The Bourne-Again Shell

Chet Ramey
chet.ramey@gmail.com

1. Introduction

A Unix shell is a program that, at its base, provides an interface to the features the operating system provides for running commands. It allows a user to invoke commands with arguments that are usually treated as strings. In addition to simply executing commands, a shell is 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.

When the shell invokes a command, it can control (*redirect*) the input and output the command sees, and can connect commands by allowing the output of one command to become the input of another (a *pipeline*). The pipeline concept is central to this article. Each invoked command returns a status to the shell, which the shell uses for constructs like **if-then-else** and **while**.

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.

One of the most familiar shell constructs is the pipeline, where two or more commands are connected in a linear fashion so that the output of one command becomes the input of the next.

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.

[Figure 1 here]

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.

The name is an acronym for Bourne-Again SHell, a pun on 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, combined with the notion of rebirth through reimplementation.

Bash is an sh-compatible shell that incorporates useful features from the Korn shell (`ksh`) and other shells such as the C shell (`csh`). It is intended to be a conformant implementation of the IEEE POSIX Shell and Utilities specification (IEEE Working Group 1003). It offers functional improvements over sh for both interactive and programming use.

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 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 SFU.

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. One consequence of my never having been paid to work on Bash is that it has always been the equivalent of a hobby. While I have had to work around the rest of my life, Bash development has been relatively independent of external pressures. The lack of anything but self-imposed deadlines has had positive and negative benefits.

1.2. Posix

Posix is a name for a family of open system standards based on Unix. There are a number of aspects of Unix that have been standardized, from the basic system services at the system call and C library level to applications and tools to system administration and management. The Posix family of standards has been designated 1003 by the IEEE, and have been ratified as international standards by the ISO.

Before 1997, Posix consisted of a large number of separate standards, each considering a different aspect of the Unix interface. The Posix Shell and Utilities standard was originally developed by IEEE Working Group 1003.2 (POSIX.2), which published its final specification in 1992. This is the portion of the standard most relevant to Bash. Since 1997, the Austin Group has taken over development of the Posix standards. The current revision of the standard was published in 2008.

The portions of Posix of interest here concentrate on the command interpreter interface and on utility programs commonly executed from the command line or by other programs.

There are four primary areas of work in the Shell and Utilities standard:

- Aspects of the shell's syntax and command language. A number of special builtins such as **cd** and **exec** are specified as part of the shell, since their functionality usually cannot be implemented by a separate executable;
- A set of utilities to be called by shell scripts and applications. Examples are programs like `sed`, `tr`, and `awk`. Utilities commonly implemented as shell builtins are described in this section, such as **test** and **kill**. An expansion of this section's scope, originally termed the User Portability Extension, or UPE, has standardized interactive programs such as `vi`;
- A group of functional interfaces to services provided by the shell, such as the traditional `system()` C library function. There are functions to perform shell word expansions, perform filename expansion (globbing), obtain values of Posix system configuration variables, retrieve values of environment variables (`getenv()`), and other services;
- A suite of development utilities such as `c99` (the Posix command to invoke the C compiler), `yacc`, and `make`.

Bash is concerned with the aspects of the shell's behavior defined by Posix. The shell command language has of course been standardized, including the basic flow control and program execution constructs,

I/O redirection and pipelining, argument handling, variable expansion, and quoting. The special builtins, which must be implemented as part of the shell to provide the desired functionality, are specified as being part of the shell; examples of these are **eval** and **export**.

Other utilities appear in the sections of Posix not devoted to the shell which are commonly (and in some cases must be) implemented as builtin commands, such as **read** and **test**. Posix also specifies aspects of the shell's interactive behavior, including job control and command line editing. Interestingly, only `vi`-style line editing commands were standardized; `emacs` editing commands were left out due to objections.

There were certain areas in which Posix felt standardization was necessary, but no, or only one, existing implementation provided the proper behavior. The working group invented and standardized functionality in these areas, including reserved words (e.g., '!', which negates the exit status of a pipeline), builtin commands (**command**, which bypasses normal command lookup to skip shell functions), and word expansions (`$(( ... ))`, which treats the characters between the parentheses as an arithmetic expression and evaluates it). There existed multiple incompatible implementations of the **test** builtin, which tests files for type and other attributes and performs arithmetic and string comparisons. Posix considered none of these correct, so the standard behavior was specified in terms of the number of arguments to the command.

While Posix includes much of what the shell has traditionally provided, some important things have been omitted as being "beyond its scope." There is, for instance, no mention of a difference between a login shell and any other interactive shell (since Posix does not specify a login program). No fixed startup files are defined, either - the standard does not mention a file named by a specific shell variable (**ENV**).

2. Shell Syntactic Units and Primitives

2.1. Shell Primitives

To the shell, there are basically three kinds of *tokens*, or syntactic units: *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.

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 builtin variables referred to as *parameters*. Variable names are restricted to alphabetic characters, numbers, and the underscore ('_'), and may not begin with an underscore. Shell parameters do not follow that rule: parameter names consist of a special character, such as '@' or '!', or a number, and cannot be assigned directly. 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 substitution or removal of portions of a variable's value matching 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, where 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 implement 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.

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: **if-then-else**, **while**, a loop that iterates over a list of values, a C-like arithmetic for loop, constructs that allow the user to select from a set of alternative values, and conditional constructs. 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 a loop that is used to process it.

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.

The shell has no separate command language. All of the programming features are available when the shell is running commands interactively read from the user's terminal and when it is reading commands from a script.

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 builtin commands take word lists of arguments.

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. The lines are, as you would expect from experience, are sequences of characters terminated by newlines.

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.

Readline is the library bash uses to implement command line editing. The readline library provides a set of functions allowing users to edit command lines, functions to save command lines as they are entered and 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. Gdb and Python are two of the most well-known, but dozens of applications use the Readline interface to read input.

Readline is very extensible: applications may implement their own editing commands and either bind them to key sequences or make them available for users to do so. For instance, readline contains a set of commands that move backward and forward in the command line by words, using readline's idea of word boundaries. While these suffice for most applications, and most cases, Bash augments them with an additional set that uses shell metacharacters as word boundaries, as the bash parser treats them.

3.1.1. Editing Modes

Readline provides editing *modes*: sets of key bindings and variables that force readline to resemble the `emacs` or `vi` editors. By default, readline starts in emacs editing mode.

3.1.2. Prompting

Bash allows users great flexibility in customizing the readline prompt. It supports a number of backslash-escaped character sequences that expand to everything from the current username to an arbitrary date and time string. Bash, and readline, provide a way to mark a sequence of characters in the prompt as "invisible" — taking up no screen space — allowing users to insert terminal escape sequences into the prompt. Many multi-colored prompts and prompts that write to a window's title bar have resulted. This single feature proved the source of many redisplay bugs.

3.1.3. Key Bindings and Macros

Readline allows arbitrary key bindings. Users may 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, retrieving previous lines, and completing partially-typed words. 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.

3.1.4. Command History

Readline provides access to the command history, the set of previously-typed command lines. There are bindable readline commands to move back and forth through the history, search for words in the history list, and save and restore the history list to and from a file. There are bindable commands and options to perform csh-like history expansion ("bang history"). Bash augments the basic readline set with additional bindable commands that search the history, expand the command line in different ways, and expose the history to the word completion facilities. There are two builtin bash commands to search for and re-execute commands from the history and to manipulate the history file.

3.1.5. Word Completion

Readline provides a very general facility to complete partially-typed words. Most completion is application-specific: bash augments the base readline set with functions to complete command and variable names, hostnames, usernames, and even complete against words from the command history. Other applications using readline do the same thing: `gdb`, for instance, has functions to complete variable and function names from the symbol table of the program it's debugging.

Bash implements a per-command *programmable* word completion mechanism using the basic readline structure. In addition to a large number of built-in completions (command names, shell function names, variable names, builtin command names, etc.), programmable completion allows a user to write shell functions to generate the list of possible completions for a given word. This flexibility has resulted in the development of a large set of bash completions, a number of which are distributed as a separate free software project.

3.1.6. 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 the ability to insert 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.

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 -- 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, whether that results in characters being inserted into the buffer, the editing position being moved, or the line being partially or completely replaced, readline updates the terminal display to reflect the results. 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 seemingingly 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. Originally written by Brian Fox and Paul Placeway, it's safe to say that very little of the code has remained unexamined, if not unchanged. 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. There have also been significant efficiency improvements.

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.

3.1.7. 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. For an example of the first, applications are allowed to replace readline's default input, redisplay, and terminal initialization and restore functions. The most common example of function interposition is an application attempting word completion before readline's default. Most applications using readline implement application-specific word completion using this hook function. The final example is also most commonly used in completion: there is a set of functions that applications may use to transform words (e.g., removing quoting characters from filenames or changing character sets) when trying to match them against file system entries during word completion.

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

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 idiosyncracies 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.

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. Parsing

Parsing is the process of taking lines of input read from the terminal or obtained from readline and transforming them into commands that can be executed.

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

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 original grammar as published in Bourne's paper describing the Seventh Edition version of the shell does not even allow the command `who | wc`. Traditional shell parsers are built as a set of functions, each interpreting an individual construct in a recursive-descent fashion. However, the functions implementing the constructs in the Bourne shell each took a flag that modified their behavior in subtle context-dependent ways. 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 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. This facility is very versatile: one may use it to create mnemonics for command names, to expand a single word to a complete command name and set of arguments, and to ensure that a command is always invoked with a pre-defined set of options. Since it's completely lexical, it can even be used (or abused) to change the shell grammar: it's possible to write an alias **repeat** that implements a compound command (**repeat N** *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 *quoted* to remove their special meaning. Quoting is the only way to allow metacharacters such as '&' to appear in a command. 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 their being recognized by the parser as reserved words or metacharacters.

There are two interesting special cases of quoting: $'...' and $"...". The first is similar to single quotes, but expands backslash-escaped characters in the same fashion as ANSI C strings. The $"..."

construct allows the characters between the double quotes to be translated using standard internationalization functions like `gettext`. 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 here-document, whether it's in a case statement or a conditional command, or processing an extended shell pattern or compound assignment statement. A few examples will illustrate the various uses.

- The lexical analyzer flags assignment statements specially, since `foo=4 echo bar` is different from `echo bar foo=4`, and the parser can tell it when an assignment statement may be recognized.
- When parsing a conditional command, bash allows extended shell patterns and regular expressions, depending on the operator (`==` or `=~`). Bash doesn't require the metacharacters in the patterns be quoted, so the parser has to force the lexical analyzer to treat these metacharacters as part of the pattern rather than their normal interpretation as word separators.
- In most cases, a semicolon terminates a command or list, but while the parser is in the middle of the C-like arithmetic for command or a here-document, it has no special meaning.
- The parser knows when it is in a position to begin a command. It can tell the lexical analyzer this so the analyzer can perform alias expansion (and the analyzer can tell itself whether the next word is subject to alias expansion) or flag words that are well-formed shell assignment statements as such so they can undergo appropriate expansion later.

Almost all of the special cases are encapsulated into a single function: the aptly-named `special_case_tokens()`.

The parser's remaining work is relatively straightforward as well. Reserved words result in the creation of C objects representing the particular shell construct; the words returned by lexical analysis are converted to word lists, redirections (which are represented as simple lists of objects describing the required actions), and other elements. The objects are arranged in trees and lists to represent familiar idioms like `a; b; c` and

```
for i in 1 2 3 4 5 6 7 8 9 10; do
        j=10 s=
        while [ "$j" -ge "$i" ]; do
                s="$s $j"
                j=$(( j - 1 ))
        done
        printf "%s0 "$s"
done
```

The structure is versatile enough to represent constructs like shell functions and coprocesses that save a group of commands for future execution.

The parser implementation, even when using improved parser generators like bison, was the source of the longest-lived incompatibility with the Posix standard. One of the "new" features Posix standardized was the `$(...)` form of command substitution, an improvement over the original `` `...` `` form that permits easier nesting and more straightforward quoting. Posix requires that the command within the parentheses be parsed while it's being read, so that the command itself determines when the closing parenthesis is reached. It took a very long time and some interesting changes to the bison grammar before bash was able to implement this sort of on-the-fly parsing.

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** or `bash -c`, 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.

The problems posed by programmable word completion, which allow arbitrary commands to be executed while parsing another command, are similar, 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.

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. As noted above, shell quoting not only removes special meaning from characters, but inhibits some or all word expansion.

The word expansions operate on the words or word lists produced by the parser, and result in transformed word lists. In many cases, the expansions transform one word into several, as explained below. There are only a few cases in which a word is transformed into multiple words via the word expansions (e.g., the expansion of `"$@"` into a list of words constructed from the shell's positional parameters). Word splitting, described later in this section, is the mechanism by which bash creates multiple words from one.

The shell's traditional role is to perform what is essentially macro expansion and execute commands. There is a rich set of word expansions available, which allow the shell programmer a great deal of flexibility — sometimes arguably too much.

I won't spend a lot of time on the basic expansions, since they're so familiar. The '$' character introduces a word expansion. Most word expansions specify a shell variable name, which is expanded to its value, and that value is either returned or further transformed. That's the basic and most common expansion.

5.1. Brace Expansion

The first expansion, one that is handled completely separately from the others, is brace expansion. This is a feature that bash adopted from the C shell, and has subsequently been adopted by other Bourne-inspired shells. Brace expansion takes the form

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

and expands into separate words based on the comma-separated words between the braces. Unlike filename expansion, which is similar in appearance, the generated strings don't have to correspond to existing filenames. The real power of this construct is that the optional prefix `pre` and suffix `post` are "glued" to

the beginning and end of each string between the braces; the above expansion would result in

```
preonepost pretwopost prethreepost
```

Brace expansions precede the other word expansions, and brace expansion doesn't apply any semantic meaning to the text between the braces, leaving that for subsequent processing stages.

Since brace expansions can be nested, the generated filenames don't have to already exist, and brace expansions can contain characters understood by other expansions, it's possible to express complex sets of files in a very compact way. The classic example is

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

One of the most common uses of brace expansion is to produce sequences of words or numbers, and bash provides shorthand to effect that. There are ways to generate increasing and decreasing sequences of numbers or letters, with a user-specified increment. This is most often used to provide a list of words through which the **for**command will iterate.

5.2. Tilde Expansion

Tilde expansion is another feature that first appeared in the C shell. Like many other C shell features, it was originally intended as primarily an interactive feature — a shorthand way to refer to a user's home directory.

The Korn shell and bash both implemented tilde expansion, and the Posix committee considered it valuable enough to include in the standard. It's grown beyond a simple shorthand mechanism into a way to refer to a large number of different directories.

Bash implements a csh-like *directory stack*, which is a list of directories in which a user has indicated "interest", using the pushd and popd builtins to add and remove directories. Bash overloads tilde expansion so that it can refer to numbered elements in the directory stack. This novel extension is allowed, but not required, by Posix.

5.3. Parameter and Variable Expansions

The variable expansions are the ones users find most familiar. They provide the shell much of its expressive and programming power. Like most high-level programming languages, the shell provides variables and ways to manipulate them. 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. These constructs are most often used by shell programmers rather than casual users.

In addition to the basic expansion of a variable's name to its value, 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, the expansion `${`*parameter*`:-`*word*`}` will expand to *parameter* if it's set, and *word* if it's not set or set to the empty string. The `${`*parameter*`:+`*word*`}` construct does the opposite; it expands to nothing if *parameter* is not set or set to the empty string, and to *word* if it's set. Programmers can use different constructs to assign default values, or even to treat unset variables as errors, complete with unique error messages.

There are expansions that act on the variable's value itself. Programmers can use these to produce substrings of a variables's value, the value's length, remove portions that match a specified pattern from the beginning or end, or replace portions of the value matching a specified pattern with a new string. To this set, bash adds a set of expansions that modify the case of alphabetic characters in a variable's value, making it possible to directly express an idiom that historically required using `tr`.

One fairly powerful extension bash provides -- one that has created a fair amount of confusion -- is the notion of variable indirection. I picked the feature up from the Korn shell, but where ksh limits it to variables with a specific *nameref* attribute, bash allows it to be applied arbitrarily. This changes the expansion syntax to allow a leading exclamation point (`${!`*parameter*`}`), while leaving the rest of the expansion unchanged. The leading '!' causes the shell to use the value of *parameter* as the name of a second variable, which is expanded. It's the expanded value of the second variable that is used in the rest of the expansion, rather than the value of *parameter* itself. This has proven somewhat difficult to explain.

5.4. Command Substitution

Command substitution is a feature that the Bourne shell first provided, 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 questionable implementation decision was that the shell removes the trailing newlines in the command's output, instead of letting them be removed later by subsequent word splitting. The Bourne-style `*command*`, which bash and Posix still support, had some peculiar rules concerning the behavior of characters preceded by a backslash; the Korn shell introduced the more modern $(*command*) form, which is the Posix standard and preferred.

5.5. Process Substitution

One of the problems with command substitution is the nature of its definition: 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. The Korn shell originally implemented the feature; bash adopted and extended it. The syntax is very similar to the modern command substitution: `<(command)` or `>(command)`. Like command substitution, bash runs *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 (the first < form) or writing (the second) and exposes it as a filename. This filename, either a file in `/dev/fd` corresponding to a file descriptor, or a named pipe (FIFO), becomes the result of the expansion. Commands expecting filenames can read from or write to them as expected, and the input or output comes from or is sent to *command*. When combined with shell redirection, this is a powerful construct.

5.6. Arithmetic Expansion

Another new feature the Korn shell and bash implement, subsequently standardized by Posix, is arithmetic expansion. The Posix-invented `$((expression))` syntax 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. In one of the few cases where bash variables can have a type, variables can be declared as integers, which means this arithmetic evaluation will be performed each time the variable is assigned a value.

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.

5.7. 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**, or in some cases a sequence of one of the characters, appears in the result, bash splits the word into two. The default value of **IFS** is space, tab, and newline, which produces the familiar behavior of a variable value consisting of several space-separated words becoming separate arguments to a command. Single and double quotes both inhibit word splitting.

There are a number of rules that complicate the issue, having to do with sequences of white space characters also in the value of **IFS**. When the value of **IFS** contains whitespace characters, sequences of these characters serve to delimit fields. This means that the shell will only create null fields when performing word splitting if the value of **IFS** contains a non-whitespace character and that character is used as the field delimiter (whew!). This allows the shell to behave like `awk` when desired: a variable whose value is "nopass::1001:1001" will, when **IFS** contains a colon, result in four fields.

Word splitting, and the conditions under which it takes place (or doesn't) is another aspect of the shell that is hard to understand and explain. It's one of the things that most often trips up novices.

5.8. Filename Generation (globbing)

After the results are split, the shell interprets a special notation that generates filenames. 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. The syntax is again familiar: '*' matches any sequence of characters, '?' matches a single character, and matching '[' and ']' define a set of possible characters to match.

Bash's implementation adds a few needed enhancements to the basic Bourne shell and Posix specification. The Posix standard specifies that patterns that match nothing are left unchanged and passed to the command as written. Some users prefer the csh-like behavior of treating a pattern that matches no files as an error ("`echo: No match.`"), so bash provides an option to enable that behavior. Bash also allows the user to specify that patterns that match no files are removed entirely, to reduce the number of spurious error messages from invoked commands. There are additional options to specify that alphabetic case should be ignored when matching the pattern, and an interesting shell variable (**GLOBIGNORE**). **GLOBIGNORE** is set to a list of patterns. The results of filename generation are tested against each of the patterns in **GLOBIGNORE**, and, if a filename matches the **GLOBIGNOR**E pattern, it is removed from the filename generation results. In other words, if a pattern matches a filename, but that filename also matches one of the patterns in **$GLOBIGNORE**, the filename is treated as if it had not matched.

The basic set of pattern matching characters is similar to what Posix calls *basic* regular expressions. There are several additional pattern matching operators which bash adopted from the Korn shell that extend the filename generation facility to have capabilities similar to Posix *extended* regular expressions. These aren't enabled by default, since they're not specified by Posix, but are available via a settable option. This has caused its own set of problems, since to use these in a function, it's necessary to have the option enabled both when the function definition is read (the option changes the behavior of the shell parser) as well as when the function is executed.

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.

5.9. Implementation

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.

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 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.

6.1. Redirection

One thing that is somewhat unique to bash and other shells, a function of their role as an interface to the operating system, is the ability to redirect input and output to and from the commands it invokes. Commands normally read from their standard input and write to their standard output; in an interactive shell these are both usually connected to the user's terminal. Redirection allows users to specify different files that replace standard input and output for a single command, a group of commands, or for the shell itself. The redirection syntax is one of the things that reveals the sophistication of the shells'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 outside the 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.

Redirection is another conceptually simple but powerful mechanism. The only factor complicating its implementation is having to remember 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. (The **exec** builtin is an exception to this rule.) 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 closing a file descriptor that it allocated, or 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.

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. There are several reasons a command can be a builtin:

- it's essentially part of the shell language, like **break** or **continue**;
- it manipulates the shell's internal state, like **declare** or **set**;
- it cannot be implemented as an external command, like **cd** or **mapfile**;
- or for efficiency reasons, such as **test** and **printf**.

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. **exit** is a close second.

No matter how they are implemented, all commands should adhere to the standard Posix option and argument conventions; the Bash builtins do so.

In addition to the traditional set of builtin commands in historical versions of sh, Posix invented or standardized functionality in areas where no existing implementation provided the proper behavior. The **command** builtin was invented to allow shell functions to override builtin commands, while still permitting the function access to the builtin's capabilities. Posix standardized the behavior of the **test** command based on the number of arguments to the command.

Posix used the System V shell as the basis for the standard, but added a number of builtins and features from the ksh88 edition of the Korn shell. Some of these builtins are **alias**, **fc**, **getopts**, and the job control builtins **fg/bg/jobs**.

Bash provides several new and extended builtin commands. The **bind** builtin provides control over readline's key bindings and variables from the shell, so they can be set in bash's startup files. Readline also provides command history; the bash **history** builtin allows users to display and modify the history list and manipulate the history file. The **mapfile** builtin is very specialized: it efficiently reads lines from a file into an array variable. The programmable completion facilities are implemented using three builtins: **complete**, **compgen**, and **compopt**. The **pushd**, **popd**, and **dirs** builtins manipulate the directory stack.

With only a few exceptions, the new bash builtins expose portions of bash's internal state so it can be modified. In only a few cases is a command a builtin for purely efficiency reasons.

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 the assignment arguments as those the shell uses for variable assignments. This means that, assuming the variable X has the value "a b", the commands

```
Y=$X
export Y
```

and

```
export Y=$X
```

behave identically. Other shells assign Y the value "a" and export a variable named "b" with the null string as its value. 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.

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 remaining features.

Shell variable assignments, words of the form *var=value*, are a kind of simple command themseleves. 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 separator) 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.

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 effect 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 element of the pipeline. 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.

When a job is to be moved from the foreground to the background, the user types the *suspend* character (usually ^Z), which causes the process group to stop. Bash receives notification that the job has stopped, and, leaving the job suspended, returns to read a new command. The user can run the **bg** builtin to resume the job's execution in the background, as if it had been started that way originally. A job in the background can be moved to the foreground using the **fg** builtin, where the shell will wait for it and return its exit status as usual.

Though there are other uses for the ability to stop and restart jobs — to make sure there are enough resources for a long-running job to finish, for instance — job control is most commonly employed to multiplex the terminal between a number of different jobs. There may be any number of process groups in the background simultaneously, but only one process group may be in the foreground, since it has access to the terminal for input. (One of the more annoying aspects of job control is that by default all process groups that share the same terminal can all write to it simultaneously.) The shell provides various shorthand to refer to these jobs, including the notion of *current* and *previous* jobs. In practice, however, more than a few background jobs becomes unwieldy.

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.

Job control was first implemented in Berkeley Unix almost 30 years ago, and it is still controversial today. Its critics contend that it is messy, inelegant, and spoils the clean Unix process semantics.

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 lists of simple commands can be pipelines, or pipelines separated by **&&** or ||, which are shorthand for if-then-else-style conditional execution.

The basic compound commands should be familiar to anyone who's used or programmed a Bourne-derived shell.

Stephen Bourne was heavily influenced by his experience with Algol-68, on which he had worked immediately before joining Bell Labs and writing his shell. The shell's programming language syntax is reminiscent of "structured programming" languages, most notably Algol. One especially visible aspect of this heritage is the practice of terminating compound commands with a keyword that is the initial keyword reversed (**fi**, **esac**). We would be terminating while loops with **od** instead of **done** today if there had not already been a Unix program with that name.

Bash adds a few more compound commands to the basic Posix set. The **select** commnand allows users to pick from a list of alternatives displayed on the screen, and execute commands based on the selection. The **[[** command is a variant of the **test** command that's been incorporated into the shell language, which solves several problems, and extended with additional capabilities, such as pattern matching. There is a C-like arithmetic **for** loop, and the ((command to evaluate arithmetic expressions and return a status based on whether the expression evalutes to a non-zero value.

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. Once the list is expanded, the function must iterate through the words in the result, assigning the word to the appropriate variable, 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.

6.6. Coprocesses

Recent bash versions implement a construct named *coprocesses*, based on concepts in the Korn shell. A coprocess is a sort of cross between process substitution and a background command. Coprocesses are introduced by preceding a command with the **coproc** reserved word. The shell runs the command in the background, and establishes a two-way pipe between the shell and the command. Note the similarity to process substitution, but coprocesses are not tied to a particular command and the shell takes care of a lot of the details.

Coprocesses may be named; the default name is **COPROC**. This means that, in theory at least, there may be more than one, but the ability to control more than one coprocess hasn't yet been implemented. The shell creates an array variable with the same name as the coprocess, and assigns the file descriptors used to communicate with that coprocess to indices of the array (note the similarity to the `pipe` system call). These can be used in redirections or passed as arguments to other commands. The shell exposes the process ID of the coprocess, so the user can kill it, or wait for it to terminate.

6.7. Signals and Traps

The shell allows the user and programmer to write equivalents of the signal handlers one would write in C, providing shell programs the ability to respond to and handle exceptional conditions. The **trap** builtin is the interface to the signal handling ability; the user specifies a command or set of commands that should be executed when the shell receives a particular signal, and the shell arranges for these command to be run sometime after that signal arrives, when it is "safe" to do so.

The trap handling implementation is similar to what the operating system kernel has to do when executing a signal handler: the shell saves state, transfers control to the command associated with the signal, and restores state when that command has terminated. The shell's trap commands, unlike C signal handlers, are allowed to run arbitrary programs and shell commands.

The trap mechanism is familiar to anyone who's programmed in C, but it's fairly primitive. That hasn't stopped shells from overloading it with a number of extra conditions that don't correspond to signals sent by the kernel. The first was the **EXIT** trap, which allows the user to specify commands that are run when the shell exits. This is a useful place to put cleanup code, since the **EXIT** trap is run whenever the shell exits, whether intended or not. Bash adds the **DEBUG** trap, which is run just before each simple command is executed, and the **RETURN** trap, which is run before execution resumes after a shell function or script executed with the '.' builtin completes.

Bash provides an option that forces the shell to exit whenever a command fails. This is often used by shell scripts that do not want to check the success or failure of each command, but want to ensure that every command they execute completes successfully. The **ERR** trap is executed whenever the shell would exit due to a failed command when this option is enabled. It's another useful place to put cleanup and error handling code.

6.8. Complex Examples

When programmers combine the shell's programming language with its builtin commands and the ability to execute commands from the file system, it becomes possible to write very complex applications using the shell.

There are several good examples of large, complex shell programs. The largest, most complex ones are probably those produced by `autoconf` — the autoconf-produced configuration script for bash-4.2 runs about 33,000 lines, for instance — but one worth particular mention is the bash debugger, `bashdb`, written by Rocky Bernstein.

7. Lessons Learned

7.1. What I've Found is Important

I've spent over twenty years working on bash, and I'd like to think I've discovered a few things that are important. The lessons are more pragmatic than anything, and not Bash- or shell-specific.

The most important thing, 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 reproducible test case, or 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.) Bash has constraints placed on it by history, backwards compatibility, and standards; in my case, regression testing is essential.

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

Not only are external standards 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 write 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.

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 correct a bad decision, to fix a design misfeature, or to correct incompatibilities between parts of the shell. I would have introduced something like the formal bash "compatibility level" notion 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.

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.