# Chapter 4

# The Bourne-Again Shell

**Reviewers point out that 'bash' and 'Bash' are used more or less interchangeably: please pick one and stick to it.**

## 4.1 Introduction

A Unix shell provides an interface that lets the user interact with the operating system by running commands. But a shell is also a fairly rich programming language: there are constructs for flow control, alternation, looping, conditionals, basic mathematical operations, named functions, string variables, and two-way communication between the shell and the commands it invokes.

Shells can be used interactively, from a terminal or terminal emulator such as xterm, and non-interactively, reading commands from a file. Most modern shells, including bash, provide command-line editing, in which the command line can be manipulated using emacs- or vi-like commands while it's being entered, and various forms of a saved history of commands.

Bash processing is much like a shell pipeline: after being read from the terminal or a script, data is passed through a number of stages, transformed at each step, until the shell finally executes a command and collects its return status.

This chapter will explore bash's major components: input processing, parsing, the various word expansions and other command processing, and command execution, from the pipeline perspective. These components act as a pipeline for data read from the keyboard or from a file, turning it into an executed command.

### 4.1.1 Bash

Bash is the shell that appears in the GNU operating system, commonly implemented atop the Linux kernel, and several other common operating systems, most notably Mac OS X. It offers functional improvements over historical versions of sh for both interactive and programming use.

The name is an acronym for Bourne-Again SHell, a pun combining the name of Stephen Bourne (the author of the direct ancestor of the current Unix shell `/bin/sh`, which appeared in the Bell Labs Seventh Edition Research version of Unix) with the notion of rebirth through reimplementation. The original author of Bash was Brian Fox, an employee of the Free Software Foundation. I am the current developer and maintainer, a volunteer who works at Case Western Reserve University in Cleveland, Ohio.

Like other GNU software, Bash is quite portable. It currently runs on nearly every version of Unix and a few other operating systems—independently-supported ports exist for hosted Windows

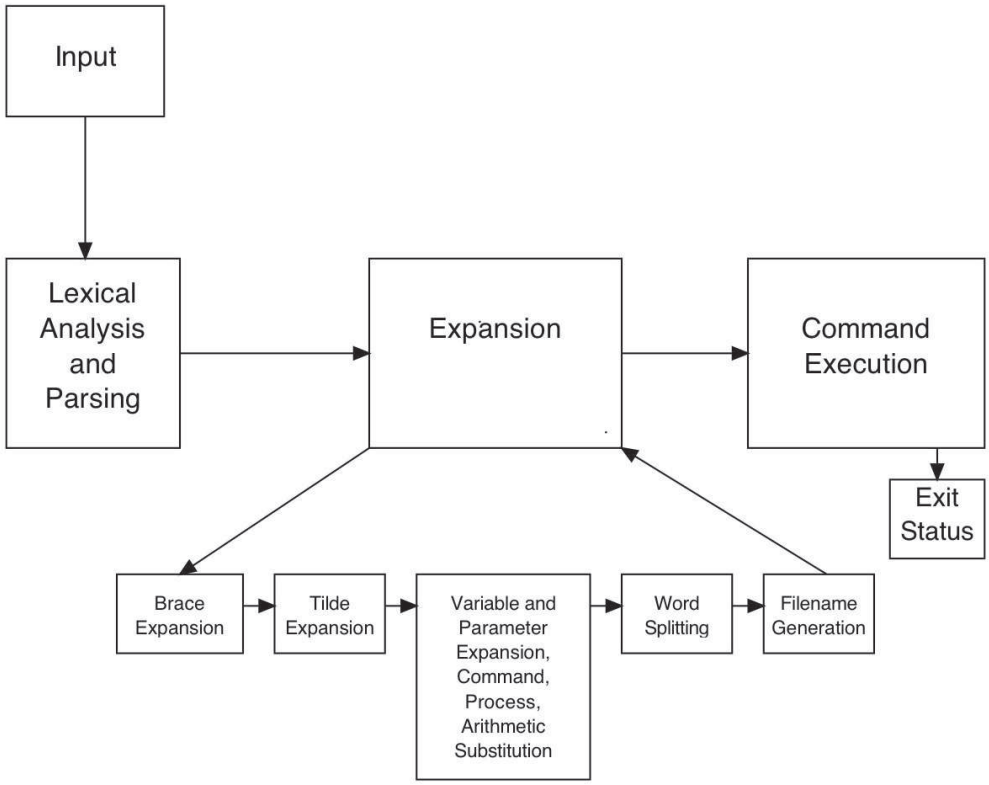


Figure 4.1: Bash Component Architecture

environments such as Cygwin and MinGW, and ports to Unix-like systems such as QNX and Minix are part of the distribution. It only requires a Posix environment to build and run, such as one provided by Microsoft's Services for Unix (SFU).

## 4.2 Syntactic Units and Primitives

### 4.2.1 Primitives

To bash, there are basically three kinds of tokens: reserved words, words, and operators. Reserved words are those that have meaning to the shell and its programming language; usually these words introduce flow control constructs, like `if` and `while`. Operators are composed of one or more metacharacters: characters that have special meaning to the shell on their own, such as `|` and `>`. The rest of the shell's input consists of ordinary words, some of which have special meaning—assignment statements or numbers, for instance—depending on where they appear on the command line.

### 4.2.2 Variables and Parameters

As in any programming language, shells provide variables: names to refer to stored data and operate on it.

The shell provides basic user-settable variables and some built-in variables referred to as parameters. Shell parameters generally reflect some aspect of the shell's internal state, and are set automatically or as a side effect of another operation.

Variable values are strings. Some values are treated specially depending on context; these will be explained later.

Variables are assigned using statements of the form `name=value`. The `value` is optional; omitting it assigns the empty string to `name`. If the value is supplied, the shell expands the value and assigns it to `name`. The shell can perform different operations based on whether or not a variable is set, but assigning a value is the only way to set a variable. Variables that have not been assigned a value, even if they have been declared and given attributes, are referred to as *unset*.

A word beginning with a dollar sign introduces a variable or parameter reference. The word, including the dollar sign, is replaced with the value of the named variable. The shell provides a rich set of expansion operators, from simple value replacement to changing or removing portions of a variable's value that match a pattern.

There are provisions for local and global variables. By default, all variables are global. Any simple command (the most familiar type of command—a command name and optional set of arguments and redirections) may be prefixed by a set of assignment statements to cause those variables to exist only for that command. The shell implements stored procedures, or shell functions, which can have function-local variables.

Variables can be minimally typed: in addition to simple string-valued variables, there are integers and arrays. Integer-typed variables are treated as numbers: any string assigned to them is expanded as an arithmetic expression and the result is assigned as the variable's value. Arrays may be indexed or associative; indexed arrays use numbers as subscripts, while associative arrays use arbitrary strings. Array elements are strings, which can be treated as integers if desired. Array elements may not be other arrays.

Bash uses hash tables to store and retrieve shell variables, and linked lists of these hash tables to implement variable scoping. There are different variable scopes for shell function calls and temporary scopes for variables set by assignment statements preceding a command. When those assignment statements precede a command that is built into the shell, for instance, the shell has to keep track of the correct order in which to resolve variable references, and the linked scopes allow bash to do that. There can be a surprising number of scopes to traverse depending on the execution nesting level.

### 4.2.3 The Shell Programming Language

A *simple* shell command, one with which most readers are most familiar, consists of a command name, such as `echo` or `cd`, and a list of zero or more arguments and redirections. Redirections allow the shell user to control the input to and output from invoked commands. As noted above, users can define variables local to simple commands.

Reserved words introduce more complex shell commands. There are constructs common to any high-level programming language, such as `if-then-else`, `while`, a `for` loop that iterates over a list of values, and a C-like arithmetic `for` loop. These more complex commands allow the shell to execute a command or otherwise test a condition and perform different operations based on the result, or execute commands multiple times.

One of the gifts Unix brought the computing world is the pipeline: a linear list of commands, in which the output of one command in the list becomes the input of the next. Any shell construct can be used in a pipeline, and it's not uncommon to see pipelines in which a command feeds data to a loop.

Bash implements a facility that allows the standard input, standard output, and standard error streams for a command to be redirected to another file or process when the command is invoked.

Shell programmers can also use redirection to open and close files in the current shell environment.

Bash allows shell programs to be stored and used more than once. Shell functions and shell scripts are both ways to name a group of commands and execute the group, just like executing any other command. Shell functions are declared using a special syntax and stored and executed in the same shell's context; shell scripts are created by putting commands into a file and executing a new instance of the shell to interpret them. Shell functions share most of the execution context with the shell that calls them, but shell scripts, since they are interpreted by a new shell invocation, share only what is passed between processes in the environment.

### 4.2.4 A Further Note

As you read further, keep in mind that the shell implements its features using only a few data structures: arrays, trees, singly-linked and doubly-linked lists, and hash tables. Nearly all of the shell constructs are implemented using these primitives.

The basic data structure the shell uses to pass information from one stage to the next, and to operate on data units within each processing stage, is the `WORD_DESC`:

```
typedef struct word_desc {
  char *word;           /* Zero terminated string. */
  int flags;            /* Flags associated with this word. */
} WORD_DESC;
```

Words are combined into, for example, argument lists, using simple linked lists:

```
typedef struct word_list {
  struct word_list *next;
  WORD_DESC *word;
} WORD_LIST;
```

`WORD_LIST`s are pervasive throughout the shell. A simple command is a word list, the result of expansion is a word list, and the built-in commands take word lists of arguments.

**In the final sentence above: can a command take multiple word lists as arguments? If not, please change to "take A word list"**

## 4.3 Input Processing

The first stage of the bash processing pipeline is input processing: taking characters from the terminal or a file, breaking them into lines, and passing the lines to the shell parser to transform into commands. As you would expect, the lines are sequences of characters terminated by newlines.

### 4.3.1 Readline and Command Line Editing

Bash reads input from the terminal when interactive, and from the script file specified as an argument otherwise. When interactive, bash allows the user to edit command lines as they are typed in, using familiar key sequences and editing commands similar to the Unix emacs and vi editors.

Bash uses the readline library to implement command line editing. This provides a set of functions allowing users to edit command lines, functions to save command lines as they are entered, to recall previous commands, and to perform csh-like history expansion. Bash is readline's primary client, and they are developed together, but there is no bash-specific code in readline. Many other projects have adopted readline to provide a terminal-based line editing interface.

Readline also allows users to bind key sequences of unlimited length to any of a large number of readline commands. Readline has commands to move the cursor around the line, insert and remove text, retrieve previous lines, and complete partially-typed words. On top of this, users may define macros, which are strings of characters that are inserted into the line in response to a key sequence, using the same syntax as key bindings. Macros afford readline users a simple string substitution and shorthand facility.

### Readline Structure

Readline is structured as a basic read/dispatch/execute/redisplay loop. It reads characters from the keyboard using `read` or equivalent, or obtains input from a macro. Each character is used as an index into a keymap, or dispatch table. Though indexed by a single eight-bit character, the contents of each element of the keymap can be several things. The characters can resolve to additional keymaps, which is how multiple-character key sequences are possible. Resolving to a readline command, such as `beginning-of-line`, causes that command to be executed. It's also possible to bind a key sequence to a command while simultaneously binding subsequences to different commands (a relatively recently-added feature); there is a special index into a keymap to indicate that this is done. Binding a key sequence to a macro provides a great deal of flexibility, from inserting arbitrary strings into a command line to creating keyboard shortcuts for complex editing sequences. Readline stores each character that is bound to the `self-insert` command in the editing buffer, which when displayed may occupy one or more lines on the screen.

**The self-insert command is dropped into the paragraph above without explanation; can be understood in context, but it it possible to reword?**

Readline manages only character buffers and strings using C `char`s, and builds multibyte characters out of them if necessary. It does not use `wchar_t` internally for both speed and storage reasons, and because the editing code existed before multibyte character support became widespread. When in a locale that supports multibyte characters, readline automatically reads an entire multibyte character and inserts it into the editing buffer. It's possible to bind multibyte characters to editing commands, but one has to bind such a character as a key sequence; this is possible, but difficult and usually not wanted. The existing emacs and vi command sets do not use multibyte characters, for instance.

Once a key sequence finally resolves to an editing command, readline updates the terminal display to reflect the results. This happens regardless of whether the command results in characters being inserted into the buffer, the editing position being moved, or the line being partially or completely replaced. Some bindable editing commands, such as those that modify the history file, do not cause any change to the contents of the editing buffer.

Updating the terminal display, while seemingly simple, is quite involved. Readline has to keep track of three things: the current contents of the buffer of characters displayed on the screen, the updated contents of that display buffer, and the actual characters displayed. In the presence of multibyte characters, the characters displayed do not exactly match the buffer, and the redisplay engine must take that into account. When redisplaying, readline must compare the current display buffer's contents with the updated buffer, figure out the differences, and decide how to most efficiently modify the display to reflect the updated buffer. This problem has been the subject of considerable research through the years (the string-to-string correction problem). Readline's approach is to identify the beginning and end of the portion of the buffer that differs, compute the cost of updating just that portion, including moving the cursor backward and forward (e.g., will it take more effort to issue terminal commands to delete characters and then insert new ones than to simply overwrite the current screen contents?), perform the lowest-cost update, then clean up by removing any characters remaining at the end of the line if necessary and position the cursor in the correct spot.

The redisplay engine is without question the one piece of readline that has been modified most

heavily. Most of the changes have been to add functionality—most significantly, the ability to have non-displaying characters in the prompt and to cope with characters that take up more than a single byte. **One reviewer asked why you'd have characters in your prompt that didn't display.**

Readline returns the contents of the editing buffer to the calling application, which is then responsible for saving the possibly-modified results in the history list.

#### Applications Extending Readline

Just as readline offers users a variety of ways to customize and extend readline's default behavior, it provides a number of mechanisms for applications to extend its default feature set.

First, bindable readline functions accept a standard set of arguments and return a specified set of results, making it easy for applications to extend readline with application-specific functions. Bash, for instance, adds more than thirty bindable commands, from bash-specific word completions to interfaces to shell builtin commands.

The second way readline allows applications to modify its behavior is through the pervasive use of pointers to hook functions with well-known names and calling interfaces. Applications can replace some portions of readline's internals, interpose functionality in front of readline, and perform application-specific transformations.

Much of readline's strength and consequent popularity stems from the ease with which it can be extended.

### 4.3.2 Non-interactive Input Processing

When the shell is not using readline, it uses either `stdio` or its own buffered input routines to obtain input. The bash buffered input package is preferable to `stdio` when the shell is not interactive because of the somewhat peculiar restrictions Posix imposes on input consumption: the shell must consume only the input necessary to parse a command and leave the rest for executed programs. This is particularly important when the shell is reading a script from the standard input. The shell is allowed to buffer input as much as it wants, as long as it is able to roll the file offset back to just after the last character the parser consumes. As a practical matter, this means that the shell must read scripts a character at a time when reading from non-seekable devices such as pipes, but may buffer as many characters as it likes when reading from files.

These idiosyncrasies aside, the output of the non-interactive input portion of shell processing is the same as readline: a buffer of characters terminated by a newline.

### 4.3.3 Multibyte Characters

Multibyte character processing was added to the shell a long time after its initial implementation, and it was done in a way designed to minimize its impact on the existing code. When in a locale that supports multibyte characters, the shell stores its input in a buffer of bytes (C `chars`), but treats these bytes as potentially multibyte characters. Readline understands how to display multibyte characters (the key is knowing how many screen positions a multibyte character occupies, and how many bytes to consume from a buffer when displaying a character on the screen), how to move forward and backward in the line a character at a time, as opposed to a byte at a time, and so on. Other than that, multibyte characters don't have much effect on shell input processing. Other parts of the shell, described later, need to be aware of multibyte characters and take them into account when processing their input.

## 4.4 Parsing

The initial job of the parsing engine is lexical analysis: to separate the stream of characters into words and apply meaning to the result. The word is the basic unit on which the parser operates. Words are sequences of characters separated by metacharacters, which include simple separators like spaces and tabs, or characters that are special to the shell language, like semicolons and ampersands.

One historical problem with the shell, as Tom Duff said in his paper about `rc`, the Plan 9 shell, is that nobody really knows what the Bourne shell grammar is. The Posix shell committee deserves significant credit for finally publishing a definitive grammar for a Unix shell, albeit one that has plenty of context dependencies. That grammar isn't without its problems—it disallows some constructs that historical Bourne shell parsers have accepted without error—but it's the best we have.

The Bash parser is derived from an early version of the Posix grammar, and is, as far as I know, the only Bourne-style shell parser implemented using Yacc or Bison. This has presented its own set of difficulties—the shell grammar isn't really well-suited to yacc-style parsing and requires some complicated lexical analysis and a lot of cooperation between the parser and the lexical analyzer.

In any event, the lexical analyzer takes lines of input from readline or another source, breaks them into tokens at metacharacters, identifies the tokens based on context, and passes them on to the parser to be assembled into statements and commands. There is a lot of context involved—for instance, the word `for` can be a reserved word, an identifier, part of an assignment statement, or other word, and the following is a perfectly valid command:

```
for for in for; do for=for; done; echo $for
```

that displays `for`.

At this point, a short digression about aliasing is in order. Bash allows the first word of a simple command to be replaced with arbitrary text using aliases. Since they're completely lexical, aliases can even be used (or abused) to change the shell grammar: it's possible to write an alias that implements a compound command that bash doesn't provide. The bash parser implements aliasing completely in the lexical phase, though the parser has to inform the analyzer when alias expansion is permitted.

Like many programming languages, the shell allows characters to be escaped to remove their special meaning, so that metacharacters such as & can appear in commands. There are three types of quoting, each of which is slightly different and permits slightly different interpretations of the quoted text: the backslash, which escapes the next character; single quotes, which prevent interpretation of all enclosed characters; and double quotes, which prevent some interpretation but allow certain word expansions (and treats backslashes differently). The lexical analyzer interprets quoted characters and strings and prevents them from being recognized by the parser as reserved words or metacharacters. There are also two special cases, `$'...'` and `$"..."`, that expand backslash-escaped characters in the same fashion as ANSI C strings and allow characters to be translated using standard internationalization functions, respectively. The former is widely used; the latter, perhaps because there are few good examples or use cases, less so.

The rest of the interface between the parser and lexical analyzer is straightforward. The parser encodes a certain amount of state and shares it with the analyzer to allow the sort of context-dependent analysis the grammar requires. For example, the lexical analyzer categorizes words according to the token type: reserved word (in the appropriate context), word, assignment statement, and so on. In order to do this, the parser has to tell it something about how far it has progressed parsing a command, whether it is processing a multiline string (sometimes called a "here document"), whether it's in a case statement or a conditional command, or whether it is processing an extended shell pattern or compound assignment statement.

Much of the work to recognize the end of the command substitution during the parsing stage is encapsulated into a single function (`parse_comsub`), which knows an uncomfortable amount of

shell syntax and duplicates rather more of the token-reading code than is optimal. This function has to know about here documents, shell comments, metacharacters and word boundaries, quoting, and when reserved words are acceptable (so it knows when it's in a `case` statement); it took a while to get that right.

When expanding a command substitution during word expansion, bash uses the parser to find the correct end of the construct. This is similar to turning a string into a command for `eval`, but in this case the command isn't terminated by the end of the string. In order to make this work, the parser must recognize a right parenthesis as a valid command terminator, which leads to special cases in a number of grammar productions and requires the lexical analyzer to flag a right parenthesis (in the appropriate context) as denoting EOF. The parser also has to save and restore parser state before recursively invoking `yyparse`, since a command substitution can be parsed and executed as part of expanding a prompt string in the middle of reading a command. Since the input functions implement read-ahead, this function must finally take care of rewinding the bash input pointer to the right spot, whether bash is reading input from a string, a file, or the terminal using readline. This is important not only so that input is not lost, but so the command substitution expansion functions construct the correct string for execution.

Similar problems are posed by programmable word completion, which allows arbitrary commands to be executed while parsing another command, and solved by saving and restoring parser state around invocations.

Quoting is also a source of incompatibility and debate. Twenty years after the publication of the first Posix shell standard, members of the standards working group are still debating the proper behavior of obscure quoting. As before, the Bourne shell is no help other than as a reference implementation to observe behavior.

The parser returns a single C structure representing a command (which, in the case of compound commands like loops, may include other commands in turn) and passes it to the next stage of the shell's operation: word expansion. The command structure is composed of command objects and lists of words. Most of the word lists are subject to various transformations, depending on their context, as explained in the following sections.

## 4.5 Word Expansions

After parsing, but before execution, many of the words produced by the parsing stage are subjected to one or more word expansions, so that (for example) `$OSTYPE` is replaced with the string `"linux-gnu"`.

### 4.5.1 Parameter and Variable Expansions

Variable expansions are the ones users find most familiar. Shell variables are barely typed, and, with few exceptions, are treated as strings. The expansions expand and transform these strings into new words and word lists.

There are expansions that act on the variable's value itself. Programmers can use these to produce substrings of a variable's value, the value's length, remove portions that match a specified pattern from the beginning or end, replace portions of the value matching a specified pattern with a new string, or modify the case of alphabetic characters in a variable's value.

In addition, there are expansions that depend on the state of a variable: different expansions or assignments happen based on whether or not the variable is set. For instance, `${parameter:-word}` will expand to `parameter` if it's set, and `word` if it's not set or set to the empty string.

### 4.5.2 And Many More

Bash does many other kinds of expansion, each of which has its own quirky rules. The first in processing order is brace expansion, which turns:

```
pre{one,two,three}post
```

into:

```
preonepost pretwopost prethreepost
```

There is also command substitution, which is a nice marriage of the shell's ability to run commands and manipulate variables. The shell runs a command, collects the output, and uses that output as the value of the expansion.

One of the problems with command substitution is that it runs the enclosed command immediately and waits for it to complete; and there's no easy way for the shell to send input to it. Bash uses a feature named process substitution, a sort of combination of command substitution and shell pipelines, to compensate for these shortcomings. Like command substitution, bash runs a command, but lets it run in the background and doesn't wait for it to complete. The key is that bash opens a pipe to the command for reading or writing and exposes it as a filename, which becomes the result of the expansion.

Next is tilde expansion. Originally intended to turn `{\textasciitilde}alan` into a reference to Alan's home directory, it has grown over the years into a way to refer to a large number of different directories.

Finally, there is arithmetic expansion. `$((expression))` causes `expression` to be evaluated according to the same rules as C language expressions. The result of the expression becomes the result of the expansion.

Variable expansion is where the difference between single and double quotes becomes most apparent. Single quotes inhibit all expansions—the characters enclosed by the quotes pass through the expansions unscathed—whereas double quotes permit some expansions and inhibit others. The word expansions and command, arithmetic, and process substitution take place—the double quotes only affect how the result is handled—but brace and tilde expansion do not.

### 4.5.3 Word Splitting

The results of the word expansions are split using the characters in the value of the shell variable `IFS` as delimiters. This is how the shell transforms a single word into more than one. Each time one of the characters in `$IFS`[1] appears in the result, bash splits the word into two. Single and double quotes both inhibit word splitting.

### 4.5.4 Globbing

After the results are split, the shell interprets each word resulting from the previous expansions as a potential pattern and tries to match it against an existing filename, including any leading directory path.

[1] Or in some cases a sequence of one of the characters.

### 4.5.5 Implementation

If the basic architecture of the shell parallels a pipeline, the word expansions are a small pipeline unto themselves. Each stage of word expansion takes a word and, after possibly transforming it, passes it to the next expansion stage. After all the word expansions have been performed, the command is executed.

The Bash implementation of word expansions builds on the basic data structures already described. The words output by the parser are expanded individually, resulting in one or more words for each input word. The `WORD_DESC` data structure has proved versatile enough to hold all the information required to encapsulate the expansion of a single word. The flags are used to encode information for use within the word expansion stage and to pass information from one stage to the next. For instance, the parser uses a flag to tell the expansion and command execution stages that a particular word is a shell assignment statement, and the word expansion code uses flags internally to inhibit word splitting or note the presence of a quoted null string (`"$x"`, where `$x` is unset or has a null value). Using a single character string for each word being expanded, with some kind of character encoding to represent additional information, would have proved much more difficult.

As with the parser, the word expansion code handles characters whose representation requires more than a single byte. For example, the variable length expansion (`$#variable`) counts the length in characters, rather than bytes, and the code can correctly identify the end of expansions or characters special to expansions in the presence of multibyte characters.

## 4.6 Command Execution

The command execution stage of the internal bash pipeline is where the real action happens. Most of the time, the set of expanded words is decomposed into a command name and set of arguments, and passed to the operating system as a file to be read and executed with the remaining words passed as the rest of the elements of `argv`.

The description thus far has deliberately concentrated on what Posix calls simple commands—those with a command name and a set of arguments. This is the most common type of command, but bash provides much more.

The input to the command execution stage is the command structure built by the parser and a set of possibly-expanded words. This is where the real bash programming language comes into play. The programming language uses the variables and expansions discussed previously, and implements the constructs one would expect in a high-level language: looping, conditionals, alternation, grouping, selection, conditional execution based on pattern matching, expression evaluation, and several higher-level constructs specific to the shell.

### 4.6.1 Redirection

One reflection of the shell's role as an interface to the operating system is the ability to redirect input and output to and from the commands it invokes. The redirection syntax is one of the things that reveals the sophistication of the shell's early users: until very recently, it required users to keep track of the file descriptors they were using, and explicitly specify by number any other than standard input, output, and error.

A recent addition to the redirection syntax allows users to direct the shell to choose a suitable file descriptor and assign it to a specified variable, instead of having the user choose one. This reduces the programmer's burden of keeping track of file descriptors, but adds extra processing: the shell has to duplicate file descriptors in the right place, and make sure they are assigned to the specified variable. This is another example of how information is passed from the lexical analyzer to the parser

through to command execution: the analyzer classifies the word as a redirection containing a variable assignment; the parser, in the appropriate grammar production, creates the redirection object with a flag indicating assignment is required; and the redirection code interprets the flag and ensures that the file descriptor number is assigned to the correct variable.

The hardest part of implementing redirection is remembering how to undo redirections. The shell deliberately blurs the distinction between commands executed from the file system that cause the creation of a new process and commands the shell executes itself (builtins), but, no matter how the command is implemented, the effects of redirections should not persist beyond the command's completion[2]. The shell therefore has to keep track of how to undo the effects of each redirection, otherwise redirecting the output of a shell builtin would change the shell's standard output. Bash knows how to undo each type of redirection, either by closing a file descriptor that it allocated, or by saving file descriptor being duplicated to and restoring it later using `dup2`. These use the same redirection objects as those created by the parser and are processed using the same function.

Since multiple redirections are implemented as simple lists of objects, the redirections used to undo are kept in a separate list. That list is processed when a command completes, but the shell has to take care when it does so, since redirections attached to a shell function or the "`.`" builtin must stay in effect until that function or builtin completes. When it doesn't invoke a command, the `exec` builtin causes the undo list to simply be discarded, because redirections associated with `exec` persist in the shell environment.

The other complication is one Bash brought on itself. Historical versions of the Bourne shell allowed the user to manipulate only file descriptors 0-9, reserving descriptors 10 and above for the shell's internal use. Bash relaxed this restriction, allowing a user to manipulate any descriptor up to the process's open file limit. This means that Bash has to keep track of its own internal file descriptors, including those opened by external libraries and not directly by the shell, and be prepared to move them around on demand. This requires a lot of bookkeeping, some heuristics involving the close-on-exec flag, and yet another list of redirections to be maintained for the duration of a command and then either processed or discarded.

### 4.6.2 Builtin Commands

Bash makes a number of commands part of the shell itself. These commands are executed by the shell, without creating a new process.

The most common reason to make a command a builtin is to maintain or modify the shell's internal state. `cd` is a good example; one of the classic exercises for introduction to Unix classes is to explain why `cd` can't be implemented as an external command.

Bash builtins use the same internal primitives as the rest of the shell. Each builtin is implemented using a C language function that takes a list of words as arguments. The words are those output by the word expansion stage; the builtins treat them as command names and arguments. For the most part, the builtins use the same standard expansion rules as any other command, with a couple of exceptions: the bash builtins that accept assignment statements as arguments (e.g., `declare` and `export`) use the same expansion rules for the assignment arguments as those the shell uses for variable assignments. This is one place where the `flags` member of the `WORD_DESC` structure is used to pass information between one stage of the shell's internal pipeline and another.

### 4.6.3 Simple Command Execution

Simple commands are the ones most commonly encountered. The search for and execution of commands read from the file system, and collection of their exit status, covers many of the shell's

[2]The `exec` builtin is an exception to this rule.

remaining features.

Shell variable assignments (i.e., words of the form `var=value`) are a kind of simple command themselves. Assignment statements can either precede a command name or stand alone on a command line. If they precede a command, the variables are passed to the executed command in its environment (if they precede a builtin command or shell function, they persist, with a few exceptions, only as long as the builtin or function executes). If they're not followed by a command name, the assignment statements modify the shell's state.

When presented a command name that is not the name of a shell function or builtin, bash searches the file system for an executable file with that name. The value of the `PATH` variable is used as a colon-separated list of directories in which to search. Command names containing slashes (or other directory separators) are not looked up, but are executed directly.

When a command is found using a `PATH` search, bash saves the command name and the corresponding full pathname in a hash table, which it consults before conducting subsequent `PATH` searches. If the command is not found, bash executes a specially-named function, if it's defined, with the command name and arguments as arguments to the function. Some Linux distributions use this facility to offer to install missing commands.

If bash finds a file to execute, it forks and creates a new execution environment, and executes the program in this new environment. The execution environment is an exact duplicate of the shell environment, with minor modifications to things like signal disposition and files opened and closed by redirections.

### 4.6.4 Job Control

The shell can execute commands in the foreground, in which it waits for the command to finish and collects its exit status, or the background, where the shell immediately reads the next command. Job control is the ability to move processes (commands being executed) between the foreground and background, and to suspend and resume their execution. To implement this, Bash introduces the concept of a job, which is essentially a command being executed by one or more processes. A pipeline, for instance, uses one process for each of its elements. The process group is a way to join separate processes together into a single job. The terminal has a process group ID associated with it, so the foreground process group is the one whose process group ID is the same as the terminal's.

The shell uses a few simple data structures in its job control implementation. There is a structure to represent a child process, including its process ID, its state, and the status it returned when it terminated. A pipeline is just a simple linked list of these process structures. A job is quite similar: there is a list of processes, some job state (running, suspended, exited, etc.), and the job's process group ID. The process list usually consists of a single process; only pipelines result in more than one process being associated with a job. Each job has a unique process group ID, and the process in the job whose process ID is the same as the job's process group ID is called the process group leader. The current set of jobs is kept in an array, conceptually very similar to how it's presented to the user. The job's state and exit status are assembled by aggregating the state and exit statuses of the constituent processes.

Like several other things in the shell, the complex part about implementing job control is bookkeeping. The shell must take care to assign processes to the correct process groups, make sure that child process creation and process group assignment are synchronized, and that the terminal's process group is set appropriately, since the terminal's process group determines the foreground job (and, if it's not set back to the shell's process group, the shell itself won't be able to read terminal input). Since it's so process-oriented, it's not straightforward to implement compound commands such as `while` and `for` loops so an entire loop can be stopped and started as a unit, and few shells have done so.

### 4.6.5 Compound Commands

Compound commands consist of lists of one or more simple commands and are introduced by a keyword such as `if` or `while`. This is where the programming power of the shell is most visible and effective.

The implementation is fairly unsurprising. The parser constructs objects corresponding to the various compound commands, and interprets them by traversing the object. Each compound command is implemented by a corresponding C function that is responsible for performing the appropriate expansions, executing commands as specified, and altering the execution flow based on the command's return status. The function that implements the `for` command is illustrative. It must first expand the list of words following the `in` reserved word. The function must then iterate through the expanded words, assigning each word to the appropriate variable, then executing the list of commands in the `for` command's body. The for command doesn't have to alter execution based on the return status of the command, but it does have to pay attention to the effects of the `break` and `continue` builtins. Once all the words in the list have been used, the `for` command returns. As this shows, for the most part, the implementation follows the description very closely.

## 4.7 Lessons Learned

### 4.7.1 What I Have Found is Important

I have spent over twenty years working on bash, and I'd like to think I have discovered a few things. The most important—one that I can't stress enough—is that it's vital to have detailed change logs. It's good when you can go back to your change logs and remind yourself about why a particular change was made. It's even better when you can tie that change to a particular bug report, complete with a reproducible test case, or a suggestion.

If it's appropriate, extensive regression testing is something I would recommend building into a project from the beginning. Bash has thousands of test cases covering virtually all of its non-interactive features. I have considered building tests for interactive features—Posix has them in its conformance test suite—but did not want to have to distribute the framework I judged it would need.

Standards are important. Bash has benefited from being an implementation of a standard. It's important to participate in the standardization of the software you're implementing. In addition to discussions about features and their behavior, having a standard to refer to as the arbiter can work well. Of course, it can also work poorly—it depends on the standard.

External standards are important, but it's good to have internal standards as well. I was lucky enough to fall into the GNU Project's set of standards, which provide plenty of good, practical advice about design and implementation.

Good documentation is another essential. If you expect a program to be used by others, it's worth having comprehensive, clear documentation. If software is successful, there will end up being lots of documentation for it, and it's important that the developer writes the authoritative version.

There's a lot of good software out there. Use what you can: for instance, gnulib has a lot of convenient library functions (once you can unravel them from the gnulib framework). So do the BSDs and Mac OS X. Picasso said "Great artists steal" for a reason.

Engage the user community, but be prepared for occasional criticism, some that will be head-scratching. An active user community can be a tremendous benefit, but one consequence is that people will become very passionate. Don't take it personally.

### 4.7.2 What I Would Have Done Differently

Bash has millions of users. I've been educated about the importance of backwards compatibility. In some sense, backwards compatibility means never having to say you're sorry. The world, however, isn't quite that simple. I've had to make incompatible changes from time to time, nearly all of which generated some number of user complaints, though I always had what I considered to be a valid reason, whether that was to correct a bad decision, to fix a design misfeature, or to correct incompatibilities between parts of the shell. I would have introduced something like formal bash compatibility levels earlier.

Bash's development has never been particularly open. I have become comfortable with the idea of milestone releases (e.g., bash-4.2) and individually-released patches. There are reasons for doing this: I accommodate vendors with longer release timelines than the free software and open source worlds, and I've had trouble in the past with beta software becoming more widespread than I'd like. If I had to start over again, though, I would have considered more frequent releases, using some kind of public repository.

No such list would be complete without an implementation consideration. One thing I've considered multiple times, but never done, is rewriting the Bash parser using straight recursive-descent rather than using `bison`. I once thought I'd have to do this in order to make command substitution conform to Posix, but I was able to resolve that issue without changes that extensive. Were I starting Bash from scratch, I probably would have written a parser by hand. It certainly would have made some things easier.

## 4.8 Conclusions

Bash is a good example of a large, complex piece of free software. It has had the benefit of more than twenty years of development, and is mature and powerful. It runs nearly everywhere, and is used by millions of people every day, many of whom don't realize it.

Bash has been influenced by many sources, dating back to the original 7th Edition Unix shell, written by Stephen Bourne. The most significant influence is the Posix standard, which dictates a significant portion of its behavior. This combination of backwards compatibility and standards compliance has brought its own challenges.

Bash has profited by being part of the GNU Project, which has provided a movement and a framework in which bash exists. Without GNU, there would be no bash.

Bash has also benefited from its active, vibrant user community. Their feedback has helped to make bash what it is today—a testament to the benefits of free software.