially a copy of file descriptor 1. The shell processes the redirect statements from left to right. Also note that the `1` could be omitted: `1>&2` and `>&2` are the same.

`zsh` only supports file descriptors 0 through 9, but `bash` supports higher numbered file descriptors. The shell always opens file descriptors 0, 1, and 2, commonly called `stdin`, `stdout`, and `stderr`, for each simple command that it invokes. If additional file descriptors are specified, those are also passed to the command. For example, if `foo` were invoked as:

```
$ foo 3> /tmp/bar.txt
```

then it could contain a system call which writes to file descriptor 3 without opening it first, e.g.

```
write(3, msg, strlen(msg));
```

Paths in the `/dev` directory can be used in place of `&1`, `&2`, ...

```
$ ls 3> /dev/fd/1 1> /dev/fd/2 2> /dev/fd/3 | wc

$ ls 3> /dev/stdout 1> /dev/stderr 2>&3 | wc
```

tcsh:

It is possible to redirect stdout and stderr to different files:

```
$ ( ls > /tmp/stdout.txt ) >& /tmp/stderr.txt
```

Echo and Read

	bash	fish	ksh	tcsh	zsh
echo *with newline, without newline*	`echo` *arg …* `echo -n` *arg …*	`echo` *arg …* `echo -n` *arg …*	`echo` *arg …* `echo -n` *arg …*	`echo` *arg …* `echo -n` *arg …*	`echo` *arg …* `echo -n` *arg …*
printf	`printf` *fmt arg …*	`printf` *fmt arg …*	`printf` *fmt arg …*	`printf` *fmt arg …*	`printf` *fmt arg …*
read *read values separated by* IFS*; with prompt; without backslash escape*	`read` *var …* `read -p` *str var* `read -r` *var …*	`read` *var …* `read -p 'echo` *str*`' ` *var*	`read` *var …* `read` *var*`?`*str* `read -r` *var …*	`echo -n` *str* `set` *var*`=$<`	`read` *var …* `read` *var*`\?`*str* `read -r` *var …*

echo

How to echo the arguments separated by spaces and followed by a newline; how to suppress the trailing newline.

The POSIX standard says that `echo` should not have any options. It also says, perhaps contradicting itself, that if the first argument is `-n` then the behavior is implementation dependent.

The POSIX standard also says that if any of the arguments contain backslashes, then the behavior is implementation dependent. Historically implementations have used the `-E` and `-e` options to enable or disable the interpretation of C-style backslash escape sequences.

`fish` provides an `-s` option for printing the arguments without spaces in-between.

Because if the ill-defined behavior of `echo`, POSIX-compliant scripts use `printf` instead.

printf

`printf` is an external command line tool, though `zsh` also has a built-in version.

man 3 printf

Like its counterpart from the C standard library, `printf` does not write a newline to stdout unless one is specified in the format using a backslash escape sequence.

Unfortunately, the supported backslash ecscapes are system dependent, though some of them are mandated by POSIX:

	posix	bsd	gnu
backslash escapes	`\a \b \c \f \n \r \t \v \\` `\`*o* `\`*oo* `\`*ooo*	`\a \b \c \f \n \r \t \v \\ \'` `\`*o* `\`*oo* `\`*ooo*	`\a \b \c \e \f \n \r \t \v \\ \"` `\`*o* `\`*oo* `\`*ooo* `\x`*hh* `\u`*hhhh* `\U`*hhhhhhhh*

An interesting backslash escape is \c, which causes the rest of the format to be ignored.

In a printf format, format specifiers are of the form `%d`, `%f` and `%s`.

	posix	bsd	gnu
format specifiers		`diouxX` `fFaAeEgG` `csb`	`diouxX` `feEgG` `csb`

format specifiers; many of which are useless in this context because of fewer types

how invalid arguments are handled

%%

extra specifiers with floats

extra specifiers with strings

read

How to read a line of input into one or more variables.

When multiple variables are specified the value of `IFS` which by default contains the whitespace characters is used to split the input. If there are fewer variables than split values, then the last variable will contain a concatenation of the remaining values with their original separators. If there are fewer values then the extra variables are set to the empty string.

`bash` and `dash` use the `-p` option to set a prompt. `ksh` and `zsh` use a ?*str* suffix appended to the first variable to set the prompt.

`fish` uses the `-p` option, but it evaluates the string to produce the prompt. This makes it possible to set the color of the prompt:

```
read -p 'set_color green; echo -n "> "; set_color normal' foo
```

The user can put a backslash in front of a newline to split the input up over multiple lines. The backslash and newline are stripped from the input. The user can put backslash into the variable by entering two backslashes. The `-r` option disables this feature, allowing the user to enter literal backslashes with a single keystroke.

`tcsh` gets input from the user by reading from the special variable `$<`. Backslashes are always interpreted literally.

Files and Directories

	bash	fish	ksh	tcsh	zsh
change current directory *change dir, to home dir, to previous dir, show physical dir, no symlink dir*	`cd` *dir* `cd` `cd -` `cd -P` *dir* *none*	`cd` *dir* `cd` `cd -` *none* *none*	`cd` *dir* `cd` `cd -` `cd -P` *dir* *none*	`cd` *dir* `cd` `cd -` *none* *none*	`cd` *dir* `cd` `cd -` `cd -P` *dir* `cd -s` *dir*
directory stack: *push, pop, list*	`pushd` *dir* `popd` `dirs`	`pushd` *dir* `popd` `dirs`		`pushd` *dir* `popd` `dirs`	`pushd` *dir* `popd` `dirs`
print current directory	`pwd`	`pwd`	`pwd`	`pwd`	`pwd`
source	`source` *file arg …* `.` *file arg …*	`source` *file arg …* `.` *file*	`source` *file arg …* `.` *file arg …*	`source` *file arg …*	`source` *file arg …* `.` *file arg …*
umask *set umask in octal, in symbolic chmod format; show umask in octal, in symbolic chmod format*	`umask 022` `umask g-w,o-w` `umask` `umask -S`	`umask 022` `umask g-w,o-w` `umask` `umask -S`	`umask 022` `umask g-w,o-w` `umask` `umask -S`	`umask 022` *none* `umask` *none*	`umask 022` `umask g-w,o-w` `umask` `umask -S`

change current directory

Change the current directory to the specified directory. If the directory starts with a slash '/' then it is taken to be an absolute path. If it does not it is treated as a relative path and CDPATH is used as a colon separated list of starting directories. By default CDPATH is empty in which case the current directory '.' is used as a starting point. See also the section on tilde expansion.

If there is no argument then the current directory is changed to $HOME.

If the argument is a hyphen '-' then the current directory is changed to $OLDPWD which is the most recent former current directory.

When the `-P` option is used, `PWD` will be set to the physical path of the current directory; i.e. any symbolic links will be resolved. If the current directory is being displayed in the prompt this will also be set the physical path.

zsh:

When the `-s` option is used, attempting to change directory into a path containing symlinks will fail.

directory stack

Push a directory provided as an argument onto the directory stack. The directory becomes the current directory.

Pop a directory off the directory stack. The popped directory becomes the current directory.

List the directory stack.

print current directory

Show the current directory. The same as executing:

```
echo $PWD
```

source

The `source` built-in executes the commands in another file using the current shell process and environment.

Some shells have a non-POSIX feature which allows arguments to be passed to the file being sourced; i.e. the following invocation would set `$1`, `$2`, and `$3` to `bar`, `baz`, and `quux` while executing `foo.sh`:

```
source foo.sh bar baz quux
```

The `.` syntax is part of the POSIX standard, but the `source` syntax is not.

The file to be sourced may be specified with an absolute path. Some shells will also search the working directory or `PATH` for the file to be sourced:

	bash	fish	ksh	tcsh	zsh
searches working directory	yes	yes	no	yes	`.` no, `source` yes
searches PATH	yes	no	no	no	yes

umask

Set the shell file mode creation mask. `umask` is a POSIX syscall.

The mask consists of 3 octal digits which apply to the user, group, and other permissions respectively. Each octal digit contains 3 bits of information. In order of most to least significant the bits apply to the read, write, and execute permissions.

Setting a bit in the mask guarantees that the corresponding bit in the file permissions will not be set when a file is created. The logic for computing the file permissions can be expressed with the following shell code:

```
mask=8#022
perms=8#777

printf "0%o\n" $(( $perms & ~ $mask ))
```

Here is the same logic in C code:

```
unsigned int mask = 0022;
unsigned int perms = 0777;

printf("%o\n", perms & ~mask);
```

If `umask` is given a numeric argument it is always interpreted as octal; a leading zero is not required.

`umask` also supports the symbolic notation used by chmod. In this case the argument is one or more 3 character sequences of the format `[agou][-+][rwx]` separated by commas.

Process and Job Control

	bash	fish	ksh	tcsh	zsh
run job in background	bg	bg	bg	bg	bg
protect job from hangup signal	disown	*does not SIGHUP background jobs on exit*	disown		disown
execute file	exec [-c]	exec	exec	exec	exec
exit	exit [n]	exit	exit	exit	exit bye
run job in foreground	fg	fg	fg	fg	fg
				hup	
list jobs	jobs [-lnprs]	jobs	jobs	jobs	jobs
send signal	kill	*external, but …* kill	kill	kill	kill
				limit	limit
				login	
	logout			logout	logout
				nice	
				nohup	
				onintr	
				sched	sched
			sleep		
				stop	
	suspend		suspend	suspend	suspend
			time	time	time
	times		times		times
	trap	trap	trap		trap
	ulimit		ulimit		ulimit
		ulimit		unlimit	unlimit
	wait		wait	wait	wait

`xargs` splits standard input on spaces and newlines and feeds the arguments to argument of `xargs` which is executed as a command. The input delimiter can be changed to null characters with the -0 flag (useful with `find -print0`) or to the value of the -d flag argument.

By default if the length of the input is more than 4096 characters the input will be broken up and the command run multiple times. This number can be increased with the -s flag up to system configuration variable ARG_MAX. It is also possible to call the command multiple times feeding it a prescribed number of arguments each time using the -n flag. The -t flag will write to standard error the command that is being invoked and its arguments before each invocation.

The -P flag can be used to for parallelization. The argument is the max number of simultaneous processes.

run job in background

protect job from hangup signal

execute file

History

history commands				
bash	fish	ksh	tcsh	zsh

command history: *list recent, list all, list with time, unnumbered list*	`fc -l` `history` *`set`* `HISTTIMEFORMAT` `fc -ln`	`history | nl | head` `history | nl` `cat ~/.config` `/fish/fish_history` `history`	*`??`* `fc -l 1` *`none`* *`??`*	`history` `15` `history` `history` `-T` *`none`*	`history` `history 1` `history` `-f` `history` `-n`
command history: *run, find and run*	*`!num`* `fc -s` *`str`*		`r` *`num`* `fc -s`	*`none`* *`none`*	*`!num`* *`??`*
command history: *delete from history, clear history*	`history -d` *`num`* `history -c`		*`none`* *`none`*	*`none`* `history` `-c`	*`none`* *`none`*
command history: *fix, find and substitute*	`fc` *`num`* `fc -s` *`old=new str`*		`fc` *`num`* `fc -s` *`old=new`* *`str`*		`fc` *`num`* *`none`*
command history: *write to file, append to file, read from file*	`history -w` *`path`* `history -a` *`path`* `history -r` *`path`*				`fc -W` *`path`* `fc -A` *`path`* `fc -R` *`path`*

command history: listing

How to list recent commands; how to list all commands; how to list commands with the time they were run.

command history: running

How to run a command in the history by command number; how to run the most recent command in the history matching a prefix.

command history: deleting

How to delete a command from the history by command number; how to clear the command history.

command history: fixing

Use the following syntax to edit commands from the history list and run them:

```
fc [-e EDIT_CMD] [-r] [FIRST [LAST]]
```

If EDIT_CMD is not specified, the value in the FCEDIT or EDITOR environment variable is used.

If FIRST and LAST are specified, these indicate the numbers of the range of commands to edit. If FIRST is specified but LAST is not, only that command at that number is edited and run. If neither is specified the last command is edited and run.

The -r flag reverses the order of the commands.

To simply list commands the following flags can be used:

```
fc -l[r] [FROM]
fc -l[r] -NUMBER_CMDS
```

If neither FROM nor -NUMBER_CMDS is specified the last 16 commands is printed. Use -NUMBER_CMDS (i.e. a negative number) to list the last NUMBER_CMDS commands. Use FROM (i.e. a positive number) to list all commands from FROM on.

The -r flag reverses the order of the commands

To rerun a recent command without editing it use:

```
fc -s [PAT=REP] [START_OF_CMD]
```

If START_OF_CMD is specified the last command that starts with START_OF_CMD will be run. If START_OF_CMD is not specified the last command will be run.

If PAT=REP is specified then each occurrence of PAT will be replaced with REP in the command before it is run.

ksh:

`hist` is a synonym for `fc` with the sole difference that HISTEDIT is the environment variable that determines the editor instead of FCEDIT.

zsh:

`r` is an alias for `fc -s`

command history file

history expansion					
	bash	fish	ksh	tcsh	zsh
most recent command	`!!`	*none*	*none*	`!!`	`!!`
n-th command	*`!n`*	*none*	*none*	*`!n`*	*`!n`*

most recent command starting with str	*!str*	*none*	*none*	*!str*	*!str*
most recent command with substitution	*^pattern^replacement*	*none*	*none*	*^pattern^replacement*	*^pattern^replacement*
nth command with substitution	*!n*:s/*pattern*/*replacement*/	*none*	*none*	*!n*:s/*pattern*/*replacement*/	*!n*:s/*pattern*/*replacement*/
n-th command with global substitution	*!n*:gs/*pattern*/*replacement*/	*none*	*none*	*!n*:gs/*pattern*/*replacement*/	*!n*:gs/*pattern*/*replacement*/
most recent arguments	!*	*none*	*none*		!*
first of most recent arguments	!:1	*none*	*none*		!:1
range of most recent arguments	!:*n-m*	*none*	*none*		!:*n-m*
last of most recent arguments	!$	*none*	*none*		!$
most recent command without arguments	!:0	*none*	*none*		!:0
m-th argument of n-th command	*!n:m*	*none*	*none*		*!n:m*

history file					
	bash	fish	ksh	tcsh	zsh
location	HISTFILE=~/.bash_history	~/.config/fish/fish_history	HISTFILE=~/.ksh_history	set histfile ~/.tcsh_history	HISTFILE=~/.zsh_history
memory size	HISTSIZE=2000		HISTSIZE=2000		HISTSIZE=2000
file size	HISTFILESIZE=2000			set savehist=2000	SAVEHIST=2000
format	*lines of input*				
timestamps	HISTTIMEFORMAT=%s				
update time	*on exit*				*on exit*
update method	*appends to file;* *to only keep most recent dupe:* HISTCONTROL=erasedups			*appends to file;* *to sort in memory file and most recent by timestamp and only keep the most recent, use:* set savehist=2000 merge	
ignore	HISTIGNORE=history:whoami				

Key Bindings

	bash	fish	ksh	tcsh	zsh
list keybindings	bind -P	bind		bindkey	bindkey
list keymaps	help bind	*none*		*none*	bindkey -l
current keymap name	bind -V | grep keymap	*none*		*none*	
change keymap	bind 'set keymap emacs'	*none*		*none*	bindkey -A emacs main
list bindable functions	bind -l	bind -f		bindkey -l	
bind key to function	bind C-a:beginning-of-line	bind \ca beginning-of-line			
restore default binding for key					

bash and zsh have keymaps

how to create a new keymap with zsh

alternate fish syntax referring to keys

Startup Files

	bash	fish	ksh	tcsh	zsh
non-interactive shell startup files	$BASH_ENV	~/.config/fish/config.fish	$ENV	/etc/csh.cshrc ~/.tcshrc ~/.cshrc	/etc/zshenv $ZDOTDIR/.zshenv
login shell startup files	/etc/profile ~/.bash_profile ~/.bash_login ~/.profile	~/.config/fish/config.fish	/etc/profile ~/.profile $ENV	/etc/csh.login ~/.login	*non-interactive startup files* /etc/zprofile $ZDOTDIR/.zprofile /etc/zshrc $ZDOTDIR/.zshrc /etc/zlogin $ZDOTDIR/.zlogin
other interactive shell startup files	~/.bashrc	~/.config/fish/config.fish	$ENV	*none*	*non-interactive startup files* /etc/zshrc $ZDOTDIR/.zshrc

login shell logout files	`~/.bash_logout`	*none*	*none*	`/etc/csh.logout` `~/.logout`	`$ZDOTDIR/.zlogout` `/etc/zlogout`

bash:

When logging in `bash` will only execute one of `~/.bash_profile`, `~/.bash_login`, or `~/.profile`. It executes the first file that exists.

fish:

The startup file `.config/fish/config.fish` is run by all shells. Here is how to put code in it which only executes at login:

```
if status --is-login
  set PATH $PATH ~/bin
end
```

How to define an exit handler:

```
function on_exit --on-process %self
  echo fish is exiting ...
end
```

Prompt Customization

	bash	fish	ksh	tcsh	zsh
set primary prompt	`PS1='$ '`	`function fish_prompt` `  echo -n '$ '` `end`	`PS1='$ '`	`set prompt='$ '`	`PS1='$ '`
set continued line prompt	`PS2='> '`	*none*	`PS2='> '`	`set prompt2='> '`	`PS2='> '`
set select prompt	`PS3='? '`	*none*	`PS='? '`	*none*	`PS3='? '`
set right prompt	*none*	`function fish_right_prompt` `  date` `end`		`set rprompt='%Y-%W-%D %p'`	`RPS1='%D{%F %T}'`
set right continued line prompt	*none*	*none*		*none*	`RSP2='...'`
dynamic information					
working directory	*none*	`pwd`		`%/`	`%d` `%/`
working directory with tilde abbrev	`\w`	*abbreviate path components other than basename with single letter:* `prompt_pwd`		`%~`	`%~`
trailing components of working directory				`%3C`	`%3d`
command number in history	`\!`		`!`	`!` `%!` `%h`	`%!` `%h`
command number in session	`\#`				
shell version	`\v`				
shell level	`$SHLVL`				
environment variable	`$var`	`echo -n $var`	`$var`	`%$var`	`$var`
command substitution	`$(cmd)`		`$(cmd)`		`$(cmd)`
host name	`\h` `\H`				`%m` `%M`
user	`\u`			`%n`	`%n`
number of jobs	`\j`			`%j`	`%j`
tty					`%y`
last command exit status				`%?`	`%?`
conditional expression					
shell privilege indicator					`%#`
continued line info					
date and time	`\D{strftime_format}`				`%D{strftime_format}`
text effects and escapes					
escapes	`\\ \[ \]`			`%% %{ %}`	`%% %{ %}`
bold				`%B %b`	`%B %b`
underline				`%U %u`	`%U %u`
standout				`%S %s`	`%S %s`
foreground color					`%F{red} %f`
background color					`%K{green} %k`

Most shells permit a user to customize the prompt by setting an environment variable. `fish` requires that the user define a callback function.

The *primary prompt* is the prompt the user sees the most often.

The *continued line prompt* is used when the user types an incomplete command. This can happen when there are open parens, braces, or quote in the command, or the user backslash escaped the newline.

The *select prompt* is used to prompt the user to make a multiple choice selection. It corresponds to the select execution control statement.

The *right prompt* appears at the far right side of the input line. If the user types enough input to need the space, the right prompt disappears.

dynamic information

`bash`, `tcsh`, and `zsh` provide a set of special character sequences for putting dynamic information in the prompt. In the case of `bash` the sequences start with a backslash and in the case of `tcsh` and `zsh` a percent sign.

`bash`, `ksh`, `tcsh`, and `zsh` will also perform variable expansion on anything that starts with a dollar sign and looks like a variable before each display of the prompt. `bash`, `ksh`, and `zsh` will also perform command substitution before each display of the prompt when they encounter the `$( )` syntax in the prompt.

text effects and escapes

Autoload

fish:

zsh:

bash (1989)

bash

The Bourne Again shell is a GNU replacement for the Bourne shell. It can run almost all Bourne scripts and POSIX compliant scripts, and operating systems often use `bash` as `/bin/sh`. Because `bash` has many extensions it is not a good shell to use for determining POSIX compliance.

csh (1978)

csh

The C shell was written by Bill Joy and released as part of the second Berkeley Standard Distribution.

It introduced features that were widely adopted by other shells: history expansion, aliases, tilde notation, and job control.

The C shell was so named because it looked more like C than the Bourne shell. It still used keywords to mark off blocks instead of curly braces, but its expressions were delimited by parens instead of square brackets and relational operators such as < and <= could be used instead of -lt and -le. The Unix community nevertheless eventually chose a derivation of the Bourne shell as the standard scripting language and writing scripts for the C shell is not recommended.

The classic Macintosh operating system had a development environment called The Mac Programmer's Workbench. It included a shell that was derived from the C shell.

dash (2002)

dash

The Debian Almquist shell, `dash`, was originally a Linux port of the NetBSD Almquist shell, `ash`. It is POSIX compliant. It is also smaller than the other shells: on Ubuntu Linux the executable is about 100k whereas the other shells are in the 300k–900k range.

`dash` does not keep a command history or offer command line editing. It does have job control, though.

fish (2005)

Fish user documentation

ksh (1983)

ksh

The Korn shell added history and job control but otherwise stayed consistent with the Bourne shell. The POSIX standard for the shell was based on the Korn shell.

The Korn shell was proprietary software until 2000, which is why clones such as `pdksh` were written. Also, `zsh` can be used to emulate `ksh`; both Mac OS X and Ubuntu link `ksh` to `zsh`.

rc (1989)

The `rc` shell was released as part of 10th Edition Unix. It was also the Plan 9 shell.

sh

POSIX 2008

A succession of shells have been installed at `/bin/sh` which are known today by the engineers who implemented them: the Thompson shell, the Mashey shell, and the Bourne shell.

The Bourne shell appeared in 1977. It introduced the execution control structures that are used in most of the modern Unix shells. These control structures, with their distinctive reversed words for marking the end of blocks: `fi` and `esac`, were borrowed from Algol 68. However, where Algol 68 uses `od` the Bourne shell uses `done`. This was because a Unix command named `od` already existed. The Bourne shell also introduced arbitrary length variable names; the Mashey shell by contrast was limited to single letter variable names.

Whatever is installed at `/bin/sh` should probably be POSIX compliant. Mac OS X uses `bash`, which changes its behavior somewhat and operates in POSIX mode when invoked as `sh`. One can also get this behavior by invoking `bash` with the `--posix` flag.

Ubuntu makes `/bin/sh` a symlink to `/bin/dash`.

tcsh (1981)

tcsh

The TENEX C shell, `tcsh`, was upgraded version of the C Shell which added tab completion, a feature originally used in the TENEX operating system.

`tcsh` is backwardly compatible with `csh` and on many systems `csh` is simply a symlink to `tcsh`.

`tcsh` is the default shell on FreeBSD and it was the default shell on Mac OS X until version 10.3 was introduced in 2003.

Writing scripts in `tcsh` is not recommended for the same reasons writing scripts in `csh` is not recommended.

The following `tcsh` built-ins interact with the terminal settings:

- echotc
- settc
- setty
- telltc
- termname

zsh (1990)

The Z shell, `zsh`, is documented by multiple man pages:

man page	topics covered
zshall	all topics in one man page
zsh	startup files
zshoptions	options
zshbuiltins	built-ins
zshcompwid, zshcompsys	tab completion
zshcompctl	old tab completion system
zshexp	history expansion; parameter expansion; process, tilde, command, and pathname expansion
zshmisc	grammar; keywords; quoting; redirection; arithmetic and conditional expressions; prompt customization
zshparam	special variables
zshzle	readline

`zsh` has these builtins for managing the completion module:

- comparguments
- compcall
- compctl
- compdescribe
- compfiles
- compgroups
- compquote
- comptags
- comptry
- compvalues

The following `zsh` built-ins interact with the terminal settings:

- echotc
- echoti
- getcap
- ttyctl

Special `zsh` builtins:

- autoload
- zcompile
- zformat
- zmodload
- zparseopts
- zstyle