Principia Softwarica: The Plan 9 Shell `rc`

version 1.0

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Tom Duff

June 17, 2026

Contents

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a shell.

1.1 Motivations

Why a shell? Because I think you are a better programmer if you fully understand how things work under the hood, and the shell is the central piece of the *command-line user interface* (CLI), a place where programmers spend lots of time. The shell is a thin layer around the kernel (hence its name) allowing you to run commands in a terminal.

Most users now use a *graphical user interface* (GUI) to execute programs (e.g., macOS, Microsoft Windows, X Window), but most programmers still spend a significant portion of their time in a shell to run compilation commands, editors, debuggers, or *scripts* to automate repetitive tasks. Integrated Development Environments (IDEs) can handle some of those use cases, but the shell still reigns when a programmer needs more flexibility. The power of pipes, redirections, variables, and basic control flow constructs allows sometimes in one command-line to perform tasks that would require hundreds of lines in a regular programming language[1].

In fact, the rise of AI coding assistants like Claude Code has given the command-line a second wind. These tools operate as shell programs, composing existing commands through pipes and scripts rather than through graphical menus. The shell's text-based, composable nature turns out to be a perfect interface for AI agents that can read, write, and chain commands together—something much harder to automate inside a GUI-based IDE.

There are unfortunately very few books explaining how a shell works. I can cite *Advanced UNIX Programming* [Roc04], but it explains only the code of a mini-shell. This is a pity because the implementation of a real shell covers many interesting topics: programming language design, compilation, interpretation, system programming, and more as you will see soon in this book.

Here are a few questions I hope this book will answer:

- What is the difference between a shell and a terminal? What is the difference between a shell and a login program?
- What happens when the user types `ls` in a terminal? What is the trace of such a command through the different layers of the software stack, and especially through the shell?
- What are the main features of a shell?
- How are redirections and pipes implemented? What are the system calls involved?
- Why `ls` and `rm` are regular programs but not `cd`? Why `cd` has to be a shell builtin? What is a shell builtin?

[1] For example, [BKM86] contrasts writing a program to count words in Pascal and in a shell.

- How does `^C`, which interrupts a process, work? Which process receives the signal after an interruption? The shell or the command run from the shell?

1.2 The Plan 9 shell: `rc`

I will explain in this book the code of the Plan 9 shell `rc` [Duf90][2] (for `Run Commands`), which contains about 5700 lines of code (LOC). `rc` is written mostly in C, with its parser using also Yacc [Joh79].

Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. Moreover, `rc` is arguably simpler to use and to understand than `sh` [Bou79], the UNIX shell, or any of its derivatives (e.g., `bash` [Ram94], the most popular shell under Linux). For example, by treating the content of any variables uniformly as a list of strings, `rc` does not need the extra operator `$@` used in `sh` (`$*` is enough). Moreover, the syntax of `rc`, specified formally and succintly by a small grammar, is also easier to learn than the (unspecified) syntax of `sh`.

`rc` itself lacks many of the interactive features found in other shells (e.g., `bash`) such as filename completion, command-line editing, job control, etc. This is partly because under Plan 9, the terminal of the windowing system `rio` is providing instead those features (see the WINDOWS book [Pad16e]).

1.3 Other shells

Here are a few shells that I considered for this book, but which I ultimately discarded:

- The first UNIX shell, originally called `sh`, was written by Ken Thompson, the original author of the UNIX kernel. `sh` started as an assembly program in UNIX V1[3] and finished as a C program in UNIX V6[4]. Ken Thompson's shell introduced the pipe (`'|'`), redirections (`'>'` and `'<'`), as well as wildcard matching[5] (`'*'`). The syntax of those features remained the same in all subsequent shells. Ken Thompson's shell contains only 899 LOC, but it is just a basic command interpreter, not a full scripting language like `rc`.

- The Bourne shell [Bou79], also called `sh`, superseded Ken Thompson's shell (which was renamed `osh`) in UNIX V7[6]. It was written by Stephen Bourne and contains 4145 LOC of C. It is arguably the most famous shell because it defined the main features that we expect now from a shell: the ability to run commands with pipes and redirections, but also the use of variables and control flow constructs to write scripts. The Bourne shell is both an interactive command interpreter and a full programming language. Most subsequent shells tried to remain backward compatible with the Bourne shell.

  The syntax of the Bourne shell was inspired by Algol with pair of keywords (e.g., `if/fi`, `do/done`, `case/esac`) instead of the curly braces of C[7]. The code of `sh` is smaller than the code of `rc`, but it is also harder to understand. For example, there is no clear grammar as in `rc`, but instead a complex recursive descent parser.

- Bash [Ram94, New95][8] (for Bourne-Again Shell) is an open-source shell similar to the Bourne shell (hence its name) from the GNU project[9]. It is the default shell in most Linux distributions. It provides many interactive features not found in the Bourne shell such as filename completion, command-line editing,

[2] See http://plan9.bell-labs.com/magic/man2html/1/rc for its manual page.
[3] http://tuhs.org/cgi-bin/utree.pl?file=V1/sh.s
[4] http://tuhs.org/cgi-bin/utree.pl?file=V6/usr/source/s2/sh.c
[5] Also known as globbing.
[6] http://tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/sh/
[7] Stephen Bourne was such as fan of Algol that the C source code of `sh` itself looks like Algol, thanks to macros such as `THEN`, `BEGIN`, `END`, etc.
[8] https://www.gnu.org/software/bash/
[9] http://www.gnu.org

interactive history, job control, etc. Many of those features were partly inspired by the C shell [Joy86], an older shell originating in BSD UNIX using a syntax more similar to C.

Bash relies on the GNU Readline library[10] for many of those interactive features (`rc` relies instead on `rio`'s terminal to provide similar features). However, the codebase of Bash is very large with more than 100 000 LOC (not including the code of the Readline library, which would add another 33 000 LOC), which is more than an order of magnitude more code than in `rc`. Bash's grammar file `parse.y` contains alone 6268 LOC.

The 15× size difference between Bash and `rc` is not just about interactive features. Much of the complexity comes from backward compatibility with the Bourne shell (and POSIX), which forces Bash to support multiple quoting styles, arithmetic expansion, brace expansion, parameter expansion with numerous operators (`$x:-default`, `$x##pattern`, etc.), and the context-sensitive grammar inherited from `sh`.

- The Z shell [Kid05][11] is another open-source shell popular among advanced Linux users. It is extensible through plugins and themes[12]. It contains most of the features found in other shells (e.g., Bash, the C Shell, the Korn Shell [BK89, Kor94]) and introduced many new interactive features such as programmable completion, recursive wildcarding with `**/*` (eliminating the need for the program `find`), and more. However, all those features come at a price: the code of the Z shell contains more than 145 000 LOC (not including the tests).

- Nushell[13] is a modern shell, written in Rust, that questions one of the most fundamental UNIX design decisions: that commands communicate through unstructured streams of text. Instead, Nushell passes *structured* data (tables and records) between the commands of a pipeline, so that filtering and formatting can be expressed with typed operations rather than the usual text munging with `awk`, `sed`, or `cut`. It is part of a broader revival of shell design that also includes Fish[14] (focused on interactive friendliness) and Oils[15] (a principled reimplementation of Bash with a real programming language underneath). Here too the price is size: Nushell weighs around 250 000 LOC, making `rc` look minuscule in comparison.

Figure 1.1 presents a timeline of major UNIX shells (and a few non-UNIX shells). I think `rc` represents the best compromise for this book: it implements the essential features of a shell while still having a small and understandable codebase (5700 LOC).

1.4 Getting started

To play with `rc`, you will first need to install the Plan 9 fork used in Principia Softwarica (see `https://www.principia-softwarica.org/getting-started.html`). Once installed, you do not need to do anything to run `rc` because it is the program started by default by the kernel (see the KERNEL book [Pad14]). Once the kernel finished to boot, you should see a percent sign, called the *prompt*, after which you can type any commands as in the following:

```
1   % ls /
2   bin
3   boot
4   ...
5   srv
```

[10] https://tiswww.case.edu/php/chet/readline/
[11] http://www.zsh.org/
[12] See https://github.com/robbyrussell/oh-my-zsh for a large repository of such contributions.
[13] https://www.nushell.sh/
[14] https://fishshell.com/
[15] https://www.oilshell.org/

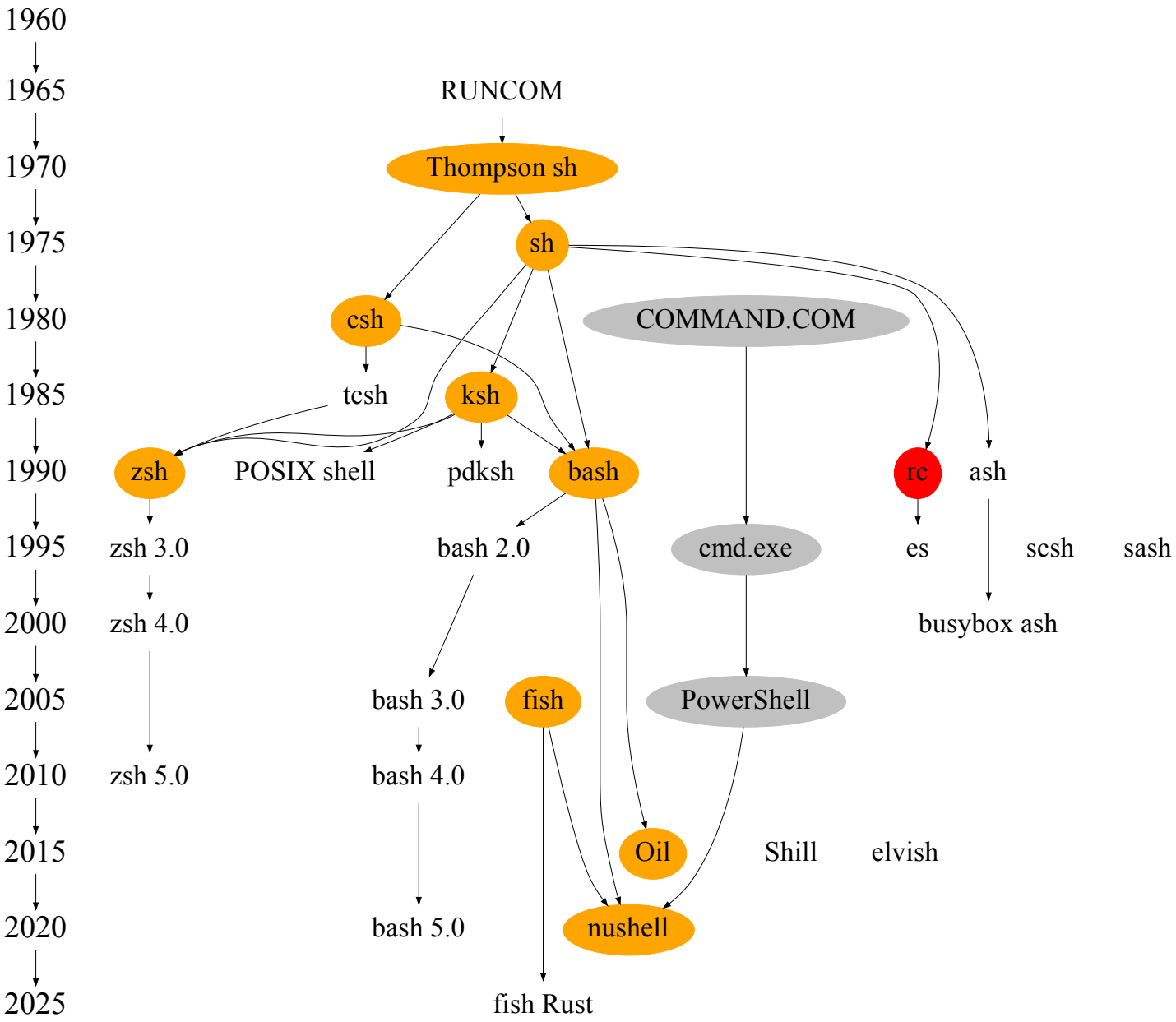


Figure 1.1: Shells timeline

```
6   % rc -help
7   Usage: rc [-SsrdiIlxepvV] [-c arg] [-m command] [file [arg...]]
8   % rc
9   % prompt='$ '
10  $ ls /
11  bin
12  boot
13  ...
14  srv
15  $ exit
16  %
```

Line 1 runs the program `ls` to list the content of the root directory. Line 6 and Line 8 show that `rc` is a regular program (just like `ls` at Line 1): you can run a shell program under a shell. Line 8 and Line 9 seem to indicate that typing `rc` has no effect, but the percent sign shown at the beginning of Line 9 is in fact displayed by the `rc` process started by Line 8, not the original `rc` process started by the kernel. Line 9 modifies the *special variable* `prompt` (see Section 10.3 for a list of those special variables) to better differentiate the two shell processes. Indeed, Line 10 shows that `'$ '` is the new prompt replacing the percent sign. Finally, Line 15 shows the use of the *builtin* `exit` (see Chapter 9 for more information on builtins) to exit from the shell. Doing so goes back to the preceding shell process (the one launched by the kernel) with the original percent prompt at Line 16.

You can also run `rc` on Linux, macOS, or Windows through plan9port[16] or Goken9cc[17], where it is compiled natively using gcc or clang.

1.5 Requirements

Because most of this book is made of C source code, you will need a good knowledge of the C programming language [KR88] to understand it. To understand Chapter 7, you will also need to know Yacc [BLM92].

Note that this book is not an introduction to the shell or to shell scripting. I assume you are already familiar with at least one shell, for instance a derivative of `sh` such as `bash`, and so are familiar with concepts such as pipes, redirections, what `#!` means, or what a script is. If not, I suggest you to read either the book introducing the UNIX programming environment [KP84], or the original `sh` tutorial [Bou79], or any more recent books on shell scripting [Mic08, New95, Kid05].

A shell is a thin layer on top of a kernel, and so a shell relies heavily on the services offered by the kernel: system calls (e.g., `rfork()` `exec()`, `wait()`, `chdir()`), but also device files (e.g., `/dev/cons` to read and write characters on the terminal). Thus, it can be useful to know how the Plan 9 kernel works (see the KERNEL book [Pad14]), or at least be familiar with system programming under UNIX [Roc04, Ste94], to fully understand some of the code in this book.

A shell is also a full programming language, and as you will see soon, `rc` is internally both a compiler and a bytecode interpreter. I assume you also have a basic understanding of how a compiler works, and for example that you know what a lexer or parser is (see the COMPILER book [Pad16c]).

If, while reading this book, you have specific questions on the syntax and interface of `rc`, I suggest you to consult the man page of `rc` at `docs/man/1/rc` in my Plan 9 repository.

Note that the `shells/docs/` directory in my Plan 9 repository contains documents describing either `rc` [Duf00] or `sh` [Bou15]. Those documents are useful to understand some of the design decisions presented in this book, especially how and why `rc` differs from `sh`.

[16] https://9fans.github.io/plan9port/
[17] https://github.com/aryx/goken9cc

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

1.8 Acknowledgments

I would like to acknowledge of course the author of `rc`, Tom Duff, who wrote in some sense most of this book.

Chapter 2

Overview

Before showing the source code of `rc` in the following chapters, I first give an overview in this chapter of the general principles and features of a shell. I also quickly describe the command-line interface of `rc` and the specific language supported by `rc`. I also define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Shell principles and features

A shell is a strange beast: it must be both an *interactive command interpreter*, allowing to run commands easily on one line in a terminal, and a proper *programming language*, allowing to write complex *scripts* in a file. The following subsections unpack this dual nature and place it in the broader context of terminals, scripting languages, and the modern revival of the command line.

2.1.1 Shell versus terminal

A source of persistent confusion—even among experienced developers—is the difference between a shell and a terminal. A *terminal* handles *character* I/O: it displays characters on a screen, reads keystrokes, and sends them to whatever program is attached. Historically a terminal was a physical device like a DEC VT100; today we usually use *terminal emulators* like xterm, iTerm2, or Windows Terminal (on respectively Linux, macOS, and Windows). A *shell* (`rc`, `bash`, `zsh`) is the program that reads *commands* and executes them. It does not know about screens, cursors, or fonts—it just reads from file descriptor 0 (standard input) and writes to file descriptor 1 (standard output).

The full chain from power-on to prompt makes the roles clear. At boot, `init` (the first process) starts the system. On a traditional UNIX, `getty` opens a terminal device (a serial port or virtual console), `login` authenticates the user and starts the user's shell. On Plan 9, `init` starts `rc` directly, and `rio` provides virtual terminals (each window is a terminal that gets its own `rc`, see the WINDOWS book [Pad16e]). On a modern machine, the terminal emulator plays the role of `getty`: it creates a pseudo-terminal device, forks a shell, and connects the two. The shell never knows whether it is talking to a physical VT100, an `xterm`, or a `rio` window—all it sees is a file descriptor.

2.1.2 Running commands

The first job of a shell is to allow the user to run commands[1]. Most shells offer a minimalist syntax for running commands, so we can type those commands quickly. Indeed, contrast the shell command `ls /` with the

[1] The very first shell, which was written for the CTSS operating system in 1964, was called `RUNCOM`[Pou00].

equivalent C program:

⟨lsroot.c 15a⟩≡
```
#include <u.h>
#include <libc.h>

void main(){
  char* args[] = {"/", nil};
  exec("/bin/ls", args);
}
```

Here are a few notes on the minimalist syntax used by shells:

- *No Parenthesis:* to call a program, you do not need any parenthesis; just type the program name next to its arguments.
- *No quotes:* to pass arguments to the program, you usually do not need any quotes or commas; just type the arguments separated by space. For example, `foo`, `--help`, `42`, `/a/b/c` are all valid arguments[2]. The only time you need to enclose an argument in a quote is when you want to use one of the special character used by the shell (e.g., `'#'`, `'$'`, `'&'`, etc. with `rc`), also known as *meta-characters*.
- *No full path:* to call a program, you do not need to specify its path in the filesystem; just type the name of the program and the shell will automatically find its location. Under UNIX, the `PATH` environment variable stores the candidate locations to find programs.
- *No special ending character:* to finish your command, you do not need any special character like a semicolon; just type a newline and the shell will start interpreting your command[3].
- *No types:* to enter a command, you do not need to specify any types such as `char*` as in C. The arguments are always strings.

Once the command you ran finished, the shell gets back in control and displays another *prompt* to indicate that you can type another command.

To make this concrete, here is a minimal shell in about 20 lines of C. It does almost nothing—just reads a command name, forks a process, and `exec`s the program—but it already captures the essence of what a shell is: a loop that reads, forks, execs, and waits.

⟨minishell1.c 15b⟩≡
```
#include <u.h>
#include <libc.h>

void main() {
  char buf[256];
  char *args[2];
  int n;

  for(;;) {
    write(STDOUT, "$ ", 2);
    n = read(STDIN, buf, sizeof(buf)-1);
    if(n <= 0)
      break;
    buf[n-1] = '\0';  /* strip newline */
    args[0] = buf;
    args[1] = nil;
```

[2]You can even sometimes avoid to specify fully all the arguments by using shortcuts, for example, wildcards as explained in Chapter 12.

[3]You can also enter commands on multiple lines, which requires to *escape* the newline as explained in Section 5.2.2.

```
    if(rfork(RFPROC|RFFDG) == 0) {
      exec(buf, args);
      exits("exec failed");
    }
    await(buf, sizeof(buf));
  }
  exits(nil);
}
```

The code above relies on system calls covered in the KERNEL book [Pad14]: `rfork()` creates the child process (its flags say what to copy or share—`RFPROC` for the process itself, `RFFDG` for a private *file-descriptor group*), `exec()` replaces it with the named program, and `await()` makes the parent wait for the child to finish.

2.1.3 Redirections

The second important feature of a shell is to allow to *redirect* the output and input of a program, as in `ls / > listing.txt`.

Many programs (e.g., `ls`, `find`, `rm`, `grep`) live in the command-line world and simply use text for their input and output. By using redirections, you can easily save the output of those programs in a file, or use a previously saved file as input for those programs[4], without even changing the code of those programs. In fact, because under UNIX and Plan 9 "everything is a file", including devices, you can use the same program in many different ways. For example, you can print the listing of the root directory by simply typing `ls / > /dev/printer`.

2.1.4 Combining commands: the pipe

Because of the *universality* of *plain text*, you can easily combine many command-line programs. For example, you can list all the C files in your home directory by combining `find` and `grep` with the two commands `find /home/pad/ > /tmp/list.txt` and `grep '\.c$' < /tmp/list.txt`. In fact, thanks to another great feature of the shell, the *pipe*, you can just use one command: `find /home/pad | grep '\.c$'`. This is not only shorter, but also more efficient, and it gives results more quickly. You can even use multiple pipes in the same command as in `find | grep '\.c$' | xargs cat | grep foo`.

Pipes allow to combine easily full programs, just like you can combine functions in a functional language. Pipes are one of the greatest innovations introduced by UNIX, and one of the first programming construct allowing a form of *component-oriented programming*. Each program can be treated as a component, which can be combined with other components. You do not program at the granularity of functions but at the higher granularity of programs.

To support redirections and pipes, one only needs to manipulate file descriptors in the forked child before calling `exec()`. Here is a second mini-shell that handles `>` for output redirection and `|` for a single pipe:

⟨minishell2.c 16⟩≡

```
#include <u.h>
#include <libc.h>

void runcmd(char *cmd) {
  char *args[2];
  args[0] = cmd;
  args[1] = nil;
  exec(cmd, args);
  exits("exec failed");
}

void main() {
  char buf[256];
```

[4] `rc` allows to redirect not just the standard input and output, as explained in Section 14.4.7.

```
  char *p;
  int n;
  fdt fd, pfd[2];

  for(;;) {
    write(STDOUT, "$ ", 2);
    n = read(STDIN, buf, sizeof(buf)-1);
    if(n <= 0)
      break;
    buf[n-1] = '\0';

    if((p = strchr(buf, '>')) != nil) {
      *p++ = '\0';
      while(*p == ' ') p++;
      if(rfork(RFPROC|RFFDG) == 0) {
        fd = create(p, OWRITE, 0666);
        dup(fd, STDOUT);
        close(fd);
        runcmd(buf);
      }
      await(buf, sizeof(buf));
    } else if((p = strchr(buf, '|')) != nil) {
      *p++ = '\0';
      while(*p == ' ') p++;
      pipe(pfd);
      if(rfork(RFPROC|RFFDG) == 0) {
        close(pfd[0]);
        dup(pfd[1], STDOUT);
        close(pfd[1]);
        runcmd(buf);
      }
      if(rfork(RFPROC|RFFDG) == 0) {
        close(pfd[1]);
        dup(pfd[0], STDIN);
        close(pfd[0]);
        runcmd(p);
      }
      close(pfd[0]);
      close(pfd[1]);
      await(buf, sizeof(buf));
      await(buf, sizeof(buf));
    } else {
      if(rfork(RFPROC|RFFDG) == 0)
        runcmd(buf);
      await(buf, sizeof(buf));
    }
  }
  exits(nil);
}
```

Notice how little code it takes: the key insight is that `rfork()` gives the child its own copy of the file descriptor table, so we can `dup()` a file or pipe end onto stdin or stdout before `exec()`—and the executed program never knows the difference. This is precisely the beauty of the `fork()`/`exec()` split that the rest of this book will explore in detail.

2.1.5 Storing commands in a script

Just like you can type a series of commands separated by newlines in a terminal, you can save those same commands in a file, a *script*, and execute this script with the shell. As the author of the first shell said [Pou00]:

"commands should be usable as building blocks for writing more commands, just like subroutine libraries".

Once a script is saved in a file, you can make it executable and run it directly, just like a compiled binary, by typing `% myscript` instead of `% rc myscript`. The trick is the `#!` line: when the first two bytes of an executable file are `#!`, the rest of that first line names the *interpreter* to use, for example `#!/bin/rc`. When asked to `exec()` such a file, the kernel notices this magic sequence and instead runs the named interpreter, passing the script as its argument (see the KERNEL book [Pad14]). The `#!` sequence is colloquially called the *shebang*.

This mechanism, introduced in UNIX V8, is what makes scripts indistinguishable from compiled programs: they can be used as filters in pipes, as components in other scripts, or as standalone commands. Before the kernel handled shebangs, only the shell could run scripts—you had to type `rc foo.rc` explicitly, and a C program could not `exec()` a script.

Why call it a *script* rather than a *program*? The word *program* had come to mean a self-contained computation compiled into a standalone executable (think of a C program). A shell script is a different kind of artifact: it is not compiled, and it rarely computes anything itself—it *directs* other programs (`ls`, `grep`, `sed`, ...), specifying which to run and in what order. The theatrical sense of the word captures this exactly: a screenplay does not act, it tells the actors what to do and when. In the same way a shell script orchestrates existing commands rather than doing the work itself, and is *interpreted* line by line rather than compiled.

2.1.6 A shell as a scripting language

Once you can write a sequence of commands in a script, you quickly want more, such as the ability to use conditionals or loops. Thus, most shells are also full programming languages, often referred to as *scripting languages*, with variables and many control flow constructs, and even function definitions. A scripting language acts like a glue, allowing to combine many programs together.

Because a scripting language must also support the basic commands you type on the command-line, it shares many characteristics with those basic commands: a scripting language does not use types and is interpreted. Moreover, a scripting language operates mainly on strings. Because the string arguments you type on the command-line do not usually require any quotes, the use of variables requires then a special operator. Most shells use `'$'` as a prefix to differentiate variables from regular string arguments, for example, `find $home`.

In `rc`, *all values are lists of strings*. This eliminates the `sh` distinction between `$*` (all arguments as one string) and `$@` (all arguments as separate words), and means that `$path` does not need the colon-separated encoding that `$PATH` uses in `sh`.

Most shells do not use booleans: control flow is based instead on the *exit status* of the command you run. Perhaps confusingly, the exit code 0 in UNIX actually means success (in Plan 9 the empty string means success, which can also be confusing). In `rc`, the only comparison operator is `~` (pattern matching on strings), and for numerical tests, the external program `test` is used (see Appendix E.3 for its code), similar to `sh`.

The "scripting language" label that shells pioneered has since outgrown its origins. Perl (Larry Wall, 1987) started as a better `awk`/`sed`/shell combination and grew into a language people wrote web servers in. Python (Guido van Rossum, 1991) started as a scripting language and now drives machine learning. JavaScript (Brendan Eich, 1995) was designed for 10-line browser scripts and now runs servers, databases, and desktop applications. The line between "scripting language" and "real programming language" disappeared sometime in the 2000s, and the distinction today says more about the speaker's prejudices than about any technical property of the language. `rc` though sits at the minimalist end of this spectrum: it is a shell and nothing more, deliberately leaving the "real programming" to C.

Meanwhile, shell scripting itself never went away. Most build systems, deployment pipelines, and system-administration workflows still use shell scripts, and AI coding agents like Claude Code have given the tiny-tools style a new audience by generating `grep`/`sed`/`awk` pipelines for developers who might never have learned those tools otherwise.

2.1.7 Globbing

A core shell convenience is letting the user write a pattern that stands for many filenames. Typing `ls *.c` should list every C file in the current directory without forcing the user to enumerate them by hand. Globbing (a.k.a. filename expansion, or wildcard matching) is the mechanism that turns `*.c` into the actual list `a.c b.c c.c` before `ls` even runs.

The fundamental design choice is who *expands* the pattern. UNIX shells and `rc` take it on themselves: when the shell parses `ls *.c`, it walks the current directory, matches each entry against the pattern, and rewrites the command into `ls a.c b.c c.c` before calling `exec`. The `ls` program never sees the `*`—it just receives a normal `argv` of literal filenames and has nothing special to do. On Windows the convention is the opposite: `cmd.exe` passes `*.c` through verbatim and each program is responsible for expanding its own arguments.

The shell-side approach has two big consequences. First, every program automatically gets globbing for free, with identical semantics, just by being launched from the shell. Second, since expansion happens before `exec`, the shell needs a way to suppress it—hence quoting: `ls '*.c'` tells the shell "don't expand this, pass it through literally". Forgetting the quotes around a pattern that was meant as a literal is one of the most common shell bugs.

Globbing patterns are usually a tiny grammar: `*` for any sequence of characters, `?` for any single character, and `[a-z]` for a character class. `rc` supports exactly these three. `bash` and `zsh` extend the grammar with `**` for recursive directory walks, `{a,b,c}` for brace expansion, and various extended-glob predicates; `zsh`'s globbing is famously powerful enough to replace `find` for many tasks. Chapter 12 shows how `rc` implements its three-operator subset.

2.1.8 Quoting and escaping

The previous subsection showed that the shell expands `*` before `exec`, which is convenient until the user actually wants a literal star. The same problem applies to every metacharacter the shell recognizes—spaces, `$`, `&`, `|`, `<`, `>`, parentheses, semicolons, newline. The shell needs a way to mark a stretch of text as "please pass through verbatim". That mechanism is *quoting*.

Most UNIX shells offer several quoting modes with overlapping but not identical rules. `sh` and `bash` have single quotes (`'...'`), double quotes (`"..."`), and backslash escapes (`\`). Single quotes suppress everything; double quotes still expand `$x` and command substitutions but suppress globbing and *word splitting*; backslash escapes only the next character. The interactions are subtle: `\$` inside double quotes is a literal dollar, but `\$` inside single quotes is the two literal characters `\` and `$`. The same character means different things depending on which kind of quote it sits in.

The deeper trap is word splitting on *unquoted* variable expansion. In `sh` and `bash`, if `x` holds `hello world`, then `ls $x` runs `ls hello world` (two arguments), while `ls "$x"` runs `ls 'hello world'` (one argument). The word splitting depends on a global variable `IFS` that controls which characters split words (usually spaces, tabs, newlines). Forgetting to quote a variable that might contain spaces is the source of an enormous class of UNIX shell bugs, and explains the defensive (ugly) idiom of writing `"$@"` instead of `$@` in scripts.

`rc` eliminates almost all of this complexity. There is exactly one quoting form: single quotes, with the convention that an embedded single quote is written by doubling it (`'it''s'`). There are no double quotes, no backslash escapes, no `IFS`, and no word splitting at all—because in `rc` every variable is already a list of strings, not a single string. If `x` holds two elements `hello` and `world`, then `ls $x` always passes two arguments (no need for a variable word splitting extra mechanism). If you wanted one argument `hello world` you would have stored it that way to begin with. The whole class of "forgot to quote a variable" bugs simply does not exist, which is why `rc`'s designers consider it one of the best things the shell does differently from `sh` [Duf90].

This `sh` (and `bash`) fragility also explains a defensive idiom common in UNIX scripts. Where one would ideally just write `test $v = yes`, careful authors instead write `test "x$v" = "xyes"`. Two separate problems force the change: an unquoted `$v` is word-split (so an empty or multi-word value breaks the command), and even quoted, the old `test` inspects the *values* of its arguments, so if `$v` is empty or starts with a `-` (or is itself `=`

or `!`), `test` mistakes the operand for an option. The quotes fix the first problem and the literal `x` the second.

In `rc` the clean form survives: you write `~ $v yes` using the built-in `~` (match) operator. There is no word splitting (every variable is already a list), and `~`'s subject and patterns are separate grammatical operands, so neither the quotes nor the `x` padding has any reason to exist.

2.1.9 An interactive interpreter

The previous subsections covered mostly the *scripting* half of the shell. The other half is the *interactive* side: most of the time a shell is used live at a terminal, and the interactive mode adds a cluster of features—line editing, history recall, tab completion, —that have nothing to do with running commands and everything to do with making the typing experience pleasant.

The fundamental design choice is where those features live. `bash`, `zsh`, and `fish` bake them all into the shell itself: `bash` uses the GNU readline library for line editing and history, `zsh` has its own `zle` line editor with hundreds of widgets, and `fish` was created largely to have first-class syntax highlighting and autosuggestions out of the box. This makes for a great interactive experience but inflates the shell—`bash` is around 100 000 LOC, `zsh` over 145 000 LOC, mostly because of the interactive layer.

`rc` takes the opposite position: the shell does as little interactive work as possible. There is no built-in line editor, no history recall, no tab completion. Reading a command is just a `read()` from file descriptor 0 (Section 4.8 shows the exact code). If you want line editing, you run `rc` inside a `rio` or `acme` window, where the terminal or editor itself provides editing, search, and history through its normal commands. The shell stays around 5700 LOC; the interactive features stay in the tools that already do them well.

2.1.10 Asynchronous execution

A long-running command should not block the shell. If you type `compile-everything &`, you want the prompt back immediately and the command to keep running in the background. This is asynchronous execution, and every shell has some form of it—but how much machinery sits behind the `&` varies enormously.

At one end of the spectrum is the minimal approach: `&` forks a child, the shell does not `wait` for it, the child is isolated just enough that the shell's terminal and signals do not interfere with it, and that is the entire feature. The user can later type a `wait` *builtin* to reap any finished children, and that is the only way the shell tracks them. `rc` takes this minimal path: no job table, no `%1`/`%2` job specs, no `fg`/`bg`, no notification when a background job finishes. The shell starts the process, isolates it, and forgets about it.

At the other end is full *job control*, introduced by `csh` in 1980 and inherited by `bash` and `zsh`. The shell maintains a job table indexed by job number; each job is its own *process group*; `Ctrl-Z` sends `SIGTSTP` to the foreground group to suspend it; `fg`/`bg` resumes it in the foreground or background; a `SIGCHLD` handler updates the table when children change state; the prompt notifies the user about completed background jobs. This is a substantial amount of code, much of it concerned with terminal-foreground-group management (`tcsetpgrp`) so that the shell can hand the controlling terminal back and forth between itself and its children.

Why does `rc` get away with the minimal approach? Largely because Plan 9 replaces a lot of what job control gives you with cleaner OS primitives. A backgrounded process can still be addressed by its `pid` and controlled through `/proc/<pid>/` (read state, send notes, debug with `acid`, see the DEBUGGER book [Pad16d]). Moreover, `rio` makes it natural to start each long command in its own window instead of multiplexing multiple jobs through one terminal. The shell-side feature exists only where the OS does not already cover it. Section 8.4.6 will show that the `&` case reduces to a fork-without-wait plus a few lines to put the child in its own note group and redirect its stdin to `/dev/null`.

2.1.11 The CLI renaissance

Note that the command line is experiencing its biggest growth period in decades. Cloud infrastructure is entirely CLI-driven (`docker`, `kubectl`, `terraform`, `aws`). Modern terminal emulators are now GPU-accelerated

with ligatures and inline images (Kitty, Alacritty, Ghostty, Windows Terminal). Terminal User Interface (TUI) frameworks (charm.sh's Bubbletea, Python's Textual and Rich) are creating a new generation of rich terminal applications[5], AI coding agents like Claude Code (itself a TUI program) live in the terminal and generate shell pipelines as their primary mode of operation, and finally VS Code's integrated terminal has made the command line the default companion to the graphical editor, even for developers who never used a standalone terminal before.

The shell this book studies is not historical nostalgia—it is the program at the center of this revival. Every `docker build`, every `kubectl apply`, every AI-generated `grep`/`sed`/`awk` pipeline runs inside a shell. Understanding how `rc` parses a command, forks a process, wires up a pipe, and waits for the result is understanding the mechanism that makes all of this work.

2.2 `rc` command-line interface

I just described the main features of a shell and illustrated a few of those features with `bash`, `rc`, or `zsh`. I will now focus exclusively on `rc` and give more details about its command-line interface.

It is rare to run `rc` itself on the command-line, like you run `ls`, `cp`, `grep`, etc. After all, if you have a command-line, you already are in a shell. However, as I said in Section 1.4, the shell under UNIX (and Plan 9) is a regular program. You can run a shell under a shell, which can be useful for example if you do not like the default shell when you log-in[6]. To run `rc`, simply type `rc` on the command-line without any arguments, as I did in Section 1.4 Line 8.

You can also pass to `rc` the name of a script as an argument, as well as the arguments of this script, for example, `rc foo.rc arg1 arg2`. However, this is usually not necessary because most scripts use `#!/bin/rc` at the beginning, which is recognized by the kernel (see the KERNEL book [Pad14]).

Here is the full command-line interface of `rc`:

```
% rc -help
Usage: rc [-SsrdiIlxepvV] [-c arg] [-m command] [file [arg...]]
```

The `-c` flag allows to execute commands from a string passed as an argument instead of from the content of a script (see Section 14.1 for the code handling `rc -c`).

The `-m` flag allows to specify the initialization script of `rc`. I will explain fully the complex initialization process of `rc` and the code of `rc -m` in Chapter 11.

Finally, `rc` supports a few options to provide advanced features or to help debug `rc` itself. I will present gradually those options in this book.

2.3 `boot.rc`

In this section, I will present an example of an `rc` script showing a few features of `rc`'s scripting language. The goal here is to illustrate the general features of a shell you have seen in Section 2.1 with the concrete syntax of `rc`'s scripting language. This will also help you understand the grammar of `rc` I will present in Chapter 7. Finally, this example will introduce a few concepts specifics to `rc` that are useful to have in mind when I explain the code of `rc` in the rest of the document.

`boot.rc`, below, is the first program executed by the kernel when running on an ARM machine (see the KERNEL book [Pad14]). The following sections will explain the main features of `rc` used in this script.

⟨*kernel/init/user/boot/arm/boot.rc* 21⟩≡
```
#!/boot/rc -m /boot/rcmain
```

[5] Where a CLI reads a command and prints back a stream of text, a TUI takes over the whole terminal screen and reacts to keystrokes in real time, like `vi` or `acme`.

[6] Under UNIX, you can also use the special program `chsh` to change your login shell.

```
/boot/echo booooooooting...

path=(/bin /boot)

# basic devices
bind -c '#e' /env
...

# storage
bind -a '#S' /dev
fdisk -p /dev/sdM0/data >/dev/sdM0/ctl
dossrv
mount -c /srv/dos /root /dev/sdM0/dos
bind -a -c /root /

bind -a /arm/bin /bin
bind -a /rc/bin /bin
...

# to use 5c, 5a, 5l by default in mk
objtype=arm
...

. -i '#d/0'
```

2.3.1 Comments

Comments in `rc` start with `#` and extend to the end of the line.

The first line of `boot.rc` is `#!/boot/rc -m /boot/rcmain`. This is the *shebang* line described earlier in Section 2.1.5.

Note that `#!` also works as a comment, since `#` starts a comment in `rc`, so the shebang line is simply ignored when the file is read by the shell itself (as in `rc boot.rc`).

2.3.2 Initialization script: `rcmain`

The `-m /boot/rcmain` flag specifies the initialization script. By default `rc` sources `/rc/lib/rcmain` at startup, but during boot the root filesystem is not yet mounted, so the boot script must use `-m` to point to a copy stored in the boot partition at `/boot/rcmain`.

Under UNIX, the equivalent mechanism is the cascade of initialization files: `.profile` for the Bourne shell, `.login` for the C shell, `.bashrc` for Bash, etc. In Plan 9, there is a single initialization script `rcmain` shared by all users (see Section 11.5 for details).

2.3.3 Variables and list of strings

The `boot.rc` script above shows a variable assignment: `path=(/bin /boot)`. In `rc`, a variable holds a *list of strings*, and parentheses are used to create lists: `(a b c)` is a three-element list. This is a key design difference from `sh`, where variables hold a single string and lists are simulated by splitting on `IFS` characters. By treating all values as lists, `rc` avoids many of the quoting pitfalls that plague `sh` scripts.

2.3.4 `$path`

The variable `$path` is a *special variable*: it tells `rc` where to look for programs. When you type `ls`, the shell searches each directory in `$path` until it finds an executable named `ls`.

In sh and bash, the equivalent variable is $PATH, which encodes the list as a colon-separated string (e.g., /bin:/usr/bin). In rc, because variables are already lists, $path is simply (/ /bin)—no special separator needed.

Under Plan 9, the path list is typically short because the system uses *union directories*: multiple directories are bound together into a single mount point like /bin, so there is no need for a long search path. This also means you never need the /usr/bin/env trick commonly used in UNIX shebang lines. Under UNIX, different systems install programs in different locations (e.g., /usr/bin/python vs. /usr/local/bin/python), so script authors write #!/usr/bin/env python instead of hardcoding the path: env searches $PATH and runs the first match. Under Plan 9, with union directories, /bin/rc is always the right path regardless of the architecture. See Section 8.3.5 for the implementation.

2.3.5 Quoting

The boot.rc script uses single quotes in bind -c '#e' /env: the # must be quoted, otherwise the shell would treat it as a comment.

In rc, single quotes are the only quoting mechanism. To include a literal single quote inside a quoted string, you double it: 'it''s'. There are no backslash escapes and no double quotes (the " character has a different meaning in rc: variable stringification, see Section 14.4.4).

2.3.6 The environment

The boot.rc script also assigns objtype=arm. This is not just a local shell variable: in Plan 9, all shell variables are automatically exported to the environment—there is no need for an export command like in bash[7]. When mk (the build tool) is later run from this shell, it inherits $objtype and uses it to select the correct compiler and assembler (e.g., 5c, 5a, 5l for ARM).

Under Plan 9, the environment is stored as files in the /env filesystem: each variable becomes a file /env/varname whose content is the variable's value. See Chapter 10 for the implementation.

2.3.7 Builtins

The last line of boot.rc is . -i '#d/0'. The . (dot) is a shell *builtin*: a command that is executed inside the shell process rather than in a child. Builtins exist because some operations must modify the shell's own state. For example, cd changes the shell's current directory—if cd were a regular program, it would run in a forked child, the child would change *its* directory and exit, and the parent shell would be unaffected.

The . builtin sources commands from a file in the current shell. Here that "file" is '#d/0', the console on file descriptor 0, and -i makes the sourced loop interactive (it prints a prompt). So this single line turns the boot shell *itself* into the interactive shell. The commented-out alternative, exec /boot/rc -m /boot/rcmain -i, would instead replace the boot shell with a brand-new rc—one that re-sources rcmain and redoes all the initialization this shell has already done. Sourcing the console directly reuses the running shell and skips that duplicated startup work. See Chapter 9 for the full list of builtins.

2.3.8 Other features

The boot.rc script is simple enough that it does not use many of rc's advanced features: no control flow, no functions. no pipes. Those features—pipes, redirections, conditionals, loops, pattern matching, functions, command substitution, and glob patterns—will be presented gradually in the following chapters.

[7] If you need environment isolation, rfork e creates a private copy of /env/; see Section 14.5.3.

2.4 Code organization

Table 2.1 presents short descriptions of the source files of `rc`, together with the main entities (e.g., structures, functions, globals) the file defines, and the corresponding chapter in this document in which the code contained in the file is primarily discussed.

Function	Ch.	File	Entities	LOC
main data structures	3	`rc.h`	`Tree`[31a] `Code`[33c] `Word`[36b] `Var`[38c]	169
execution data structures	3	`exec.h`	`Thread`[34d] `List`[38a]	78
AST helpers	3	`tree.c`	`newtree()`[32a] `tree1()`[32c] `tree2()`[32d]	117
globals	3	`globals.c`	`codebuf`[33b] `runq`[35a]	19
list of strings (words)	3	`words.c`	`newword()`[36c] `count()`[37a] `freewords()`[37b]	69
shell variables	3	`var.c`	`setvar()`[39b] `vlook()`[39c] `gvar`[39e] `gvlook()`[39f]	60
function prototypes	3	`fns.h`		70
entry point	4	`main.c`	`main()`[42b]	81
execution utilities	4	`executils.c`	`start()`[45b] `pushlist()`[46a] `pushword()`[46b]	134
character input and prompt	5	`input.c`	`getnext()`[49] `pprompt()`[52a] `nextc()`[53a]	155
lexer	6	`lex.c`	`yylex()`[55a]	324
parser	7	`syn.y`	`yyparse()`[65b]	116
bytecode generation	8	`code.c`	`emitf()`[75d] `compile()`[75g] `outcode()`[76b]	483
process status	8	`status.c`	`setstatus()`[78a] `getstatus()`[78b]	38
simple command bytecode	8	`simple.c`	`Xsimple()`[81] `execexec()`[83b] `doredir()`[91a]	138
`$path` management	8	`path.c`	`searchpath()`[84a]	18
process-related bytecodes	8	`processes.c`	`execforkexec()`[83a] `Xpipe()`[97c] `Xasync()`[99b]	304
other bytecodes	8	`exec.c`	`Xtrue()`[88a] `Xjump()`[101g] `Xfn()`[107a]	663
shell builtins	9	`builtins.c`	`execcd()`[116c] `execexit()`[118c] `execdot()`[119]	463
shell environment	10	`env.c`	`addenv()`[124c] `Vinit()`[124d]	6
initialization (bootstrap)	11	`main.c`	`Rcmain`[129b] `dotcmds`[120b]	
wildcard matching	12	`glob.c`	`glob()`[135a] `match()`[138b]	245
signal management	13	`trap.c`	`dotrap()`[145b] `Trapinit()`[144e]	49
here documents	14	`here.c`	`readhere()`[158c] `heredoc()`[159h]	145
AST dumper	A	`pcmd.c`	`pcmd()`[173b]	130
bytecode dumper	A	`pfnc.c`	`pfnc()`[177a]	75
error management	B	`error.c`	`panic()`[180c]	27
memory management	C	`utils.c`	`emalloc()`[181a] `efree()`[181c]	52
command-line flags	C	`getflags.[ch]`	`getflags()`[182i] `usage()`[184]	228
buffered IO	C	`io.[ch]`	`openfd()`[186h] `rchr()`[188c]	125
pretty printer	C	`fmt.c`	`pfmt()`[190b]	154
Plan 9 host-specific code	G	`plan9.c`	`Waitfor()`[85b] `notifyf()`[144f] `Globsize()`[139b]	408
Unix host-specific code	G	`unix.c`	`Waitfor()` `Trapinit()` `gettrap()`[241]	506
Total				5678

Table 2.1: Chapters and associated `rc` source files.

Every source file in `rc` includes the same set of headers. `rc.h` defines the main data structures, `exec.h` defines

the execution data structures, `fns.h` contains all function prototypes, `getflags.h` provides the command-line flag parsing API, and `io.h` defines the buffered IO layer.

⟨*includes* 25⟩≡ (240 239 238 237 236 235 234 233 230 229 228 227b 226)

```
#include "rc.h"
#include "exec.h"
#include "fns.h"
#include "getflags.h"
#include "io.h"
```

2.5 Software architecture

Before describing the internal architecture of `rc` itself, I will first present the broader context in which `rc` operates: the command-line user interface and the other system components (kernel, device drivers, terminal) that `rc` interacts with. Then I will describe `rc`'s own components and trace the execution of a simple command through them.

2.5.1 The command-line user interface

The shell is only one component of the command-line user interface (CLI). Figure 2.1 shows the components supporting the CLI under Plan 9 (the components for the CLI under UNIX are very similar).

To run commands, the shell needs first a kernel to create processes. The kernel provides services to applications (including the shell) through its *system call interface* (also known as *syscalls*). For example, under Plan 9, `rfork()` creates a new process in which the shell can then `exec()` a program (see KERNEL book [Pad14]).

A shell needs also device drivers in the kernel handling the terminal with its keyboard and monitor. To access those devices, the kernel provides a *filesystem interface* (called a *namespace* under Plan 9). An application can `open()`, `read()`, `write()`, or `close()` files in this filesystem. Under UNIX and Plan 9, *devices are represented as files*. For example, `/dev/cons` (for console) is a file representing the terminal. To read characters from the keyboard, an application can simply read characters from `/dev/cons`. To write characters on the monitor, an application can simply write characters in `/dev/cons`.

A key UNIX innovation is that the shell is a regular user program, not part of the kernel. When the kernel boots, it starts an initial process whose stdin and stdout are connected to `/dev/cons`. Every process forked from it inherits this connection, including `rc`. Because the shell is just a program, you can replace it—or run a different shell inside the current one—without changing the kernel.

2.5.2 `rc`'s components

Figure 2.2 describes the main data flow of `rc`, whereas Figure 2.3 describes the main control flow of `rc`. At its core, `rc` is both a compiler and a bytecode interpreter. As shown by Figure 2.2, given a series of characters (coming either from what you typed in the terminal or from the content of a script), `rc` first groups those characters in tokens (with `yylex()`[55a]). Then, `rc` parses those tokens (with `yyparse()`[65b]) and builds an abstract syntax tree (AST) of the program (see `Tree`[31a]). Finally, `rc` transforms this tree in a series of bytecodes (see `Code`[33c]). This is similar to what a compiler such as `5c` does (see COMPILER book [Pad16c]), except a bytecode here is not an instruction from a concrete machine but from a *virtual machine*.

After this compilation, `rc` goes through the series of bytecodes and interprets them. `rc` keeps track of what is currently executing in a `Thread`[34d] data structure and in a queue `runq`[35a]. This data structure is called a "thread" because `rc` must sometimes manage multiple threads of execution. Indeed, some of the bytecodes can create new processes running concurrently (e.g., `Xpipe()`[97c], the bytecode handling pipes).

In fact, `rc` does not start by compiling, but by interpreting. Indeed, `rc` starts first by interpreting some special bytecodes, called the *bootstrap*, that contains a bytecode (`Xrdcmds()`[47a]) that then triggers the compiler. I will now explain briefly the control flow of `rc`, starting from the top of Figure 2.3.

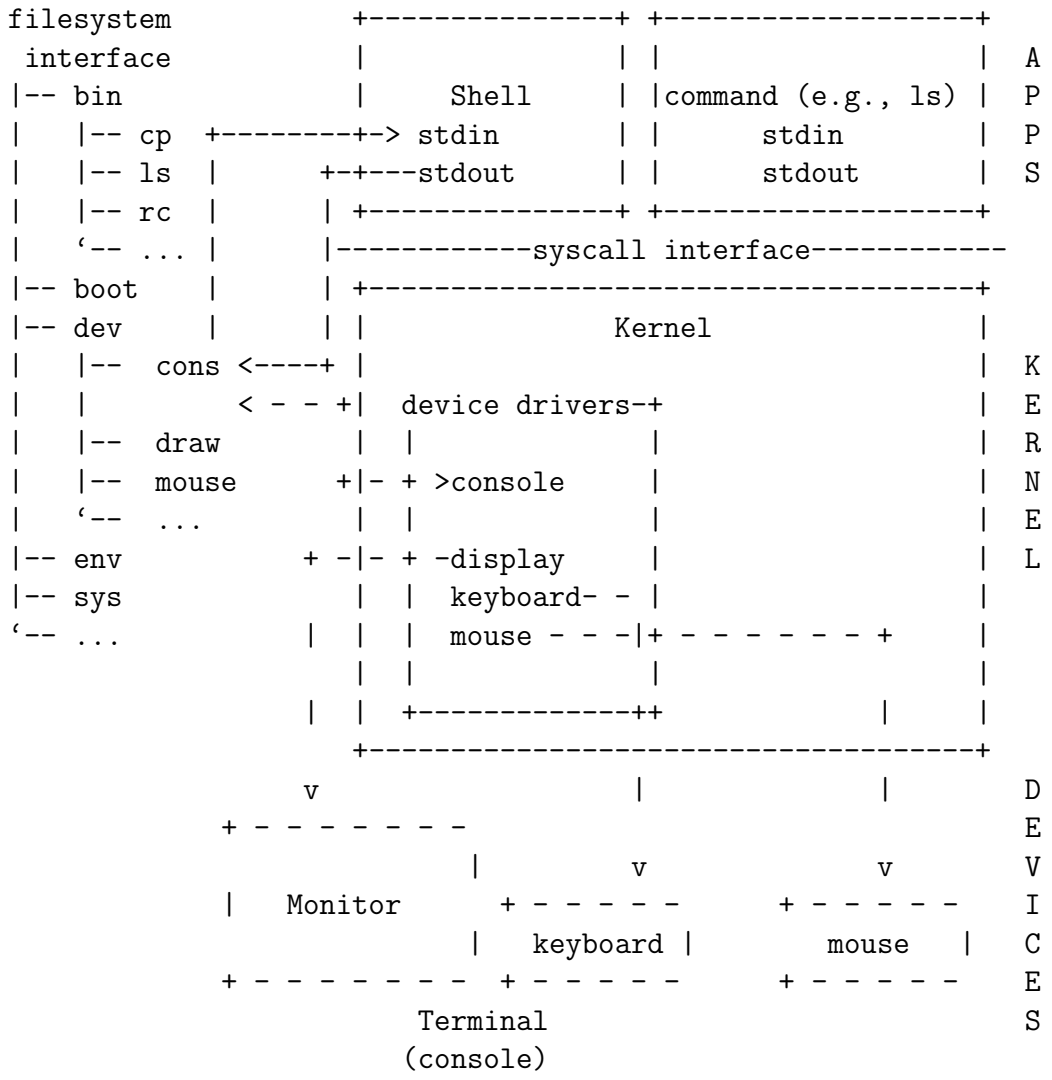


Figure 2.1: Components of the command-line user interface under Plan 9.

```
Input characters --> tokens --> AST --> bytecodes --> threads --> processes
```

Figure 2.2: Data flow diagram of `rc`.

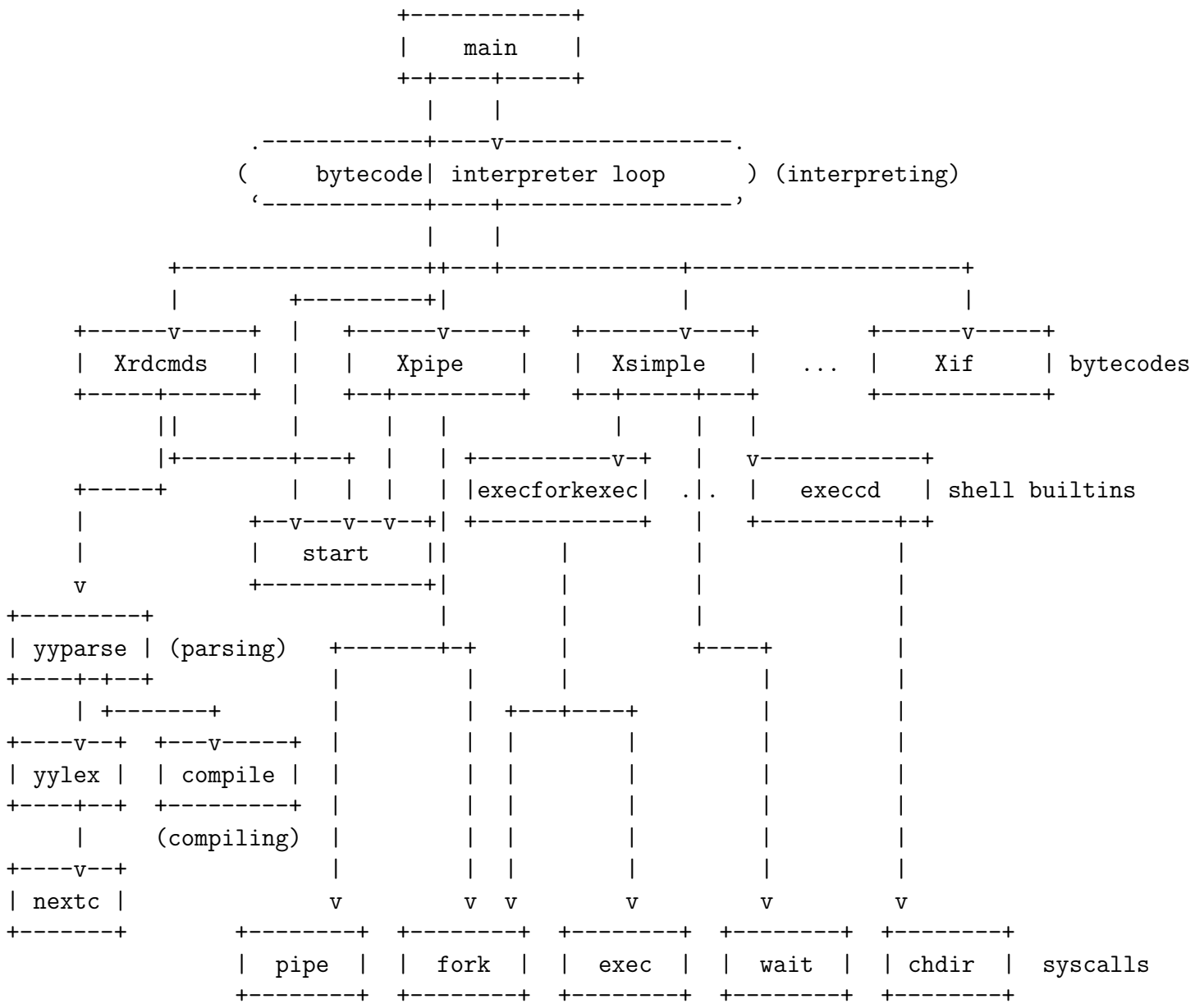


Figure 2.3: Control flow diagram of `rc`.

After some basic initializations, `main()`[42b] sets its local `bootstrap`[42a] to contain the initial set of bytecodes to execute. It then calls `start()`[45b] with `bootstrap` as an argument, to start interpreting this series of bytecodes. Internally, `start()` just modifies the global `runq` to point to a newly created `Thread`, and sets the field `Thread.code`[34d] to the content of `bootstrap`. After `start()` returns, `main()` goes in a loop that interprets the bytecodes in `runq->code`.

The most important bytecode in `bootstrap` is `Xrdcmds()`, which reads a command (hence its name) and starts a new `Thread`. `Xrdcmds()` first calls the parser `yyparse()`, which calls the lexer `yylex()`, which calls the input routine `nextc()`[53a] to get the next character. By default, `nextc()` reads characters from the standard input, and so it waits for the characters you type in a terminal. Once you finish entering a command with a newline, `yyparse()` will call `compile()`[75g] with the tree it built during parsing as an argument. Internally, `compile()` modifies the global `codebuf`[33b] to store the bytecodes deriving from the tree. Then, after `yyparse()` returned, `Xrdcmds()` calls `start()` to start a new thread with this time `codebuf` as an argument (not `bootstrap`). `start()` then modifies again the global `runq`, and when `Xrdcmds()` returns, the main bytecode interpreter loop will process a new series of bytecodes stored in `runq.code`.

Note that before modifying `runq`, `start()` first links the newly created thread with the old thread (through the `Thread.ret`[35b] field). Moreover, `compile()` adds the bytecode `Xreturn()`[96c] at the end of the series of bytecodes deriving from your command. Thus, after `main()` finished interpreting the bytecodes of your command, it will process `Xreturn()`, which will modify `runq` to point to the old thread, the one containing the bootstrap bytecodes. Then, after `Xreturn()` returns, the main bytecode interpreter loop will process again the bytecodes from the bootstrap, which will read another command through `Xrdcmds()`.

In `rc`, the bytecodes are represented by regular C functions starting by convention with an `X` (e.g., `Xrdcmds()`, `Xpipe()`, `Xif()`[100b]). Thus, the bytecode interpreter is mainly a function dispatcher. Internally those functions perform system calls to the kernel to create a pipe (`pipe()`), to fork a new process (`fork()`), to wait for a child process (`wait()`), or to change directory (`chdir()`). An important bytecode is `Xsimple()`[81], which `rc` uses to run a "simple" command. `Xsimple()` represents the essence of a shell: with the series of system calls `fork()`, `exec()`, and `wait()`, `rc` can run a command in a new process and wait for its termination (or interruption). `Xsimple()` is also responsible for managing the multiple shell builtins. Indeed, if the name of the "simple" command is a builtin (e.g., `cd`), then `Xsimple()` dispatches the appropriate function (e.g., `execcd`) instead of forking a new process.

2.5.3 Trace of a simple command: `ls /`

You can see the bytecodes and the threads created internally by `rc` by running `rc` with the `-r` flag. Here is an example of a trace of the simple command `ls /`:

```
% rc -r
pid 39 cycle 0002D930 1 Xmark ()
...
pid 38 cycle 0001984C 9 Xrdcmds
% ls /
pid 38 cycle 0002CBD0 1 Xmark
pid 38 cycle 0002CBD0 2 Xword ()
pid 38 cycle 0002CBD0 4 Xword (/)
pid 38 cycle 0002CBD0 6 Xsimple (ls /)
bin
boot
...
srv
pid 38 cycle 0002CBD0 7 Xreturn
pid 38 cycle 0001984C 9 Xrdcmds
```

It is not important to fully understand the format of this trace (see Section A.2 for the full explanation and for the code handling `rc -r`), but you can recognize a few of the bytecodes I mentioned before: `Xrdcmds()`[47a], `Xsimple()`[81], and `Xreturn()`[96c]. I will explain `Xmark()`[77b] and `Xword()`[77a] later in this document.

2.6 Book structure

You now have enough background to understand the source code of `rc`. The rest of the book is organized as follows. I will start by describing the core data structures of `rc` in Chapter 3. Then, I will use a top-down approach, starting with Chapter 4 with the description of `main()`[42b], the initialization of `rc`, the bytecode interpreter loop, and the function `Xrdcmds()`[47a]. The following chapters will describe the main components of the compilation pipeline: Chapter 5 will present the input routines, Chapter 6 the lexer, Chapter 7 the parser, and finally Chapter 8 the bytecode generator and bytecode interpreter for the main features of `rc`. In Chapter 9, I will present the code of the different shell builtins (e.g., `execcd()`[116c] for `cd`). Chapter 10 contains the code to manage the environment of the processes launched from the shell, and Chapter 11 presents the actual code initializing `rc`. Indeed, Chapter 4 presents only a simplified initialization to not introduce too much complexity early-on. Then, I will present advanced features of `rc` that I did not present before to simplify the explanations, for instance, wildcard matching (e.g., `ls *.c`) in Chapter 12, the code to manage the signals sent to the processes (also known as *notes* under Plan 9) in Chapter 13, or command substitutions (e.g., `` `{ls} ``) in Chapter 14. Those advanced features tend to crosscut many components of `rc` with extensions to the lexer, the parser, the bytecode generator, and the bytecode interpreter. Finally, Chapter 15 concludes and gives pointers to other books in the Principia Softwarica series.

Some appendices present the code of non-functional properties: code to help debug `rc` itself in Appendix A and code to manage errors in Appendix B. Appendix C contains the code of utility functions used by `rc` but that are not specific to `rc` (e.g., a library to manage string buffers). Appendix D presents examples of `rc` scripts. Finally, Appendix E presents the source code of small utility programs closely tied to the shell (such as `echo`), used constantly in scripts and interactive sessions.

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

In this chapter, I will present the core data structures of `rc`. Most of those data structures are listed in Figure 2.2, which presented the data flow of `rc`. The first three sections will present the data structures of the compiler part of `rc`: the token, the abstract syntax tree, and the bytecode. The bytecode is a data structure used both by the compiler and interpreter. The following sections will present the remaining data structures of the interpreter part of `rc`: the thread and the variable, where the variable is itself defined by a list of words.

Note that the original source code of `rc` has been slightly modified to be arguably easier to read. Moreover, in addition to my explanations, I have also sometimes added a few comments in the code itself. The original comments from Tom Duff are using the standard C syntax `/**/` while my comments are using the `//` syntax, so they are easy to differentiate.

3.1 Token

The first step of the compilation pipeline is to transform a stream of characters into a stream of tokens (also called lexemes). A token groups one or more characters that logically belong together (e.g., `while` is a single keyword token, not five character tokens). The lexer `yylex()`[55a] (described in Chapter 6) is responsible for this transformation.

The token kinds in `rc` are declared below using the `yacc` syntax `%token` (see CompilerGenerator book [Pad16a]). I have grouped them in categories: the keyword tokens (`FOR`, `WHILE`, `IF`, etc.), the operator tokens (`REDIR`, `PIPE`, `ANDAND`, etc.), and the catch-all `WORD` token for anything else (e.g., command names, arguments, numbers, or file paths). Note that some single-character operators like `'$'`, `';'`, or `'='` do not have a dedicated token kind; the lexer simply returns their ASCII value directly (see `yylex()` and `pcmd()`[173b]).

⟨*token declarations* 30⟩≡ (65a) 31b▷

```
%token FOR IN WHILE IF NOT SWITCH FN
%token TWIDDLE BANG  /** ~ ! */
%token REDIR PIPE /** {>, <, <<, >>} | */
%token ANDAND ORAND /** && || */
%token COUNT SUB /** $# ( */
%token WORD /** anything else (e.g. foo, --help, 42, /a/b/c, etc) */
```

3.2 Abstract syntax `Tree`

The second step of the compilation pipeline transforms the stream of tokens into an abstract syntax tree (AST). The parser `yyparse()`[65b] (described in Chapter 7) is responsible for this transformation. `rc` has a *command language*: it has variables, control flow statements (`if`, `while`, `for`, `switch`), functions (`fn`), and operators (pipes, redirections, etc.). All those constructs are represented in the AST using the `Tree` structure below.

⟨*struct* `Tree` 31a⟩≡ (223b)
```
struct Tree {

    // either<enum<Token_kind>, char>
    int type;
    // string of the token or AST dump of the whole subtree for certain nodes
    // option<ref_own<string>>
    char *str;

    // array<option<ref_own<Tree>>
    tree *child[3];

    ⟨Tree redirection and pipe specific fields 61a⟩
    ⟨Tree word specific fields 59a⟩

    // Extra
    ⟨Tree extra fields 31d⟩
};
```

The `Tree` structure is a generic tree node with a `type` field indicating what kind of node it is (e.g., a `PIPE`, an `IF`, a `WORD`, or even a single character like `';'` for sequencing), a `str` field to store the string content of the token (e.g., the command name or the variable name), and up to three children. Three children are enough to represent all the constructs of `rc`: for example, an `if` uses `child[0]` for the condition, `child[1]` for the then-body, and leaves `child[2]` unused. A `for` uses all three: `child[0]` for the variable, `child[1]` for the list, and `child[2]` for the body.

The parser also introduces a few extra node types that do not correspond to tokens produced by the lexer, but that are useful to structure the AST. For example, `SIMPLE` represents a simple command with its arguments, and `ARGLIST` groups those arguments into a list.

⟨*token declarations* 31b⟩+≡ (65a) ◁30 72a▷
```
/* not used in syntax */
%token SIMPLE
%token ARGLIST WORDS
%token BRACE PAREN
```

All tree nodes allocated during parsing are linked together in a global list `treenodes`[31c] through the `Tree.next` field further below. This makes it easy to free all nodes at once (via `freenodes()`[32b]) after the tree has been compiled to bytecodes in `Xrdcmds()`[47a], instead of having to walk the tree recursively.

⟨*global* `treenodes` 31c⟩≡ (233b)
```
// list<ref_own<Tree>> (next = Tree.next)
tree *treenodes;
```

⟨`Tree` *extra fields* 31d⟩≡ (31a)
```
// list<ref_own<Tree>> (head = treenodes)
tree *next;
```

The functions below are used by the parser to build AST nodes. `newtree()`[32a] allocates a fresh node and links it into `treenodes`. The convenience wrappers `tree1()`[32c], `tree2()`[32d], and `tree3()`[32e] create a node with

a given type and one, two, or three children respectively.

⟨*function* `newtree` 32a⟩≡ (233b)

```
/*
 * create and clear a new tree node, and add it
 * to the node list.
 */
tree*
newtree(void)
{
    tree *t = new(tree);
    t->str = nil;
    t->child[0] = t->child[1] = t->child[2] = nil;

    // add_list(t, treenodes)
    t->next = treenodes;
    treenodes = t;

    return t;
}
```

Uses `treenodes` 31c.

⟨*function* `freenodes` 32b⟩≡ (233b)

```
/// Xrdcmds -> <>
void
freenodes(void)
{
    tree *t, *u;
    for(t = treenodes;t;t = u){
        u = t->next;
        if(t->str)
            efree(t->str);
        efree((char *)t);
    }
    treenodes = nil;
}
```

Uses `efree()` 181c and `treenodes` 31c.

⟨*function* `tree1` 32c⟩≡ (233b)

```
tree*
tree1(int type, tree *c0)
{
    return tree3(type, c0, (tree *)nil, (tree *)nil);
}
```

Uses `tree3()` 32e.

⟨*function* `tree2` 32d⟩≡ (233b)

```
//@Scheck: used by syn.y
tree* tree2(int type, tree *c0, tree *c1)
{
    return tree3(type, c0, c1, (tree *)nil);
}
```

Uses `tree3()` 32e.

⟨*function* `tree3` 32e⟩≡ (233b)

```
tree*
tree3(int type, tree *c0, tree *c1, tree *c2)
{
    tree *t;
```

```
    ⟨tree3 if some empty sequence 33a⟩
    // else
    t = newtree();
    t->type = type;
    t->child[0] = c0;
    t->child[1] = c1;
    t->child[2] = c2;
    return t;
}
```

Uses `newtree()` 32a.

The special case below is an optimization for the sequencing operator `';'`: if one side of a sequence is empty (which happens frequently since each newline produces an empty command), `tree3()` simply returns the other side instead of creating a useless node.

⟨tree3 *if some empty sequence* 33a⟩≡ (32e)

```
if(type==';'){
    if(c0==nil)
        return c1;
    if(c1==nil)
        return c0;
}
```

3.3 ByteCode and codebuf

The third step of the compilation pipeline transforms the AST into a series of bytecodes. The function `compile()`[75g] (described in Chapter 8) is responsible for this transformation, and stores the resulting bytecodes in the global `codebuf`[33b] below. Like Java or Python, `rc` does not interpret the AST directly but compiles it first into a lower-level representation that is then interpreted by a small stack-based virtual machine (described in Chapter 4).

⟨*global* codebuf 33b⟩≡ (226)

```
// growing_array<ref_own<Code>>
code *codebuf;    /* compiler output */
```

A bytecode is represented by the `Code` union below. Each element in the `codebuf` array is either: a function pointer `f` (the bytecode itself, e.g., `Xsimple`[81], `Xpipe`[97c]), an integer `i` (used for jump offsets or file descriptor numbers), or a string `s` (used for inline string arguments like in `Xword`[77a]). The first element of any code vector is always a reference count integer, managed by `codecopy()`[34a] and `codefree()`[34b] below. This reference counting is needed because bytecode vectors can be shared, for instance when a function is defined with `fn`.

⟨*struct* Code 33c⟩≡ (223b)

```
/*
 * The first word of any code vector is a reference count.
 * Always create a new reference to a code vector by calling codecopy(.).
 * Always call codefree(.) when deleting a reference.
 */
union Code {
    void (*f)(void); // Xxxx() bytecode
    int i;
    // ref_own<string>
    char *s;
};
```

⟨*function* `codecopy` 34a⟩≡ (236a)
```
/// start | Xfn -> <>
code*
codecopy(code *cp)
{
    cp[0].i++;
    return cp;
}
```

⟨*function* `codefree` 34b⟩≡ (236a)
```
/// Xreturn | Xfn | Xdelfn -> <>
void
codefree(code *cp)
{
    code *p;
    // check ref count
    if(--cp[0].i != 0)
        return;
    // else
    for(p = cp+1; p->f; p++){
        ⟨codefree() in loop over code cp, switch bytecode cases 34c⟩
    }
    efree((char *)cp);
}
```
Uses `efree()` 181c.

`codefree()` walks the code vector to release it once its reference count reaches zero, and the walk must stay aligned to opcode boundaries. Most bytecodes occupy a single slot (just their function pointer), but the ones listed here are each followed by one inline operand—a jump target, a file descriptor, and so on. The `p++` skips that slot so the loop does not mistake the operand for the next function pointer. Opcodes with no inline operand need nothing, which keeps this first version simple; we will add the other cases as we meet them.

⟨`codefree()` *in loop over code* `cp`*, switch bytecode cases* 34c⟩≡ (34b) 80h▷
```
if(p->f==Xfalse || p->f==Xtrue
|| p->f==Xread || p->f==Xwrite || p->f==Xrdwr
|| p->f==Xappend || p->f==Xclose
|| p->f==Xasync || p->f==Xbackq || p->f==Xcase
|| p->f==Xfor || p->f==Xjump
|| p->f==Xsubshell)
    p++;
```
Uses `Xappend()` 94b, `Xasync()` 99b, `Xbackq()` 151c, `Xcase()` 105a, `Xclose()` 164e, `Xfalse()` 88b, `Xfor()` 102b, `Xjump()` 101g, `Xrdwr()` 162a, `Xread()` 93b, `Xsubshell()` 155d, `Xtrue()` 88a, and `Xwrite()` 92b.

3.4 `Thread` and `runqeue`

Once the bytecodes are generated, `rc` needs a data structure to keep track of the current execution state. This is the role of the `Thread`[34d] structure. A `Thread` is similar to a *stack frame* in a regular program: it contains a pointer to a code vector (`code`) and a program counter (`pc`) indexing the next bytecode to execute. The name "thread" might be confusing since these are not OS-level threads or processes. They are *units of execution* inside the `rc` interpreter. When a bytecode like `Xpipe()`[97c] or `Xasync()`[99b] needs to run commands concurrently, it creates actual OS processes (with `fork()`), but within the interpreter itself, those commands are also represented as threads, as units of execution.

⟨*struct* `Thread` 34d⟩≡ (225)
```
struct Thread {
    code *code;  /* code for this thread */
    int pc;   /* code[pc] is the next instruction */
```

⟨Thread *other fields* 35c⟩

// Extra
⟨Thread *extra fields* 35b⟩
};

The global `runq`[35a] (for "run queue") points to the top of this stack of threads. Threads are linked through the `ret` field: when the current thread finishes (via `Xreturn()`[96c]), `rc` pops it and resumes the thread pointed to by `ret`. At the bottom of the stack sits the bootstrap thread, which loops forever reading new commands via `Xrdcmds()`[47a].

⟨*global* runq 35a⟩≡ (226)
```
// stack<ref_own<Thread>> (next = Thread.ret)
thread *runq;
```

⟨Thread *extra fields* 35b⟩≡ (34d)
```
thread *ret;  /* who continues when this finishes */
```

One of the most important field of a thread is `argv`, the argument stack. It is a list of lists of words: bytecodes like `Xmark()`[77b] push a new empty list on this stack, and `Xword()`[77a] pushes a word onto the current list. When a bytecode like `Xsimple()`[81] needs to run a command, it pops the top list from `argv` and uses it as the command and its arguments.

⟨Thread *other fields* 35c⟩≡ (34d) 36a▷
```
// list<list<ref_own<word>>> (next = List.next)
list *argv;  /* argument stack */
```

The two-level structure is easier to grasp on a concrete example. Running the single command `ls -l /tmp` compiles roughly to the sequence `Xmark; Xword "/tmp"; Xword "-l"; Xword "ls"; Xsimple`. Here is what `runq->argv` looks like after each of those four bytecodes has executed (bytecodes push to the front of the current word list, so words end up in source order):

```
1: after Xmark              2: after Xword "/tmp"

 argv ---+                  argv ---+
         v                          v
       +---+                      +---+
       |nil|                      |   |-----> Word("/tmp") -> nil
       +---+                      +---+
         |                          |
         v                          v
       (rest of stack)            (rest of stack)

3: after Xword "-l"         4: after Xword "ls"

 argv ---+                  argv ---+
         v                          v
       +---+                      +---+
       |   |->W("-l")->W("/tmp")  |   |->W("ls")->W("-l")->W("/tmp")
       +---+                      +---+

 Xsimple then pops the top List, forks, and calls
 exec("ls", {"ls","-l","/tmp",nil}).
```

Why a two-level list rather than one flat word list? Because the interpreter needs to stack arguments: a pipeline like `ls /tmp | wc -l` must build two argument lists concurrently, an `if`-condition must suspend the argument list of its surrounding command while it evaluates the test, and a command substitution like `echo ‘{date}` must evaluate an inner thread whose argv pushes don't interfere with the outer `echo`. `Xmark` introduces a new top-of-stack whenever we are about to start accumulating a fresh word list; the previous lists sit underneath, ready to be restored when the current one is consumed.

Each thread also has a list of local variables. These are created by the `local` builtin or by constructs like `for` that bind a variable for the duration of a block. When looking up a variable with `vlook()`[39c], `rc` first searches the current thread's locals before falling back to the global variable table.

⟨`Thread` *other fields* 36a⟩+≡ (34d) ◁35c 47b▷
```
// list<ref_own<Var>> (next = Var.next)
var *local;  /* list of local variables */
```

3.5 Words and lists of words

I mentioned the `argv` field in the `Thread` structure above, which is a list of lists of words. Let me now present the `Word` and `List` structures that make up this two-level list.

3.5.1 `Words`

A `Word` is simply a string with a `next` pointer, forming a linked list. This is the fundamental data type in `rc`: a command like `ls -l /tmp` is represented as a list of three words. Variables in `rc` can also hold multiple values (e.g., `x=(a b c)`), which are represented as a list of words.

⟨*struct* `Word` 36b⟩≡ (223b)
```
/*
 * word lists are in correct order,
 * i.e. word0->word1->word2->word3->nil
 */
struct Word {
    // ref_own<string>
    char *word;

    // Extra
    word *next;
};
```

The functions below provide the usual linked-list operations for word lists: `newword()`[36c] allocates a new word and prepends it to a list, `count()`[37a] returns the length, `copywords()`[37d] duplicates a list, and `freewords()`[37b] deallocates it.

⟨*function* `newword` 36c⟩≡ (239a)
```
word*
newword(char *wd, word *next)
{
    word *p = new(struct Word);
    p->word = strdup(wd);
    p->next = next;
    return p;
}
```
Uses `Word` 36b.

⟨*function* count 37a⟩≡ (239a)
```
int
count(word *w)
{
    int n;
    for(n = 0;w;n++)
        w = w->next;
    return n;
}
```

⟨*function* freewords 37b⟩≡ (239a)
```
void
freewords(word *w)
{
    word *nw;
    while(w){
        efree(w->word);
        nw = w->next;
        efree((char *)w);
        w = nw;
    }
}
```
Uses efree() 181c.

copynwords()[37c] and copywords() use the classic C pointer-to-pointer tail linking idiom: end starts as &v (address of the head pointer), and after each node is created, it advances to &(*end)->next (address of the new node's next field). This avoids special-casing the first element and builds the list in forward order without reversing at the end.

⟨*function* copynwords 37c⟩≡ (239a)
```
word*
copynwords(word *a, word *tail, int n)
{
    word *v = nil;
    word **end = &v;

    while(n-- > 0){
        *end = newword(a->word, 0);
        end = &(*end)->next;
        a = a->next;
    }
    *end = tail;
    return v;
}
```
Uses newword() 36c.

⟨*function* copywords 37d⟩≡ (239a)
```
/*
 * copy arglist a, adding the copy to the front of tail
 */
word*
copywords(word *a, word *tail)
{
    word *v = nil;
    word **end;

    for(end=&v;a;a = a->next,end=&(*end)->next)
        *end = newword(a->word, nil);
    *end = tail;
    return v;
```

```
}
```
Uses newword() 36c.

3.5.2 List of sequence of words

The List structure adds one more level of indirection on top of word lists. It is used for Thread.argv, the argument stack: each element in this stack is a List node containing a word list. For example, when the interpreter processes ls -l /tmp, the bytecodes Xmark()[77b] and Xword()[77a] will first push the words ls, -l, /tmp onto a word list, and this word list sits inside a List node on the argv stack.

⟨*struct* List 38a⟩≡ (225)
```
struct List {
    // list<ref_own<Word>> (next = Word.next)
    word *words;

    // Extra
    list *next;
};
```

freelist()[38b] is functionally identical to freewords()[37b]: both walk a word list and free each string and each node. The two names are just historical duplication—by convention freewords() releases a variable's value while freelist() releases the word lists popped off the argv and local stacks—but neither does anything the other does not.

⟨*function* freelist 38b⟩≡ (239a)
```
void
freelist(word *w)
{
    word *nw;
    while(w){
        nw = w->next;
        efree(w->word);
        efree((char *)w);
        w = nw;
    }
}
```
Uses efree() 181c.

3.6 Variables

I already mentioned local variables in Thread.local[36a] above. Let me now present the Var structure itself and the global variable table. Variables are needed whenever you write something like x=1 or path=(/bin /usr/bin) in the shell. They are also used internally by rc for *special variables* like $status (the exit status of the last command) or $* (the script arguments).

A Var has a name (the variable name as a string, without the leading dollar), a val (the variable value as a list of words, since in rc all variables can hold multiple values), and a next pointer for chaining in hash buckets or local lists.

⟨*struct* Var 38c⟩≡ (223b)
```
struct Var {
    // key
    char *name;  /* ascii name */
    // value
    // ref_own<list<Word>>
    word *val;  /* value */
```

```
    ⟨Var other fields 105d⟩
    // Extra
    ⟨Var extra fields 39a⟩
};
```

⟨Var *extra fields* 39a⟩≡ (38c)
```
// list<ref_own<var>> (head = runq->local list or gvar hash)
var *next;  /* next on hash or local list */
```

The main interface to the variable consists of `setvar()`[39b] to assign a value, and `vlook()`[39c] to look up a variable by name. `vlook()` first searches the local variables of the current thread (stored in `runq->local`), then falls back to the global variable table via `gvlook()`[39f].

⟨*function* setvar 39b⟩≡ (229a)
```
void
setvar(char *name, word *val)
{
    var *v = vlook(name);
    freewords(v->val);
    v->val = val;
    v->changed = true;
}
```
Uses `freewords()` 37b and `vlook()` 39c.

⟨*function* vlook 39c⟩≡ (229a)
```
var*
vlook(char *name)
{
    var *v;
    if(runq)
        for(v = runq->local;v;v = v->next)
            if(strcmp(v->name, name)==0)
                return v;

    return gvlook(name);
}
```
Uses `gvlook()` 39f and `runq` 35a.

Global variables are stored in a simple hash table of 521 buckets, with chaining through `Var.next` for collisions. `gvlook()` looks up a global variable and automatically creates it (with a `nil` value) if it does not exist yet.

⟨*constant* NVAR 39d⟩≡ (223b)
```
#define NVAR 521
```

⟨*global* gvar 39e⟩≡ (229a)
```
// map<string, ref_own<Var>> (next = Var.next in bucket list)
var *gvar[NVAR];  /* hash for globals */
```
Uses `NVAR` 39d.

⟨*function* gvlook 39f⟩≡ (229a)
```
var*
gvlook(char *name)
{
    int h = hash(name, NVAR);
    var *v;

    for(v = gvar[h];v;v = v->next)
        if(strcmp(v->name, name)==0)
            return v;
```

```
    gvar[h] = newvar(strdup(name), gvar[h]);
    return gvar[h];
}
```

Uses NVAR 39d, gvar 39e, hash() 40a, and newvar() 40b.

⟨*function* hash 40a⟩≡ (229a)

```
unsigned
hash(char *as, int n)
{
    int i = 1;
    unsigned h = 0;
    uchar *s;

    s = (uchar *)as;
    while (*s)
        h += *s++ * i++;
    return h % n;
}
```

⟨*function* newvar 40b⟩≡ (229a)

```
var*
newvar(char *name, var *next)
{
    var *v = new(var);
    v->name = name;
    v->val = nil;

    v->fn = nil;
    v->changed = false;
    v->fnchanged = false;

    v->next = next;
    return v;
}
```

3.7 Putting it together: the data structures of `hello.rc`

It is worth pausing here to see how the core data structures fit together before `main()`[42b] starts using them. Rather than listing fields in the abstract, Figure 3.1 shows the actual instances `rc` creates for the single command `ls -l /tmp`, as it flows through the four stages of the shell: lexing, parsing, compiling, and interpreting.

The same data is reshaped at each stage. The lexer (`yylex()`[55a]) splits the line into a stream of tokens; the parser (`yyparse()`[65b]) nests those tokens into an AST of `Tree` nodes; `compile()`[75g] flattens the tree into a `Code`[33c] vector of bytecodes; and running that vector rebuilds the arguments as a `Word`[36b] list inside the `Thread`[34d]. In the figure, a `*` marks a pointer field that an arrow follows.

Each stage is the subject of a later chapter: lexing in Chapter 6, parsing in Chapter 7, and both compilation and interpretation in Chapter 8. The two-level `List`/`Word` structure of `runq->argv`, only sketched here, is explained in Section 3.4.

```
SOURCE       ls -l /tmp <newline>

(1) LEXING   yylex() returns one token at a time
             WORD "ls"    WORD "-l"    WORD "/tmp"    '\n'

(2) PARSING  yyparse() nests the tokens into an AST of Tree nodes
             +-----------------------------+
             | SIMPLE   str="ls -l /tmp"   |
             | child[0] *--+               |
             +-------------|---------------+
                           v
             +-----------------------------+
             | ARGLIST                     |
             | child[0] *--+   child[1] *------> WORD str="/tmp"
             +-------------|---------------+
                           v
             +-----------------------------+
             | ARGLIST                     |
             | child[0] *------> WORD str="ls"
             | child[1] *------> WORD str="-l"
             +-----------------------------+

(3) COMPILING  compile()/outcode() flatten the AST into a code vector
             codebuf:
               [0]  .i = 1          <- reference count (held by runq->code)
               [1]  .f = Xmark
               [2]  .f = Xword      [3]  .s = "/tmp"
               [4]  .f = Xword      [5]  .s = "-l"
               [6]  .f = Xword      [7]  .s = "ls"
               [8]  .f = Xsimple
               [9]  .f = Xreturn    [10] .f = nil
             (one Xword per argument, emitted in reverse since each
              Xword prepends to the current word list)

(4) INTERPRETING  the runq Thread walks the code vector
             +------------------------+
    runq *-->| Thread                 |
             | code  *--> codebuf above
             | pc         steps [1]..[10]
             | argv  *--+             |
             | local = nil            |
             +---------|--------------+
                       v
             +-----------+  words
             | List      |*-----> Word "ls" -> Word "-l" -> Word "/tmp" -> nil
             +-----------+
             (Xmark pushed the List; the three Xword built the Word
              chain; Xsimple reads it as argv and fork/execs "ls")

VARIABLES    global table (var.c), consulted by Xsimple via $path:
             gvar[h] *--> Var{ name:"path", val: Word "." -> Word "/bin" -> nil }
```

Figure 3.1: The core data structures of `rc`, instantiated for `ls -l /tmp`.

Chapter 4

`main()`

I now switch from the bottom-up approach of Chapter 3 to a top-down approach: I will describe in the following chapters the main functions of `rc`, starting in this chapter with `main()`[42b], the entry point of `rc`.

4.1 `main()` skeleton

The `main()`[42b] function of `rc` performs four tasks: it processes the command-line arguments, it initializes the shell (keywords, traps, environment), it sets up the bootstrap bytecodes and the initial thread, and then it enters the bytecode interpreter loop. I will present in this chapter a simplified version of the bootstrap; the actual bootstrap code (which sources `rcmain`) is described in Chapter 11.

⟨`main()` *locals* 42a⟩≡ (42b) 44c▷
```
code bootstrap[17];
```

⟨*function* `main` 42b⟩≡ (240a)
```
/*
 * get command line flags.
 * initialize keywords & traps.
 * get values from environment.
 * set $pid, $cflag, $*
 * fabricate bootstrap code and start it (*=(argv);. /usr/lib/rcmain $*)
 * start interpreting code
 */
void main(int argc, char *argv[])
{
    ⟨main() locals 42a⟩

    ⟨main() argc argv processing, modify flags 43d⟩

    ⟨main() initialisation 44a⟩
    ⟨main() initialize boostrap 129a⟩
    ⟨main() initialize runq with bootstrap code 45a⟩
    ⟨main() initialize runq->argv 45d⟩

    ⟨main() interpreter loop 46c⟩
}
```

4.2 Command-line arguments processing

`rc` accepts many flags (e.g., `-i` for interactive mode, `-l` for login mode, `-e` for exit-on-error). Flags are stored in a global array indexed by the flag character, where a non-nil entry means the flag is set. Some flags like `-c`

and `-m` take a string argument, which is why `flag` stores string arrays rather than simple booleans.

⟨*global* `flag` 43a⟩≡ (227a)
```
// map<char, option<array<string>>>
char **flag[NFLAG];
```
Uses `NFLAG` 43b.

⟨*constant* `NFLAG` 43b⟩≡ (222)
```
#define NFLAG 128
```

⟨*global* `flagset` 43c⟩≡ (227a)
```
char *flagset[] = {"<flag>"};
```

⟨`main()` *argc argv processing, modify flags* 43d⟩≡ (42b) 43e▷
```
argc = getflags(argc, argv, "SsrdiIlxepvVc:1m:1[command]", 1);
if(argc==-1)
    usage("[file [arg ...]]");
```
Uses `getflags()` 182i.

`getflags()`[182i] is defined in Appendix C. The format string encodes which flags exist and which ones take an argument (`c:1` means `-c` takes one argument). It populates the global `flag` array declared above.

4.2.1 Login mode: `rc -l`

When `rc` is started as a login shell, its parent (the login or console service that execs it) sets `argv[0]` to `"-rc"` (starting with a dash) rather than passing an explicit `-l` flag. This is a UNIX convention: the code below detects the leading dash and sets `flag['l']` accordingly, so that `rc` will later source the user's profile (`$home/lib/profile`, see Appendix D for an example).

⟨`main()` *argc argv processing, modify flags* 43e⟩+≡ (42b) ◁43d 43f▷
```
if(argv[0][0]=='-')
    flag['l'] = flagset;
```
Uses `flag` 43a and `flagset` 43c.

4.2.2 Interactive mode: `rc -i`

Interactive mode determines whether `rc` will display a prompt and handle notes (signals) differently (e.g., not exiting on `^C`). The logic below is subtle: `rc` is interactive by default when no script file is given (`argc==1`) and standard input is a terminal. The `-I` flag forces non-interactive mode.

⟨`main()` *argc argv processing, modify flags* 43f⟩+≡ (42b) ◁43e
```
if(flag['I'])
    flag['i'] = nil;
else
    if(flag['i']==nil && argc==1 && Isatty(STDIN))
       flag['i'] = flagset;
```
Uses `flag` 43a and `flagset` 43c.

Under Plan 9, there is no `isatty()` system call. Instead, `Isatty()`[43g] checks whether the file descriptor corresponds to `/dev/cons` (the console device) by inspecting its path with `fd2path()`.

⟨*function* `Isatty` *(plan9.c)* 43g⟩≡ (240b)
```
bool
Isatty(fdt fd)
{
    char buf[64];

    if(fd2path(fd, buf, sizeof buf) != 0)
```

```
        return false;

    /* might be #c/cons during boot - fixed 22 april 2005, remove this later */
    if(strcmp(buf, "#c/cons") == 0)
        return true;

    /* might be /mnt/term/dev/cons */
    return strlen(buf) >= 9 && strcmp(buf+strlen(buf)-9, "/dev/cons") == 0;
}
```

5.2.3

4.3 Initialization

The initialization first sets up `err`, a buffered IO handle for standard error (see Appendix C for `openfd()`[186h]). Then it initializes three subsystems: `kinit()`[63e] registers the shell keywords (`if`, `for`, etc.) so the lexer can recognize them, `Trapinit()`[144e] sets up the signal handler (see Chapter 13), and `Vinit()`[124d] reads the process environment (from `/env`) and populates the global variable table (see Chapter 10).

⟨`main()` *initialisation* 44a⟩≡ (42b) 44b▷
```
err = openfd(STDERR);
```
Uses `err` 179b and `openfd()` 186h.

⟨`main()` *initialisation* 44b⟩+≡ (42b) ◁44a 129d▷
```
kinit();    // initialize keywords
Trapinit(); // notify() function setup
Vinit();    // read environment variables and add them in gvar
```
Uses `Vinit()` 124d and `kinit()` 63e.

4.4 Bootstrapping bytecodes (simplified)

As I mentioned earlier, `rc` does not start by compiling source code; it starts by *interpreting* a small series of hand-crafted bytecodes called the *bootstrap*. In its simplest form, the bootstrap just contains `Xrdcmds`[47a], the bytecode that reads a command, compiles it, and starts a new thread to execute it. The first element is the reference count (set to 1), and the last element is a zero sentinel marking the end of the code vector. The actual bootstrap is more complex (see Chapter 11): it sources the `rcmain` initialization script, which sets up the default `$path` and runs the user's profile, but for now it is simpler to assume the simpler bootstrap below.

⟨`main()` *locals* 44c⟩+≡ (42b) ◁42a 129c▷
```
int i;
```

⟨`main()` *initialize* `boostrap` *(simplified)* 44d⟩≡
```
memset(bootstrap, 0, sizeof bootstrap);

i = 0;
bootstrap[i++].i = 1; // reference count
bootstrap[i++].f = Xrdcmds;
bootstrap[i].i = 0;
```

4.5 Setting `runq`

`start()`[45b] creates a new thread and pushes it on top of `runq`[35a]. The program counter starts at 1 (not 0) because element 0 of any code vector is reserved for the reference count. Note how `start()` links the new thread

to the old `runq` through `p->ret = runq`: this is how the interpreter knows where to resume after the current thread finishes.

⟨`main()` *initialize* `runq` *with bootstrap code* 45a⟩≡ (42b) 47c▷
```
start(bootstrap, 1, (var *)nil);
```
Uses `start()` 45b.

⟨*function* `start` 45b⟩≡ (230a)
```
/// main | Xrdcmds -> <>
void
start(code *c, int pc, var *local)
{
    struct Thread *p = new(struct Thread);

    p->code = codecopy(c);
    p->pc = pc;

    p->argv = nil;
    p->local = local;

    p->cmdfile = nil;
    p->cmdfd = nil;
    p->lineno = 1;
    p->eof = false;
    p->iflag = false;

    ⟨start() set redir 90b⟩

    // add_stack(runq, p)
    p->ret = runq;
    runq = p;
}
```
Uses `Thread` 34d, `codecopy()` 34a, and `runq` 35a.

4.6 Setting `runq->argv`

Now that `runq`[35a] is set, `main()`[42b] populates its argument stack with the command-line arguments. These arguments are pushed in reverse order because `pushword()`[46b] prepends to the list, so the first argument ends up at the top. The actual bootstrap code will later assign this list to `$*` (the script arguments variable).

⟨*global* `argv0` 45c⟩≡ (230a)
```
/*
 * Start executing the given code at the given pc with the given redirection
 */
char *argv0="rc";
```
Uses `argv0` 45c.

⟨`main()` *initialize* `runq->argv` 45d⟩≡ (42b)
```
/* prime bootstrap argv */
pushlist();
argv0 = strdup(argv[0]);
for(i = argc-1; i!=0; --i)
    pushword(argv[i]);
```
Uses `argv0` 45c, `pushlist()` 46a, and `pushword()` 46b.

For example, after `rc myscript.rc foo bar` the data structures look like this:

```
runq ----> Thread
             code = bootstrap [1:Xrdcmds, 0]
             pc   = 1
             argv ----> List
                          words -> "myscript.rc" -> "foo" -> "bar" -> nil
                          next  = nil
             ret  = nil   (bottom of stack)
```

The real bootstrap code will then assign this word list to `$*` and source `rcmain`, which sets up `$path` and runs the script.

`pushlist()`[46a] pushes a new empty word list onto `runq->argv`, and `pushword()` adds a word to the current top list. These two functions are the main interface for building the argument stack and are used extensively by bytecodes like `Xmark()`[77b] and `Xword()`[77a].

⟨*function* `pushlist` 46a⟩≡ (230a)

```
/// main | Xmark ->  <>
void
pushlist(void)
{
    list *p = new(list);

    // add_list(p, runq->argv)
    p->next = runq->argv;
    p->words = nil;
    runq->argv = p;
}
```

Uses `runq` 35a.

⟨*function* `pushword` 46b⟩≡ (230a)

```
void
pushword(char *wd)
{
    if(runq->argv==nil)
        panic("pushword but no argv!", 0);
    runq->argv->words = newword(wd, runq->argv->words);
}
```

Uses `newword()` 36c, `panic()` 180c, and `runq` 35a.

4.7 Bytecode interpreter loop

The heart of `rc` is the infinite loop below. It increments the program counter, then calls the function pointer at the previous position. The increment-then-call-at-minus-one pattern is a common idiom: the `pc` is advanced *before* calling the bytecode function, so that if the bytecode needs to read inline arguments (e.g., a string after `Xword`[77a]), it can read them at `runq->code[runq->pc]` and advance `pc` itself.

⟨`main()` *interpreter loop* 46c⟩≡ (42b)

```
for(;;){
    ⟨main() debug runq in interpreter loop 176a⟩

    runq->pc++;
    (*runq->code[runq->pc-1].f)();

    ⟨main() handing trap if necessary in interpreter loop 145a⟩
}
```

Uses `runq` 35a.

Note that between bytecodes, the loop also checks for pending signals (traps).

4.8 Reading commands: `Xrdcmds()`

`Xrdcmds()`[47a] is the bytecode that implements the read-eval loop of `rc`. It calls `yyparse()`[65b] to read and compile one command line, then calls `start()`[45b] to push a new thread executing the compiled bytecodes.

⟨*function* `Xrdcmds` 47a⟩≡ (230b)
```
void
Xrdcmds(void)
{
    struct Thread *p = runq;
    bool error;
    ⟨Xrdcmds() other locals 51e⟩

    ⟨Xrdcmds() flush errors and reset error count 179d⟩
    ⟨Xrdcmds() print status if -s 177b⟩
    ⟨Xrdcmds() set promptstr if interactive mode 51f⟩

    ⟨Xrdcmds() calls Noerror() before yyparse() 146c⟩
    // read one cmd line, compiles it, and modifies codebuf global
    error = yyparse();

    ⟨Xrdcmds() if yyparse() returned an error 48⟩
    else{
        ⟨Xrdcmds() reset ntrap 146e⟩
        --p->pc; /* re-execute Xrdcmds after codebuf runs */
        // modifies runq, new thread (linked to bootstrap one)
        start(codebuf, 1, runq->local);
    }
    freenodes(); // allocated in yyparse()
}
```
Uses `Thread` 34d, `codebuf` 33b, `freenodes()` 32b, `runq` 35a, `start()` 45b, and `yyparse()`.

The key subtlety is the `--p->pc` before `start()`: this makes the bootstrap thread re-execute `Xrdcmds()` when the command thread finishes (via `Xreturn()`[96c]). This is how `rc` loops forever reading commands without an explicit loop in the bootstrap bytecodes themselves. Note that `yyparse()` internally calls `compile()`[75g], which modifies the global `codebuf`[33b]. So when `Xrdcmds()` calls `start(codebuf, ...)`, it starts a thread on the freshly compiled bytecodes.

Each thread also carries input-related fields: `cmdfd` below is the buffered IO handle from which the lexer reads characters, `cmdfile` is the filename (for error messages), and `iflag` determines whether this thread displays a prompt. These are per-thread fields (not globals) because the `.` (dot) builtin (see Section 9.4) and `-c` flag (see Section 14.1) create new threads that read from different sources (a script file, a string), and each thread needs its own input state.

⟨`Thread` *other fields* 47b⟩+≡ (34d) ◁36a 50c▷
```
struct Io *cmdfd; /* file descriptor for Xrdcmd */
char *cmdfile;  /* file name in Xrdcmd */
bool iflag;  /* interactive? */
```
Uses `Io` 186f.

⟨`main()` *initialize* `runq` *with bootstrap code* 47c⟩+≡ (42b) ◁45a
```
runq->cmdfd = openfd(STDIN); // reading from stdin
runq->cmdfile = "<stdin>";
runq->iflag = flag['i']? true : false;// interactive mode; will print a prompt
```
Uses `flag` 43a, `openfd()` 186h, and `runq` 35a.

When `yyparse()` returns an error, the behavior depends on the mode. In non-interactive mode (or on a genuine EOF), `rc` closes the input and returns to the parent thread. In interactive mode, it simply prints a newline after an interrupt (`^C`) and goes back to read the next command.

⟨`Xrdcmds()` *if yyparse() returned an error* 48⟩≡ (47a)

```
if(error){
    if(!p->iflag  ||  p->eof && !Eintr()){
        if(p->cmdfile)
            efree(p->cmdfile);
        closeio(p->cmdfd);

        Xreturn(); /* should this be omitted? */
    }else{
        if(Eintr()){
            pchr(err, '\n');
            p->eof = false;
        }
        --p->pc; /* go back for next command */
    }
}
```

Uses `Xreturn()` 96c, `closeio()` 189a, `efree()` 181c, `err` 179b, and `pchr()` 188a.

Chapter 5

Input

Recall the main flow from Figure 2.3: `main()`[42b] → `Xrdcmds()`[47a] → `yyparse()`[65b] → `yylex()`[55a]. Before describing the lexer, I will present the input routines it relies on. These routines handle reading characters from the current input source (terminal or script file), managing the prompt, and providing the one-character lookahead that the lexer needs.

5.1 Overview

The call chain for reading input is shown below:

```
yylex()
  |
  +---> nextc(); advance()   one-character lookahead (cached in 'future')
          |
          +---> getnext()      raw character read + prompt + escaped newlines
                  |
                  +---> rchr(runq->cmdfd)    buffered IO read
                          |
                          +---> read()       system call (terminal or file)
```

`getnext()`[49] reads a raw character and handles escaped newlines (for multi-line commands) and prompt display. `nextc()`[53a] adds a one-character lookahead on top of `getnext()`, and `advance()`[53b] consumes that lookahead.

5.2 Reading a character: `getnext()`

`getnext()`[49] reads one character from the current thread's input stream (`runq->cmdfd`). Besides the actual read (via `rchr()`[188c], the buffered IO read function), it handles three concerns: returning a previously peeked character (`peekc`), displaying the prompt before blocking on input, and converting escaped newlines (`\<newline>`) into spaces to support multi-line commands.

⟨*function* `getnext` 49⟩≡ (228)

```
/// yylex -> nextc -> <>
/*
 * read a character from the input stream
 */
int
getnext(void)
{
    int c;
    ⟨getnext() other locals 51a⟩
```

```
    ⟨getnext() before read, peekc handling 51b⟩
    ⟨getnext() before read, return if already at EOF 50d⟩
    ⟨getnext() before read, prompt management 51g⟩

    c = rchr(runq->cmdfd);

    ⟨getnext() after read, handle backslash 50e⟩
    ⟨getnext() after read, prompt management 52c⟩
    ⟨getnext() after read, if EOF 50b⟩
    ⟨getnext() after read, if not EOF but verbose mode, print character read 178⟩

    return c;
}
```

Uses rchr() 188c and runq 35a.

5.2.1 End-of-file management

When rchr()[188c] returns EOF, getnext()[49] sets the eof flag on the current thread. On subsequent calls, getnext() returns EOF immediately without trying to read again. This flag is also checked in Xrdcmds()[47a] to decide whether to close the input and return to the parent thread.

⟨constant EOF (rc/io.h) 50a⟩≡ (223a)
```
#define EOF (-1)
```

⟨getnext() after read, if EOF 50b⟩≡ (49)
```
if(c==EOF)
    runq->eof = true;
```

Uses EOF 50a and runq 35a.

⟨Thread other fields 50c⟩+≡ (34d) ◁47b 52b▷
```
bool eof;  /* is cmdfd at eof? */
```

⟨getnext() before read, return if already at EOF 50d⟩≡ (49)
```
if(runq->eof)
    return EOF;
```

Uses EOF 50a and runq 35a.

5.2.2 Multiple lines commands and escaped newlines

A backslash before a newline tells the shell that the command continues on the next line. This lets you spread a long command—a pipeline with many stages, or a command with many arguments—over several lines for readability, without the shell running it as soon as you press return. getnext()[49] detects this sequence and returns a space instead of the newline, effectively joining the two lines. If the backslash is followed by any other character, it is not consumed: the backslash is returned normally, and the next character is saved in peekc for the next call.

⟨getnext() after read, handle backslash 50e⟩≡ (49)
```
if(!inquote && c=='\\'){

    c = rchr(runq->cmdfd);

    if(c=='\n' && !incomm){  /* don't continue a comment */
        ⟨getnext() when backslash and newline, set doprompt 52d⟩
        c=' ';
    }
```

```
    else{
        peekc = c;
        c='\\';
    }
}
```

Uses incomm 56e, inquote 58a, rchr() 188c, and runq 35a.

⟨getnext() *other locals* 51a⟩≡ (49)
```
static int peekc = EOF;
```

Uses EOF 50a.

⟨getnext() *before read, peekc handling* 51b⟩≡ (49)
```
if(peekc!=EOF){
    c = peekc;
    peekc = EOF;
    return c;
}
```

Uses EOF 50a.

5.2.3 Displaying the prompt

In rc, the $prompt variable is a two-element list: the first element is the main prompt (e.g., "% "), and the second element is the continuation prompt displayed when a command spans multiple lines (e.g., "\t"). The prompt is printed by the input layer, just before it reads the next character—not by the command-execution code when a command finishes (You do still see the prompt after each command, but only because the shell then loops back to read the next line.). The mechanism is the global doprompt flag, set to true after each newline (or EOF) and reset to false once the prompt has been printed.

⟨*global* doprompt 51c⟩≡ (228)
```
bool doprompt = true;
```

Uses doprompt 51c.

⟨*global* promptstr 51d⟩≡ (226)
```
char *promptstr;
```

⟨Xrdcmds() *other locals* 51e⟩≡ (47a)
```
// list<ref_own<word> (2 elts)
word *prompt;
```

⟨Xrdcmds() *set promptstr if interactive mode* 51f⟩≡ (47a)
```
if(runq->iflag){
    prompt = vlook("prompt")->val;
    if(prompt)
        promptstr = prompt->word;
    else
        promptstr="% ";
}
```

Uses promptstr 51d, runq 35a, and vlook() 39c.

⟨getnext() *before read, prompt management* 51g⟩≡ (49)
```
if(doprompt)
    // pprompt() internally set doprompt back to false at the end
    pprompt();
```

Uses doprompt 51c and pprompt() 52a.

The subtlety in `pprompt()`[52a] is that after printing the current prompt string, it switches `promptstr` to the *second* element of `$prompt` (the continuation prompt). This way, if the lexer needs more input for a multi-line command, the next call to `pprompt()` will display the continuation prompt instead of the main one. The main prompt is restored in `Xrdcmds()`[47a] before each new command as shown above.

⟨*function* `pprompt` 52a⟩≡ (228)
```
void
pprompt(void)
{
    var *prompt;

    if(runq->iflag){
        // printing the prompt
        pstr(err, promptstr);
        flush(err);

        // set promptstr for the next pprompt()
        prompt = vlook("prompt");
        if(prompt->val && prompt->val->next)
            promptstr = prompt->val->next->word;
        else
            promptstr="\t";
    }
    runq->lineno++;
    doprompt = false;
}
```
Uses `doprompt` 51c, `err` 179b, `flush()` 187b, `promptstr` 51d, `pstr()` 191a, `runq` 35a, and `vlook()` 39c.

Each thread also tracks the current line number, incremented in `pprompt()` after each newline. This is used for error reporting when running scripts: a syntax error in line 42 of a script is much easier to find than one at an unknown location.

⟨`Thread` *other fields* 52b⟩+≡ (34d) ◁50c 90d▷
```
int lineno;   /* linenumber */
```

⟨`getnext()` *after read, prompt management* 52c⟩≡ (49)
```
doprompt = doprompt || c=='\n' || c==EOF;
```
Uses `EOF` 50a and `doprompt` 51c.

⟨`getnext()` *when backslash and newline, set doprompt* 52d⟩≡ (50e)
```
doprompt = true;
```
Uses `doprompt` 51c.

5.3 Looking ahead: `nextc()` and `advance()`

Lexers often need to look one character ahead to decide what token to produce. For example, when the lexer sees `&`, it must peek at the next character to decide whether it is a lone `&` (background operator) or part of `&&` (logical and). `nextc()`[53a] provides this lookahead by caching the next character in the global `future`. Multiple calls to `nextc()` return the same character until `advance()`[53b] is called to consume it. The helper `nextis()`[53c] combines both: it checks whether the next character matches, and consumes it if so.

⟨*global* `future` 52e⟩≡ (228)
```
int future = EOF;
```
Uses `EOF` 50a and `future` 52e.

⟨*function* `nextc` 53a⟩≡ (228)

```
/// yylex | advance | skipnl | nextis -> <>
/*
 * Look ahead in the input stream
 */
int
nextc(void)
{
    if(future==EOF)
        future = getnext();
    return future;
}
```

Uses `EOF` 50a, `future` 52e, and `getnext()` 49.

⟨*function* `advance` 53b⟩≡ (228)

```
/// nextis -> <>
/*
 * Consume the lookahead character.
 */
int
advance(void)
{
    int c = nextc();
    ⟨advance() save future in lastc 180b⟩
    future = EOF;
    return c;
}
```

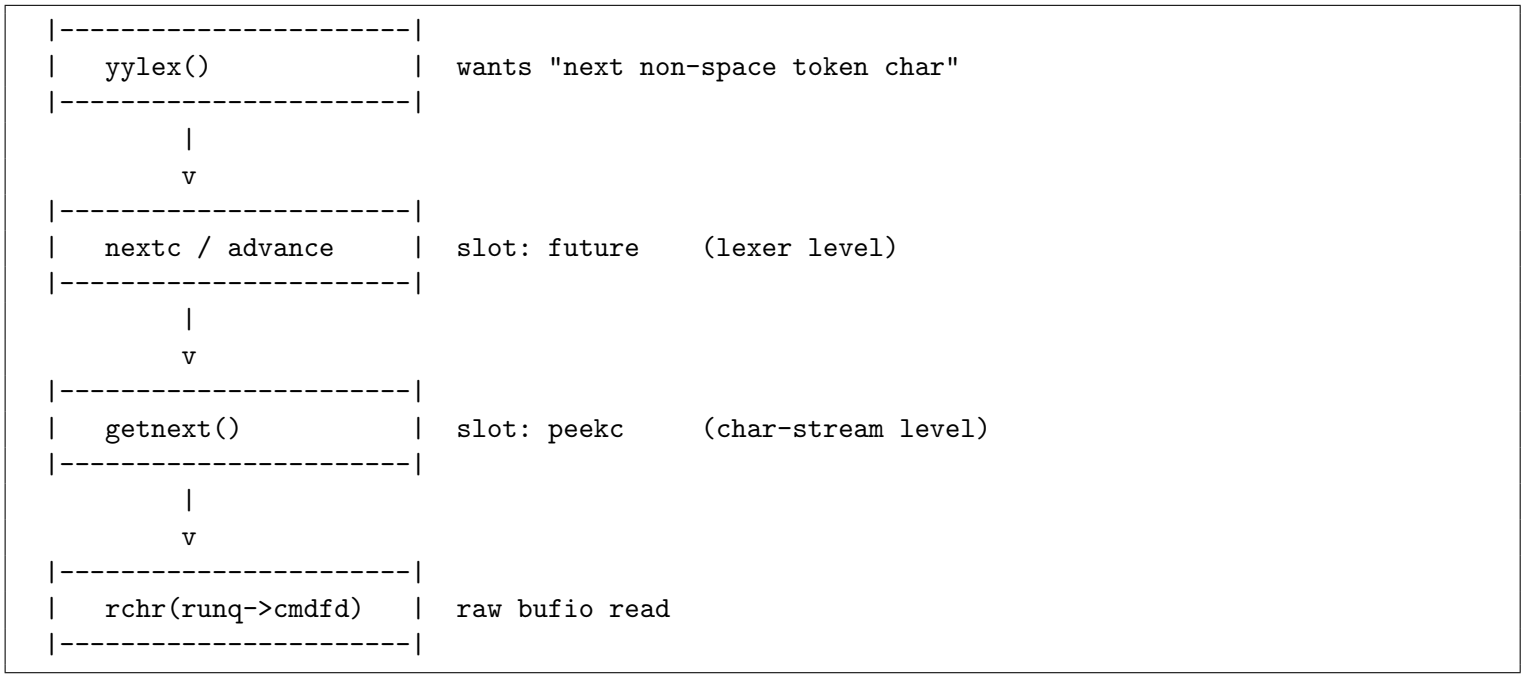
Uses `EOF` 50a, `future` 52e, and `nextc()` 53a.

⟨*function* `nextis` 53c⟩≡ (228)

```
/// yylex -> <>
bool
nextis(int c)
{
    if(nextc()==c){
        advance();
        return true;
    }
    return false;
}
```

Uses `advance()` 53b and `nextc()` 53a.

`rc` actually has two separate one-character lookahead slots, each living at a different layer:



`peekc` is internal to `getnext()`[49] and exists only to handle the backslash-escape case: when `getnext()` reads a `'\\'` and then finds that the next byte is not a newline, it has already consumed a byte it cannot "return," so it stashes it in `peekc` for the next call. `future`, by contrast, is the lookahead for the lexer proper—it lets `yylex()`[55a] probe the next character without committing to consume it, which is how `&` / `&&` and `|` / `||` disambiguation works. The two slots never interact: `future` is always filled by a fresh `getnext()` call, and `peekc` is an invisible fast-path inside that call.

Chapter 6

Lexing

The lexer of `rc` is hand-written (not generated by `lex`), which gives more control over subtle interactions with the rest of the shell, at the cost of some complexity. The main function is `yylex()`[55a], called by the `yacc`-generated parser whenever it needs the next token.

6.1 `yylex()`

The structure of `yylex()`[55a] is a switch on the next character. After skipping whitespace, it dispatches on the character read: operators and special characters are handled as individual cases, and anything else is treated as a word character.

⟨*function* `yylex` 55a⟩≡ (234a)
```
//@Scheck: called from yyparse()
/// yyparse -> <>
int yylex(void)
{
    int c;
    ⟨yylex() other locals 58d⟩

    ⟨yylex() before read, hack for SUB 64c⟩
    ⟨yylex() initialisations 58b⟩
    skipwhite();

    switch(c = advance()){
    ⟨yylex() switch c cases 56b⟩
    }
    // else
    ⟨yylex() if c is not a word character 62b⟩
    // else
    ⟨yylex() if c is a word character 62c⟩
}
```
Uses `advance()` 53b and `skipwhite()` 57a.

`wordchr()`[55b] and `idchr()`[56a] below classify input bytes. A *word* character is anything that is not whitespace, a shell metacharacter (`#;&|^$=‘’()<>`), or `EOF`—this is what the lexer gathers into a bare word such as a command name or a filename. An *identifier* character is the stricter set allowed in a variable name after `$`; note that `/` is a word character (filenames contain slashes) but not an identifier character. Both tests are byte-oriented yet UTF-8 safe: every byte of a multibyte rune is $\geq$ `0x80` while the listed metacharacters are all ASCII, so such a byte always fails the `strchr()` test and stays part of the word.

⟨*function* `wordchr` 55b⟩≡ (234a)
```
int
wordchr(int c)
```

```
{
    return !strchr("\n \t#;&|^$=‘’{}()<>", c) && c!=EOF;
}
```

Uses EOF 50a.

⟨*function* idchr 56a⟩≡ (234a)

```
int
idchr(int c)
{
    /*
     * Formerly:
     * return 'a'<=c && c<='z' || 'A'<=c && c<='Z' || '0'<=c && c<='9'
     * || c=='_' || c=='*';
     */
    return c > ' ' && !strchr("!\"#$%&'()+,-./:;<=>?@[\\]^‘{|}~", c);
}
```

The body of `yylex()` is one big `switch` on the first character of the token; we start with the simplest case, `EOF`, and add the cases for quotes, operators, and words gradually through this chapter. The `EOF` case records the token name in `tok`, the buffer in which the lexer accumulates the text of the current token.

⟨`yylex()` *switch c cases* 56b⟩≡ (55a) 58g▷

```
case EOF:
    lastdol = false;
    strcpy(tok, "EOF");
    return EOF;
```

Uses EOF 50a, lastdol 60b, and tok 56c.

⟨*global* tok 56c⟩≡ (234a)

```
char tok[NTOK + UTFmax];
```

Uses NTOK-10 56d.

⟨*constant* NTOK 56d⟩≡ (234a)

```
#define NTOK 8192  /* maximum bytes in a word (token) */
```

`tok` is sized `NTOK + UTFmax` rather than `NTOK` so that appending a final multibyte rune when the word is already near the `NTOK` limit cannot overflow it—`UTFmax` is the largest number of bytes a single rune can occupy.

There are a few lexing subtleties worth noting as you will see soon: newlines are sometimes skipped (via `skipnl()`[57b]) after certain operators to support multi-line commands; keywords like `if` and `for` are recognized as keywords only in command position (not as arguments); and the `(` character is handled specially depending on whether it follows a `$` (array subscript) or not (grouping/function definition).

6.2 Spaces and comments (’#’)

`skipwhite()`[57a] skips spaces, tabs, and comments. In `rc`, comments start with `#` and extend to the end of the line. Note that the `incomm`[56e] global is needed by `getnext()`[49]: inside a comment, a backslash-newline should *not* be treated as a line continuation.

⟨*global* incomm 56e⟩≡ (228)

```
bool incomm;
```

⟨*function* `skipwhite` 57a⟩≡ (234a)

```
/// yylex -> | skipnl <>
void
skipwhite(void)
{
    int c;
    for(;;){
        c = nextc();
        /* Why did this used to be  if(!inquote && c=='#') ?? */
        if(c=='#'){
            incomm = true;
            for(;;){
                c = nextc();
                if(c=='\n' || c==EOF) {
                    incomm = false;
                    break;
                }
                // else
                advance();
            }
        }
        if(c==' ' || c=='\t')
            advance();
        else
            return;
    }
}
```

Uses `EOF` 50a, `advance()` 53b, `incomm` 56e, and `nextc()` 53a.

6.3 Newlines

There is no dedicated `case '\n'` in the `yylex()`[55a] switch: a newline is not a word character, so it falls through to the generic "not a word character" path shown later in this chapter, which returns the character as a single-byte token—the `'\n'` the grammar treats as a command terminator. Lone metacharacters like `;` are returned the same way.

In `rc`, a newline normally acts as a command terminator (like `;` in C). However, after certain operators like `|`, `&&`, or `||`, a newline should be ignored because the user is clearly continuing the command on the next line. `skipnl()`[57b] is called by the lexer after those operators to consume any whitespace and newlines until meaningful input arrives. This is why you can write things like:

```
ls |
wc -l
```

and have it work as expected.

⟨*function* `skipnl` 57b⟩≡ (234a)

```
/// yylex | yyparse -> <>
void
skipnl(void)
{
    int c;
    for(;;){
        skipwhite();
        c = nextc();
        if(c!='\n')
            return;
```

```
        // else consume the newline and continue
        advance();
    }
}
```

Uses advance() 53b, nextc() 53a, and skipwhite() 57a.

6.4 Quoted strings ('...')

In rc, strings are quoted with single quotes. To include a literal single quote inside a quoted string, you double it: 'it''s' produces it's. The inquote[58a] flag is needed by getnext()[49] because inside a quoted string, backslash-newline should *not* be treated as a line continuation. The lastword[58c] flag tracks whether the previous token was a word, which the grammar uses to decide whether adjacent tokens should be concatenated.

⟨*global* inquote 58a⟩≡ (228)
```
bool inquote;
```

⟨yylex() *initialisations* 58b⟩≡ (55a) 58f▷
```
inquote = false;
```
Uses inquote 58a.

⟨*global* lastword 58c⟩≡ (234a)
```
// used also by syn.y
bool lastword; /* was the last token read a word or compound word terminator? */
```

⟨yylex() *other locals* 58d⟩≡ (55a) 58e▷
```
char *w = tok;
```
Uses tok 56c.

w is the write cursor into tok[56c]: as the lexer reads the bytes of a word it appends them (via addutf()[149e]) and advances w, so w always points just past the last byte stored. The rest of yylex()[55a] relies on it to build up the token text before terminating it with a '\0' and wrapping tok in a WORD node.

⟨yylex() *other locals* 58e⟩+≡ (55a) ◁58d 64b▷
```
struct Tree *t;
```
Uses Tree.

The lexer does not really parse; it just packages each word as a *leaf* of the AST. The semantic value yacc passes around (yylval.tree) is a Tree[31a], so when yylex() recognizes a word it wraps it in a single WORD node (t) and returns that. yyparse()[65b] still does the real work of combining these leaves into larger trees; having the lexer pre-build the leaves merely avoids a separate token-to-node conversion step.

⟨yylex() *initialisations* 58f⟩+≡ (55a) ◁58b
```
yylval.tree = nil;
```
Uses yylval.

With tok, the write cursor w, and the quoting flags in place, we can finally show the lexer code that scans a single-quoted string.

⟨yylex() *switch c cases* 58g⟩+≡ (55a) ◁56b 59c▷
```
case '\'':
    inquote = true;
    lastword = true;
    lastdol = false;
    for(;;){
        c = advance();
        if(c==EOF)
            break;
        if(c=='\''){
```

```
            if(nextc()!='\'')
                break;
            // else
            advance();
        }
        w = addutf(w, c);
    }
    if(w != nil)
        *w='\0';

    t = token(tok, WORD);
    t->quoted = true;

    yylval.tree = t;
    return t->type; // WORD
```

Uses EOF 50a, WORD, addutf() 149e, advance() 53b, inquote 58a, lastdol 60b, lastword 58c, nextc() 53a, tok 56c, token() 59b, and yylval.

⟨Tree *word specific fields* 59a⟩≡ (31a)
```
bool quoted;
```

⟨*function* token 59b⟩≡ (228)
```
tree*
token(char *str, int type)
{
    tree *t = newtree();

    t->type = type;
    t->str = strdup(str);
    return t;
}
```

Uses newtree() 32a.

6.5 Operators ('&', '&&', '|', '||', '<', '>', '$', ...)

Each operator is handled as a case in the main switch of yylex()[55a]. For multi-character operators like && or ||, the lexer uses nextis()[53c] to check the second character. Note that after binary operators like &&, ||, and |, the lexer calls skipnl()[57b] so that newlines are treated as whitespace, allowing the command to continue on the next line. The lastdol flag tracks whether the previous token was $, which affects how (is lexed (see Section 6.8).

⟨yylex() *switch c cases* 59c⟩+≡ (55a) ◁58g 60a▷
```
case '&':
    lastdol = false;
    if(nextis('&')){
        skipnl();
        strcpy(tok, "&&");
        return ANDAND;
    }
    // else
    strcpy(tok, "&");
    return '&';
```

Uses ANDAND, lastdol 60b, nextis() 53c, skipnl() 57b, and tok 56c.

⟨yylex() *switch c cases* 60a⟩+≡ (55a) ◁59c 60c▷
```
case '$':
    lastdol = true;
    if(nextis('#')){
        strcpy(tok, "$#");
        return COUNT;
    }
    // else
    if(nextis('"')){
        strcpy(tok, "$\"");
        return '"';
    }
    // else
    strcpy(tok, "$");
    return '$';
```
Uses COUNT, lastdol 60b, nextis() 53c, and tok 56c.

⟨*global* lastdol 60b⟩≡ (234a)
```
bool lastdol; /* was the last token read '$' or '$#' or '"'? */
```

⟨yylex() *switch c cases* 60c⟩+≡ (55a) ◁60a
```
case '|':
    lastdol = false;
    if(nextis('|')){
        skipnl();
        strcpy(tok, "||");
        return OROR;
    }
    // else FALLTHROUGH
case '<':
case '>':
    lastdol = false;
    ⟨yylex() in switch when redirection character 60d⟩
```
Uses OROR, lastdol 60b, nextis() 53c, skipnl() 57b, and tok 56c.

Redirections and pipes are the most complex part of the lexer. Unlike simple operators like && or || which map directly to a token, redirections carry extra information: the type (WRITE, READ, APPEND, HERE), and which file descriptors are involved (fd0, fd1). The lexer builds the Tree node right here, rather than in the parser, for a lexical reason rather than mere convenience. Inside a redirection like >[2=1] the 2 and 1 are file descriptors, but everywhere else a digit is an ordinary word character that wordchr()[55b] folds into a word. Emitting the digits and brackets as separate grammar tokens would clash with that normal word lexing, so rc recognizes the whole bracketed redirection here instead; the comment below sketches the little grammar it accepts.

⟨yylex() *in switch when redirection character* 60d⟩≡ (60c)
```
/*
 * funny redirection tokens:
 * redir: arrow | arrow '[' fd ']'
 * arrow: '<' | '<<' | '>' | '>>' | '|'
 * fd: digit | digit '=' | digit '=' digit
 * digit: '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
 * some possibilities are nonsensical and get a message.
 */
*w++=c;
t = newtree();
switch(c){
⟨yylex() in switch when redirection character, switch c cases 61e⟩
}
⟨yylex() in switch when redirection character, if bracket after 162b⟩
*w='\0';
```

```
yylval.tree = t;
if(t->type==PIPE)
    skipnl();
return t->type;
```

Uses PIPE, newtree() 32a, skipnl() 57b, and yylval.

⟨Tree *redirection and pipe specific fields* 61a⟩≡ (31a)

```
//enum<Redirection_kind>
int rtype;

// For a pipe, fd0 is the left fd of the pipe, and fd1 the right fd.
// For a redirection, fd0 is what we redirect (stdout for >, stdin for <)
// and fd1 is what we possibly redirect to (when DUP).
fdt fd0;
fdt fd1; /* details of REDIR PIPE DUP tokens */
```

WRITE, READ, and APPEND are the tags stored in a redirection node's rtype field to record which kind it is: > (truncate and write), < (read), and >> (append). The actual numbers are arbitrary, only their distinctness matters

⟨*constant* WRITE 61b⟩≡ (223b)

```
#define WRITE 2
```

⟨*constant* READ 61c⟩≡ (223b)

```
#define READ 3
```

⟨*constant* APPEND 61d⟩≡ (223b)

```
#define APPEND 1
```

Each redirection character is handled as a case below. For a pipe, fd0 defaults to 1 (stdout of the left command) and fd1 to 0 (stdin of the right command), which is the standard cmd1 | cmd2 behavior. For >, fd0 defaults to 1 (redirect stdout); for <, fd0 defaults to 0 (redirect stdin). A double >> sets the type to APPEND. These defaults can be overridden with the bracket syntax (e.g., >[2] to redirect stderr); see Section 14.4.7.

⟨yylex() *in switch when redirection character, switch c cases* 61e⟩≡ (60d) 61f▷

```
case '|':
    t->type = PIPE;
    t->fd0 = STDOUT; // left fd of pipe (stdout of left cmd)
    t->fd1 = STDIN; // right fd of pipe (stdin of right cmd)
    break;
```

Uses PIPE.

⟨yylex() *in switch when redirection character, switch c cases* 61f⟩+≡ (60d) ◁61e 62a▷

```
case '>':
    t->type = REDIR;
    if(nextis('>')){
        t->rtype = APPEND;
        *w++=c;
    }
    else
        t->rtype = WRITE;
    t->fd0 = STDOUT;
    break;
```

Uses APPEND 61d, REDIR, WRITE 61b, and nextis() 53c.

⟨yylex() *in switch when redirection character, switch c cases* 62a⟩+≡ (60d) ◁61f

```
case '<':
    t->type = REDIR;
    ⟨yylex() in switch when redirection character, if here document 157b⟩
    ⟨yylex() in switch when redirection character, if read/write redirect 161c⟩
    else
        t->rtype = READ;
    t->fd0 = STDIN;
    break;
```

Uses READ 61c and REDIR.

6.6 Remaining non-word characters

Any character that is not a word character and was not handled by an earlier `switch` case ends up here: the lexer simply returns it as a single-character token. This is the path the newline `'\n'` falls through to (as noted earlier), and it likewise covers lone metacharacters such as `;` and the closing brace and parenthesis that the grammar consumes directly.

⟨yylex() *if c is not a word character* 62b⟩≡ (55a)

```
if(!wordchr(c)){
    lastdol = false;
    tok[0] = c;
    tok[1]='\0';
    return c;
}
```

Uses lastdol 60b, tok 56c, and wordchr() 55b.

6.7 Keywords and identifiers (`if`, `for`, `while`, `switch`, `fn`, ...)

When the lexer encounters a word character (anything not an operator or whitespace), it reads characters until a non-word character appears. After a `$`, only identifier characters (`idchr`) are accepted, which excludes slashes and other path characters.

⟨yylex() *if c is a word character* 62c⟩≡ (55a)

```
for(;;){
    ⟨yylex() when c is a word character, if glob character 133a⟩
    w = addutf(w, c);

    c = nextc();
    if(lastdol ? !idchr(c) : !wordchr(c))
        break;
    advance();
}

lastword = true;
lastdol = false;
if(w!=nil)
    *w='\0';

t = klook(tok);
if(t->type != WORD)
    lastword = false;

t->quoted = false;
```

```
  yylval.tree = t;
  return t->type;
```

Uses WORD, addutf() 149e, advance() 53b, idchr() 56a, klook() 63a, lastdol 60b, lastword 58c, nextc() 53a, tok 56c, wordchr() 55b, and yylval.

Once the word is assembled, `klook()`[63a] checks whether it matches a keyword. Keywords are stored in a small hash table initialized by `kinit()`[63e] at startup. If the word is a keyword, `klook()` returns the appropriate token type; otherwise it returns WORD.

⟨*function* klook 63a⟩≡ (228)

```
  tree*
  klook(char *name)
  {
      struct Kw *p;
      tree *t = token(name, WORD); // default if actually not a keyword

      for(p = kw[hash(name, NKW)];p;p = p->next)
          if(strcmp(p->name, name)==0){
              t->type = p->type;
              break;
          }
      return t;
  }
```

Uses Kw 63b, NKW-8 63d, WORD, hash() 40a, kw 63c, and token() 59b.

⟨*struct* Kw 63b⟩≡ (228)

```
  struct Kw {
      char *name;
      int type;

      struct Kw *next;
  };
```

Uses Kw 63b.

⟨*global* kw 63c⟩≡ (228)

```
  struct Kw *kw[NKW];
```

Uses Kw 63b and NKW-8 63d.

⟨*constant* NKW 63d⟩≡ (228)

```
  #define NKW 30
```

⟨*function* kinit 63e⟩≡ (228)

```
  void
  kinit(void)
  {
      kenter(FOR, "for");
      kenter(IN, "in");
      kenter(WHILE, "while");
      kenter(IF, "if");
      kenter(NOT, "not");
      kenter(SWITCH, "switch");
      kenter(FN, "fn");

      kenter(TWIDDLE, "~");
      kenter(BANG, "!");
      kenter(SUBSHELL, "@");
  }
```

Uses BANG, FN, FOR, IF, IN, NOT, SUBSHELL, SWITCH, TWIDDLE, WHILE, and kenter() 64a.

⟨*function* `kenter` 64a⟩≡ (228)

```
void
kenter(int type, char *name)
{
    int h = hash(name, NKW);
    struct Kw *p = new(struct Kw);
    p->type = type;
    p->name = name;
    p->next = kw[h];
    kw[h] = p;
}
```

Uses `Kw` 63b, `NKW-8` 63d, `hash()` 40a, and `kw` 63c.

6.8 Array subscript (<arr>(<n>))

This is one of the trickiest parts of the lexer. When a word is immediately followed by `(`, as in `$x(1)`, it must be treated as an array subscript (`SUB`) rather than a regular parenthesis. Additionally, when two words are adjacent without whitespace (e.g., `-x$flag`), the lexer inserts an implicit `^` (caret, the concatenation operator) between them. Without this, the lexer would produce three separate tokens and there would be no way to distinguish `-x$flag` (one argument) from `-x $flag` (two arguments).

⟨`yylex()` *other locals* 64b⟩+≡ (55a) ◁58e

```
int d = nextc();
```

Uses `nextc()` 53a.

⟨`yylex()` *before read, hack for SUB* 64c⟩≡ (55a)

```
/*
 * Embarassing sneakiness:  if the last token read was a quoted or unquoted
 * WORD then we alter the meaning of what follows.  If the next character
 * is '(', we return SUB (a subscript paren) and consume the '('. Otherwise,
 * if the next character is the first character of a simple or compound word,
 * we insert a '^' before it.
 */
if(lastword){
    lastword = false;
    if(d=='('){
        advance();
        strcpy(tok, "( [SUB]");
        return SUB;
    }
    // else
    if(wordchr(d) || d=='\'' || d=='`' || d=='$' || d=='"'){
        strcpy(tok, "^");
        return '^';
    }
    // else fallthrough
}
```

Uses `SUB`, `advance()` 53b, `lastword` 58c, `tok` 56c, and `wordchr()` 55b.

Chapter 7

Parsing

Now that we have seen how the lexer produces tokens from the input stream, we can describe the parser, which assembles those tokens into an AST.

7.1 Overview

The parser of `rc` is written using `yacc` (see CompilerGenerator book [Pad16a] for an introduction to `yacc`). The `.y` file follows the standard `yacc` layout: a prologue section (`%{...%}`) with C includes, then declarations (`%union`, `%token`, `%left`, `%type`), and finally the grammar rules after `%%`. The `%union` declaration tells `yacc` that semantic values (`yylval`, `$1`, `$2`, etc.) are `Tree` pointers, since every grammar rule builds or passes along AST nodes.

⟨*rc/syn.y* 65a⟩≡
```
%{
#include "rc.h"
#include "fns.h"
%}

%union {
 struct Tree *tree;
};

⟨token declarations 30⟩
⟨priority and associativity declarations 65c⟩
⟨type declarations 65d⟩

%%
⟨grammar 66a⟩
```

⟨*function* `yyparse` 65b⟩≡
```
... generated code from syn.y by yacc ...
```

⟨*priority and associativity declarations* 65c⟩≡ (65a)
```
/* operator priorities -- lowest first */
%left IF WHILE FOR SWITCH ')' NOT
%left ANDAND ORORR
%left BANG SUBSHELL
%left PIPE
%left '^'
%right '$' COUNT '"'
%left SUB
```

⟨*type* `declarations` 65d⟩≡ (65a) ?? ▷
```
%type<tree> line cmd simple word comword
```

The grammar is quite compact, which reflects the simplicity of the `rc` language. The top-level rule `rc` parses a single line: either an empty line (EOF), or a `line` followed by a newline. When a complete line is parsed, the action calls `compile()`[75g] to transform the AST into bytecodes. Note that newlines have a semantic role in `rc`: they terminate commands (like `;` in C). This is what makes the shell convenient for interactive use — you do not need an explicit semicolon to run a command.

⟨*grammar* 66a⟩≡ (65a)
```
rc:
    /*empty*/       { return ERROR_1;}
|   line '\n'       { return !compile($1);}
```

⟨*line rule* 66b⟩
⟨*cmd rule* 66c⟩
⟨*simple rule* 66d⟩

⟨*word rule* 68c⟩
⟨*comword rule* 68b⟩

⟨*other rules* 68a⟩

The following sections will gradually introduce the remaining rules: `line`, `cmd`, `word`, etc.

7.2 Simple commands (<cmd> <arg1>...<argn>)

The grammar is layered: a `line` is one or more `cmd`s separated by `;` or `&`, a `cmd` is a `simple` command (or a control flow statement), and a `simple` command is a `first` word followed by zero or more `word` arguments. The distinction between `first` and `word` is important: a `first` cannot be a keyword, so that `if` in command position is recognized as a keyword, but `echo if` treats `if` as a regular word argument.

⟨*line rule* 66b⟩≡ (66a)
```
line:
    cmd
```
⟨*line rule other cases* 69a⟩

⟨*cmd rule* 66c⟩≡ (66a)
```
cmd:
    /*empty*/           {$$=nil;}
|   simple              {$$=simplemung($1);}
```
⟨*cmd rule other cases* 69c⟩

⟨*simple rule* 66d⟩≡ (66a)
```
simple:
    first
|   simple word         {$$=tree2(ARGLIST, $1, $2);}
```
⟨*simple rule other cases* 70a⟩

Here is what the AST looks like for `ls -l /tmp`. `simple` is left-recursive, so each additional argument wraps the previous one in an `ARGLIST` node; the `simplemung()`[67a] post-pass then hoists the whole thing under a `SIMPLE` node:

```
after parsing simple:          after simplemung():

        ARGLIST                          SIMPLE
         /   \                             |
    ARGLIST  WORD("/tmp")                 ARGLIST
     /   \                                /   \
   WORD  WORD("-l")                  ARGLIST  WORD("/tmp")
    |                                  /   \
"ls"                                WORD  WORD("-l")
                                      |
                                    "ls"
```

⟨*function* `simplemung` 67a⟩≡ (233b)

```
/*
 * Add a SIMPLE node at the root of t and percolate all the redirections
 * up to the root.
 */
//@Scheck: used by syn.y
tree* simplemung(tree *t)
{
    tree *u;
    struct Io *s;

    t = tree1(SIMPLE, t);

    s = openstr();
    pfmt(s, "%t", t);
    t->str = strdup((char *)s->strp);
    closeio(s);

    ⟨simplemung() percolate redirections up to the root 67b⟩
    return t;
}
```

Uses `Io` 186f, `SIMPLE`, `closeio()` 189a, `openstr()` 189b, `pfmt()` 190b, and `tree1()` 32c.

`simplemung()` wraps a simple command in a `SIMPLE` node and also stores a string representation of the whole command in `Tree.str`. This string is used later to store function definitions in the environment (see Chapter 10). The `"%t"` here is not a standard `printf` verb: `rc`'s own `pfmt()`[190b] recognizes `%t` as "format a `Tree`[31a]" and dispatches to the AST printer `pcmd()`[173b] (Appendix A). So this call renders the whole command subtree back into its textual form, which is what gets stored in `t->str`.

`simplemung()` also "percolates" redirections to the root of the subtree, so that the bytecode generator can handle them before executing the command.

⟨`simplemung()` *percolate redirections up to the root* 67b⟩≡ (67a)

```
for(u = t->child[0]; u->type==ARGLIST; u = u->child[0]){
    if(u->child[1]->type==REDIR || u->child[1]->type==DUP){
        u->child[1]->child[1] = t;
        t = u->child[1];
        u->child[1] = nil;
    }
}
```

Uses `ARGLIST`, `DUP`, and `REDIR`.

For example, when parsing `grep pattern < input > output`, the parser first builds a flat `ARGLIST` tree with the redirections mixed among the arguments. Then `simplemung()` percolates the redirections up to the root, producing an AST where redirections wrap the command:

```
 Before simplemung():            After simplemung():

 SIMPLE                          > "output" fd0=1
    |                              |
  ARGLIST                         < "input" fd0=0
  /     \                          |
ARGLIST  > "output"               SIMPLE
  /     \                          |
ARGLIST   < "input"               ARGLIST
  /      \                         /     \
 "grep"  "pattern"               "grep"  "pattern"
```

This way, the bytecode generator encounters the redirections first and can set them up before executing the command.

The rules below cover the remaining pieces of a simple command. `first` is the command name: it is deliberately *not* allowed to be a keyword, so an `if` or `for` at the start of a line is read as a control structure, not a command. The argument rule `word` is more permissive—it accepts a `keyword` and immediately demotes it to a plain `WORD`—which is what lets you pass keyword-looking arguments to ordinary commands, as in `grep for *.c`, without the parser mistaking them for control structures.

⟨*other rules* 68a⟩≡ (66a) 68d▷
```
first:
    comword
|   first '^' word      {$$=tree2('^', $1, $3);}
```

⟨*comword rule* 68b⟩≡ (66a)
```
comword:
    WORD
⟨comword rule other cases 72g⟩
```

⟨*word rule* 68c⟩≡ (66a)
```
word:
    comword
|   word '^' word       {$$=tree2('^', $1, $3);}
|   keyword             {lastword=true; $1->type=WORD;}
```

⟨*other rules* 68d⟩+≡ (66a) ◁68a 68e▷
```
keyword: FOR|IN|WHILE|IF|NOT|TWIDDLE|BANG|SUBSHELL|SWITCH|FN
```

⟨*other rules* 68e⟩+≡ (66a) ◁68d 69b▷
```
words:
    /*empty*/       {$$=(struct Tree*)nil;}
|   words word      {$$=tree2(WORDS, $1, $2);}
```

7.3 Operators

Operators are spread across different grammar levels, which implicitly defines their priorities. Sequencing (`;` and `&`) is at the `line` level (lowest priority), logical operators (`&&`, `||`) and pipes (`|`) are at the `cmd` level, and redirections are at the `simple` level (highest priority, closest to the command words).

On top of these structural levels, the grammar's preamble also declares explicit `yacc` precedences with `%left` and `%right` (for instance `%left PIPE` sits below `%left ANDAND OROR`, so pipes bind tighter than the logical operators). These resolve the shift/reduce ambiguities the level structure alone leaves open; the `%prec BANG` used for assignments later in this chapter is the same mechanism.

7.3.1 Sequences (';', '&')

The ; operator sequences two commands. The & operator runs a command asynchronously (in the background) and is treated as a unary postfix operator: `x & y` is parsed as `(x&); y`.

⟨*line rule other cases* 69a⟩≡ (66b)
```
|   cmdsa line      {$$=tree2(';', $1, $2);}
```

⟨*other rules* 69b⟩+≡ (66a) ◁68e 70b▷
```
cmdsa:
    cmd ';'
|   cmd '&'         {$$=tree1('&', $1);}
```

7.3.2 Logical operators ('&&', '||', '!')

⟨*cmd rule other cases* 69c⟩≡ (66c) 69d▷
```
|   cmd ANDAND cmd      {$$=tree2(ANDAND, $1, $3);}
|   cmd OROR cmd        {$$=tree2(OROR, $1, $3);}
```

⟨*cmd rule other cases* 69d⟩+≡ (66c) ◁69c 69f▷
```
|   BANG cmd            {$$=mung1($1, $2);}
```

⟨*function* mung1 69e⟩≡ (233b)
```
//@Scheck: used by syn.y
tree* mung1(tree *t, tree *c0)
{
    t->child[0] = c0;
    return t;
}
```

7.3.3 String matching ('~')

The ~ (twiddle) operator is the fundamental comparison operator in rc. Since all values in rc are (list of) strings, there are no arithmetic comparisons; instead, ~ performs pattern matching: `~ $x foo*` tests whether $x matches the pattern foo*[1]. It can also be used for simple equality tests (without wildcards) and even for counting: `~ $#list 0` tests if a list is empty. This is a major simplification compared to the Bourne shell, which relies on the external test program (or [ ]). rc still provides test (Appendix E gives its code) for the file tests ~ cannot do, such as `test -f $file`.

⟨*cmd rule other cases* 69f⟩+≡ (66c) ◁69d 69h▷
```
|   TWIDDLE word words  {$$=mung2($1, $2, $3);}
```

⟨*function* mung2 69g⟩≡ (233b)
```
//@Scheck: used by syn.y
tree* mung2(tree *t, tree *c0, tree *c1)
{
    t->child[0] = c0;
    t->child[1] = c1;
    return t;
}
```

7.3.4 Pipe ('|')

⟨*cmd rule other cases* 69h⟩+≡ (66c) ◁69f 70c▷
```
|   cmd PIPE cmd        {$$=mung2($2, $1, $3);}
```

[1] The * here is interpreted by the twiddle operator itself using match()[138b] not by the shell's globbing mechanism which expands filenames (see Chapter 12).

You can see below the AST for the `ls -l | wc -l` command: the `PIPE` binds two `SIMPLE` subtrees, and at the top a `cmd` rule would normally wrap the whole thing. Note that the `PIPE` carries the `fd0`/`fd1` pair (defaults 1/0) on the tree node itself, which is why `Tree` has redirection-specific fields—the lexer emitted a `PIPE` tree already, and the parser just hangs (with `mung2()`[69g]) the two children on it:

```
                PIPE (fd0=1, fd1=0)
               /    \
          SIMPLE      SIMPLE
            |           |
         ARGLIST      ARGLIST
         /    \       /    \
      WORD  WORD("-l")  WORD  WORD("-l")
        |                  |
       "ls"               "wc"
```

7.3.5 Redirections ('>', '<')

Redirections are at the `simple` level in the grammar, which means they bind tighter than pipes and logical operators. For example, `ls | grep bi* > foo` is parsed as `ls | (grep bi* > foo)`, not `(ls | grep bi*) > foo`—the redirection applies only to `grep`, not to the whole pipeline. If you want to redirect the output of an entire pipeline, you need braces: `{ls | grep bi*} > foo`. A command can have multiple redirections (e.g., `ls < foo > bar`), and they can appear mixed with arguments. The `simplemung()`[67a] function later percolates again them to the root of the AST subtree.

⟨*simple rule other cases* 70a⟩≡ (66d)
```
  |   simple redir        {$$=tree2(ARGLIST, $1, $2);}
```

⟨*other rules* 70b⟩+≡ (66a) ◁69b 70d▷
```
  redir:
      REDIR word      {$$=mung1($1, $1->rtype==HERE ? heredoc($2) : $2);}
  ⟨redir rule other cases 163c⟩
```

⟨*cmd rule other cases* 70c⟩+≡ (66c) ◁69h 70f▷
```
  |   brace epilog        {$$=epimung($1, $2);}
```

⟨*other rules* 70d⟩+≡ (66a) ◁70b 71e▷
```
  epilog:
      /*empty*/           {$$=nil;}
  |   redir epilog        {$$=mung2($1, $1->child[0], $2);}
```

⟨*function* `epimung` 70e⟩≡ (233b)
```
  //@Scheck: used by syn.y
  tree* epimung(tree *comp, tree *epi)
  {
      tree *p;
      if(epi==0)
          return comp;
      for(p = epi;p->child[1];p = p->child[1]);
      p->child[1] = comp;
      return epi;
  }
```

⟨*cmd rule other cases* 70f⟩+≡ (66c) ◁70c 71a▷
```
  |   redir cmd  %prec BANG
          {$$=mung2($1, $1->child[0], $2);}
```

7.4 Control flow statements (`if`, `if not`, `while`, `switch`, `for`)

Note that `rc` uses `if not` instead of `else`. This is because `rc` parses and executes one line at a time. When the user types `if(test) cmd` and presses Enter, `rc` must parse, compile, and execute this line immediately. The problem with `else` is that it would need to appear on the *next* line:

```
if(test) cmd1
else cmd2
```

But `rc` has already fully processed the first line before it even sees `else`; it cannot retroactively turn `if(test) cmd1` into the first half of an if/else in an AST. The Bourne shell solves this with `if...then...fi`: the explicit `fi` terminator tells the parser when the `if` block ends, so `else` can appear unambiguously between `then` and `fi`. But this requires more keywords (`then`, `fi`). `rc` takes a different approach: `if not` is a completely separate command that checks the exit status left by the previous `if`. No lookahead is needed, no extra keywords, and each line is self-contained.

⟨*cmd rule other cases* 71a⟩+≡ (66c) ◁70f 71b▷
```
|   IF paren {skipnl();} cmd  {$$=mung2($1, $2, $4);}
|   IF NOT   {skipnl();} cmd  {$$=mung1($2, $4);}
```

Like the lexer's `skipnl()`[57b] calls after `&&`, `||`, and `|`, these rules also swallow a newline so the body can sit on the next line—but here the call is a mid-rule grammar action (`skipnl();`) rather than lexer code. The difference is who knows that the newline should be ignored: for a binary operator the lexer can tell as soon as it scans the operator, whereas after `if(...)` or `while(...)` only the parser knows, once it has reduced the condition and is about to read the body.

⟨*cmd rule other cases* 71b⟩+≡ (66c) ◁71a 71c▷
```
|   WHILE paren {skipnl();} cmd    {$$=mung2($1, $2, $4);}
|   SWITCH word {skipnl();} brace  {$$=tree2(SWITCH, $2, $4);}
```

The `for` loop has two forms: `for(x in a b c)` iterates over the explicit list, and `for(x)` iterates over `$*` (the script arguments).

⟨*cmd rule other cases* 71c⟩+≡ (66c) ◁71b 72d▷
```
  /*
   * if ``words'' is nil, we need a tree element to distinguish between
   * for(i in ) and for(i), the former being a loop over the empty set
   * and the latter being the implicit argument loop.  so if $5 is nil
   * (the empty set), we represent it as "()".  don't parenthesize non-nil
   * functions, to avoid growing parentheses every time we reread the
   * definition.
   */
|   FOR '(' word IN words ')' {skipnl();} cmd
    {$$=mung3($1, $3,    $5 ? $5 : tree1(PAREN, $5), $8);}

|   FOR '(' word ')' {skipnl();} cmd
    {$$=mung3($1, $3, (struct Tree *)0, $6);}
```

⟨*function* mung3 71d⟩≡ (233b)
```
//@Scheck: used by syn.y
tree* mung3(tree *t, tree *c0, tree *c1, tree *c2)
{
    t->child[0] = c0;
    t->child[1] = c1;
    t->child[2] = c2;
    return t;
}
```

⟨*other rules* 71e⟩+≡ (66a) ◁70d 72b▷
```
paren:  '(' body ')'        {$$=tree1(PCMD, $2);}
```

⟨*token declarations* 72a⟩+≡ (65a) ◁31b 153e▷
```
%token PCMD
```

⟨*other rules* 72b⟩+≡ (66a) ◁71e 72c▷
```
brace:  '{' body '}'        {$$=tree1(BRACE, $2);}
```

⟨*other rules* 72c⟩+≡ (66a) ◁72b 72f▷
```
body:
    cmd
|   cmdsan body     {$$=tree2(';', $1, $2);}

cmdsan:
    cmdsa
|   cmd '\n'
```

7.5 Functions (`fn`)

Function definition in `rc` is written as `fn name {body}`. The second form, `fn name` without a body, *deletes* the function definition. Note that `words` (not just a single word) is accepted below as the function name: this allows defining a function for multiple names at once (e.g., `fn sigint sighup {...}` to handle multiple signals with the same handler).

⟨*cmd rule other cases* 72d⟩+≡ (66c) ◁71c 72e▷
```
|   FN words brace  {$$=tree2(FN, $2, $3);}
|   FN words        {$$=tree1(FN, $2);}
```

7.6 Variables (`<x>=...`)

Variable assignment in `rc` is written `x=value`. The interesting aspect is that an assignment can be followed by a command: `x=value cmd` makes the assignment local to that command only. When `cmd` is empty, the assignment is global. The assignment appears *before* the command, following the Bourne shell convention (which was itself motivated by programs like `make` that accept `x=y` on the command line).[2]

⟨*cmd rule other cases* 72e⟩+≡ (66c) ◁72d 155b▷
```
|   assign cmd                    %prec BANG
      {$$=mung3($1, $1->child[0], $1->child[1], $2);}
```

The `%prec BANG` directive above tells `yacc` to resolve ambiguities (e.g., `x=1 ls && echo done`) using the priority of `BANG`, giving assignment lower priority than pipes but higher than logical operators.

⟨*other rules* 72f⟩+≡ (66a) ◁72c
```
assign: first '=' word      {$$=tree2('=', $1, $3);}
```

A `comword` is the smallest standalone argument the grammar recognizes. Besides a bare `WORD`, it covers variable expansion (`$x`), the element count `$#x` (`COUNT`), and subscripting `$x(n)` (`SUB`). The last relies on a small lexer trick: a `(` normally begins a list, but when it comes right after a word the lexer (seeing its `lastword` flag set) returns the distinct `SUB` token instead and consumes the `(`, so `$x(1 2)` is parsed as a subscript rather than as `$x` next to a separate list.

⟨*comword rule other cases* 72g⟩≡ (68b) 73▷
```
|   '$' word        {$$=tree1('$', $2);}
|   COUNT word      {$$=tree1(COUNT, $2);}
|   '$' word SUB words ')'  {$$=tree2(SUB, $2, $4);}
```

[2]The two are not the same: `x=y mk` is a shell assignment, so `rc` sets `x` in the environment that `mk` inherits, whereas `mk x=y` passes `x=y` as an argument that `mk` itself parses as an `mkfile` variable assignment. Both end up defining `x` for the build—since `mk` also imports the environment—but by different paths.

7.7 Lists `(...)`

In `rc`, parentheses create lists: `(a b c)` is a list of three words. This is how you assign multiple values to a variable: `x=(a b c)`. Lists are first-class values in `rc`, which is one of its main improvements over the Bourne shell.

⟨*comword rule other cases* 73⟩+≡ (68b) ◁72g 151a▷

```
| '(' words ')'  {$$=tree1(PAREN, $2);}
```

This already completes the grammar of `rc` which has a very simple syntax. Note that there is no checking or type-checking pass: `rc` has no types—every value is a list of strings—so once `yyparse()`[65b] has built the AST it is handed straight to the bytecode generator, the subject of the next chapter.

Chapter 8

Bytecode Generation and Interpretation

I decided to merge the bytecode generation and interpretation in the same chapter because it helps to see the bytecodes generated from a given AST node together with the functions that interpret those bytecodes. We could have also grouped them with the related parsing grammar rules, but the parser can be understood in isolation.

8.1 Bytecode vs tree-walking interpretation

Why bytecode at all? Most shells—`bash`, `zsh`—execute commands by walking the AST node by node, firing a function for each node type. `rc` takes a different route: it compiles the AST to a stack-machine bytecode and runs the bytecodes through a dispatch loop. The compiled form lives in a `codebuf`[33b] of `code` cells, each holding either an opcode function pointer or an inline argument (a string or an integer). Running the program means advancing a `runq->pc` index through that buffer.

The argument for tree-walking is simplicity: no separate compile step, the parser hands its tree directly to the interpreter, and there is one less data structure to design (in fact some programs do not even bother to generate an AST). The argument for bytecode—which `rc` picks—is that tree-walking conflates control flow with tree shape, which gets awkward for things that do not match the tree (jumps out of loops, function returns, signal handlers that need to resume execution at a known point, sourcing one script from inside another via the `.` builtin). With bytecodes, all of those become PC manipulations on a single linear buffer, which is exactly the same model as a CPU running machine code. The `runq` thread structure that holds `pc` alongside its arguments and locals (Section 3.4) is the shell's equivalent of a userspace process control block, and its existence is what lets `rc` interleave multiple bytecode programs (the bootstrap, the user script, sourced files, function calls) without any explicit reentrancy in the C code. The point is where the execution state lives. A tree-walker keeps it in C local variables and the C call stack, so pausing one program to run another—a function body, a sourced file, the left side of a pipe—means re-entering the interpreter recursively and arranging non-local jumps for returns and errors. `rc` instead keeps all per-program state in a heap-allocated `runq Thread`, so the single dispatch loop in `main()`[42b] is never re-entered: starting a nested program allocates a new `Thread` and points `runq` at it (its `ret` field links back), and finishing restores the previous one. Switching execution contexts is a pointer assignment, not a recursive C call.

8.2 Overview

Recall the main flow: `main()`[42b] → `Xrdcmds()`[47a] → `yyparse()`[65b] → `compile()`[75g]. After `yyparse()` builds the AST, it calls `compile()` which walks the tree and emits bytecodes into `codebuf`[33b]. The bytecodes are then interpreted by the main loop in `main()`.

This overview introduces the machinery before the per-construct code generation that follows. We start with the `emitf()`[75d] family of helpers that append to `codebuf`, then `compile()`, which drives one line, and finally

the skeleton of `outcode()`, the recursive walker. The later sections fill in `outcode()`'s cases one construct at a time (simple commands, pipes, loops, ...).

8.2.1 Emitting bytecodes: `emitxxx()`

The emit functions append elements to `codebuf`[33b], growing it as needed. There are three variants: `emitf()`[75d] for function pointers (the bytecodes), `emiti()`[75c] for integers (jump offsets, file descriptors), and `emits()`[75e] for strings (inline arguments like the string after `Xword`[77a]). `codep`[75a] is the current write position in the buffer, and `ncode`[75b] is the allocated capacity.

⟨*global* `codep` 75a⟩≡ (236a)
```
// idx in codebuf
int codep;
```

⟨*global* `ncode` 75b⟩≡ (236a)
```
int ncode;
```

⟨*function* `emiti` 75c⟩≡ (236a)
```
#define emiti(x) ((codep!=ncode || morecode()), codebuf[codep].i = (x), codep++)
```

⟨*function* `emitf` 75d⟩≡ (236a)
```
#define emitf(x) ((codep!=ncode || morecode()), codebuf[codep].f = (x), codep++)
```

⟨*function* `emits` 75e⟩≡ (236a)
```
#define emits(x) ((codep!=ncode || morecode()), codebuf[codep].s = (x), codep++)
```

⟨*function* `morecode` 75f⟩≡ (236a)
```
//@Scheck: used by the macros above (why marked as dead then??? TODO)
int morecode(void)
{
    ncode+=100;
    codebuf = (code *)realloc((char *)codebuf, ncode*sizeof codebuf[0]);
    if(codebuf==nil)
        panic("Can't realloc %d bytes in morecode!", ncode*sizeof(code));
    return OK_0;
}
```

Uses `codebuf` 33b, `ncode` 75b, and `panic()` 180c.

8.2.2 Compiling a line: `compile()`

`compile()`[75g] allocates a fresh code buffer, then calls `outcode()`[76b] to recursively walk the AST and emit bytecodes. It appends `Xreturn()`[96c] at the end so that when the bytecodes finish executing, the interpreter returns to the previous thread. Note that `compile()` is called for *each line* entered interactively (or each line of a script). So each line gets its own bytecode sequence ending with `Xreturn()`.

⟨*function* `compile` 75g⟩≡ (236a)
```
//@Scheck: called from syn.y
/// yyparse -> <> -> outcode
error0 compile(tree *t)
{
    ncode = 100;
    codep = 0;
    codebuf = (code *)emalloc(ncode*sizeof(code));

    emiti(0);           /* reference count */
    outcode(t, flag['e'] ? true : false);

    ⟨compile() check nerror 76a⟩
```

```
    ⟨compile() after outcode and error management, read heredoc 158b⟩

    emitf(Xreturn);
    emitf(nil);

    return OK_1;
}
```

Uses Xreturn() 96c, codebuf 33b, codep 75a, emalloc() 181a, emitf-64 75d, emiti-65 75c, flag 43a, and ncode 75b.

The new allocation for codebuf above does not leak the previous line's buffer. compile() only fills the global codebuf as a handoff: the caller (Xrdcmds()[47a]) immediately passes it to start()[45b], which calls codecopy[34a] to take a counted reference (the emiti(0) above seeded the count at 0). That thread then *owns* the buffer and codefree()[34b] releases it when the thread finishes, so by the time the next line is compiled the old buffer already has its proper owner.

⟨compile() *check nerror* 76a⟩≡ (75g)

```
if(nerror){
    efree((char *)codebuf);
    return ERROR_0;
}
```

Uses codebuf 33b, efree() 181c, and nerror 179c.

8.2.3 Walking the AST: outcode()

outcode()[76b] is the main recursive function that walks the AST and emits bytecodes. It switches on the node type and emits the appropriate sequence of bytecodes for each construct. The eflag parameter propagates the -e flag (exit on error, see Section 14.2)): when set, an Xeflag[149d] bytecode is emitted after certain commands to check if they failed and exit accordingly.

⟨*function* outcode 76b⟩≡ (236a)

```
/// main -> Xrdcmds -> yyparse -> compile -> <>
void
outcode(tree *t, bool eflag)
{
    ⟨outcode() locals 87b⟩

    if(t==nil)
        return;

    ⟨outcode() set iflast before switch 101d⟩
    switch(t->type){
    ⟨outcode() cases 80d⟩
    default:
        pfmt(err, "bad type %d in outcode\n", t->type);
        break;
    }
    ⟨outcode() set iflast after switch 101c⟩
}
```

8.2.4 argv management

Before describing the compilation of specific AST nodes, let me present the bytecodes used to manage the argument stack. Most compilation schemes follow the same pattern: emit Xmark[77b] to start a new word list on

`argv`, emit a series of `Xword`[77a] to push words onto it, then emit a bytecode (like `Xsimple`[81]) that consumes the list. For example, `ls -l /tmp` compiles to: `Xmark`, `Xword "ls"`, `Xword "-l"`, `Xword "/tmp"`, `Xsimple`.

⟨*function* `Xword` 77a⟩≡ (230b)
```
void
Xword(void)
{
    pushword(runq->code[runq->pc++].s);
}
```
Uses `pushword()` 46b and `runq` 35a.

⟨*function* `Xmark` 77b⟩≡ (230b)
```
void
Xmark(void)
{
    pushlist();
}
```
Uses `pushlist()` 46a.

⟨*function* `poplist` 77c⟩≡ (230a)
```
void
poplist(void)
{
    list *p = runq->argv;
    if(p==nil)
        panic("poplist but no argv", 0);
    freelist(p->words);
    runq->argv = p->next;
    efree((char *)p);
}
```
Uses `efree()` 181c, `freelist()` 38b, `panic()` 180c, and `runq` 35a.

⟨*function* `popword` 77d⟩≡ (230a)
```
/// execexec -> <>
void
popword(void)
{
    word *p;
    ⟨popword() sanity check argv 77e⟩
    p = runq->argv->words;
    ⟨popword() sanity check argv words 77f⟩
    runq->argv->words = p->next;
    efree(p->word);
    efree((char *)p);
}
```
Uses `efree()` 181c and `runq` 35a.

⟨`popword()` *sanity check argv* 77e⟩≡ (77d)
```
if(runq->argv==nil)
    panic("popword but no argv!", 0);
```
Uses `panic()` 180c and `runq` 35a.

⟨`popword()` *sanity check argv words* 77f⟩≡ (77d)
```
if(p==nil)
    panic("popword but no word!", 0);
```
Uses `panic()` 180c.

8.2.5 Process status management

The exit status of the last command is stored in the `$status` variable. `setstatus()`[78a] sets it, and `getstatus()`[78b] and `truestatus()`[78c] read it. In Plan 9 and `rc`, a status is a string (not an integer like in UNIX and the Bourne shell). An empty string or the string `"0"` means success (which can be confusing); anything else means failure. This mirrors the Plan 9 system call `exits()` (see KERNEL book [Pad14]), which takes a string argument rather than a numeric exit code.

⟨*function* `setstatus` 78a⟩≡ (237a)
```
void
setstatus(char *s)
{
    setvar("status", newword(s, (word *)nil));
}
```
Uses `newword()` 36c and `setvar()` 39b.

⟨*function* `getstatus` 78b⟩≡ (237a)
```
char*
getstatus(void)
{
    var *status = vlook("status");
    return status->val ? status->val->word : "";
}
```
Uses `vlook()` 39c.

The `|` check in `truestatus()` below deserves explanation. When a pipeline like `ls | wc` finishes, `rc` combines the exit statuses of all commands into a single string separated by `|`, for example `"0|0"` if both succeed or `"error|0"` if the first fails. So `truestatus()` considers a status "true" if it contains only `0` and `|` characters—meaning every stage of the pipeline succeeded.

⟨*function* `truestatus` 78c⟩≡ (237a)
```
bool
truestatus(void)
{
    char *s;
    for(s = getstatus();*s;s++)
        if(*s!='|' && *s!='0')
            return false;
    return true;
}
```
Uses `getstatus()` 78b.

Why not reduce the pipeline status to a single value earlier? Because keeping the full string lets the user inspect `$status` and see exactly which stage failed (e.g., `"0|error|0"` tells you the middle command had a problem). With numeric exit codes (as in UNIX), reducing is easy: just take the maximum or the last non-zero value. But with string statuses, there is no natural way to merge `"can't open"` and `"0"` into a single meaningful string, so `rc` preserves them all. In `bash`, `$?` only reports the exit status of the *last* command in the pipeline—a failure in an earlier stage is silently lost. `bash` later added the `PIPESTATUS` array to work around this, but it requires explicit use.

This is another example of `rc` getting the design right from the start, while `bash` accumulates workarounds: pipeline statuses are visible by default in `$status` (vs. `PIPESTATUS` added later in `bash`), all variables are automatically exported via `/env/` (vs. the error-prone `export` command), `if not` avoids the parsing ambiguity of `else` (vs. the heavyweight `if...then...fi` syntax), and string exit statuses naturally carry diagnostic information (vs. opaque numeric codes that require `strerror()` or a manual to decode).

8.2.6 Subprocesses management

`rc` keeps track of all child processes it has spawned in the `waitpids` array. This is needed for the `wait` builtin (which waits for all children, see Section 9.6) and for proper cleanup when `rc` exits.

⟨*global* `waitpids` 79a⟩≡ (233a)
```
// growing_array<pid> (but really a list)
int *waitpids;
```

⟨*global* `nwaitpids` 79b⟩≡ (233a)
```
int nwaitpids;
```

⟨*function* `addwaitpid` 79c⟩≡ (233a)
```
void
addwaitpid(int pid)
{
    ⟨addwaitpid() grow waitpids if needed 79d⟩
    waitpids[nwaitpids++] = pid;
}
```
Uses `nwaitpids` 79b and `waitpids` 79a.

⟨`addwaitpid()` *grow* `waitpids` *if needed* 79d⟩≡ (79c)
```
waitpids = realloc(waitpids, (nwaitpids+1)*sizeof waitpids[0]);
if(waitpids == nil)
    panic("Can't realloc %d waitpids", nwaitpids+1);
```
Uses `nwaitpids` 79b, `panic()` 180c, and `waitpids` 79a.

⟨*function* `delwaitpid` 79e⟩≡ (240b)
```
void
delwaitpid(int pid)
{
    int r, w;

    for(r=w=0; r<nwaitpids; r++)
        if(waitpids[r] != pid)
            waitpids[w++] = waitpids[r];
    nwaitpids = w;
}
```
Uses `nwaitpids` 79b and `waitpids` 79a.

⟨*function* `clearwaitpids` 79f⟩≡ (233a)
```
void
clearwaitpids(void)
{
    nwaitpids = 0;
}
```
Uses `nwaitpids` 79b.

⟨*function* `havewaitpid` 79g⟩≡ (240b)
```
bool
havewaitpid(int pid)
{
    int i;

    for(i=0; i<nwaitpids; i++)
        if(waitpids[i] == pid)
            return true;
    return false;
}
```
Uses `nwaitpids` 79b and `waitpids` 79a.

8.3 Simple commands

Now that we have seen the helper infrastructure—argument stack management, process status, and subprocess tracking—we can describe the bytecode generation and interpretation for simple commands, which is the core of the shell.

Throughout `outcode()`[76b], `c0`, `c1`, and `c2` are shorthand for the three children of the current node `t` (`t->child[0]` etc.). They keep the per-construct cases terse: for a `SIMPLE` node `c0` is the argument list, for a pipe `c0` and `c1` are the two sides, and so on.

⟨*constant c0 (rc/code.c)* 80a⟩≡ (236a)
```
#define c0  t->child[0]
```

⟨*constant c1 (rc/code.c)* 80b⟩≡ (236a)
```
#define c1  t->child[1]
```

⟨*constant c2 (rc/code.c)* 80c⟩≡ (236a)
```
#define c2  t->child[2]
```

8.3.1 Bytecode generation

A `SIMPLE` command compiles to: `Xmark`[77b] (start a new word list), followed by the bytecodes for the arguments (which push words onto the stack), followed by `Xsimple`[81] (which pops the word list and executes the command).

⟨`outcode()` *cases* 80d⟩≡ (76b) 80e▷
```
case SIMPLE:
    emitf(Xmark);
    outcode(c0, eflag); // the arguments and argv0
    emitf(Xsimple);
    ⟨outcode() emit Xeflag after Xsimple 148c⟩
    break;
```

Note that `ARGLIST` and `WORDS` emit their children in reverse order: since `Xword`[77a] prepends to the word list, emitting the last word first ensures the list ends up in the correct order.

⟨`outcode()` *cases* 80e⟩+≡ (76b) ◁80d 80f▷
```
case ARGLIST:
    outcode(c1, eflag);
    outcode(c0, eflag);
    break;
```

⟨`outcode()` *cases* 80f⟩+≡ (76b) ◁80e 80g▷
```
case WORDS:
    outcode(c1, eflag);
    outcode(c0, eflag);
    break;
```

⟨`outcode()` *cases* 80g⟩+≡ (76b) ◁80f 87a▷
```
case WORD:
    emitf(Xword);
    emits(strdup(t->str));
    break;
```

⟨`codefree()` *in loop over code* `cp`*, switch bytecode cases* 80h⟩+≡ (34b) ◁34c 96a▷
```
else if(p->f==Xword || p->f==Xdelhere)
        efree((++p)->s);
```
Uses `Xdelhere()` 159b, `Xword()` 77a, and `efree()` 181c.

For example, `echo hello world` produces the following AST and bytecodes:

```
AST:                          codebuf:

SIMPLE                        Xmark
 |                            Xword "world"
ARGLIST                       Xword "hello"
 / \                          Xword "echo"
ARGLIST  "world"              Xsimple
 / \
"echo"  "hello"
```

8.3.2 `Xsimple()`

`Xsimple()`[81] is the bytecode that executes a simple command. It is the heart of `rc` and handles three cases, in order:

1. If the command name matches a shell *function* (defined with `fn`), it runs the function's bytecodes in a new thread.
2. If the command name matches a *builtin* (e.g., `cd`, `exit`, `.`), it calls the corresponding C function directly (without forking).
3. Otherwise, it forks a child process with `execforkexec()`[83a] and waits for it to finish.

The skeleton below covers just the last case.

⟨*function* `Xsimple` 81⟩≡ (234c)

```
void
Xsimple(void)
{
    word *a;
    thread *p = runq;
    int pid;
    ⟨Xsimple() other locals 108a⟩

    ⟨Xsimple() initializations, globlist() 133e⟩

    a = runq->argv->words;
    ⟨Xsimple() sanity check a 82b⟩
    ⟨Xsimple() if -x 177c⟩

    ⟨Xsimple() if argv0 is a function 108b⟩
    else{
        ⟨Xsimple() if argv0 is a builtin 116a⟩
        ⟨Xsimple() if exitnext() 86a⟩
        else{
            flush(err);
            Updenv(); /* necessary so changes don't go out again */
            if((pid = execforkexec()) < 0){
                Xerror("try again");
                return;
            }

            /* interrupts don't get us out */
            poplist();
            while(Waitfor(pid, true) < 0)
```

```
                ;
            }
        }
    }
```

Uses Xerror() 82a, err 179b, execforkexec() 83a, flush() 187b, poplist() 77c, and runq 35a.

The `while(Waitfor(...) < 0)` loop above retries the wait if it was interrupted by a signal (e.g., `^C`): the signal kills the child, but the parent still needs to reap it. `Waitfor()`[85b] waits for the single child this simple command forked. A pipeline does not change that: each side of `cmd1 | cmd2` is itself a simple command waiting for its own exec'd process, and the extra step—waiting for the forked child `rc` that ran the left side—is handled separately by `Xpipewait`[98d] (the `Pipe` subsection below). So at this level the shell always waits for exactly one `pid`.

⟨*function* Xerror 82a⟩≡ (230a)

```
void
Xerror(char *s)
{
    if(strcmp(argv0, "rc")==0 || strcmp(argv0, "/bin/rc")==0)
        pfmt(err, "rc: %s: %r\n", s);
    else
        pfmt(err, "rc (%s): %s: %r\n", argv0, s);
    flush(err);
    setstatus("error");

    while(!runq->iflag)
        Xreturn();
}
```

Uses Xreturn() 96c, argv0 45c, err 179b, flush() 187b, pfmt() 190b, runq 35a, and setstatus() 78a.

The `else` branch above fires whenever `argv0` is neither `"rc"` nor `"/bin/rc"`—most commonly when `rc` is running a script, where `argv0` is the script's name. The message then reads `rc (script): ...` so you can tell which script reported the error.

⟨Xsimple() *sanity check a* 82b⟩≡ (81)

```
if(a==nil){
    Xerror1("empty argument list");
    return;
}
```

Uses Xerror1() 82c.

`Xerror1()`[82c] is the same as `Xerror()`[82a] but without the `%r`: `Xerror()` appends the system error string (`%r`), so it is used after a failed system call (an `exec` or `open` that set the kernel's error), whereas `Xerror1()` is for `rc`'s own logical errors that have no associated OS error—like the `"empty argument list"` and `"variable name not singleton"` checks above. Both set the status to `"error"` and unwind the thread stack through `Xreturn`[96c].

⟨*function* Xerror1 82c⟩≡ (230a)

```
void
Xerror1(char *s)
{
    if(strcmp(argv0, "rc")==0 || strcmp(argv0, "/bin/rc")==0)
        pfmt(err, "rc: %s\n", s);
    else
        pfmt(err, "rc (%s): %s\n", argv0, s);
    flush(err);
    setstatus("error");

    while(!runq->iflag)
        Xreturn();
}
```

Uses Xreturn() 96c, argv0 45c, err 179b, flush() 187b, pfmt() 190b, runq 35a, and setstatus() 78a.

8.3.3 fork()

`execforkexec()`[83a] implements the fundamental `fork+exec+wait` pattern that is the essence of a shell. It forks a child process, and in the child, calls `execexec()`[83b] to replace the child with the actual command. If `exec` fails (e.g., the command does not exist), the child exits with an error; the parent continues normally.

⟨*function* `execforkexec` 83a⟩≡ (233a)
```
int
execforkexec(void)
{
    int pid;
    int n;
    char buf[ERRMAX];

    // fork()!!
    switch(pid = fork()){
    case -1:
        return -1;
    case 0: // child
        clearwaitpids();
        pushword("exec");
        execexec();

        // should not be reached! unless the command did not exist
        strcpy(buf, "can't exec: ");
        n = strlen(buf);
        errstr(buf+n, ERRMAX-n);
        Exit(buf, __LOC__);
    }
    // parent
    addwaitpid(pid);
    return pid;
}
```
Uses `addwaitpid()` 79c, `clearwaitpids()` 79f, `execexec()` 83b, and `pushword()` 46b.

The `pushword("exec")` before `execexec()` is a trick: `execexec()` is also used as the `exec` builtin (see Section 14.5.1), where `exec` appears as the first word in `argv`. Here we add it artificially so `execexec()` can `popword()` it uniformly.

8.3.4 exec()

⟨*function* `execexec` 83b⟩≡ (234c)
```
/// execforkexec | builtin "exec" -> <>
void
execexec(void)
{
    popword(); /* "exec" */
    ⟨execexec() sanity check arguments 83c⟩
    ⟨execexec() perform the redirections 90a⟩

    Execute(runq->argv->words, searchpath(runq->argv->words->word));
    // should not be reached! unless command did not exist
    poplist();
}
```
Uses `Execute()` 84c, `poplist()` 77c, `popword()` 77d, `runq` 35a, and `searchpath()` 84a.

⟨`execexec()` *sanity check arguments* 83c⟩≡ (83b)
```
if(runq->argv->words==nil){
    Xerror1("empty argument list");
```

```
    return;
}
```

Uses Xerror1() 82c and runq 35a.

8.3.5 $path management

When the command name starts with /, ./, ../, or # (device path in Plan 9), it is treated as an absolute path and no search is performed. Otherwise, searchpath()[84a] returns the list of directories in $path to try. Execute()[84c] then iterates over these directories, prepending each to the command name and calling exec(). If exec() fails, it tries the next directory. This is the standard UNIX path search mechanism.

⟨*function* searchpath 84a⟩≡ (238a)

```
word*
searchpath(char *w)
{
    word *path;

    if(strncmp(w, "/", 1)==0
    || strncmp(w, "#", 1)==0
    || strncmp(w, "./", 2)==0
    || strncmp(w, "../", 3)==0
    || (path = vlook("path")->val)==nil)
        path=&nullpath;
    return path;
}
```

Uses nullpath 84b and vlook() 39c.

⟨*global* nullpath 84b⟩≡ (238a)

```
struct Word nullpath = { "", nil};
```

Uses Word 36b.

⟨*function* Execute 84c⟩≡ (233a)

```
void
Execute(word *args, word *path)
{
    char **argv = mkargv(args);
    char file[1024];
    char errstr[1024];
    int nc;

    Updenv();
    errstr[0] = '\0';

    for(;path;path = path->next){
        nc = strlen(path->word);
        if(nc < sizeof file - 1){ /* 1 for / */
            strcpy(file, path->word);
            if(file[0]){
                strcat(file, "/");
                nc++;
            }
            if(nc + strlen(argv[1]) < sizeof file){
                strcat(file, argv[1]);

                // The actual exec() system call!
                exec(file, argv+1);

                // reached if the file does not exist
```

```
                rerrstr(errstr, sizeof errstr);
                /*
                 * if file exists and is executable, exec should
                 * have worked, unless it's a directory or an
                 * executable for another architecture.  in
                 * particular, if it failed due to lack of
                 * swap/vm (e.g., arg. list too long) or other
                 * allocation failure, stop searching and print
                 * the reason for failure.
                 */
                if (strstr(errstr, " allocat") != nil ||
                    strstr(errstr, " full") != nil)
                    break;
            }
            else werrstr("command name too long");
        }
    }
    // should not be reached if found an actual binary to exec
    pfmt(err, "%s: %s\n", argv[1], errstr);
    efree((char *)argv);
}
```

Uses `efree()` 181c, `err` 179b, `mkargv()` 85a, and `pfmt()` 190b.

⟨*function* `mkargv` 85a⟩≡ (233a)

```
char **
mkargv(word *a)
{
    char **argv = (char **)emalloc((count(a)+2) * sizeof(char *));
    char **argp = argv+1; /* leave one at front for runcoms */

    for(;a;a = a->next)
        *argp++=a->word;
    *argp = nil;
    return argv;
}
```

Uses `count()` 37a and `emalloc()` 181a.

8.3.6 `wait()`

Back in the parent process, `Waitfor()`[85b] loops calling `wait()` until it reaps the child with the expected `pid`. If `wait()` returns a different child (from a background process started with `&`), its status is recorded but the loop continues. If `wait()` is interrupted by a signal, `Waitfor()` returns `-1`, and the caller (`Xsimple`) retries.

⟨*function* `Waitfor` *(plan9.c)* 85b⟩≡ (240b)

```
/// Xsimple -> <>
int
Waitfor(int pid, bool _persist)
{
    thread *p;
    Waitmsg *w;
    char errbuf[ERRMAX];

    if(pid >= 0 && !havewaitpid(pid))
        return 0;

    // wait()!! until we found it
    while((w = wait()) != nil){
```

```
            delwaitpid(w->pid);

            if(w->pid==pid){
                setstatus(w->msg);
                free(w);
                return 0;
            }
            ⟨Waitfor() in while loop, if wait returns another pid 98e⟩
        }

        errstr(errbuf, sizeof errbuf);
        if(strcmp(errbuf, "interrupted")==0)
            return -1;
        return 0;
    }
```

Uses delwaitpid() 79e, havewaitpid() 79g, and setstatus() 78a.

8.3.7 Fork optimization

If the next bytecode after Xsimple[81] is Xexit[96b] (possibly preceded by Xpopredir[92c] calls), the shell is about to exit anyway, so forking a child and waiting for it is pointless. In that case, Xsimple calls execexec()[83b] directly, replacing the current process with the command. This is the same optimization that the exec builtin provides, applied automatically. This matters in practice for scripts like rc -c 'ls': without this optimization, rc would fork, exec ls in the child, wait, and then exit—wasting a process.

⟨Xsimple() *if exitnext()* 86a⟩≡ (81)

```
    if(exitnext()){
        /* fork and wait is redundant */
        pushword("exec");
        execexec();
        Xexit();
    }
```

Uses Xexit() 96b, execexec() 83b, exitnext() 86b, and pushword() 46b.

⟨*function* exitnext 86b⟩≡ (234c)

```
    /*
     * Search through the following code to see if we're just going to exit.
     */
    int
    exitnext(void){
        union Code *c = &runq->code[runq->pc];
        while(c->f==Xpopredir)
            c++;
        return c->f==Xexit;
    }
```

Uses Code 33c, Xexit() 96b, Xpopredir() 92c, and runq 35a.

That completes the code for simple commands—by far the largest piece of the interpreter so far, and a striking contrast with the twenty-odd lines of the minishell1.c we saw in Section 2.1.2. The extra machinery (a code vector, the runq thread, the redirection stack, the fork optimization) is what buys rc its flexibility: the pipes, redirections, functions, and control flow in the rest of this chapter all build on these same pieces.

8.4 Operators

With simple commands handled, the remaining constructs—sequencing, pipes, logical operators, and the control-flow keywords—are just more cases in outcode()[76b], each emitting a small pattern of bytecodes, often with jumps. The next sections walk through them.

8.4.1 Basic sequence

The sequencing operator `;` simply emits the bytecodes of both children one after the other.

⟨outcode() *cases* 87a⟩+≡ (76b) ◁80g 87c▷
```
case ';':
    outcode(c0, eflag);
    outcode(c1, eflag);
    break;
```

8.4.2 Logical operators

`&&` and `||` use a forward jump technique. For `cmd1 && cmd2`, the compiler emits: the bytecodes of `cmd1`, then `Xtrue`[88a] followed by a placeholder jump offset, then the bytecodes of `cmd2`. After emitting `cmd2`, `stuffdot()`[87d] patches the placeholder with the actual address, so `Xtrue` can jump past `cmd2` if the status is false.

⟨outcode() *locals* 87b⟩≡ (76b) 95a▷
```
int p;
```

⟨outcode() *cases* 87c⟩+≡ (76b) ◁87a 87e▷
```
case ANDAND:
    outcode(c0, false);
    emitf(Xtrue);
    p = emiti(0);
    outcode(c1, eflag);
    stuffdot(p);
    break;
```

⟨*function* stuffdot 87d⟩≡ (236a)
```
void
stuffdot(int a)
{
    if(a<0 || codep<=a)
        panic("Bad address %d in stuffdot", a);
    codebuf[a].i = codep;
}
```
Uses codebuf 33b, codep 75a, and panic() 180c.

This is the classic assembler trick of a *forward jump with backpatching* (see the ASSEMBLER book [Pad15]). In assembly you would write

```
        <cmd1>
        Xtrue L1     # jump past cmd2 if status is false
        <cmd2>
L1:
```

but when `Xtrue` is emitted the address of `L1` is not yet known (`cmd2` has not been generated). So the compiler emits a placeholder slot, remembers its index in `p` (`p = emiti(0)`), and once it reaches the label—after `cmd2`—`stuffdot()` fills that slot with the current position (`codebuf[p].i = codep`). Here `p` is just the unresolved forward reference to `L1`.

`Xtrue` and `Xfalse`[88b] are symmetric: `Xtrue` jumps if the status is *false* (skip the second command), and `Xfalse` jumps if the status is *true*.

⟨outcode() *cases* 87e⟩+≡ (76b) ◁87c 88c▷
```
case OROR:
    outcode(c0, false);
    emitf(Xfalse);
    p = emiti(0);
    outcode(c1, eflag);
    stuffdot(p);
    break;
```

⟨*function* `Xtrue` 88a⟩≡ (230b)
```
void
Xtrue(void)
{
    if(truestatus()) runq->pc++;
    else runq->pc = runq->code[runq->pc].i;
}
```
Uses `runq` 35a and `truestatus()` 78c.

⟨*function* `Xfalse` 88b⟩≡ (230b)
```
void
Xfalse(void)
{
    if(truestatus()) runq->pc = runq->code[runq->pc].i;
    else runq->pc++;
}
```
Uses `runq` 35a and `truestatus()` 78c.

⟨`outcode()` *cases* 88c⟩+≡ (76b) ◁87e 88e▷
```
case BANG:
    outcode(c0, eflag);
    emitf(Xbang);
    break;
```

⟨*function* `Xbang` 88d⟩≡ (230b)
```
void
Xbang(void)
{
    setstatus(truestatus()? "false" : "");
}
```
Uses `setstatus()` 78a and `truestatus()` 78c.

Recall `rc`'s status convention: an *empty* status string (or `"0"`) means success—the command counts as "true"—while any non-empty string means failure. That is why `Xbang` (the `!` negation) sets the status to `"false"` for failure and to the empty string for success, and why `truestatus()`[78c] simply tests for emptiness (it can be a little confusing that the *string* `"true"` would itself count as failure, being non-empty; only the empty string is "true".)

8.4.3 String matching

The `~` operator pushes two word lists onto `argv` (the *subject* and the *patterns*), then calls `Xmatch`[89a]. Two `Xmark`[77b] bytecodes are needed because both the subject and the patterns can be multi-word (e.g., if they come from variable expansion), and they must be kept separate.

⟨`outcode()` *cases* 88e⟩+≡ (76b) ◁88c 91d▷
```
case TWIDDLE:
    emitf(Xmark);
    outcode(c1, eflag);
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xmatch);
    ⟨outcode() emit Xeflag after Xmatch 149a⟩
    break;
```

`Xmatch()`[89a] converts the subject words to a single string (joining with spaces), then tries *each* pattern word against it using `match()`[138b] (described in Chapter 12).

⟨*function* `Xmatch` 89a⟩≡ (230b)
```
void
Xmatch(void)
{
    word *p;
    char *subject;

    subject = list2str(runq->argv->words);
    setstatus("no match");
    for(p = runq->argv->next->words;p;p = p->next)
        if(match(subject, p->word, '\0')){
            setstatus("");
            break;
        }
    efree(subject);
    poplist();
    poplist();
}
```
Uses `efree()` 181c, `list2str()` 89b, `poplist()` 77c, `runq` 35a, and `setstatus()` 78a.

⟨*function* `list2str` 89b⟩≡ (230b)
```
char*
list2str(word *words)
{
    char *value, *s, *t;
    int len = 0;
    word *ap;

    for(ap = words;ap;ap = ap->next)
        len += 1+strlen(ap->word);
    value = emalloc(len+1);

    s = value;
    for(ap = words;ap;ap = ap->next){
        for(t = ap->word;*t;)
            *s++=*t++;
        *s++=' ';
    }
    if(s==value)
        *s='\0'; // replace last space
    else s[-1]='\0';
    return value;
}
```
Uses `emalloc()` 181a.

8.4.4 Redirection

Redirections in `rc` are handled in two phases. When the interpreter encounters a redirection bytecode (e.g., `Xwrite`[92b] we will see soon), it opens the target file and pushes a `Redir` record onto the thread's redirection stack. But it does *not* actually redirect the file descriptors yet. The actual redirection happens later, in `execexec()`[83b] inside the forked child process, just before `exec()`: `doredir()`[91a] walks the redirection stack and performs the `dup()` and `close()` calls. This two-phase design is necessary because the parent shell must not modify its own file descriptors; only the child (which is about to be replaced by `exec()`) should have its stdin/stdout/stderr

redirected.

⟨`execexec()` *perform the redirections* 90a⟩≡ (83b)
```
doredir(runq->redir);
```
Uses `doredir()` 91a and `runq` 35a.

⟨`start()` *set redir* 90b⟩≡ (45b)
```
p->redir = p->startredir = runq ? runq->redir : nil;
```
Uses `runq` 35a.

The `startredir` field in `start()`[45b] deserves attention: when a new thread is created, it inherits its parent's redirection stack (so subshells see the parent's redirections), but `startredir` remembers where the inherited portion ends. This way, if an error occurs, the thread can pop only the redirections it added, without disturbing the parent's state.

`Redir`

The `Redir` structure records one redirection the interpreter has to carry out. Its `type` is a *runtime* action enum, distinct from the *parse-time* `READ`/`WRITE`/`APPEND`/`HERE` of the lexer: the values are `ROPEN` (`dup2(from,to)` then close `from`, for a file redirection), `RDUP` (`dup2(from,to)`, for `>[2=1]`), and `RCLOSE` (close `from`, for `>[2=]`). `from` is the descriptor that was opened or duplicated and `to` the one being replaced (e.g. 1 for stdout). The thread's `redir` field chains these records as the interpreter meets redirection bytecodes.

⟨*struct* `Redir` 90c⟩≡ (225)
```
struct Redir {
    // enum<redirection_kind_bis>
    char type; /* what to do */

    /* what to do it to */
    short from;
    short to;

    // Extra
    ⟨Redir extra fields 90e⟩
};
```

⟨`Thread` *other fields* 90d⟩+≡ (34d) ◁52b 94c▷
```
// list<ref_own<Redir>> (next = Redir.next)
struct Redir *redir; /* redirection stack */
```
Uses `Redir` 90c.

⟨`Redir` *extra fields* 90e⟩≡ (90c)
```
struct Redir *next;  /* what else to do (reverse order) */
```
Uses `Redir` 90c.

⟨*function* `pushredir` 90f⟩≡ (230a)
```
void
pushredir(int type, int from, int to)
{
    redir * rp = new(redir);
    rp->type = type;
    rp->from = from;
    rp->to = to;

    // add_list(runq->redir, rp)
    rp->next = runq->redir;
    runq->redir = rp;
}
```
Uses `runq` 35a.

`doredir()`

Because `pushredir()`[90f] prepends each new record to the front of the list, the list is in reverse order relative to the command line. For `cmd > foo >[2=1]`, the list has `>[2=1]` first, then `> foo`. But the user expects the redirections to be applied left-to-right: first redirect stdout to `foo`, then duplicate fd 2 from fd 1. `doredir()`[91a] solves this by recursing to the end of the list before performing any `dup()` calls, so the deepest (oldest) redirection executes first—restoring the original left-to-right order.

⟨*function* `doredir` 91a⟩≡ (234c)
```
/// execexec -> <>
void
doredir(redir *rp)
{
    if(rp){
        // recurse first, so do them in the reverse order of the list
        doredir(rp->next);

        switch(rp->type){
        ⟨doredir() switch redir type cases 91b⟩
        }
    }
}
```
Uses `doredir()` 91a.

⟨`doredir()` *switch redir type cases* 91b⟩≡ (91a) 164f▷
```
case ROPEN:
    if(rp->from != rp->to){
        dup(rp->from, rp->to);
        close(rp->from);
    }
    break;
```
Uses `ROPEN` 91c.

⟨*constant* `ROPEN` 91c⟩≡ (225)
```
/*
 * redir types
 */
#define ROPEN 1   /* dup2(from, to); close(from); */
```

The `if(rp->from != rp->to)` guard handles the rare case where `Creat()`[193g] happens to hand back exactly the target fd (possible when stdout was previously closed), in which case the `dup` and `close` would defeat each other.

Bytecode generation

The bytecode for a `REDIR` node first evaluates the filename (with globbing), then emits the appropriate redirection bytecode (e.g., `Xwrite`[92b] for `>`), followed by the file descriptor number. After the redirected command (`c1`) finishes, `Xpopredir`[92c] cleans up by closing the opened file descriptor and removing the `Redir` record from the stack. This scoping ensures that in `cmd1 > foo; cmd2`, the redirection applies only to `cmd1`—once `Xpopredir` runs, `cmd2` sees the original file descriptors.

⟨`outcode()` *cases* 91d⟩+≡ (76b) ◁88e 95b▷
```
case REDIR:
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xglob);

    switch(t->rtype){
    ⟨outcode() when REDIR case, switch redirection type cases 92a⟩
```

```
    }
    emiti(t->fd0);
    outcode(c1, eflag);
    emitf(Xpopredir);
    break;
```

`Xwrite()`

`Xwrite()`[92b] handles the `>` redirection at runtime. It pops the filename from the argv stack, opens (or creates) the file with `Creat()`[193g], and pushes an `ROPEN` record mapping the newly opened file descriptor to the target fd (typically 1 for stdout, read from the bytecode stream). The actual `dup()` will happen later in `doredir()`[91a].

⟨`outcode()` *when REDIR case, switch redirection type cases* 92a⟩≡ (91d) 93a▷

```
case WRITE:
    emitf(Xwrite);
    break;
```

⟨*function* `Xwrite` 92b⟩≡ (230b)

```
void
Xwrite(void)
{
    char *file;
    fdt f;

    switch(count(runq->argv->words)){
    default:
        Xerror1("> requires singleton\n");
        return;
    case 0:
        Xerror1("> requires file\n");
        return;
    case 1:
        break;
    }
    file = runq->argv->words->word;
    if((f = Creat(file))<0){
        pfmt(err, "%s: ", file);
        Xerror("can't open");
        return;
    }
    pushredir(ROPEN, f, runq->code[runq->pc].i);
    runq->pc++;
    poplist();
}
```

Uses `Creat()` 193g, `ROPEN` 91c, `Xerror()` 82a, `Xerror1()` 82c, `count()` 37a, `err` 179b, `pfmt()` 190b, `poplist()` 77c, `pushredir()` 90f, and `runq` 35a.

`Xpopredir()`[92c] is the cleanup counterpart: it pops the top `Redir` record and, if it was an `ROPEN`, closes the file descriptor that was opened for the redirection. This is important because the opened fd (from `Creat()` or `Open()`) is a temporary—after `doredir()` has `dup()`'d it onto the target fd in the child process, the parent still holds the original, and it must be closed to avoid leaking file descriptors.

⟨*function* `Xpopredir` 92c⟩≡ (230a)

```
void
Xpopredir(void)
{
    struct Redir *rp = runq->redir;

    if(rp==nil)
        panic("turfredir null!", 0);
```

```
    // pop_list(runq->redir);
    runq->redir = rp->next;

    if(rp->type==ROPEN)
        close(rp->from);

    efree((char *)rp);
}
```

Uses ROPEN 91c, Redir 90c, efree() 181c, panic() 180c, and runq 35a.

Xread()

Xread()[93b] implements input redirection (<, and the here-document <<). It checks the argument is a single word, opens the named file read-only, and pushes an ROPEN redirection so the descriptor is duplicated onto the target (stdin by default).

⟨outcode() *when REDIR case, switch redirection type cases* 93a⟩+≡ (91d) ◁92a 94a▷

```
case READ:
case HERE:
    emitf(Xread);
    break;
```

⟨*function* Xread 93b⟩≡ (230b)

```
void
Xread(void)
{
    char *file;
    int f;

    switch(count(runq->argv->words)){
    default:
        Xerror1("< requires singleton\n");
        return;
    case 0:
        Xerror1("< requires file\n");
        return;
    case 1:
        break;
    }
    file = runq->argv->words->word;
    if((f = open(file, 0))<0){
        pfmt(err, "%s: ", file);
        Xerror("can't open");
        return;
    }
    pushredir(ROPEN, f, runq->code[runq->pc].i);
    runq->pc++;
    poplist();
}
```

Uses ROPEN 91c, Xerror() 82a, Xerror1() 82c, count() 37a, err 179b, pfmt() 190b, poplist() 77c, pushredir() 90f, and runq 35a.

Xread() is almost identical to Xwrite()[92b]; the difference is the open mode—Xread() opens the file read-only, whereas Xwrite() creates and truncates it for writing—and the default descriptor each one redirects (stdin versus stdout). The rest (the singleton check, the ROPEN pushredir()[90f], the poplist()[77c]) is the same boilerplate.

`Xappend()`

`Xappend()`[94b] implements `>>`. It is `Xwrite()`[92b] with one change: instead of truncating, it opens the existing file (or creates it) and seeks to the end before redirecting, so output is appended rather than overwriting.

⟨`outcode()` *when REDIR case, switch redirection type cases* 94a⟩+≡ (91d) ◁93a 161d▷
```
case APPEND:
    emitf(Xappend);
    break;
```

⟨*function* `Xappend` 94b⟩≡ (230b)
```
void
Xappend(void)
{
    char *file;
    int f;

    switch(count(runq->argv->words)){
    default:
        Xerror1(">> requires singleton");
        return;
    case 0:
        Xerror1(">> requires file");
        return;
    case 1:
        break;
    }
    file = runq->argv->words->word;
    if((f = open(file, 1))<0 && (f = Creat(file))<0){
        pfmt(err, "%s: ", file);
        Xerror("can't open");
        return;
    }
    seek(f, 0L, SEEK__END);
    pushredir(ROPEN, f, runq->code[runq->pc].i);
    runq->pc++;
    poplist();
}
```
Uses `Creat()` 193g, `ROPEN` 91c, `Xerror()` 82a, `Xerror1()` 82c, `count()` 37a, `err` 179b, `pfmt()` 190b, `poplist()` 77c, `pushredir()` 90f, and `runq` 35a.

Closing redirection opened files

⟨`Thread` *other fields* 94c⟩+≡ (34d) ◁90d 98a▷
```
struct Redir *startredir; /* redir inheritance point */
```
Uses `Redir` 90c.

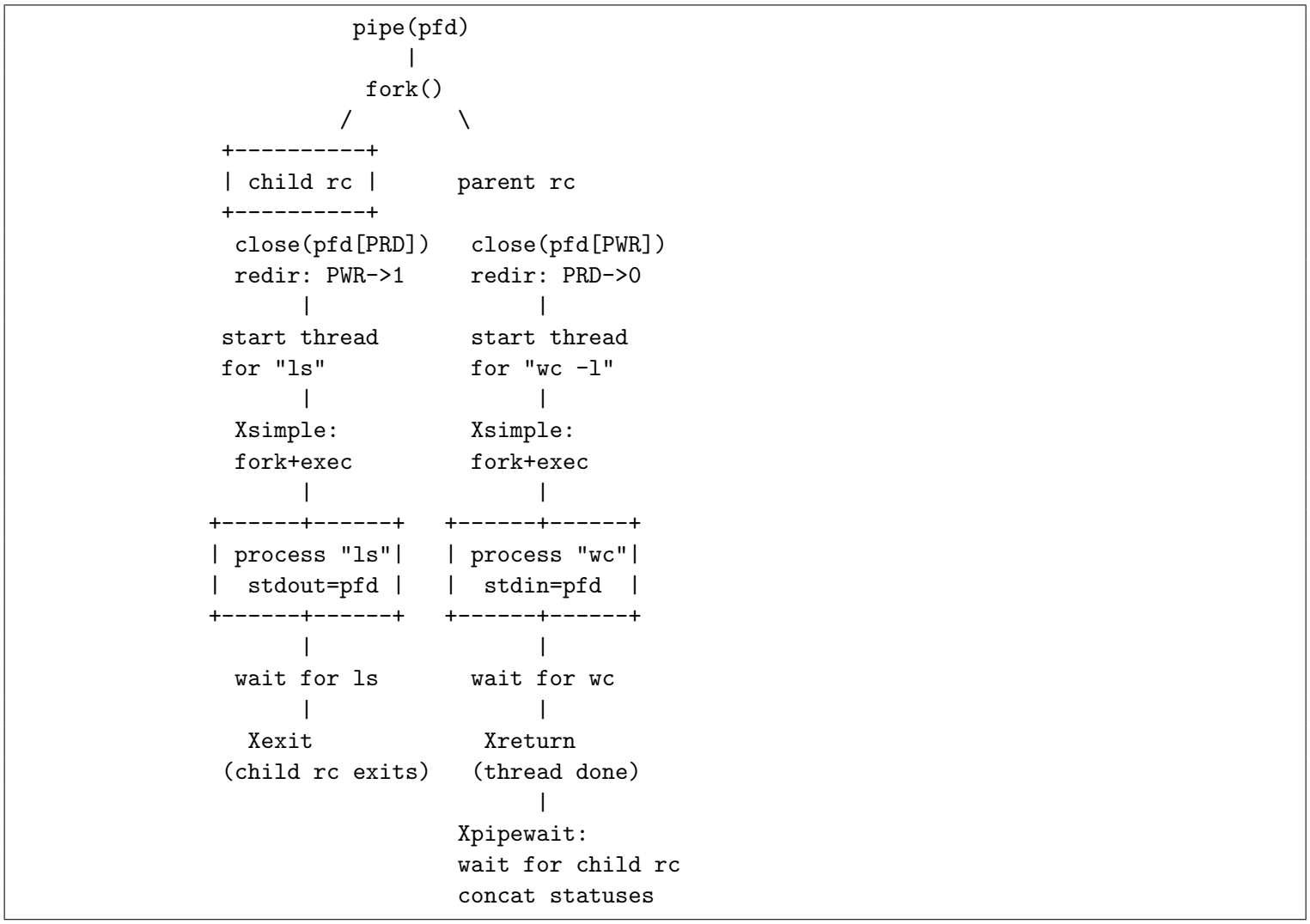
⟨*function* `turfredir` 94d⟩≡ (230a)
```
/// (Xerror | ...) -> Xreturn -> <>
void
turfredir(void)
{
    while(runq->redir != runq->startredir)
        Xpopredir();
}
```
Uses `Xpopredir()` 92c and `runq` 35a.

⟨`Xreturn()` *pop the redirections from this thread* 94e⟩≡ (96c)
```
turfredir();
```
Uses `turfredir()` 94d.

8.4.5 Pipe

Pipes are arguably the most important operator in a shell. For `cmd1 | cmd2`, four processes end up being involved: the parent `rc`, a child `rc` that interprets `cmd1`'s bytecodes, and then each side fork+exec's the actual command (e.g., `ls` and `wc`). The left side (`cmd1`) runs in a forked child `rc` process. The right side (`cmd2`) runs in the parent `rc`'s new thread. This asymmetry means that the right side can modify the shell's state (e.g., variable assignments), while the left side cannot. The compiled bytecodes contain two jump offsets: one for the right side's code, and one for the parent's `Xpipewait`[98d].

Here is the overall picture for `ls | wc -l`:



⟨outcode() *locals* 95a⟩+≡ (76b) ◁87b 109a▷

```
int q;
```

⟨outcode() *cases* 95b⟩+≡ (76b) ◁91d 99a▷

```
case PIPE:
    emitf(Xpipe);
    emiti(t->fd0); // 1 in the normal case
    emiti(t->fd1); // 0 in the normal case
    p = emiti(0);
    q = emiti(0);

    // for first child
    outcode(c0, eflag);
    emitf(Xexit);
```

```
    // for second child
    stuffdot(p);
    outcode(c1, eflag);
    emitf(Xreturn);

    // for parent (rc)
    stuffdot(q);
    emitf(Xpipewait);
    break;
```

The child runs `cmd1`'s bytecodes and ends with `Xexit` (it must terminate, not return to the parent's thread stack). The parent creates a new thread for `cmd2` that ends with `Xreturn`, then falls through to `Xpipewait` which waits for the child process and concatenates both exit statuses into a pipe status string like `"0|0"`.

⟨`codefree()` *in loop over code* `cp`*, switch bytecode cases* 96a⟩+≡ (34b) ◁80h 106b▷

```
  else if(p->f==Xpipe)
          p+=4;
```

Uses `Xpipe()` 97c.

⟨*function* `Xexit` 96b⟩≡ (230b)

```
  void
  Xexit(void)
  {
      ⟨Xexit() locals 147b⟩

      ⟨Xexit() trap management 147c⟩
      Exit(getstatus(), __LOC__);
  }
```

Uses `getstatus()` 78b.

Three layers sit behind "exiting." `exits()` is the Plan 9 libc call that actually terminates the process with a status string. `Exit()`[180e] is `rc`'s wrapper around it (defined per host): it flushes variables to `/env` with `Updenv()`[123a], records the status, then calls `exits()`. `Xexit` is the *bytecode*: it runs any exit traps and then calls `Exit()`. So `Xexit` ends the whole shell, whereas `Xreturn`[96c] (next) ends only the current thread.

`Xreturn` ends the current thread and hands control back to the one that started it: it pops this thread's redirections, frees its argument lists and code vector (`codefree()`[34b]), then restores `runq` to the saved parent `p->ret` (exiting the shell if there is no parent). Every compiled line ends in an `Xreturn` so control returns to the read loop, as does every function call and sourced file—it is the general "this program is done" return, not anything specific to pipes.

⟨*function* `Xreturn` 96c⟩≡ (230a)

```
  void
  Xreturn(void)
  {
      struct Thread *p = runq;

      ⟨Xreturn() pop the redirections from this thread 94e⟩
      // free p
      while(p->argv)
          poplist();
      codefree(p->code);
      // pop(runq)
      runq = p->ret;
      efree((char *)p);

      if(runq==nil)
          Exit(getstatus(), __LOC__);
  }
```

Uses `Thread` 34d, `codefree()` 34b, `efree()` 181c, `getstatus()` 78b, `poplist()` 77c, and `runq` 35a.

`PRD` and `PWR` just name the two ends of the array filled by `pipe(pfd)`: `pfd[PRD]` is the read end and `pfd[PWR]` the write end. Naming them keeps the pipe-setup code readable—close the end you do not use, `dup` the other onto stdin or stdout—instead of sprinkling bare 0s and 1s.

⟨*constant* `PRD` 97a⟩≡ (223b)
```
/*
 * Which fds are the reading/writing end of a pipe?
 * Unfortunately, this can vary from system to system.
 * 9th edition Unix doesn't care, the following defines
 * work on plan 9.
 */
#define PRD 0
```

⟨*constant* `PWR` 97b⟩≡ (223b)
```
#define PWR 1
```

With the helpers in place, `Xpipe`[97c] is the runtime half of the pipe: the bytecode that actually calls `pipe()`, forks the child `rc`, wires the two ends onto the right descriptors, and launches a thread for each side—the concrete realization of the diagram above.

⟨*function* `Xpipe` 97c⟩≡ (233a)
```
void
Xpipe(void)
{
    struct Thread *p = runq;
    int pc = p->pc;
    int forkid;
    fdt lfd = p->code[pc++].i;
    fdt rfd = p->code[pc++].i;
    fdt pfd[2];

    if(pipe(pfd)<0){
        Xerror("can't get pipe");
        return;
    }
    switch(forkid = fork()){
    case -1:
        Xerror("try again");
        break;
    case 0: // child
        clearwaitpids();
        // pc+2 so jump the jump addresses
        start(p->code, pc+2, runq->local);
        runq->ret = nil;
        close(pfd[PRD]);
        pushredir(ROPEN, pfd[PWR], lfd);
        break;
    default: // parent
        addwaitpid(forkid);
        start(p->code, p->code[pc].i, runq->local);
        close(pfd[PWR]);
        pushredir(ROPEN, pfd[PRD], rfd);
        p->pc = p->code[pc+1].i;
        p->pid = forkid;
        break;
    }
}
```

Uses `PRD` 97a, `PWR` 97b, `ROPEN` 91c, `Thread` 34d, `Xerror()` 82a, `addwaitpid()` 79c, `clearwaitpids()` 79f, `pushredir()` 90f, `runq` 35a, and `start()` 45b.

⟨Thread *other fields* 98a⟩+≡ (34d) ◁94c 98b▷
```
// option<int> (None = -1)
int pid;  /* process for Xpipewait to wait for */
```

⟨Thread *other fields* 98b⟩+≡ (34d) ◁98a 101b▷
```
char status[NSTATUS]; /* status for Xpipewait */
```
Uses NSTATUS 98c.

⟨*constant* NSTATUS 98c⟩≡ (225)
```
#define NSTATUS ERRMAX   /* length of status (from plan 9) */
```

⟨*function* Xpipewait 98d⟩≡ (230b)
```
void
Xpipewait(void)
{
    char status[NSTATUS+1];
    if(runq->pid==-1)
        setstatus(concstatus(runq->status, getstatus()));
    else{
        strncpy(status, getstatus(), NSTATUS);
        status[NSTATUS]='\0';
        Waitfor(runq->pid, true);
        runq->pid=-1;
        setstatus(concstatus(getstatus(), status));
    }
}
```
Uses NSTATUS 98c, concstatus() 98f, getstatus() 78b, runq 35a, and setstatus() 78a.

⟨Waitfor() *in while loop, if wait returns another pid* 98e⟩≡ (85b)
```
// else
for(p = runq->ret;p;p = p->ret)
    if(p->pid==w->pid){
        p->pid=-1;
        strcpy(p->status, w->msg);
    }
free(w);
```
Uses runq 35a.

⟨*function* concstatus 98f⟩≡ (237a)
```
char*
concstatus(char *s, char *t)
{
    static char v[NSTATUS+1];
    int n = strlen(s);
    strncpy(v, s, NSTATUS);
    if(n<NSTATUS){
        v[n]='|';
        strncpy(v+n+1, t, NSTATUS-n-1);
    }
    v[NSTATUS]='\0';
    return v;
}
```
Uses NSTATUS 98c.

8.4.6 Asynchronous execution

The `&` operator runs a command in the background. The child's stdin is redirected to `/dev/null` (so it does not compete with the shell for terminal input), and the child is put in a new note group (`RFNOTEG`) so that interrupts sent to the shell do not kill it (see Chapter 13 for more information on notes). The parent records the child's PID in `$apid` (for "asynchronous PID"), allowing the user to wait for it later or send signals to it.

⟨`outcode()` *cases* 99a⟩+≡ (76b) ◁95b 100a▷

```
case '&':
    emitf(Xasync);
    p = emiti(0);
    outcode(c0, eflag);
    emitf(Xexit);
    stuffdot(p);
    break;
```

⟨*function* `Xasync` 99b⟩≡ (233a)

```
void
Xasync(void)
{
    fdt null = open("/dev/null", 0);
    int pid;
    char npid[10];

    if(null<0){
        Xerror("Can't open /dev/null\n");
        return;
    }
    switch(pid = rfork(RFFDG|RFPROC|RFNOTEG)){
    case -1:
        close(null);
        Xerror("try again");
        break;
    case 0: // child
        clearwaitpids();
        pushredir(ROPEN, null, STDIN);
        // start a new Thread runq->pc+1 so skip pointer to code after &
        start(runq->code, runq->pc+1, runq->local);
        runq->ret = 0;
        break;
    default: // parent
        addwaitpid(pid);
        close(null);
        // jump to code after &
        runq->pc = runq->code[runq->pc].i;
        inttoascii(npid, pid);
        setvar("apid", newword(npid, (word *)nil));
        break;
    }
}
```

Uses `ROPEN` 91c, `Xerror()` 82a, `addwaitpid()` 79c, `clearwaitpids()` 79f, `inttoascii()` 193b, `newword()` 36c, `pushredir()` 90f, `runq` 35a, `setvar()` 39b, and `start()` 45b.

8.5 Control flow statements

The control-flow keywords—`if`, `while`, `for`, and `switch`—all compile with the same forward-jump and back-patching technique introduced for `&&` and `||`: emit a condition, a conditional jump with a placeholder offset, the body, then patch the offset once the body's end is known. The following subsections show each one.

8.5.1 `if`

The `if` compilation uses the same forward-jump technique as `&&`: emit the condition, then `Xif`[100b] with a jump offset to skip the body if the condition is false. The `Xwastrue`[100d] bytecode at the end resets the `ifnot` flag. This flag is how `rc` connects `if` and `if not`: `Xif` sets `ifnot` to `true`, and `Xifnot`[100f] checks it to decide whether to execute the else-body. If any other command is executed between `if` and `if not`, `Xwastrue` resets the flag, making `if not` a no-op.

⟨`outcode()` *cases* 100a⟩+≡ (76b) ◁99a 100e▷
```
case IF:
    outcode(c0, false); // the condition
    emitf(Xif);
    p = emiti(0);
    outcode(c1, eflag); // then body
    emitf(Xwastrue);
    stuffdot(p);
    break;
```

⟨*function* `Xif` 100b⟩≡ (230b)
```
void
Xif(void)
{
    ifnot = true;
    if(truestatus())
        runq->pc++;
    else
        runq->pc = runq->code[runq->pc].i;
}
```
Uses `ifnot` 100c, `runq` 35a, and `truestatus()` 78c.

⟨*global* `ifnot` 100c⟩≡ (230b)
```
bool ifnot;  /* dynamic if not flag */
```

⟨*function* `Xwastrue` 100d⟩≡ (230b)
```
void
Xwastrue(void)
{
    ifnot = false;
}
```
Uses `ifnot` 100c.

⟨`outcode()` *cases* 100e⟩+≡ (76b) ◁100a 101e▷
```
case NOT:
    ⟨outcode() when NOT, sanity check last command was an if 101a⟩
    emitf(Xifnot);
    p = emiti(0);
    outcode(c0, eflag);
    stuffdot(p);
    break;
```

⟨*function* `Xifnot` 100f⟩≡ (230b)
```
void
Xifnot(void)
{
    if(ifnot)
        runq->pc++;
    else
        runq->pc = runq->code[runq->pc].i;
}
```
Uses `ifnot` 100c and `runq` 35a.

`Xifnot` above is the *dynamic* side of `if not`: at run time it consults the `ifnot` flag to decide whether to run the else-body. There is also a *static* side, checked at compile time in `outcode()`[76b]—a sanity check that an `if not` actually follows an `if(...)`, using the `iflast` flag discussed earlier.

⟨`outcode()` *when NOT, sanity check last command was an if* 101a⟩≡ (100e)
```
if(!runq->iflast)
    yyerror("'if not' does not follow 'if(...)'");
```

⟨`Thread` *other fields* 101b⟩+≡ (34d) ◁98b
```
bool iflast;  /* static 'if not' checking */
```

"Static" here means the check happens at compile time—inside `outcode()` as the code is generated, not while it runs. It only verifies the syntactic structure (that an `if not` follows an `if(...)`) and ignores the runtime exit status. The flag lives on `runq` rather than in a plain global because `runq` is what persists across the compile-then-run cycle of consecutive lines, so the bit set while compiling one line survives to the next.

⟨`outcode()` *set iflast after switch* 101c⟩≡ (76b)
```
if(t->type!=NOT && t->type!=';')
    runq->iflast = t->type==IF;
else
   if(c0)
       runq->iflast = c0->type==IF;
```

⟨`outcode()` *set iflast before switch* 101d⟩≡ (76b)
```
if(t->type!=NOT && t->type!=';')
    runq->iflast = false;
```

8.5.2 `while`

The `while` loop compiles to: the condition bytecodes, `Xtrue`[88a] (jump out if false), the body bytecodes, `Xjump`[101g] (jump back to the condition). If the condition is empty (e.g., `while()`), an `Xsettrue`[101f] bytecode is emitted to make it loop forever.

⟨`outcode()` *cases* 101e⟩+≡ (76b) ◁100e 102a▷
```
case WHILE:
    q = codep;
    outcode(c0, false);
    if(q==codep)
        emitf(Xsettrue);    /* empty condition == while(true) */
    emitf(Xtrue);
    p = emiti(0);
    outcode(c1, eflag);
    emitf(Xjump);
    emiti(q);
    stuffdot(p);
    break;
```

⟨*function* `Xsettrue` 101f⟩≡ (230b)
```
void
Xsettrue(void)
{
    setstatus("");
}
```
Uses `setstatus()` 78a.

⟨*function* `Xjump` 101g⟩≡ (230b)
```
void
Xjump(void)
{
    runq->pc = runq->code[runq->pc].i;
}
```
Uses `runq` 35a.

8.5.3 `for`

The `for` loop is more complex. It first evaluates the list to iterate over (or expands `$*` if no list is given), creates a local variable for the loop variable (via `Xlocal`[110a]), and then loops with `Xfor`[102b]. `Xfor()`[102b] pops the next word from the list, assigns it to the local variable, and executes the body. When the list is exhausted, it jumps past the body and `Xunlocal`[110b] removes the local binding.

⟨`outcode()` *cases* 102a⟩+≡ (76b) ◁101e 103a▷

```
case FOR:
    emitf(Xmark);
    if(c1){
        outcode(c1, eflag);
        emitf(Xglob);
    }
    else{
        emitf(Xmark);
        emitf(Xword);
        emits(strdup("*"));
        emitf(Xdol);
    }
    emitf(Xmark);       /* dummy value for Xlocal */
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xlocal);
    p = emitf(Xfor);
    q = emiti(0);
    outcode(c2, eflag);
    emitf(Xjump);
    emiti(p);
    stuffdot(q);
    emitf(Xunlocal);
    break;
```

The two jumps are easiest to see laid out as bytecode:

```
for(i in a b c) { body }   compiles to:

        Xmark            (dummy value for Xlocal)
        Xmark
        "i"              (the loop variable)
        Xlocal           bind i, take over the word list
  p:    Xfor   q         if the list is empty, jump forward to q
        <body>
        Xjump  p         else jump back to p for the next iteration
  q:    Xunlocal         restore i and leave the loop
```

`Xfor` pulls the next word off the list into `i` and falls through to the body, or—when the list is exhausted—jumps to `q`. The trailing `Xjump p` forms the back-edge. `p` is the address of the `Xfor` (saved by `p = emitf(Xfor)`) and `q` the forward placeholder patched by `stuffdot()`[87d].

⟨*function* `Xfor` 102b⟩≡ (230b)

```
void
Xfor(void)
{
    if(runq->argv->words==0){
        poplist();
        runq->pc = runq->code[runq->pc].i;
    }
    else{
```

```
        freelist(runq->local->val);
        runq->local->val = runq->argv->words;
        runq->local->changed = true;
        runq->argv->words = runq->argv->words->next;
        runq->local->val->next = 0;
        runq->pc++;
    }
}
```

Uses `freelist()` 38b, `poplist()` 77c, and `runq` 35a.

8.5.4 `switch`

The `switch` statement is the most complex control flow construct to compile because it involves multiple jump targets. The generated bytecodes push the switch value onto the stack, then for each `case`, push the case patterns and call `Xcase`[105a], which uses `match()`[138b] (the same glob matcher used by the `~` operator) to compare. If the case matches, execution falls through to the case body; otherwise, it jumps to the next case. After a case body executes, it jumps to the end of the switch (like an implicit `break;` in C). The comment in `codeswitch()`[103b] below shows the layout.

⟨`outcode()` *cases* 103a⟩+≡ (76b) ◁102a 105b▷

```
case SWITCH:
    codeswitch(t, eflag);
    break;
```

⟨*function* `codeswitch` 103b⟩≡ (236a)

```
/*
 * switch code looks like this:
 *  Xmark
 *  (get switch value)
 *  Xjump   1f
 * out: Xjump   leave
 * 1:   Xmark
 *  (get case values)
 *  Xcase   1f
 *  (commands)
 *  Xjump   out
 * 1:   Xmark
 *  (get case values)
 *  Xcase   1f
 *  (commands)
 *  Xjump   out
 * 1:
 * leave:
 *  Xpopm
 */
void
codeswitch(tree *t, bool eflag)
{
    int leave;      /* patch jump address to leave switch */
    int out;        /* jump here to leave switch */
    int nextcase;   /* patch jump address to next case */
    tree *tt;

    // c1 is BRACE { ; ; ; }
    if(c1->child[0]==nil
    || c1->child[0]->type != ';'
    || !iscase(c1->child[0]->child[0])){
        yyerror("case missing in switch");
```

```
            return;
        }

        emitf(Xmark);
        outcode(c0, eflag);
        emitf(Xjump);

        nextcase = emiti(0);
        out = emitf(Xjump);
        leave = emiti(0);

        stuffdot(nextcase);

        // from now on c0, c1, ... refer to this new t
        t = c1->child[0];
        while(t->type==';'){
            tt = c1;
            emitf(Xmark);
            for(t = c0->child[0];t->type==ARGLIST;t = c0)
                outcode(c1, eflag);
            emitf(Xcase);
            nextcase = emiti(0);
            t = tt;
            for(;;){
                if(t->type==';'){
                    if(iscase(c0))
                        break;
                    outcode(c0, eflag);
                    t = c1;
                }
                else{
                    if(!iscase(t))
                        outcode(t, eflag);
                    break;
                }
            }
            emitf(Xjump);
            emiti(out);
            stuffdot(nextcase);
        }
        stuffdot(leave);
        emitf(Xpopm);
    }
```

Uses ARGLIST, Xcase() 105a, Xjump() 101g, Xmark() 77b, Xpopm() 104a, c0-61 80a, c1-62 80b, emitf-64 75d, emiti-65 75c, iscase() 104b, stuffdot() 87d, and yyerror() 179e.

⟨*function* Xpopm 104a⟩≡ (230b)

```
    void
    Xpopm(void)
    {
        poplist();
    }
```

Uses poplist() 77c.

⟨*function* iscase 104b⟩≡ (236a)

```
    bool
    iscase(tree *t)
    {
        if(t->type!=SIMPLE)
            return false;
```

```
    do { t = c0; } while(t->type==ARGLIST);
    return t->type==WORD && !t->quoted && strcmp(t->str, "case")==0;
}
```

Uses ARGLIST, SIMPLE, WORD, and c0-61 80a.

⟨*function* Xcase 105a⟩≡ (230b)

```
void
Xcase(void)
{
    word *p;
    char *s;
    bool ok = false;

    s = list2str(runq->argv->next->words);
    for(p = runq->argv->words;p;p = p->next){
        if(match(s, p->word, '\0')){
            ok = true;
            break;
        }
    }
    efree(s);
    if(ok)
        runq->pc++;
    else
        runq->pc = runq->code[runq->pc].i;
    poplist();
}
```

Uses efree() 181c, list2str() 89b, poplist() 77c, and runq 35a.

8.5.5 Blocks: '{...}'

Blocks and parenthesized expressions are transparent to the bytecode generator—they simply recurse into their child node. The distinction between PAREN, PCMD, and BRACE matters to the parser (for grouping word lists vs. command groups), but by the time we generate bytecodes, they all just mean "compile the contents."

⟨outcode() *cases* 105b⟩+≡ (76b) ◁103a 105c▷

```
case PAREN:
    outcode(c0, eflag);
    break;
```

⟨outcode() *cases* 105c⟩+≡ (76b) ◁105b 106a▷

```
case PCMD:
case BRACE:
    outcode(c0, eflag);
    break;
```

8.6 Functions

In rc, functions are stored in variables: the Var structure has an fn field pointing to the function's bytecode vector and a pc field indicating where in that vector the function body starts. This design unifies functions and variables in the same namespace and allows functions to be exported to the environment (see Chapter 10).

⟨Var *other fields* 105d⟩≡ (38c) 124b▷

```
code *fn;  /* pointer to function's code vector */
int pc;  /* pc of start of function */
bool fnchanged;
```

8.6.1 Function definitions (`fn <foo> ... }`)

When defining a function with `fn foo {body}`, `Xfn`[107a] stores the current code vector (via `codecopy`[34a]) in the variable named `foo`. This is why bytecode vectors are reference-counted: the function's code vector is shared with the code that defined it. When `fn foo` is used without a body, `Xdelfn`[107b] deletes the function by setting `v->fn` to `nil`.

⟨`outcode()` *cases* 106a⟩+≡ (76b) ◁105c 109b▷
```
case FN:
    emitf(Xmark);
    outcode(c0, eflag); // name(s)
    if(c1){
        emitf(Xfn);
        p = emiti(0);
        emits(fnstr(c1));
        outcode(c1, eflag); // body of the function
        emitf(Xunlocal);    /* get rid of $* */
        emitf(Xreturn);
        stuffdot(p);
    }
    else
        emitf(Xdelfn);
    break;
```

⟨`codefree()` *in loop over code* `cp`, *switch bytecode cases* 106b⟩+≡ (34b) ◁96a 164b▷
```
else if(p->f==Xfn){
          efree(p[2].s);
          p+=2;
      }
```
Uses `Xfn()` 107a and `efree()` 181c.

`fnstr()`[106c] turns a function's AST back into source text, used to store the definition in the environment (under `/env/fn#name`) so child processes inherit it. It reuses the `%t` AST printer (like `simplemung()`[67a]), but first sets the global newline character `nl` to `;` so the body is serialized with semicolons instead of newlines—keeping the whole definition on one logical line that can be saved in an `/env` file and re-parsed later.

⟨*function* `fnstr` 106c⟩≡ (236a)
```
char*
fnstr(tree *t)
{
    void *v;
    extern char nl;
    char svnl = nl;
    io *f = openstr();

    nl = ';';
    pfmt(f, "%t", t);
    nl = svnl;
    v = f->strp;
    f->strp = nil;
    closeio(f);
    return v;
}
```
Uses `closeio()` 189a, `nl` 173a, `openstr()` 189b, and `pfmt()` 190b.

The key move in `Xfn()`[107a] is that it stores a pointer to the *same* bytecode vector the surrounding script is currently executing, plus an offset into it. It does not compile a separate closure object. After defining `fn greet { echo hi }`, the global variable `greet` looks like this:

```
var{ name="greet" }
  |
  | v->fn ---------+         v->pc -----+
  | v->pc          |                    |
  +----------------+                    |
                   v                    v
runq->code (shared bytecode vector, refcnt++):
  ... Xmark Xfn <end> "greet;" Xassign ... Xecho "hi" Xunlocal Xreturn ...
              ^         ^     ^                        ^
              |         |     |                        |
              pc here   |     v->pc points here        Xreturn ends call
              at Xfn    stuffdot patches this
                        to "end of body"
```

The `end` slot was emitted as a forward reference by `outcode()` and patched by `stuffdot(p)` once the body had been emitted. At run time, defining the function is almost a no-op for the enclosing script: `Xfn` just records the body's location in the variable and uses `end` to *skip over* the body (`runq->pc = end`). The body runs only later, when the function is invoked: `execfunc()`[108c] calls `start(func->fn, func->pc, ...)`, creating a new thread that resumes just after the `Xfn` and runs to the matching `Xreturn`. Since several variables can share one body (`fn a b c { ... }` defines three functions from a single vector), that vector is reference-counted: `codecopy` bumps the count for each variable that points at it, and `codefree()` drops it when a function is redefined or deleted.

⟨*function* `Xfn` 107a⟩≡ (230b)

```
void
Xfn(void)
{
    var *v;
    word *a;
    int end;

    end = runq->code[runq->pc].i;
    globlist();
    for(a = runq->argv->words;a;a = a->next){
        v = gvlook(a->word);
        if(v->fn)
            codefree(v->fn);
        v->fn = codecopy(runq->code);
        v->pc = runq->pc+2;
        v->fnchanged = true;
    }
    runq->pc = end;
    poplist();
}
```

Uses `codecopy()` 34a, `codefree()` 34b, `globlist()` 134b, `gvlook()` 39f, `poplist()` 77c, and `runq` 35a.

⟨*function* `Xdelfn` 107b⟩≡ (230b)

```
void
Xdelfn(void)
{
    var *v;
    word *a;

    for(a = runq->argv->words;a;a = a->next){
        v = gvlook(a->word);
        if(v->fn)
```

```
            codefree(v->fn);
        v->fn = nil;
        v->fnchanged = true;
    }
    poplist();
}
```

Uses codefree() 34b, gvlook() 39f, poplist() 77c, and runq 35a.

8.6.2 Function uses (<foo>(...))

When Xsimple()[81] detects that the command name matches a function, it calls execfunc()[108c]. This function pops the command arguments from argv, starts a new thread with the function's bytecodes, and binds the arguments to a local $* variable. This is how function arguments work in rc: inside a function, $* contains the arguments passed to the function, shadowing the script's global $*.

⟨Xsimple() *other locals* 108a⟩≡ (81) 115c▷

```
var *v;
```

⟨Xsimple() *if argv0 is a function* 108b⟩≡ (81)

```
v = gvlook(a->word);
if(v->fn)
    execfunc(v);
```

Uses execfunc() 108c and gvlook() 39f.

⟨*function* execfunc 108c⟩≡ (234c)

```
void
execfunc(var *func)
{
    word *starval;

    popword();
    starval = runq->argv->words;
    runq->argv->words = nil;
    poplist();
    start(func->fn, func->pc, runq->local);
    runq->local = newvar(strdup("*"), runq->local);
    runq->local->val = starval;
    runq->local->changed = true;
}
```

Uses newvar() 40b, poplist() 77c, popword() 77d, runq 35a, and start() 45b.

8.7 Variables

We have already met variables in passing—$* bound as a local inside functions, and the special variables of the environment. This section collects how variable *definitions* and *uses* compile to bytecode, starting with assignment.

8.7.1 Variable definitions (<x>=...)

Variable assignment has two modes: when followed by a command (e.g., x=1 cmd), the assignment is *local* to that command (using Xlocal[110a]/Xunlocal[110b]); when standalone (e.g., x=1), it is *global* (using Xassign[109c]). The code detects which mode by following the chain of '=' nodes through child[2]: if it eventually reaches a

non-assignment node, there is a command and the assignments are local. Multiple chained assignments like `A=b C=d cmd` are supported: each creates a local binding, and they are all "unlocalized" after the command finishes.

⟨`outcode()` *locals* 109a⟩+≡ (76b) ◁95a
```
tree *tt;
```

⟨`outcode()` *cases* 109b⟩+≡ (76b) ◁106a 110c▷
```
case '=':
    tt = t;
    for(;t && t->type=='='; t = c2);

    if(t){                  /* var=value cmd */
        for(t = tt;t->type=='=';t = c2){
            emitf(Xmark);
            outcode(c1, eflag);
            emitf(Xmark);
            outcode(c0, eflag);
            emitf(Xlocal);      /* push var for cmd */
        }
        outcode(t, eflag);      /* gen. code for cmd */
        for(t = tt; t->type == '='; t = c2)
            emitf(Xunlocal);    /* pop var */
    }
    else{                   /* var=value */
        for(t = tt;t;t = c2){
            emitf(Xmark);
            outcode(c1, eflag);
            emitf(Xmark);
            outcode(c0, eflag);
            emitf(Xassign); /* set var permanently */
        }
    }
    t = tt; /* so tests below will work */
    break;
```

⟨*function* `Xassign` 109c⟩≡ (230b)
```
void
Xassign(void)
{
    var *v;
    if(count(runq->argv->words)!=1){
        Xerror1("variable name not singleton!");
        return;
    }
    deglob(runq->argv->words->word); // remove the special \001 mark
    v = vlook(runq->argv->words->word);
    poplist();

    globlist();
    freewords(v->val);
    v->val = runq->argv->words;
    v->changed = true;
    runq->argv->words = nil;
    poplist();
}
```
Uses `Xerror1()` 82c, `count()` 37a, `deglob()` 137a, `freewords()` 37b, `globlist()` 134b, `poplist()` 77c, `runq` 35a, and `vlook()` 39c.

⟨*function* `Xlocal` 110a⟩≡ (230b)

```
void
Xlocal(void)
{
    if(count(runq->argv->words)!=1){
        Xerror1("variable name must be singleton\n");
        return;
    }
    deglob(runq->argv->words->word);
    runq->local = newvar(strdup(runq->argv->words->word), runq->local);
    poplist();
    globlist();
    runq->local->val = runq->argv->words;
    runq->local->changed = true;
    // change ownership
    runq->argv->words = nil;
    poplist();
}
```

Uses `Xerror1()` 82c, `count()` 37a, `deglob()` 137a, `globlist()` 134b, `newvar()` 40b, `poplist()` 77c, and `runq` 35a.

⟨*function* `Xunlocal` 110b⟩≡ (230b)

```
void
Xunlocal(void)
{
    var *v = runq->local, *hid;
    if(v==0)
        panic("Xunlocal: no locals!", 0);
    runq->local = v->next;
    hid = vlook(v->name);
    hid->changed = true;
    efree(v->name);
    freewords(v->val);
    efree((char *)v);
}
```

Uses `efree()` 181c, `freewords()` 37b, `panic()` 180c, `runq` 35a, and `vlook()` 39c.

8.7.2 Variable uses (`$<x>`)

`Xdol()` (dollar) handles variable expansion. It looks up the variable name and pushes its value (a list of words) onto the argument stack. A subtle feature: if the variable name is a number (e.g., `$1`, `$2`), `Xdol()` treats it as a positional parameter, indexing into `$*` instead of looking up a named variable. This is how a function (or a script) reads its arguments: inside `fn f { echo $1 }`, `$1` is the first word passed to `f`, `$2` the second, and so on, with `$*` the whole list.

⟨`outcode()` *cases* 110c⟩+≡ (76b) ◁109b 111a▷

```
case '$': //$
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xdol);
    break;
```

⟨*function* `Xdol` 110d⟩≡ (230b)

```
void
Xdol(void)
{
    word *a, *star;
    char *s, *t;
    int n;
```

```
    if(count(runq->argv->words)!=1){
        Xerror1("variable name not singleton!");
        return;
    }
    s = runq->argv->words->word;
    deglob(s);

    n = 0;
    for(t = s;'0'<=*t && *t<='9';t++)
        n = n*10+*t-'0';

    a = runq->argv->next->words;
    if(n==0 || *t)
        a = copywords(vlook(s)->val, a);
    else{
        star = vlook("*")->val;
        if(star && 1 <= n && n <= count(star)){
            while(--n)
                star = star->next;
            a = newword(star->word, a);
        }
    }
    poplist();
    runq->argv->words = a;
}
```

8.7.3 Variable count ($#<var>)

`$#x` gives the number of elements in the list variable `x`. It compiles like `$x` but with `Xcount` in place of `Xdol`: `Xcount` looks up the variable and pushes the `count()` of its word list as a string. (For a numeric name like `$#1` it instead reports whether that positional argument exists, 1 or 0.)

⟨`outcode()` *cases* 111a⟩+≡ (76b) ◁110c 112a▷

```
case COUNT:
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xcount);
    break;
```

⟨*function* `Xcount` 111b⟩≡ (230b)

```
void
Xcount(void)
{
    word *a;
    char *s, *t;
    int n;
    char num[12];

    if(count(runq->argv->words)!=1){
        Xerror1("variable name not singleton!");
        return;
    }
    s = runq->argv->words->word;
    deglob(s);

    // n = int_of_string(s)
    n = 0;
    for(t = s;'0'<=*t && *t<='9';t++)
```

```
        n = n*10 + *t - '0';
    if(n==0 || *t){
        a = vlook(s)->val;
        inttoascii(num, count(a));
    }
    else{
        a = vlook("*")->val;
        inttoascii(num, (a && 1<=n && n<=count(a)) ? 1 : 0);
    }
    poplist();
    pushword(num);
}
```

8.7.4 Variable subscripting ($<var>(<idx>))

Subscripting ($x(2) or $x(2-5)) extracts elements from a list variable by index or range. subwords()[112c] does the heavy lifting: it parses each subscript (which can be a single number or a n-m range, where a bare n- means "from n to the end"), walks n-1 links into the variable's value list, and calls copynwords()[37c] to extract the slice. The recursion on sub->next handles multiple subscripts like $x(1 3 5), accumulating results in reverse so the final list comes out in the right order.

⟨outcode() *cases* 112a⟩+≡ (76b) ◁111a 113a▷

```
case SUB:
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xmark);
    outcode(c1, eflag);
    emitf(Xsub);
    break;
```

⟨*function* Xsub 112b⟩≡ (230b)

```
void
Xsub(void)
{
    word *a, *v;
    char *s;
    if(count(runq->argv->next->words)!=1){
        Xerror1("variable name not singleton!");
        return;
    }
    s = runq->argv->next->words->word;
    deglob(s);
    a = runq->argv->next->next->words;
    v = vlook(s)->val;
    a = subwords(v, count(v), runq->argv->words, a);
    poplist();
    poplist();
    runq->argv->words = a;
}
```

Uses Xerror1() 82c, count() 37a, deglob() 137a, poplist() 77c, runq 35a, and vlook() 39c.

⟨*function* subwords 112c⟩≡ (230b)

```
word*
subwords(word *val, int len, word *sub, word *a)
{
    int n, m;
    char *s;
    if(!sub)
```

```
        return a;
    a = subwords(val, len, sub->next, a);
    s = sub->word;
    deglob(s);
    m = 0;
    n = 0;
    while('0'<=*s && *s<='9')
        n = n*10+ *s++ -'0';
    if(*s == '-'){
        if(*++s == 0)
            m = len - n;
        else{
            while('0'<=*s && *s<='9')
                m = m*10+ *s++ -'0';
            m -= n;
        }
    }
    if(n<1 || n>len || m<0)
        return a;
    if(n+m>len)
        m = len-n;
    while(--n > 0)
        val = val->next;
    return copynwords(val, a, m+1);
}
```

8.8 Concatenation: <x1> ^ <x2>

Concatenation with ^ is far more than string joining. When one side is a single word and the other a list, the word is glued onto *every* element—so `foo/^$list` prepends `foo/` to each member of `$list` (the shell's equivalent of a map). When both sides are equal-length lists, ^ joins them element by element (a zip), so `$a^$b` pairs them up; mismatched non-singleton lengths are an error. Most uses of ^ are in fact *implicit*: the lexer inserts a ^ between adjacent words that touch, so `-x$flag` is really `-x ^ $flag`.

⟨outcode() *cases* 113a⟩+≡ (76b) ◁112a 151b▷

```
case '^':
    emitf(Xmark);
    outcode(c1, eflag);
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xconc);
    break;
```

⟨*function* Xconc 113b⟩≡ (230b)

```
void
Xconc(void)
{
    word *lp = runq->argv->words;
    word *rp = runq->argv->next->words;
    word *vp = runq->argv->next->next->words;
    int lc = count(lp), rc = count(rp);

    if(lc!=0 || rc!=0){
        if(lc==0 || rc==0){
            Xerror1("null list in concatenation");
            return;
        }
        if(lc!=1 && rc!=1 && lc!=rc){
```

```
            Xerror1("mismatched list lengths in concatenation");
            return;
        }
        vp = conclist(lp, rp, vp);
    }
    poplist();
    poplist();
    runq->argv->words = vp;
}
```

Uses Xerror1() 82c, conclist() 114, count() 37a, poplist() 77c, and runq 35a.

⟨*function* conclist 114⟩≡ (230b)

```
word*
conclist(word *lp, word *rp, word *tail)
{
    char *buf;
    word *v;

    if(lp->next || rp->next)
        tail = conclist(lp->next==0? lp: lp->next,
            rp->next==0? rp: rp->next, tail);
    buf = emalloc(strlen(lp->word)+strlen((char *)rp->word)+1);
    strcpy(buf, lp->word);
    strcat(buf, rp->word);
    v = newword(buf, tail);
    efree(buf);
    return v;
}
```

Uses conclist() 114, efree() 181c, emalloc() 181a, and newword() 36c.

Chapter 9

Builtins

Shell builtins are commands that must be executed inside the shell process itself, rather than in a forked child. The most obvious example is `cd`: if it were an external program, the `chdir()` would happen in the child process and the parent (the shell) would remain in the old directory.

9.1 Overview

Builtins are stored in a simple array of name/function pairs. When `Xsimple()`[81] detects that the command name matches a builtin, it calls the corresponding function directly.

⟨*struct* `Builtin` 115a⟩≡ (225)
```
struct Builtin {
    char *name;
    void (*fnc)(void);
};
```

⟨*global* `builtins` 115b⟩≡ (237b)
```
builtin builtins[] = {
    "cd"     ,      execcd,
    "exit"   ,      execexit,
    "."      ,      execdot,
    "eval"   ,      execeval,

    "whatis" ,      execwhatis,

    "exec"   ,      execexec,   /* but with popword first */
    "rfork"  ,      execnewpgrp,
    "wait"   ,      execwait,

    "shift"  ,      execshift,
    "finit"  ,      execfinit,
    "flag"   ,      execflag,
    //TODO: unix-specific "umask", execumask,
    0
};
```
Uses `execcd()` 116c, `execdot()` 119, `execeval()` 121b, `execexec()` 83b, `execexit()` 118c, `execfinit()` 126b, `execflag()` 168, `execnewpgrp()` 167, and `execwait()` 122b.

⟨`Xsimple()` *other locals* 115c⟩+≡ (81) ◁108a
```
struct Builtin *bp;
```
Uses `Builtin` 115a.

⟨Xsimple() *if argv0 is a builtin* 116a⟩≡ (81)
```
⟨Xsimple() if argv0 is the builtin keyword 116b⟩
for(bp = builtins;bp->name;bp++)
    if(strcmp(a->word, bp->name)==0){
        (*bp->fnc)();
        return;
    }
```
Uses builtins 115b.

The `builtin` keyword allows bypassing a function that shadows a builtin: `builtin cd` always calls the builtin `cd`, even if a function named `cd` exists.

⟨Xsimple() *if argv0 is the builtin keyword* 116b⟩≡ (116a)
```
if(strcmp(a->word, "builtin")==0){
    if(count(a)==1){
        pfmt(err, "builtin: empty argument list\n");
        setstatus("empty arg list");
        poplist();
        return;
    }
    a = a->next;
    popword();
}
```
Uses count() 37a, err 179b, pfmt() 190b, poplist() 77c, popword() 77d, and setstatus() 78a.

We now look at the most important builtins one at a time, each implemented by an `execxxx()` function (`execcd()`[116c], `execexit()`[118c], `execdot()`[119], ...) that `Xsimple()` dispatches to when the command name matches one in the builtin table.

9.2 % cd

`cd` with no argument changes to `$home`. With one argument, it searches the `$cdpath` directories (similar to how `$path` works for commands). When the directory is found via `$cdpath`, `cd` prints the resolved path so the user knows where they ended up.

⟨*function* execcd 116c⟩≡ (237b)
```
void
execcd(void)
{
    word *a = runq->argv->words;
    word *cdpath;
    char *dir;

    setstatus("can't cd");
    cdpath = vlook("cdpath")->val;

    switch(count(a)){
    case 1:
        a = vlook("home")->val;
        if(count(a)>=1){
            if(dochdir(a->word)>=0)
                setstatus("");
            else
                pfmt(err, "Can't cd %s: %r\n", a->word);
        }
        else
            pfmt(err, "Can't cd -- $home empty\n"); //$
        break;
    case 2:
```

```
        if(a->next->word[0]=='/' || cdpath==nil)
            cdpath = &nullpath;
        for(; cdpath; cdpath = cdpath->next){
            if(cdpath->word[0] != '\0')
                dir = appfile(cdpath->word, a->next->word);
            else
                dir = strdup(a->next->word);

            if(dochdir(dir) >= 0){
                if(cdpath->word[0] != '\0' &&
                    strcmp(cdpath->word, ".") != 0)
                    pfmt(err, "%s\n", dir);
                free(dir);
                setstatus("");
                break;
            }
            free(dir);
        }
        if(cdpath==nil)
            pfmt(err, "Can't cd %s: %r\n", a->next->word);
        break;
    default:
        pfmt(err, "Usage: cd [directory]\n");
        break;
    }
    poplist();
}
```

Uses `appfile()` 117a, `count()` 37a, `dochdir()` 117b, `err` 179b, `nullpath` 84b, `pfmt()` 190b, `poplist()` 77c, `runq` 35a, `setstatus()` 78a, and `vlook()` 39c.

`appfile()`[117a] joins a directory and a path component with a `/`, returning a freshly allocated string (the Plan 9 equivalent of UNIX's path join). `cd` uses it to build each candidate directory when searching `$cdpath`.

⟨*function* `appfile` 117a⟩≡ (237b)

```
static char *
appfile(char *dir, char *comp)
{
    int dirlen, complen;
    char *s, *p;

    dirlen = strlen(dir);
    complen = strlen(comp);
    s = emalloc(dirlen + 1 + complen + 1);
    memmove(s, dir, dirlen);
    p = s + dirlen;
    *p++ = '/';
    memmove(p, comp, complen);
    p[complen] = '\0';
    return s;
}
```

Uses `emalloc()` 181a.

⟨*function* `dochdir` 117b⟩≡ (237b)

```
errorneg1
dochdir(char *word)
{
    ⟨dochdir() locals 118a⟩

    // the actual syscall
    if(chdir(word)<0)
```

```
        return ERROR_NEG1;

    ⟨dochdir() adjust /dev/wdir if run under rio 118b⟩
    return OK_1;
}
```

In Plan 9, the window manager `rio` (see WINDOWS book [Pad16e]) tracks the current working directory of each window via the `/dev/wdir` file. When `cd` succeeds in an interactive shell, it writes the new directory to this file so that `rio` can display it in the window title or use it when spawning new windows. The file descriptor is cached in a static variable and opened only once (the `-2` sentinel means "not yet tried").

⟨dochdir() *locals* 118a⟩≡ (117b)
```
/* report to /dev/wdir if it exists and we're interactive */
static fdt wdirfd = -2;
```

⟨dochdir() *adjust* /dev/wdir *if run under* rio 118b⟩≡ (117b)
```
if(flag['i']!=nil){
    if(wdirfd==-2)  /* try only once */
        wdirfd = open("/dev/wdir", OWRITE); // TODO: |OCEXEC but plan9 specific?
    if(wdirfd>=0) {
        //fcntl(wdirfd, F_SETFD, FD_CLOEXEC);
        write(wdirfd, word, strlen(word));
    }
}
```
Uses `flag` 43a.

9.3 % exit

`execexit()`[118c] implements the `exit` builtin: with an optional status argument it records that status, then calls `Xexit`[96b] to terminate the shell—so it goes through the same trap handling and `Exit()`[180e] cleanup as any other exit.

⟨*function* execexit 118c⟩≡ (237b)
```
void
execexit(void)
{
    switch(count(runq->argv->words)){
    default:
        pfmt(err, "Usage: exit [status]\nExiting anyway\n");
        // FALLTHROUGH
    case 2:
        setstatus(runq->argv->words->next->word);
        // FALLTHROUGH
    case 1:
        Xexit();
    }
}
```
Uses `Xexit()` 96b, `count()` 37a, `err` 179b, `pfmt()` 190b, `runq` 35a, and `setstatus()` 78a.

9.4 % . (source)

The . (dot) builtin "sources" a script: it reads and executes the commands from a file in the current shell (without forking a new process). This is how `rc` loads initialization files like `rcmain` and the user's profile. It

creates a new thread whose `cmdfd` reads from the sourced file, so `Xrdcmds()`[47a] will read commands from that file instead of the terminal.

⟨*function* `execdot` 119⟩≡ (237b)

```
void
execdot(void)
{
    thread *p = runq;
    bool iflag = false;
    fdt fd;
    list *av;
    char *zero; // new stdin
    char *file;
    word *path;
    ⟨execdot() other locals 120c⟩

    ⟨execdot() if first 120d⟩
    ⟨execdot() if not first execution 149c⟩

    popword(); // "."
    if(p->argv->words && strcmp(p->argv->words->word, "-i")==0){
        iflag = true;
        popword();
    }

    /* get input file */
    if(p->argv->words==nil){
        Xerror1("Usage: . [-i] file [arg ...]");
        return;
    }
    zero = strdup(p->argv->words->word);
    popword();

    fd = -1;
    for(path = searchpath(zero); path; path = path->next){
        if(path->word[0] != '\0')
            file = appfile(path->word, zero);
        else
            file = strdup(zero);

        fd = open(file, 0);
        free(file);
        if(fd >= 0)
            break;
    }
    if(fd<0){
        pfmt(err, "%s: ", zero);
        setstatus("can't open"); // what for? it is reseted by Xerror anyway
        Xerror(".: can't open");
        return;
    }

    /* set up for a new command loop */
    start(dotcmds, 1, (struct Var *)nil);

    pushredir(RCLOSE, fd, 0);
    runq->cmdfile = zero;
    runq->cmdfd = openfd(fd);
    runq->iflag = iflag;
```

```
    runq->iflast = false;

    /* push $* value */
    pushlist();
    runq->argv->words = p->argv->words;

    /* free caller's copy of $* */
    av = p->argv;
    p->argv = av->next;
    efree((char *)av);

    /* push $0 value */
    pushlist();
    pushword(zero);

    ndot++;
}
```

Uses RCLOSE 164d, Var 38c, Xerror() 82a, Xerror1() 82c, appfile() 117a, dotcmds 120b, efree() 181c, err 179b, ndot 120a, openfd() 186h, pfmt() 190b, popword() 77d, pushlist() 46a, pushredir() 90f, pushword() 46b, runq 35a, searchpath() 84a, setstatus() 78a, and start() 45b.

⟨*global* ndot 120a⟩≡ (226)

```
/*
 * How many dot commands have we executed?
 * Used to ensure that -v flag doesn't print rcmain.
 */
int ndot;
```

⟨*global* dotcmds 120b⟩≡ (237b)

```
union Code dotcmds[14];
```

Uses Code 33c.

⟨execdot() *other locals* 120c⟩≡ (119)

```
static bool first = true;
```

⟨execdot() *if first* 120d⟩≡ (119)

```
if(first){
    dotcmds[0].i = 1;
    dotcmds[1].f = Xmark;
      dotcmds[2].f = Xword;
      dotcmds[3].s = "0";
    dotcmds[4].f = Xlocal; // will pop_list twice

    dotcmds[5].f = Xmark;
      dotcmds[6].f = Xword;
      dotcmds[7].s="*";
    dotcmds[8].f = Xlocal; // will pop_list twice

    dotcmds[9].f = Xrdcmds; // =~ a REPL

    dotcmds[10].f = Xunlocal;
    dotcmds[11].f = Xunlocal;
    dotcmds[12].f = Xreturn;

    first = false;
}
```

Uses Xlocal() 110a, Xmark() 77b, Xrdcmds() 47a, Xreturn() 96c, Xunlocal() 110b, Xword() 77a, and dotcmds 120b.

9.5 % eval

`eval` takes a string and executes it as a command. This is useful when the command to execute is constructed dynamically (e.g., from a variable). The standard use is option parsing: a script declares its flags and runs `aux/getflags` (Appendix E) at the top:

⟨*myscript* 121a⟩≡
```
#!/bin/rc
flagfmt='v,o output'                    # -v boolean, -o takes an arg
eval `{aux/getflags $*} || aux/usage
...
# use $flagv, $flago, etc.
```

where `getflags` prints an `rc` script of assignments that `eval` executes. After those two lines, running the script as `myscript -v -o log foo` leaves the shell as if one had written

```
flagv=1   flago=(log)   *=(foo)
```

so the rest of the script simply reads `$flagv`, `$flago`, and `$*`. Without `eval`, the substituted `flagv=1` would instead be taken as the name of a command to run.

`execeval()`[121b] joins the arguments into a single string, then creates a new thread reading from that string (via `opencore()`[189c], which creates a buffered IO from memory).

⟨*function* `execeval` 121b⟩≡ (237b)
```
void
execeval(void)
{
    char *cmdline, *s, *t;
    int len = 0;
    word *ap;

    if(count(runq->argv->words)<=1){
        Xerror1("Usage: eval cmd ...");
        return;
    }
    eflagok = true;
    for(ap = runq->argv->words->next;ap;ap = ap->next)
        len+=1+strlen(ap->word);

    cmdline = emalloc(len);
    s = cmdline;
    for(ap = runq->argv->words->next;ap;ap = ap->next){
        for(t = ap->word;*t;) *s++=*t++;
        *s++=' ';
    }
    s[-1]='\n';
    poplist();

    execcmds(opencore(cmdline, len));
    efree(cmdline);
}
```

Uses `Xerror1()` 82c, `count()` 37a, `eflagok` 149b, `efree()` 181c, `emalloc()` 181a, `execcmds()` 122a, `opencore()` 189c, `poplist()` 77c, and `runq` 35a.

⟨*global* `rdcmds` 121c⟩≡ (234c)
```
union Code rdcmds[4];
```

Uses `Code` 33c.

⟨*function* `execcmds` 122a⟩≡ (234c)

```
void
execcmds(io *f)
{
    static bool first = true;
    if(first){
        rdcmds[0].i = 1;
        rdcmds[1].f = Xrdcmds;
        rdcmds[2].f = Xreturn;
        first = false;
    }

    start(rdcmds, 1, runq->local);
    runq->cmdfd = f;
    runq->iflast = false;
}
```

Uses `Xrdcmds()` 47a, `Xreturn()` 96c, `rdcmds` 121c, `runq` 35a, and `start()` 45b.

9.6 % wait

`execwait()`[122b] implements the `wait` builtin: with a `pid` argument it waits for that process, and with no argument it waits for *all* of the shell's children (`Waitfor(-1, ...)`). It is the companion to `&` (background execution): after `cmd &`, you call `wait $apid` to block until the job finishes. This is `rc`'s minimal substitute for `bash`-style job control.

⟨*function* `execwait` 122b⟩≡ (237b)

```
void
execwait(void)
{
    switch(count(runq->argv->words)){
    default:
        Xerror1("Usage: wait [pid]");
        return;
    case 2:
        Waitfor(atoi(runq->argv->words->next->word), false);
        break;
    case 1:
        Waitfor(-1, false);
        break;
    }
    poplist();
}
```

Uses `Xerror1()` 82c, `count()` 37a, `poplist()` 77c, and `runq` 35a.

`rc` has more builtins than the few shown here—`shift`, `whatis`, `flag`, `rfork`, and others appear later in Chapter 14—but `cd`, `exit`, `eval`, and `wait` are the ones worth understanding in detail. `rc` also keeps this set deliberately small: commands that other shells build in, such as `read` and `test`, are ordinary external programs in Plan 9 (you write `x = '{read}`, whose code is in Appendix E.

Chapter 10

Environment

Under Plan 9 each environment variable is stored as a file under `/env/`: the variable `$path` is stored in `/env/path`. This is a more elegant design than UNIX because child processes automatically inherit the environment through the *namespace* (see the KERNEL book [Pad14]), and changes are visible to all processes sharing the same namespace. In UNIX the environment is part of the process *image*: `exec` copies the `environ` array (alongside `argv`) into the new process, so a child gets its own copy and can never change its parent's environment. Under Plan 9, because `/env` is just shared state in the namespace, a child *can* modify variables that its parent and siblings will then see—more flexibility. A process that instead wants a private environment forks with `rfork(RFENVG)`, which gives it its own copy of `/env`.

`rc` must synchronize its internal variable table with these files in both directions: `Updenv()`[123a] writes changed variables to `/env/`, and `Vinit()`[124d] reads variables from `/env/` at startup.

10.1 Writing variables to /env/: Updenv()

`Updenv()`[123a] iterates over all global and local variables and writes any that have changed (with `v->changed == true`) to the `/env/` filesystem. The `changed` flag avoids redundant writes.

⟨*function* Updenv 123a⟩≡ (240b)
```
/// Xsimple | Execute -> <>
void
Updenv(void)
{
    var *v, **h;

    for(h = gvar;h!=&gvar[NVAR];h++)
        for(v=*h;v;v = v->next)
            addenv(v);
    if(runq)
        updenvlocal(runq->local);
}
```
Uses NVAR 39d, addenv() 124c, gvar 39e, runq 35a, and updenvlocal() 123b.

⟨*function* updenvlocal 123b⟩≡ (240b)
```
void
updenvlocal(var *v)
{
    if(v){
        updenvlocal(v->next);
        addenv(v);
    }
}
```
Uses addenv() 124c and updenvlocal() 123b.

⟨*enum* `MiscPlan9` 124a⟩≡ (240b)
```
enum {
    Maxenvname = 256, /* undocumented limit */
};
```

⟨`Var` *other fields* 124b⟩+≡ (38c) ◁105d
```
bool changed;
```

⟨*function* `addenv` *(plan9.c)* 124c⟩≡ (240b)
```
void
addenv(var *v)
{
    char envname[Maxenvname];
    word *w;
    int f;
    io *fd;

    if(v->changed){
        v->changed = false;
        snprint(envname, sizeof envname, "/env/%s", v->name);
        if((f = Creat(envname))<0)
            pfmt(err, "rc: can't open %s: %r\n", envname);
        else{
            for(w = v->val;w;w = w->next)
                write(f, w->word, strlen(w->word)+1L);
            close(f);
        }
    }
    if(v->fnchanged){
        v->fnchanged = false;
        snprint(envname, sizeof envname, "/env/fn#%s", v->name);
        if((f = Creat(envname))<0)
            pfmt(err, "rc: can't open %s: %r\n", envname);
        else{
            if(v->fn){
                fd = openfd(f);
                pfmt(fd, "fn %q %s\n", v->name, v->fn[v->pc-1].s);
                closeio(fd);
            }
            close(f);
        }
    }
}
```
Uses `Creat()` 193g, `Maxenvname-2` 124a, `closeio()` 189a, `err` 179b, `openfd()` 186h, and `pfmt()` 190b.

10.2 Reading variables from `/env/`: `Vinit()`

`Vinit()`[124d] reads the `/env/` directory at startup and populates the global variable table. Each environment file contains the variable's words separated by null bytes. `Vinit()` reads the file, splits on nulls, and calls `setvar()`[39b] for each variable. It then resets the `changed` flag so that `Updenv()`[123a] does not write them back immediately. The null byte separator is the only safe choice: unlike spaces or newlines, null bytes cannot appear in filenames or variable values, so the encoding is unambiguous. This is the same insight behind UNIX tools like `find -print0` and `xargs -0`.

⟨*function* `Vinit` 124d⟩≡ (240b)
```
/// main -> <>
void
```

```
Vinit(void)
{
    int dir, f, len, i, n, nent;
    char *buf, *s;
    char envname[Maxenvname];
    word *val;
    Dir *ent;

    dir = open("/env", OREAD);
    if(dir<0){
        pfmt(err, "rc: can't open /env: %r\n");
        return;
    }
    ent = nil;
    for(;;){
        nent = dirread(dir, &ent);
        if(nent <= 0)
            break;
        for(i = 0; i<nent; i++){
            len = ent[i].length;
            if(len && strncmp(ent[i].name, "fn#", 3)!=0){
                snprint(envname, sizeof envname, "/env/%s", ent[i].name);
                if((f = open(envname, 0))>=0){
                    buf = emalloc(len+1);
                    n = readn(f, buf, len);
                    if (n <= 0)
                        buf[0] = '\0';
                    else
                        buf[n] = '\0';
                    val = 0;
                    /* Charitably add a 0 at the end if need be */
                    if(buf[len-1])
                        buf[len++]='\0';
                    s = buf+len-1;
                    for(;;){
                        while(s!=buf && s[-1]!='\0') --s;
                        val = newword(s, val);
                        if(s==buf)
                            break;
                        --s;
                    }
                    setvar(ent[i].name, val);
                    vlook(ent[i].name)->changed = false;
                    close(f);
                    efree(buf);
                }
            }
        }
        free(ent);
    }
    close(dir);
}
```

Uses `Maxenvname-2` 124a, `efree()` 181c, `emalloc()` 181a, `err` 179b, `newword()` 36c, `pfmt()` 190b, `setvar()` 39b, and `vlook()` 39c.

⟨*global* `envdir` 125⟩≡ (230b)

```
fdt envdir;
```

10.3 Special variables

The environment is also where `rc`'s *special variables* live: ordinary variables by name, but ones the shell itself consults to decide how it behaves. Here are a few of the most important, many of them given their defaults in `rcmain` (see Section 11.5):

- `$prompt` — the two prompt strings (main and continuation), printed by the input loop (Section 5.2.3).
- `$status` — the exit status of the last command, as a string (`rc`'s answer to the Bourne shell's `$?`), set through `setstatus()`[78a]/`getstatus()`[78b].
- `$home` — the user's home directory, used by `cd` with no argument and to locate `$home/lib/profile`. Unlike the UNIX shells, `rc` offers no `~` shortcut for it; you spell out `$home`.
- `$path` — the list of directories searched for external commands by `searchpath()`[84a].
- `$cdpath` — the list of directories `cd` searches when given a relative directory name, the `cd` analogue of `$path`; `cd` echoes the directory it lands in when it was reached through a `$cdpath` entry rather than the current directory.
- `$*` — the argument list of the script or function; `$1`, `$2`, ... index into it (via `Xdol()`).
- `$ifs` — the field separators used to split the output of a backquote substitution `` `{...} `` into words.

One small but telling difference: `rc` spells these `$home`, `$path`, `$prompt`, where the UNIX shells shout `HOME`, `PATH`, `PS1`. The uppercase convention exists in `sh` only to keep exported environment variables from colliding with the lowercase names a script picks for its own scratch variables. `rc` has no second class of names to stay clear of—every variable already lives in the one `/env` namespace—so it can use the quieter lowercase spellings.

10.4 % finit

The `finit` builtin re-reads function definitions from the `/env` filesystem. This is Plan 9-specific: when another process (e.g., a parent shell) defines a function, it appears as a file `/env/fn#name`. `finit` scans `/env` for these `fn#` files and executes them to import the function definitions into the current shell. It uses a small hand-crafted bytecode loop (`rdfns`) similar to the bootstrap code, with `Xrdfn` reading one function file per iteration.

⟨*global* `rdfns` 126a⟩≡ (237b)
```
union Code rdfns[4];
```
Uses `Code` 33c.

⟨*function* `execfinit` 126b⟩≡ (237b)
```
/// "finit" builtin -> <>
void
execfinit(void)
{
    //TODO: commented for goken for now because Linux does not have a /env
    // probably need to move to Plan9.c
    //static bool first = true;
    //if(first){
    //    rdfns[0].i = 1;
    //    rdfns[1].f = Xrdfn;
    //    rdfns[2].f = Xjump;
    //    rdfns[3].i = 1;
    //    first = false;
    //}
```

```
    //Xpopm(); // pop_list()
    //envdir = open("/env", 0);
    //if(envdir<0){
    //    pfmt(err, "rc: can't open /env: %r\n");
    //    return;
    //}
    //start(rdfns, 1, runq->local);
}
```

⟨*function* `Xrdfn` 127⟩≡ (240b)

```
void
Xrdfn(void)
{
    int f, len;
    Dir *e;
    char envname[Maxenvname];
    static Dir *ent, *allocent;
    static int nent;

    for(;;){
        if(nent == 0){
            free(allocent);
            nent = dirread(envdir, &allocent);
            ent = allocent;
        }
        if(nent <= 0)
            break;
        while(nent){
            e = ent++;
            nent--;
            len = e->length;
            if(len && strncmp(e->name, "fn#", 3)==0){
                snprint(envname, sizeof envname, "/env/%s", e->name);
                if((f = open(envname, 0))>=0){
                    execcmds(openfd(f));
                    return;
                }
            }
        }
    }
    close(envdir);
    Xreturn();
}
```

Uses `Maxenvname-2` 124a, `Xreturn()` 96c, `envdir` 125, `execcmds()` 122a, and `openfd()` 186h.

We have now seen how `finit` works: it scans `/env` for `fn#` files and runs each one to recreate the functions a parent shell exported. What we have not yet seen is `finit` actually being called. That happens in the `/lib/rc/rcmain` startup script of the next chapter, which is also where the rest of `rc`'s initialization is written in `rc` itself.

Chapter 11

Initialization

In Section 4.4, I showed a simplified `bootstrap` to introduce the interpreter loop. This chapter reveals the real initialization sequence: how `rc` fabricates its own bootstrap bytecodes, sources the `rcmain` script to set up the default environment, and transitions into the read-eval-print loop that makes an interactive shell usable.

11.1 Shell initialization across shells

Initialization is the single most confusing topic in the UNIX shell world, and it is worth understanding the problem before looking at how `rc` solves it. A shell can be invoked in three different contexts, each wanting a different subset of setup. A *login session* (the one the display manager or `ssh` starts) needs the full environment configured: `$path`, `$home`, `$term`, locale, mail spool, anything other programs will inherit. An *interactive non-login* shell (every new terminal window you open inside that session) only needs personalization— prompt, aliases, key bindings—because the heavy environment is already inherited from the login shell. A *non-interactive* shell running a script needs neither; both kinds of customization would just slow the script down or, worse, change its behavior in ways the script author did not anticipate.

`bash` handles this by spreading the work across several files and asking the user to remember which file runs in which case. A login shell sources `/etc/profile` then the first of `~/.bash_profile`, `~/.bash_login`, or `~/.profile` that exists. An interactive non-login shell sources `/etc/bash.bashrc` then `~/.bashrc`. A non-interactive shell sources nothing unless `BASH_ENV` is set. The result is that `~/.bashrc` is not read by login shells unless `~/.bash_profile` explicitly sources it. `zsh` has a four-file ladder—`.zshenv` then `.zprofile` then `.zshrc` then `.zlogin`, invoked in that order depending on shell type. `fish` collapses everything into a single `~/.config/fish/config.fish`. None of these are obviously right; they are just compromises between "run the right things" and "stay backwards compatible with /bin/sh".

`rc` takes a much simpler position. Every invocation, login or not, runs exactly one file: `/rc/lib/rcmain` (sourced by the bootstrap bytecodes shown below). For login shells, `rcmain` then sources `~/lib/profile` for user customization (actually `$home/lib/profile` because `~` is not handled by `rc`). That is the entire ladder—two files instead of six, with the same logic regardless of how the shell was started. The price is that there is no separate "interactive only" file: if you want a setting in interactive shells only, you test `$prompt` inside `profile` yourself. The benefit is that nothing surprises you, because nothing depends on which entry point invoked the shell.

The sections below walk through the actual bootstrap bytecodes that source `rcmain`, the `-m` flag for picking a different one, and how the `$home/lib/profile` mechanism plugs into the chain.

11.2 Actual bootstrapping code

The bootstrap performs two operations encoded as hand-crafted bytecodes: first, it assigns the command-line arguments to `$*` (with `*=(argv)`), then it executes `. rcmain $*` to source the initialization script. The

bootstrap is essentially the bytecode equivalent of:

```
*=(argv)
. /rc/lib/rcmain $*
```

The two-line script above is encoded as a sequence of bytecodes filling the `bootstrap` array.

⟨main() *initialize* boostrap 129a⟩≡ (42b)
```
memset(bootstrap, 0, sizeof bootstrap);

i = 0;
bootstrap[i++].i=1; // ref count
// runq->argv is populated with the arguments to rc (argv)
// we just need to add '*=...'
bootstrap[i++].f = Xmark;
  bootstrap[i++].f = Xword;
  bootstrap[i++].s = "*";
bootstrap[i++].f = Xassign; // will pop_list() x2

bootstrap[i++].f = Xmark;
  bootstrap[i++].f = Xmark;
    bootstrap[i++].f = Xword;
    bootstrap[i++].s = "*";
  bootstrap[i++].f = Xdol; // will pop_list()
  bootstrap[i++].f = Xword;
  bootstrap[i++].s = rcmain;
  bootstrap[i++].f = Xword;
  bootstrap[i++].s = ".";
bootstrap[i++].f = Xsimple; // will pop_list()

bootstrap[i++].f = Xexit;

bootstrap[i].i = 0;
```
Uses `Xassign()` 109c, `Xexit()` 96b, `Xmark()` 77b, `Xsimple()` 81, and `Xword()` 77a.

The first group (`Xmark`, `Xword "*"`, `Xassign`) assigns the command-line arguments to `$*`. The second group builds the argument list for the `.` builtin: it pushes `$*` (via `Xdol`), the path to `rcmain`, and the `.` command itself, then calls `Xsimple`.

11.3 Intitialization script and `rc -m /path/to/rcmain`

The default initialization script is `/rc/lib/rcmain`, which sets up `$path` and, for login shells, sources the user's `$home/lib/profile`. The `-m` flag allows overriding this path, which is useful during system bootstrap when the standard `rcmain` may not be available yet (like for `boot.rc` in Section 2.3).

⟨*global* Rcmain 129b⟩≡ (240b)
```
char *Rcmain = "/rc/lib/rcmain";
```

⟨main() *locals* 129c⟩+≡ (42b) ◁44c 130b▷
```
char *rcmain;
```

⟨main() *initialisation* 129d⟩+≡ (42b) ◁44b 130c▷
```
rcmain = flag['m'] ? flag['m'][0] : Rcmain;
```
Uses `flag` 43a.

11.4 Actual environment

Before entering the interpreter loop, `rc` also sets up a few built-in variables such as `$pid` (the shell's own process id, useful for creating unique temporary filenames).

⟨*global* `mypid` 130a⟩≡ (226)
```
int mypid;
```

⟨`main()` *locals* 130b⟩+≡ (42b) ◁129c
```
char num[12];
```

⟨`main()` *initialisation* 130c⟩+≡ (42b) ◁129d 131a▷
```
mypid = getpid();
inttoascii(num, mypid);
setvar("pid", newword(num, (word *)nil));
```
Uses `inttoascii()` 193b, `mypid` 130a, `newword()` 36c, and `setvar()` 39b.

11.5 /rc/lib/rcmain

I can finally show the initialization script sourced by the bootstrap bytecodes. It sets up default values for the special variables `$home`, `$ifs`, `$prompt`, and `$path`, calls `finit` to import shell functions from the environment (see Section 10.4), and then dispatches based on how `rc` was invoked: with `-c` (run a command string), with `-i` (interactive, source the user's profile and read from the console), or with a script file argument.

This short script doubles as a tour of `rc`'s control flow. It uses the `~` match operator to test counts (`~ $#home 0`), a `switch`, and especially `if not`—`rc`'s spelling of "else", which binds to the preceding `if` so that the chain of `if not if` reads as a cascade of conditionals.

⟨*rc/rcmain.rc* 130d⟩≡
```
# /lib/rc/rcmain, Plan 9 version
if(~ $#home 0) home=/
# space, tab, newline
if(~ $#ifs 0) ifs='
'

switch($#prompt){
case 0
        prompt=('% ' '  ')
case 1
        prompt=($prompt '       ')
}
⟨rcmain.rc possibly set broken prompt 131b⟩

if(flag p) path=/bin
if not{
        finit
        if(~ $#path 0) path=(. /bin)
}

⟨rcmain.rc delete sigexit 131c⟩

⟨rcmain.rc if -c flag 148b⟩
if not if(flag i){
        ⟨rcmain.rc if -l, source user's profile 195a⟩
        status=''
        # if arguments, probably a script, source the script (and pass args)
        if(! ~ $#* 0) . $*
        # otherwise interactive shell! source stdin
```

```
        . -i '#d/0'
}
if not if(~ $#* 0) . '#d/0'
if not{
        status=''
        . $*
}
exit $status
```

Notice the division of labor. The C-level initialization—reading the environment into globals (`Vinit`), installing keywords and the note handler—all ran back in `main()`[42b] before the bootstrap code fired. What you are reading now is the rest of the initialization, written in `rc` itself: a small script that the freshly built interpreter runs as its very first program.

The `-p` flag (tested here as `flag p`, see Section 14.5.4 for the code of this builtin) selects a more cautious `rc`. Normally the shell calls `finit`, which imports function definitions from `/env` (see Section 10.4)—the exported functions of whatever started it—and defaults `$path` to `(. /bin)`, current directory included. With `-p` neither happens: `$path` becomes just `/bin` and no functions are inherited. This matters in an untrusted context (say a set-user-id script), where an attacker-supplied function in the environment, or a `.` early in `$path`, could hijack a plain command name.

The file `#d/0` above is the `#d` file-descriptor device opened at entry 0—the shell's own standard input. Sourcing it with `. -i` turns that input into an interactive command loop: the `-i` flag makes the nested `.` set `iflag` on its `runq`, so it prints a prompt between commands and does not abort on the first error—exactly what is wanted when a human is typing. The same file sourced without `-i` (the `. '#d/0'` on the lines below and before, for a non-interactive `rc` reading a piped script) runs silently and stops at end of input.

`$rcname` below records the name `rc` was invoked as (`argv[0]`); the chunk just below uses it to notice when the shell is a freshly compiled, not-yet-installed binary (the default `?.out` output name) and warns you with a "broken!" prompt.

⟨`main()` *initialisation* 131a⟩+≡ (42b) ◁130c 148a▷

```
setvar("rcname", newword(argv[0], (word *)nil));
```

Uses `newword()` 36c and `setvar()` 39b.

⟨`rcmain.rc` *possibly set broken prompt* 131b⟩≡ (130d)

```
if(~ $rcname ?.out) prompt=('broken! ' ' ')
```

`finit` just imported every exported function from the environment, and one of those may be `sigexit`—the function `rc` runs when it exits. A child shell should not silently inherit and re-run its parent's exit handler, so `fn sigexit` with an empty body deletes any such inherited definition.

⟨`rcmain.rc` *delete* `sigexit` 131c⟩≡ (130d)

```
fn sigexit
```

Chapter 12

Globbing

Globbing, also called wildcard expansion or pathname expansion, is one of the key features of a shell: when you write `ls *.c`, the shell *expands* `*.c` into the list of matching filenames *before* passing them to `ls`. This is a deliberate design choice: the expansion is done by the shell, not by the individual programs, which factorizes the functionality.

The glob patterns `*`, `?`, and `[]` are different from regular expressions: `*` matches any string (not "any number of the preceding character"), and `?` matches any single character. This is more practical for filenames, where `*.c` is more natural than the regex `.*\.c`.

Globbing touches several phases of the interpreter. The lexer marks glob characters with a special escape byte so they can be distinguished from literal characters. The bytecode generator emits `Xglob`[133d] instructions at the right points. And at runtime, the actual matching is done by `globlist()`[134b], which walks directories and filters filenames against the pattern using `matchfn()`[138a], as explained in the following sections.

12.1 Lexing globbing characters

Glob characters (`*`, `?`, `[]`) are marked during lexing by inserting a `GLOB` byte (`\001`) before them. This way, the rest of the shell can distinguish a literal `*` (inside quotes, no `GLOB` prefix) from a wildcard `*` (with a `GLOB` prefix). The `deglob()`[137a] function removes these markers when they are no longer needed.

Concretely, here is how the two patterns `*.c` and `'*.c'` end up in memory after lexing. Both produce a single `Word`, but the bytes differ:

```
wildcard *.c  ->   byte:  \001  '*'  '.'  'c'  '\0'
                          ^^^^   ^
                          GLOB   wild

quoted '*.c'  ->   byte:  '*'  '.'  'c'  '\0'
                          (no GLOB marker, literal star)
```

The *in-band* marker solves a real tension: the shell needs to carry these strings through `^` concatenation, variable substitution, and command substitution without losing the "is this `*` a wildcard?" bit. Using a parallel bitmap or a separate struct would require every string-manipulation function (`strdup`, `strcat`, `emalloc`) to also know about the bitmap. The `\001` byte sneaks alongside the normal UTF-8 bytes (which never contain `\001` because that is a C0-control code and not a valid start byte for any Rune sequence) so that ordinary `strcpy` just works. The only cost is that `match()`[138b] must skip over the marker, and `deglob()` must strip it once matching is done.

⟨*constant* `GLOB` 132⟩≡ (223b)

```
/*
 * Glob character escape in strings:
```

```
 * In a string, GLOB must be followed by *?[ or GLOB.
 * GLOB* matches any string
 * GLOB? matches any single character
 * GLOB[...] matches anything in the brackets
 * GLOBGLOB matches GLOB
 */
#define GLOB '\001'
```

⟨yylex() *when c is a word character, if glob character* 133a⟩≡ (62c)
```
if(c=='*' || c=='[' || c=='?' || c==GLOB)
    w = addtok(w, GLOB);
```
Uses GLOB 132 and addtok() 133b.

⟨*function* addtok 133b⟩≡ (234a)
```
char*
addtok(char *p, int val)
{
    ⟨addtok() sanity check p 133c⟩
    *p++=val;
    return p;
}
```

⟨addtok() *sanity check* p 133c⟩≡ (133b)
```
if(p==nil)
    return nil;
if(p >= &tok[NTOK]){
    *p = '\0';
    yyerror("token buffer too short");
    return nil;
}
```
Uses NTOK-10 56d, tok 56c, and yyerror() 179e.

12.2 Expanding globbing characters

The expansion happens at runtime, not at compile time. The Xglob[133d] bytecode (emitted after evaluating command arguments) scans the word list in argv and replaces any word containing GLOB markers with the list of matching filenames from the filesystem.

⟨*function* Xglob 133d⟩≡ (230b)
```
/// outcode (REDIR and FOR cases) -> <>
void
Xglob(void)
{
    globlist();
}
```
Uses globlist() 134b.

For a simple command, the argument list is globbed by calling globlist() directly inside Xsimple, rather than through a separate Xglob bytecode emitted by the compiler (as the REDIR and FOR cases do). The reason for the asymmetry is that a simple command is by far the most common thing a shell runs, and its arguments always have to be globbed right before the exec. Folding the call into Xsimple saves emitting and dispatching a separate Xglob bytecode for every command. Redirections and for loops have no such fused instruction to piggyback on, so there the compiler emits a standalone Xglob.

⟨Xsimple() *initializations,* globlist() 133e⟩≡ (81)
```
globlist();
```
Uses globlist() 134b.

`globv`[134a] is the accumulator for expansion. `globlist()`[134b] resets it to `nil`, then walks the command's word list calling `glob()`[135a] on each word; `glob()` pushes every matching filename (or, when nothing matches, the literal word) onto `globv`. So when `globlist()` returns, `globv` holds the fully expanded argument list, which is then spliced back in as the command's `argv`. It is a global rather than a parameter because the expansion recurses deeply through `globdir()`, and a shared accumulator saves threading an output list through every recursive call.

⟨*global* `globv` 134a⟩≡ (229c)
```
// list<ref_own<Word>>
struct Word *globv;
```
Uses `Word` 36b.

⟨*function* `globlist` 134b⟩≡ (229c)
```
/// Xsimple | ... -> <>
void
globlist(void)
{
    word *a;
    globv = nil;

    globlist1(runq->argv->words);
    poplist();
    pushlist();

    if(globv){
        for(a = globv;a->next;a = a->next)
            ;
        a->next = runq->argv->words;
        runq->argv->words = globv;
    }
}
```
Uses `globlist1()` 134c, `globv` 134a, `poplist()` 77c, `pushlist()` 46a, and `runq` 35a.

⟨*function* `globlist1` 134c⟩≡ (229c)
```
void
globlist1(word *gl)
{
    if(gl){
        globlist1(gl->next);
        glob(gl->word);
    }
}
```
Uses `glob()` 135a and `globlist1()` 134c.

12.3 Finding matching files: `glob()`

'`glob()`[135a] takes a pattern string and pushes all matching filenames onto `globv`[134a]. If no files match, the original pattern (with `GLOB` markers removed) is kept as-is.

⟨*global* `globname` 134d⟩≡ (229c)
```
char *globname;
```

⟨*function* `glob` 135a⟩≡ (229c)

```
/// Xglob -> globlist -> globlist1 -> <>
/*
 * Push all file names matched by p on the current thread's stack.
 * If there are no matches, the list consists of p.
 */
void
glob(void *ap)
{
    uchar *p = ap;
    word *svglobv = globv;
    int globlen = Globsize(ap);

    if(!globlen){
        deglob(p);
        globv = newword((char *)p, globv);
        return;
    }
    // else
    globname = emalloc(globlen);
    globname[0]='\0';
    globdir(p, (uchar *)globname);
    efree(globname);
    if(svglobv==globv){
        deglob(p);
        globv = newword((char *)p, globv);
    }
    else
        globsort(globv, svglobv);
}
```

Uses `deglob()` 137a, `efree()` 181c, `emalloc()` 181a, `globdir()` 135b, `globname` 134d, `globsort()` 136, `globv` 134a, and `newword()` 36c.

`globdir()`[135b] is the recursive workhorse: it scans the pattern for the first component containing a metacharacter, reads the corresponding directory, and recurses for each matching entry. The results are sorted alphabetically by `globsort()`[136].

⟨*function* `globdir` 135b⟩≡ (229c)

```
/*
 * Push names prefixed by globname and suffixed by a match of p onto the astack.
 * namep points to the end of the prefix in globname.
 */
void
globdir(uchar *p, uchar *namep)
{
    uchar *t, *newp;
    fdt f;

    /* scan the pattern looking for a component with a metacharacter in it */
    if(*p=='\0'){
        globv = newword(globname, globv);
        return;
    }
    t = namep;
    newp = p;
    while(*newp){
        if(*newp==GLOB)
            break;
        *t=*newp++;
        if(*t++=='/'){
```

```
            namep = t;
            p = newp;
        }
    }
    /* If we ran out of pattern, append the name if accessible */
    if(*newp=='\0'){
        *t='\0';
        if(access(globname, 0)==0)
            globv = newword(globname, globv);
        return;
    }
    /* read the directory and recur for any entry that matches */
    *namep='\0';
    if((f = Opendir(globname[0]?globname:"."))<0) return;
    while(*newp!='/' && *newp!='\0') newp++;
    while(Readdir(f, namep, *newp=='/')){
        if(matchfn(namep, p)){
            for(t = namep;*t;t++);
            globdir(newp, t);
        }
    }
    Closedir(f);
}
```

Uses GLOB 132, Opendir() 140d, globdir() 135b, globname 134d, globv 134a, matchfn() 138a, and newword() 36c.

⟨*function* globsort 136⟩≡ (229c)

```
void
globsort(word *left, word *right)
{
    char **list;
    word *a;
    int n = 0;

    for(a = left;a!=right;a = a->next)
        n++;
    list = (char **)emalloc(n*sizeof(char *));
    for(a = left,n = 0;a!=right;a = a->next,n++)
        list[n] = a->word;
    qsort((void *)list, n, sizeof(void *), globcmp);
    for(a = left,n = 0;a!=right;a = a->next,n++)
        a->word = list[n];
    efree((char *)list);
}
```

Uses efree() 181c, emalloc() 181a, and globcmp() 137b.

Walking through `ls src/*.c` makes the recursion concrete. After lexing, the argument is the byte string `src/\001*.c`. `globdir()` is called with `p` pointing at the start of that pattern and `namep` pointing at the start of an empty `globname` buffer. The loop copies literal bytes into `globname` until it hits the first `GLOB` marker:

```
step 1 : copy "src/" into globname, advance p past it
         globname = "src/"
         p        = "\001*.c"
         namep    = &globname[4]   (right after the slash)

step 2 : open directory "src/", iterate entries
         for each entry name, run match(name, "\001*.c")
         keeping only those that match "*.c":
            hello.c      MATCH
            hello.h      no
            Makefile     no
            util.c       MATCH

step 3 : for each match, copy the name into globname
         at namep and recurse with newp = "" (the suffix
         after the matched component is empty).
         The recursive call sees *p=='\0' and pushes
         "src/hello.c" and "src/util.c" onto globv.
```

Two subtleties of this design are worth naming. First, `globdir()` recurses per component, not per character: a pattern like `*/*.c` visits every directory in the current directory with a first-level call, then for each matching directory makes a second-level call that scans its entries. This is why `match()`[138b] stops at the first slash—it only ever matches a single component at a time. Second, if nothing matches, `glob()` returns the pattern itself (with `GLOB` bytes stripped) as the only result; this is the "no expansion" behaviour inherited from UNIX. It means that in `rc`, typing `echo *.nonsense` in an empty directory prints "`*.nonsense`" literally instead of erroring out, which occasionally surprises users but lets scripts pass glob patterns around as ordinary strings.

⟨*function* `deglob` 137a⟩≡ (229c)
```
/// Xlocal | Xcount | ... -> <>
/*
 * delete all the GLOB marks from s, in place
 */
void
deglob(void *as)
{
    char *s = as;
    char *t = s;
    do{
        if(*t==GLOB)
            t++;
        *s++=*t;
    }while(*t++);
}
```
Uses `GLOB` 132.

⟨*function* `globcmp` 137b⟩≡ (229c)
```
int
globcmp(const void *s, const void *t)
{
    return strcmp(*(char**)s, *(char**)t);
}
```

12.4 Matching one name: `match()`

`matchfn()`[138a] is the entry point called from `glob()`[135a]. It enforces one subtle rule: the dot-files `.` and `..` are only matched if the pattern explicitly starts with a dot. This prevents `*` from matching `.` and `..`, which would cause recursive globbing to loop or produce confusing results.

⟨*function* `matchfn` 138a⟩≡ (229c)

```
/*
 * Does the string s match the pattern p
 * . and .. are only matched by patterns starting with .
 * * matches any sequence of characters
 * ? matches any single character
 * [...] matches the enclosed list of characters
 */
bool
matchfn(void *as, void *ap)
{
    uchar *s = as, *p = ap;

    if(s[0]=='.' && (s[1]=='\0' || s[1]=='.' && s[2]=='\0') && p[0]!='.')
        return false;
    return match(s, p, '/');
}
```

The actual matching is done by `match()`[138b], which walks the pattern and the string in lockstep. Non-`GLOB` bytes must match literally. `GLOB` bytes introduce the special characters: `*` tries every possible suffix via recursion (backtracking), `?` matches exactly one UTF-8 character, and `[...]` matches a character class with optional ranges (`a-z`) and complement (`~`). Note that `rc` uses `~` instead of `^` for complement inside character classes, since `^` is already the concatenation operator.

The `stop` parameter is `/`: a single glob pattern is matched one path component at a time by `globdir()`[135b], so `match()` stops at slashes rather than consuming them.

⟨*function* `match` 138b⟩≡ (229c)

```
bool
match(void *as, void *ap, int stop)
{
    int compl, hit, lo, hi, t, c;
    uchar *s = as, *p = ap;

    for(; *p!=stop && *p!='\0'; s = nextutf(s), p = nextutf(p)){
        if(*p!=GLOB){
            if(!equtf(p, s)) return false;
        }
        else switch(*++p){
        case GLOB:
            if(*s!=GLOB)
                return false;
            break;
        case '*':
            for(;;){
                if(match(s, nextutf(p), stop)) return 1;
                if(!*s)
                    break;
                s = nextutf(s);
            }
            return false;
        case '?':
            if(*s=='\0')
                return false;
```

```
            break;
        case '[':
            if(*s=='\0')
                return false;
            c = unicode(s);
            p++;
            compl=*p=='~';
            if(compl)
                p++;
            hit = 0;
            while(*p!=']'){
                if(*p=='\0')
                    return false;  /* syntax error */
                lo = unicode(p);
                p = nextutf(p);
                if(*p!='-')
                    hi = lo;
                else{
                    p++;
                    if(*p=='\0')
                        return false; /* syntax error */
                    hi = unicode(p);
                    p = nextutf(p);
                    if(hi<lo){ t = lo; lo = hi; hi = t; }
                }
                if(lo<=c && c<=hi)
                    hit = 1;
            }
            if(compl)
                hit=!hit;
            if(!hit)
                return false;
            break;
        }
    }
    return *s=='\0';
}
```

⟨*constant* NDIR 139a⟩≡ (240b)

```
#define NDIR 256  /* shoud be a better way */
```

⟨*function* Globsize 139b⟩≡ (240b)

```
int
Globsize(char *p)
{
    int isglob = 0, globlen = NDIR+1;
    for(;*p;p++){
        if(*p==GLOB){
            p++;
            if(*p!=GLOB)
                isglob++;
            globlen+=*p=='*'?NDIR:1;
        }
        else
            globlen++;
    }
    return isglob?globlen:0;
}
```

Uses GLOB 132 and NDIR-3 139a.

12.5 Directory walking

`globdir()`[135b] needs to read a directory one name at a time, but Plan 9's `dirread()` hands back a whole buffer of `Dir` structures at once. This section bridges the two with a small `stdio`-like layer—`Opendir()`[140d], `Readdir()`[141a], `Closedir()`[141b]—that returns entries singly, buffering the rest in a per-file-descriptor table `dir[NFD]` indexed by the open `fd`. The one twist over a plain directory reader is the `onlydirs` hint: when the pattern still has path components left to match, only subdirectories can lead anywhere, so `Readdir()` may ask the system to skip plain files.

⟨*constant* `NFD` 140a⟩≡ (240b)
```
#define NFD 50
```

⟨*struct* `DirEntryWrapper` 140b⟩≡ (240b)
```
struct DirEntryWrapper {
    Dir *dbuf;
    int i;
    int n;
};
```

⟨*global* `dir` 140c⟩≡ (240b)
```
struct DirEntryWrapper dir[NFD];
```
Uses `DirEntryWrapper` 140b and `NFD-4` 140a.

⟨*function* `Opendir` 140d⟩≡ (240b)
```
int
Opendir(char *name)
{
    Dir *db;
    int f;
    f = open(name, 0);
    if(f==-1)
        return f;
    db = dirfstat(f);
    if(db!=nil && (db->mode&DMDIR)){
        if(f<NFD){
            dir[f].i = 0;
            dir[f].n = 0;
        }
        free(db);
        return f;
    }
    free(db);
    close(f);
    return -1;
}
```
Uses `NFD-4` 140a and `dir` 140c.

⟨*function* `trimdirs` 140e⟩≡ (240b)
```
static int
trimdirs(Dir *d, int nd)
{
    int r, w;

    for(r=w=0; r<nd; r++)
        if(d[r].mode&DMDIR)
            d[w++] = d[r];
    return w;
}
```

⟨*function* Readdir 141a⟩≡ (240b)

```
/*
 * onlydirs is advisory -- it means you only
 * need to return the directories.  it's okay to
 * return files too (e.g., on unix where you can't
 * tell during the readdir), but that just makes
 * the globber work harder.
 */
int
Readdir(int f, void *p, int onlydirs)
{
    int n;

    if(f<0 || f>=NFD)
        return 0;
Again:
    if(dir[f].i==dir[f].n){ /* read */
        free(dir[f].dbuf);
        dir[f].dbuf = 0;
        n = dirread(f, &dir[f].dbuf);
        if(n>0){
            if(onlydirs){
                n = trimdirs(dir[f].dbuf, n);
                if(n == 0)
                    goto Again;
            }
            dir[f].n = n;
        }else
            dir[f].n = 0;
        dir[f].i = 0;
    }
    if(dir[f].i == dir[f].n)
        return 0;
    strcpy(p, dir[f].dbuf[dir[f].i].name);
    dir[f].i++;
    return 1;
}
```

Uses NFD-4 140a, dir 140c, and trimdirs() 140e.

⟨*function* Closedir 141b⟩≡ (240b)

```
void
Closedir(int f)
{
    if(f>=0 && f<NFD){
        free(dir[f].dbuf);
        dir[f].i = 0;
        dir[f].n = 0;
        dir[f].dbuf = 0;
    }
    close(f);
}
```

Uses NFD-4 140a and dir 140c.

Chapter 13

Notes (Signals) Management

Under Plan 9, signals are called *notes*. Signal/note handling is critical for a shell because the shell sits between the user and every running program. When the user presses `^C`, the interrupt signal is sent to all processes in the foreground group[1]—including the shell itself. If the shell simply died along with the interrupted command, the user would lose their terminal session. Instead, the shell must catch the signal, cancel the current command, and return to its prompt, ready for the next input.

Beyond `^C`, a shell must also handle *hangup* signals (so it can clean up or notify background jobs when a terminal disconnects), and let users define their own trap handlers for scripting (e.g., cleaning up temporary files on exit). See the KERNEL book [Pad14] for more information on how the Plan 9 kernel delivers notes to processes.

13.1 Overview

When a note is received by the `rc` process (e.g., an interrupt from `^C`), the kernel calls the handler registered by `Trapinit()`[144e]. The handler does not execute the trap function immediately (which would be unsafe in the middle of interpreting bytecodes). Instead, it increments a counter (`ntrap`[144c]) and the main interpreter loop checks this counter after each bytecode (see the `handing trap if necessary` chunk in `main()`[42b]). Users can define signal handlers with `fn sigint {...}`, `fn sighup {...}`, etc. If no handler is defined, the default behavior applies (usually terminating the process).

The full path of a `^C` from keystroke to user-defined handler is therefore asynchronous in the kernel half and synchronous in the interpreter half:

[1] How does the kernel know which processes are "in front"? Plan 9 has no UNIX tty process group; instead each terminal or `rio` window runs its commands in their own *note group*, created with `rfork(RFNOTEG)`, and the console driver posts the interrupt note to that group—see KERNEL book [Pad14] and WINDOWS book [Pad16e].)

```
user        kernel                    rc interpreter loop
----        ------                    -------------------
 ^C
  |         deliver "interrupt"
  |          note to rc
  |               |
  |               v
  |         notifyf(s="interrupt")    (currently running, say,
  |           trap[SIGINT]++          inside Xsimple of a nested
  |           ntrap++                 for-loop)
  |           noted(NCONT)
  |               |
  |               v
  |         rc resumes where interrupted
  |                                       |
  |                                       v
  |                                   finish current bytecode
  |                                       |
  |                                       v
  |                                   main() loop top:
  |                                     if(ntrap) dotrap();
  |                                       |
  |                                       v
  |                                   dotrap() looks up
  |                                   vlook("sigint")->fn:
  |                                     - if user defined one,
  |                                       start() a new thread
  |                                       running the fn body
  |                                     - otherwise, pop threads
  |                                       via Xreturn() until we
  |                                       reach the iflag REPL
```

Two design points are worth calling out. First, the deferral via `ntrap` is the only way to keep the interpreter's invariants intact: a bytecode like `Xassign`[109c] is in the middle of re-linking `argv`, freeing old values, and installing new ones; if a shell function were run at an arbitrary point inside it, the word lists would be in an inconsistent state and the heap would leak. The same argument applies to any language runtime with safe-points—`rc` just happens to have very coarse safe-points (once per bytecode) because its bytecodes are large compared to a CPU instruction. Second, the "pop threads until `iflag`" unwinding in `dotrap`[145b] is how `^C` abandons a deeply nested script or for-loop and returns to the interactive prompt: each `Xreturn` pops one frame off `runq`, just as if the function had returned normally, and eventually uncovers the bootstrap thread that loops in `Xrdcmds()`[47a]. If no such interactive thread exists (for non-interactive `rc`), the final `Xreturn` terminates the shell—which is the correct behaviour for `^C` in a script.

⟨*constant* `NSIG` 143a⟩≡ (223b)
```
#define NSIG 32
```

Unlike UNIX, where a signal is a small integer, a Plan 9 *note* is an arbitrary *string* delivered to the process (for example `"interrupt"` or `"hangup"`). `syssigname`[143b] lists the note strings `rc` recognizes; the handler matches an incoming note against this table to find which one arrived.

⟨*global* `syssigname` *(plan9.c)* 143b⟩≡ (240b)
```
char *syssigname[] = {
    "exit",  /* can't happen */
```

```
    "hangup",
    "interrupt",
    "quit",  /* can't happen */
    "alarm",
    "kill",
    "sys: fp: ",
    "term",
    0
};
```

⟨*constant* SIGINT 144a⟩≡ (223b)
```
#define SIGINT 2
```

⟨*constant* SIGQUIT 144b⟩≡ (223b)
```
#define SIGQUIT 3
```

The two constants above are not the UNIX signal numbers; they are simply indices into `syssigname` (so `SIGINT`[144a] is 2 because `syssigname[2]` is `"interrupt"`). `rc` uses them only to index its own `trap` array—Plan 9 has no signal numbers at all.

⟨*global* ntrap 144c⟩≡ (234b)
```
int ntrap;    /* number of outstanding traps */
```

⟨*global* trap 144d⟩≡ (234b)
```
int trap[NSIG];    /* number of outstanding traps per type */
```

13.2 Registering handlers: `Trapinit()`

`Trapinit()`[144e] simply registers `notifyf()`[144f] as the process's note handler via `notify()`. When a note arrives, `notifyf()` matches its string against `syssigname`[143b] and bumps the corresponding `trap`[144d] counter for `dotrap()`[145b] to act on later.

⟨*function* Trapinit 144e⟩≡ (240b)
```
void
Trapinit(void)
{
    notify(notifyf);
}
```
Uses notifyf() 144f.

⟨*function* notifyf 144f⟩≡ (240b)
```
void
notifyf(void*, char *s)
{
    int i;
    for(i = 0;syssigname[i];i++)
     if(strncmp(s, syssigname[i], strlen(syssigname[i]))==0){
        if(strncmp(s, "sys: ", 5)!=0)
            interrupted = true;
        goto Out;
    }
    pfmt(err, "rc: note: %s\n", s);
    noted(NDFLT);
    return;
Out:
    if(strcmp(s, "interrupt")!=0 || trap[i]==0){
        trap[i]++;
        ntrap++;
```

```
    }
    if(ntrap>=32){ /* rc is probably in a trap loop */
        pfmt(err, "rc: Too many traps (trap %s), aborting\n", s);
        abort();
    }
    noted(NCONT);
}
```

Uses err 179b, interrupted 146a, ntrap 144c, pfmt() 190b, syssigname 143b, and trap 144d.

The "sys: " prefix it tests for above is a Plan 9 kernel convention: the kernel prefixes the notes it raises itself (faults, "sys: fp: ...", and the like) with sys:, so rc uses that prefix to tell a kernel-generated error from a user-sent note like "interrupt"—only the latter sets interrupted[146a].

noted() is the Plan 9 call that ends a note handler, and its argument says what to do next: NCONT resumes the process where the note interrupted it (here, after recording the trap for dotrap()), while NDFLT takes the default action—terminating the process—used for notes rc does not recognize.

13.3 Trap dispatch: dotrap()

dotrap()[145b] runs from the interpreter loop whenever ntrap is nonzero—that is, after a note handler has recorded a pending trap. It drains the trap counters and, for each, either runs a user-defined handler function (next section), unwinds to the interactive prompt on ^C or quit, or exits.

⟨main() *handing trap if necessary in interpreter loop* 145a⟩≡ (46c)

```
if(ntrap)
    dotrap();
```

Uses dotrap() 145b and ntrap 144c.

⟨*function* dotrap 145b⟩≡ (234b)

```
void
dotrap(void)
{
    int i;
    struct Var *trapreq;
    struct Word *starval;

    starval = vlook("*")->val;
    while(ntrap)
     for(i = 0;i!=NSIG;i++)
      while(trap[i]){
        --trap[i];
        --ntrap;
        if(getpid()!=mypid)
            Exit(getstatus(), __LOC__);

        ⟨dotrap() check if trap handled by function 147a⟩
        else if(i==SIGINT || i==SIGQUIT){
            /*
             * run the stack down until we uncover the
             * command reading loop.  Xreturn will exit
             * if there is none (i.e. if this is not
             * an interactive rc.)
             */
            while(!runq->iflag)
                Xreturn();
        }
        else
            Exit(getstatus(), __LOC__);
```

```
    }
}
```

Uses Var 38c, Word 36b, Xreturn() 96c, getstatus() 78b, mypid 130a, ntrap 144c, runq 35a, trap 144d, and vlook() 39c.

interrupted is how rc tells a ^C apart from a genuine end-of-input. notifyf()[144f] sets it when an interrupt note arrives (but not for a sys: kernel note), and Eintr()[146b] just reports its value. The interactive command loop uses it after yyparse()[65b] fails: if the read was interrupted, rc prints a newline and returns to the prompt instead of treating the failure as EOF and closing the input. Noerror()[146d] clears the flag in Xrdcmds()[47a] before each parse, so it reflects only the read just attempted.

⟨*global* interrupted 146a⟩≡ (240b)
```
bool interrupted = false;
```

Uses interrupted 146a.

⟨*function* Eintr 146b⟩≡ (240b)
```
bool
Eintr(void)
{
    return interrupted;
}
```

Uses interrupted 146a.

⟨Xrdcmds() *calls Noerror() before yyparse()* 146c⟩≡ (47a)
```
Noerror();
```

⟨*function* Noerror 146d⟩≡ (240b)
```
void
Noerror(void)
{
    interrupted = false;
}
```

Uses interrupted 146a.

⟨Xrdcmds() *reset ntrap* 146e⟩≡ (47a)
```
ntrap = 0; /* avoid double-interrupts during blocked writes */
```

Uses ntrap 144c.

13.4 User-defined signal handlers: fn sig<xxx>

A user can handle a note by defining a function named after it, such as fn sigint ... or fn sigexit .... When dotrap()[145b] finds such a function in the variable table (looked up via the name signame[i]), it runs that instead of taking the built-in action; this is how scripts clean up on exit or ignore interrupts.

⟨*global* signame*(plan9.c)* 146f⟩≡ (240b)
```
char *signame[] = {
    "sigexit",
    "sighup",
    "sigint",
    "sigquit",
    "sigalrm",
    "sigkill",
    "sigfpe",
    "sigterm",
    0
};
```

⟨dotrap() *check if trap handled by function* 147a⟩≡ (145b)

```
trapreq = vlook(signame[i]);
if(trapreq->fn){
    start(trapreq->fn, trapreq->pc, (struct Var *)nil);
    runq->local = newvar(strdup("*"), runq->local);
    runq->local->val = copywords(starval, (struct Word *)nil);
    runq->local->changed = true;
    runq->redir = runq->startredir = nil;
}
```

Uses Var 38c, Word 36b, copywords() 37d, newvar() 40b, runq 35a, start() 45b, and vlook() 39c.

⟨Xexit() *locals* 147b⟩≡ (96b)

```
struct Var *trapreq;
struct Word *starval;
static bool beenhere = false;
```

Uses Var 38c and Word 36b.

⟨Xexit() *trap management* 147c⟩≡ (96b)

```
if(getpid()==mypid && !beenhere){
    trapreq = vlook("sigexit");
    if(trapreq->fn){
        beenhere = true;
        --runq->pc;
        starval = vlook("*")->val;
        start(trapreq->fn, trapreq->pc, (struct Var *)0);
        runq->local = newvar(strdup("*"), runq->local);
        runq->local->val = copywords(starval, (struct Word *)0);
        runq->local->changed = true;
        runq->redir = runq->startredir = nil;
        return;
    }
}
```

Uses Var 38c, Word 36b, copywords() 37d, mypid 130a, newvar() 40b, runq 35a, start() 45b, and vlook() 39c.

Chapter 14

Advanced Topics

In this chapter I present features, flags, builtins, and more generally topics that are not essential for understanding the core architecture of `rc`, but that are important in practice. Many of these features—here documents, command substitution, process substitution—are what make a shell truly useful for scripting.

Note that in Plan 9, many shell-like features that other systems implement inside the shell are instead provided by the window manager `rio`: command-line editing, history, filename completion, and job control (see the WINDOWS book [Pad16e]). This keeps `rc` simple.

14.1 Reading commands from a string: `rc -c`

The `-c` flag lets you pass a command string directly on the command line, as in `rc -c 'echo hello'`. This is how other programs (like `mk`, see the BUILDER book [Pad16b]) invoke the shell to execute a recipe: they build a command string and pass it via `-c` rather than writing it to a temporary file.

⟨`main()` *initialisation* 148a⟩+≡ (42b) ◁131a
```
setvar("cflag", flag['c']? newword(flag['c'][0], (word *)nil) : (word *)nil);
```
Uses `flag` 43a, `newword()` 36c, and `setvar()` 39b.

There is no special C code in `rc` to support `-c` beyond the code above: the feature is implemented entirely in `rcmain`. The flag's argument is stashed in `$cflag`, and the script simply runs `eval $cflag`, handing the string to the `eval` builtin (Section 9.5) to be re-parsed and executed like any other command.

⟨`rcmain.rc` *if* `-c` *flag* 148b⟩≡ (130d)
```
if(! ~ $#cflag 0){
        ⟨rcmain.rc if -l, source user's profile 195a⟩
        status=''
        eval $cflag
}
```

14.2 Failing fast: `rc -e`

The `-e` flag makes `rc` exit immediately when a simple command fails (returns non-null status). This is extremely important again for build systems: `mk` invokes `rc -e` for recipes, so that a failure in any command—especially inside a loop iterating over subdirectories—stops the entire build rather than silently continuing.

Although `-e` is passed as a command-line flag, you can also enable it from within a script with the `flag` builtin (`flag e +`, see Section 14.5.4), which makes it equivalent to `bash`'s `set -e`.

The check is only inserted after simple commands, not after control flow constructs like `if` or `||`, since a false condition in `if(test ...)` should not abort the script.

⟨`outcode()` *emit Xeflag after Xsimple* 148c⟩≡ (80d)
```
if(eflag)
    emitf(Xeflag);
```

⟨outcode() *emit Xeflag after Xmatch* 149a⟩≡ (88e)
```
if(eflag)
    emitf(Xeflag);
```

The `eflagok` guard prevents `-e` from triggering during startup (while sourcing `rcmain`). It is set to true only after the first `dot` command finishes, so initialization errors do not cause an immediate exit.

⟨*global* `eflagok` 149b⟩≡ (226)
```
bool eflagok; /* kludge flag so that -e doesn't exit in startup */
```

⟨execdot() *if not first execution* 149c⟩≡ (119)
```
else
    eflagok = true;
```
Uses `eflagok` 149b.

Here, finally, is the `Xeflag` bytecode itself. It is tiny: if the guard `eflagok` is set and the last command's status was not "true" (`truestatus()`[78c] returns false for any status string carrying a non-zero digit), it calls `Xexit()`[96b] to abort the shell.

⟨*function* `Xeflag` 149d⟩≡ (230b)
```
void
Xeflag(void)
{
    if(eflagok && !truestatus())
        Xexit();
}
```
Uses `Xexit()` 96b, `eflagok` 149b, and `truestatus()` 78c.

It is worth stepping back, because error handling is one of the genuinely hard problems every language faces, and `-e` is only one answer to it. The baseline in any shell is to check `$status` by hand after each command—correct, but verbose and easy to forget. `-e` automates that check, at the cost of the surprises noted above. Richer languages reach instead for exceptions (a failure unwinds the stack until something catches it) or for return values that force the caller to acknowledge the error (Go's explicit `err`, Rust's `Result`, OCaml's `result`). A shell sits at the low end of this spectrum: a command yields a single string status, and `-e` is the pragmatic middle ground between ignoring errors entirely and checking every one by hand.

14.3 Unicode

Plan 9 was the birthplace of UTF-8, and `rc` handles Unicode transparently. The key insight is that UTF-8 is self-synchronizing: the lexer can consume multi-byte runes by reading the first byte (which encodes the length) and then pulling the appropriate number of continuation bytes. `addutf()`[149e] does exactly this during lexing: after `addtok()`[133b] stores the first byte, it reads continuation bytes by checking the leading-bit pattern.

⟨*function* `addutf` 149e⟩≡ (234a)
```
/// yylex -> <>
char*
addutf(char *p, int c)
{
    uchar b, m;
    int i;

    p = addtok(p, c); /* 1-byte UTF runes are special */
    if(c < Runeself)
        return p;
    // else
    m = 0xc0;
    b = 0x80;
    for(i=1; i < UTFmax; i++){
```

```
        if((c&m) == b)
            break;
        p = addtok(p, advance());
        b = m;
        m = (m >> 1)|0x80;
    }
    return p;
}
```

Uses addtok() 133b and advance() 53b.

Why not just call the standard chartorune() or Bgetrune()? Because both want their input already buffered—chartorune() decodes from a complete in-memory string, Bgetrune() from a Biobuf. The lexer has neither: it pulls bytes one at a time through advance()[53b], which hides the pushback slot and the several possible input sources. So addutf decodes a rune incrementally from that stream, fetching each continuation byte with advance() as it goes.

This rune-awareness is not confined to the lexer; it quietly shows up wherever rc inspects characters. The ? glob matches one whole UTF-8 rune rather than one byte (see match()[138b]), and the word-splitting in command substitution reads its input rune by rune (the rutf loop in Xbackq()[151c]) so that a multi-byte separator is recognized correctly.

A few small UTF-8 helpers round out the support. equtf tests whether two byte pointers begin with the same rune, nextutf advances a pointer by one rune (without skipping past a NUL inside a malformed sequence), and unicode decodes the rune at a pointer to its integer code point. They are used by the globber's match(), where character-class tests and comparisons must work per rune rather than per byte.

⟨*function* equtf 150a⟩≡ (229c)

```
/*
 * Do p and q point at equal utf codes
 */
bool
equtf(uchar *p, uchar *q)
{
    Rune pr, qr;
    if(*p!=*q)
        return false;

    chartorune(&pr, (char*)p);
    chartorune(&qr, (char*)q);
    return pr == qr;
}
```

⟨*function* nextutf 150b⟩≡ (229c)

```
/*
 * Return a pointer to the next utf code in the string,
 * not jumping past nuls in broken utf codes!
 */
uchar*
nextutf(uchar *p)
{
    Rune dummy;
    return p + chartorune(&dummy, (char*)p);
}
```

⟨*function* unicode 150c⟩≡ (229c)

```
/*
 * Convert the utf code at *p to a unicode value
 */
int
unicode(uchar *p)
```

```
{
    Rune r;

    chartorune(&r, (char*)p);
    return r;
}
```

14.4 Advanced constructs

The constructs in this section are where `rc` goes past the common shell core. Each is a small, self-contained extension—a grammar rule, a bytecode, and a runtime function—layered on the machinery already built.

14.4.1 Command substitution: ‘{<cmd>}

Command substitution captures a command's stdout as a list of words. The Bourne shell uses backticks `‘cmd‘`, which cannot nest; `rc` uses `‘{cmd}` with braces, which nests naturally—Plan 9 scripts routinely write things like `dir=‘{dirname ‘{cleanname $file}}`.

The output is split into words by default using `$ifs` (inter-field separators, defaulting to space, tab, newline, as shown in `rcmain` in Section 11.5). The implementation forks a child connected via a pipe; the parent reads the output, splits on `$ifs` characters, and pushes the resulting word list onto the argv stack.

⟨*comword rule other cases* 151a⟩+≡ (68b) ◁73 153b▷
```
|   ’‘’ brace       {$$=tree2(’‘’, nil, $2);}
```

⟨`outcode()` *cases* 151b⟩+≡ (76b) ◁113a 153f▷
```
case ’‘’:
    emitf(Xmark);
    ⟨outcode() in backquote case, if c0 153c⟩
    else {
        emitf(Xmark);
        emitf(Xword);
        emits(strdup("ifs"));
        emitf(Xdol);
    }
    emitf(Xbackq);
    p = emiti(0);
    outcode(c1, false);
    emitf(Xexit);
    stuffdot(p);
    break;
```

`Xbackq` is the runtime side of command substitution. It forks the command with its standard output wired to a pipe, then the parent reads that output rune by rune, breaks it into words at the separator characters, and pushes the resulting list onto `argv`.

⟨*function* `Xbackq` 151c⟩≡ (233a)
```
/*
 * Who should wait for the exit from the fork?
 */
void
Xbackq(void)
{
    int n;
    char *stop;
    fdt pfd[2];
    int pid;
    struct Io *f;
```

```
    word *v, *nextv;
    String *word; // extensible string
    ⟨Xbackq() other locals 152⟩


    stop = "";
    if(runq->argv && runq->argv->words)
        stop = runq->argv->words->word;
    if(pipe(pfd)<0){
        Xerror("can't make pipe");
        return;
    }
    switch(pid = fork()){
    case -1:
        Xerror("try again");
        close(pfd[PRD]);
        close(pfd[PWR]);
        return;
    case 0: // child
        clearwaitpids();
        close(pfd[PRD]);
        start(runq->code, runq->pc+1, runq->local);
        pushredir(ROPEN, pfd[PWR], 1);
        return;
    default: // parent
        //pad: was commented at some point to remove the only dependency to
        //str.h (libstring) that I originally didn't want to port to goken
        addwaitpid(pid);
        close(pfd[PWR]);
        f = openfd(pfd[PRD]);
        word = s_new();
        v = nil;
        ⟨Xbackq() read UTF bytes from pipe f and modify word 153a⟩
        if(s_len(word) > 0)
            v = newword(s_to_c(word), v);
        s_free(word);
        closeio(f);
        Waitfor(pid, false);
        poplist(); /* ditch split in "stop" */
        /* v points to reversed arglist -- reverse it onto argv */
        while(v){
            nextv = v->next;
            v->next = runq->argv->words;
            runq->argv->words = v;
            v = nextv;
        }
        runq->pc = runq->code[runq->pc].i;
        return;
    }
}
```

Uses Io 186f, PRD 97a, PWR 97b, ROPEN 91c, Xerror() 82a, addwaitpid() 79c, clearwaitpids() 79f, closeio() 189a, newword() 36c, openfd() 186h, poplist() 77c, pushredir() 90f, runq 35a, and start() 45b.

This is the word-splitting loop, and it works one rune at a time, not one byte at a time (see Section 14.3). `rutf` pulls the next UTF-8 rune from the pipe; `utfutf` tests whether that rune appears in the `stop` separator set. A separator ends the current word; any other rune is appended to it.

⟨Xbackq() *other locals* 152⟩≡ (151c)

```
char utf[UTFmax+1];
Rune r;
```

⟨Xbackq() *read UTF bytes from pipe* f *and modify* word 153a⟩≡ (151c)
```
/* rutf requires at least UTFmax+1 bytes in utf */
while((n = rutf(f, utf, &r)) != EOF){
    utf[n] = '\0';
    if(utfutf(stop, utf) == nil)
        s_nappend(word, utf, n);
    else
        /*
         * utf/r is an ifs rune (e.g., \t, \n), thus
         * ends the current word, if any.
         */
        if(s_len(word) > 0){
            v = newword(s_to_c(word), v);
            s_reset(word);
        }
}
```
Uses EOF 50a, newword() 36c, and rutf() 188e.

In Xbackq(), the stop separators come from runq->argv, not from a global. This is to support also the *extended command-substitution* grammar that accepts a separator word before the brace (the ‘<sep>{cmd} form): it lets each substitution name its own separators. Earlier versions of rc read the global $ifs directly, which forced you to assign $ifs, run the substitution, then restore it—awkward, and broken under nesting, since an inner substitution would see the mutated global. Passing the separator as an argument keeps it local and composable.

⟨*comword rule other cases* 153b⟩+≡ (68b) ◁151a 153d▷
```
|   '‘' word brace  {$$=tree2('‘', $2, $3);}
```

⟨outcode() *in backquote case, if* c0 153c⟩≡ (151b)
```
if(c0){
    outcode(c0, 0);
    emitf(Xglob);
}
```

14.4.2 Process substitution (command output as a file): <{<cmd>}

⟨*comword rule other cases* 153d⟩+≡ (68b) ◁153b 156a▷
```
|   REDIR brace     {$$=mung1($1, $2); $$->type=PIPEFD;}
```

Process substitution lets you use a command's output as if it were a file. cmp <{old} <{new} runs the two commands and passes their outputs as filenames to cmp.

This is the same capability bash and ksh exposes as <(cmd). All of them rely on the /dev/fd device (here Plan 9's /fd).

Syntactically this is treated as just another command word, because it really resolves to a filename (/dev/fd/N) connected to the command's output via a pipe. The implementation forks a child that runs the command with its stdout (or stdin for >) connected to one end of a pipe, while the parent pushes the filename /fd/N (the other end) as a word onto the argv stack.

⟨*token declarations* 153e⟩+≡ (65a) ◁72a 155a▷
```
%token PIPEFD
```

⟨outcode() *cases* 153f⟩+≡ (76b) ◁151b 155c▷
```
case PIPEFD:
    emitf(Xpipefd);
    emiti(t->rtype);
    p = emiti(0);
    outcode(c0, eflag);
    emitf(Xexit);
    stuffdot(p);
    break;
```

⟨*global* Fdprefix 154a⟩≡ (240b)

```
char *Fdprefix = "/fd/";
```

⟨*function* Xpipefd 154b⟩≡ (233a)

```
void
Xpipefd(void)
{
    struct Thread *p = runq;
    int pc = p->pc;
    int pid;
    char name[40];
    fdt pfd[2];
    fdt sidefd, mainfd;

    if(pipe(pfd)<0){
        Xerror("can't get pipe");
        return;
    }
    if(p->code[pc].i==READ){
        sidefd = pfd[PWR];
        mainfd = pfd[PRD];
    }
    else{
        sidefd = pfd[PRD];
        mainfd = pfd[PWR];
    }
    switch(pid = fork()){
    case -1:
        Xerror("try again");
        break;
    case 0: // child
        clearwaitpids();
        start(p->code, pc+2, runq->local);
        close(mainfd);
        pushredir(ROPEN, sidefd, p->code[pc].i==READ?1:0);
        runq->ret = 0;
        break;
    default: // parent
        addwaitpid(pid);
        close(sidefd);
        pushredir(ROPEN, mainfd, mainfd); /* isn't this a noop? */
        strcpy(name, Fdprefix);
        inttoascii(name+strlen(name), mainfd);
        pushword(name);
        p->pc = p->code[pc+1].i;
        break;
    }
}
```

Uses PRD 97a, PWR 97b, READ 61c, ROPEN 91c, Thread 34d, Xerror() 82a, addwaitpid() 79c, clearwaitpids() 79f, inttoascii() 193b, pushredir() 90f, pushword() 46b, runq 35a, and start() 45b.

14.4.3 Subshell: `@ <cmd>`

The `@` operator runs a command in a child process, so that side effects like `cd` or variable assignments do not affect the parent[1]. Braces `{...}` group commands but run them in the current process—so `{cd /tmp; ls}` would

[1]This isolation works because `fork()` gives the child its own copy of the `/env` namespace, so the assignments it makes land in that copy. Processes that deliberately *share* an environment group (an `rfork` without `RFENVG`) would, by contrast, see each other's variable changes.

change the parent's directory. The `@` operator is most useful precisely when paired with them: `@{cd /tmp; ls}` runs the whole group in a subshell, so the `cd` (and any variable assignment inside) is confined there and the parent is left untouched.

This is particularly useful in `mkfile` recipes that iterate over subdirectories:

```
for(i in $DIRS) @{
    cd $i
    mk $MKFLAGS $stem
}
```

The implementation is straightforward: fork, run the body in the child (which will `Xexit`), and `Waitfor` in the parent. The forward-jump past the body (via `stuffdot`) lets the parent skip the child's code.

⟨*token declarations* 155a⟩+≡ (65a) ◁153e 163b▷
```
%token SUBSHELL /** @ */
```

⟨*cmd rule other cases* 155b⟩+≡ (66c) ◁72e
```
|   SUBSHELL cmd        {$$=mung1($1, $2);}
```

⟨`outcode()` *cases* 155c⟩+≡ (76b) ◁153f 156b▷
```
case SUBSHELL:
    emitf(Xsubshell);
    p = emiti(0);
    outcode(c0, eflag);
    emitf(Xexit);
    stuffdot(p);
    ⟨outcode() emit Xeflag in Xsubshell 155e⟩
    break;
```

With the bytecode emitted, here finally is `Xsubshell` itself.

⟨*function* `Xsubshell` 155d⟩≡ (233a)
```
void
Xsubshell(void)
{
    int pid;
    switch(pid = fork()){
    case -1:
        Xerror("try again");
        break;
    case 0: // child
        clearwaitpids();
        start(runq->code, runq->pc+1, runq->local);
        runq->ret = nil;
        break;
    default: // parent
        addwaitpid(pid);
        Waitfor(pid, true);
        runq->pc = runq->code[runq->pc].i;
        break;
    }
}
```
Uses `Xerror()` 82a, `addwaitpid()` 79c, `clearwaitpids()` 79f, `runq` 35a, and `start()` 45b.

⟨`outcode()` *emit Xeflag in Xsubshell* 155e⟩≡ (155c)
```
if(eflag)
    emitf(Xeflag);
```

Why an `Xeflag` after the subshell, when a `-e` failure inside the body already triggers `Xexit`? Because that inner `Xexit` only terminates the *child*; it sets the child's exit status but leaves the parent shell running. The child's status becomes the subshell's status, so the parent needs its own `Xeflag` to notice that the subshell failed and honor `-e` itself.

14.4.4 Stringification of variables: `$"<foo>`

The `$"` operator converts a variable—which in `rc` is always a list—into a single string, joining the elements with spaces. If `$x` is `(a b c)`, then `$"x` is `'a b c'` as a single word. This is the inverse of `$ifs`-splitting in command substitution: command substitution splits strings into lists, while `$"` joins a list back into a string.

⟨*comword rule other cases* 156a⟩+≡ (68b) ◁153d

```
|   '"' word        {$$=tree1('"', $2);}
```

⟨`outcode()` *cases* 156b⟩+≡ (76b) ◁155c 163e▷

```
case '"':
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xqdol);
    break;
```

⟨*function* `Xqdol` 156c⟩≡ (230b)

```
void
Xqdol(void)
{
    word *a, *p;
    char *s;
    int n;
    if(count(runq->argv->words)!=1){
        Xerror1("variable name not singleton!");
        return;
    }
    s = runq->argv->words->word;
    deglob(s);
    a = vlook(s)->val;
    poplist();
    n = count(a);
    if(n==0){
        pushword("");
        return;
    }
    for(p = a;p;p = p->next) n+=strlen(p->word);
    s = emalloc(n);
    if(a){
        strcpy(s, a->word);
        for(p = a->next;p;p = p->next){
            strcat(s, " "); // no ifs here, always space to join
            strcat(s, p->word);
        }
    }
    else
        s[0]='\0';
    pushword(s);
    efree(s);
}
```

Uses `Xerror1()` 82c, `count()` 37a, `deglob()` 137a, `efree()` 181c, `emalloc()` 181a, `poplist()` 77c, `pushword()` 46b, `runq` 35a, and `vlook()` 39c.

14.4.5 Here documents: << <HERE>

Here documents let you embed multi-line *data* directly in a script, avoiding external file dependencies. The syntax `<<TAG` reads input until a line matching `TAG` appears. It is the inline cousin of the `<file` input redirection: where `<file` feeds a command's stdin from a named file, `<<TAG` feeds it from text written directly in the script.

For example, a phone-lookup script can carry its own database inline:

```
grep $1 <<EOF
Alice 555-1234
Bob   555-5678
EOF
```

The program and its data live in the same file, so the script is self-contained and easy to move around. A variable would not work as well here: shell variables are word lists, so multi-line structure (newlines, whitespace, special characters) would be lost.

The implementation is a clever two-phase trick. The body of a here document does not sit where the `<<TAG` appears—it starts on the *next* line, after the rest of the command line (other redirections, a pipeline) has been parsed. So `rc` cannot capture it on the spot. Instead it splits the work: while parsing the command line, each `<<TAG` is turned into a redirection from a freshly named temporary file and remembered on a pending list; once the whole line is compiled, `rc` goes back, reads the bodies that follow, and writes them into those temporary files. Handling several here documents on one line (e.g., `cmd <<A <<B`) then falls out for free—the pending list simply holds more than one entry.

If the here-tag was not quoted, variable substitution is performed via `psubst()`[160]; otherwise the text is copied literally. The temporary files are cleaned up by emitting `Xdelhere` bytecodes that `remove()` them after use.

⟨*constant* `HERE` 157a⟩≡ (223b)
```
#define HERE 4
```

⟨`yylex()` *in switch when redirection character, if here document* 157b⟩≡ (62a)
```
if(nextis(c)){
    t->rtype = HERE;
    *w++=c;
}
```
Uses `HERE` 157a and `nextis()` 53c.

The pending here documents live on a simple linked list of `Here` records, each pairing a tag with the temporary filename chosen for its body. `here` points at the head and `ehere` holds the tail pointer, so new entries are appended in source order.

⟨*struct* `Here` 157c⟩≡ (223b)
```
struct Here {
    tree *tag;
    char *name;
    struct Here *next;
};
```
Uses `Here` 157c.

⟨*global* `here` 157d⟩≡ (235b)
```
struct Here *here;
```
Uses `Here` 157c.

⟨*global* `ehere` 157e⟩≡ (235b)
```
struct Here **ehere;
```
Uses `Here` 157c.

⟨*constant* `NLINE` 158a⟩≡ (235b)

```
/*
 * bug: lines longer than NLINE get split -- this can cause spurious
 * missubstitution, or a misrecognized EOF marker.
 */
#define NLINE 4096
```

⟨`compile()` *after outcode and error management, read heredoc* 158b⟩≡ (75g)

```
readhere();
```

Uses `readhere()` 158c.

After `compile()`[75g] finishes emitting bytecode for the whole line, `readhere()`[158c] walks this list: for each entry it `Creat()`s the temp file, then reads one line at a time from `runq->cmdfd` until a line matches `h->tag`, writing the others to the file (with variable substitution via `psubst()` unless the tag was quoted). At runtime the `HERE` redirection becomes an ordinary `Xread`[93b] of the temp file, and a matching `Xdelhere`[159b] bytecode deletes it afterwards.

⟨*function* `readhere` 158c⟩≡ (235b)

```
void
readhere(void)
{
    int c, subst;
    char *s, *tag;
    char line[NLINE+1];
    io *f;
    struct Here *h, *nexth;

    for(h = here; h; h = nexth){
        subst = !h->tag->quoted;
        tag = h->tag->str;
        c = Creat(h->name);
        if(c < 0)
            yyerror("can't create here document");
        f = openfd(c);
        s = line;
        pprompt();
        while((c = rchr(runq->cmdfd)) != EOF){
            if(c == '\n' || s == &line[NLINE]){
                *s = '\0';
                if(tag && strcmp(line, tag) == 0)
                    break;
                if(subst)
                    psubst(f, (uchar *)line);
                else
                    pstr(f, line);
                s = line;
                if(c == '\n'){
                    pprompt();
                    pchr(f, c);
                }else
                    *s++ = c;
            }else
                *s++ = c;
        }
        flush(f);
        closeio(f);
        cleanhere(h->name);
        nexth = h->next;
        efree((char *)h);
    }
```

```
        here = nil;
        doprompt = true;
    }
```

Uses Creat() 193g, EOF 50a, Here 157c, NLINE-9 158a, cleanhere() 159a, closeio() 189a, doprompt 51c, efree() 181c, flush() 187b, here 157d, openfd() 186h, pchr() 188a, pprompt() 52a, pstr() 191a, rchr() 188c, runq 35a, and yyerror() 179e.

⟨*function* cleanhere 159a⟩≡ (236a)

```
    void
    cleanhere(char *f)
    {
        emitf(Xdelhere);
        emits(strdup(f));
    }
```

Uses Xdelhere() 159b, emitf-64 75d, and emits-66 75e.

⟨*function* Xdelhere 159b⟩≡ (230b)

```
    void
    Xdelhere(void)
    {
        remove(runq->code[runq->pc++].s);
    }
```

Uses runq 35a.

⟨*function* Unlink 159c⟩≡ (236c)

⟨*global* ser 159d⟩≡ (235b)

```
    int ser = 0;
```

Uses ser 159d.

⟨*global tmp (rc/here.c)* 159e⟩≡ (235b)

```
    char tmp[] = "/tmp/here0000.0000";
```

Uses tmp 159e.

⟨*global* hex 159f⟩≡ (235b)

```
    char hex[] = "0123456789abcdef";
```

Uses hex 159f.

⟨*function* hexnum 159g⟩≡ (235b)

```
    void
    hexnum(char *p, int n)
    {
        *p++ = hex[(n>>12)&0xF];
        *p++ = hex[(n>>8)&0xF];
        *p++ = hex[(n>>4)&0xF];
        *p = hex[n&0xF];
    }
```

Uses hex 159f.

⟨*function* heredoc 159h⟩≡ (235b)

```
    //@Scheck: used by syn.y
    tree* heredoc(tree *tag)
    {
        struct Here *h = new(struct Here);

        if(tag->type != WORD)
            yyerror("Bad here tag");
        h->next = 0;
        if(here)
```

```
        *ehere = h;
    else
        here = h;
    ehere = &h->next;
    h->tag = tag;
    hexnum(&tmp[9], getpid());
    hexnum(&tmp[14], ser++);
    h->name = strdup(tmp);
    return token(tmp, WORD);
}
```

Uses Here 157c, WORD, ehere 157e, here 157d, hexnum() 159g, ser 159d, tmp 159e, token() 59b, and yyerror() 179e.

heredoc()[159h] performs a lexical trick: on the first pass through the source, it does not capture the document body at all. Instead, it replaces the <<EOF tag *in the token stream* with a fresh /tmp/hereXXXX.YYYY filename (pid and serial counter), so the parser sees a plain <-redirection from that file. It then pushes a Here record on the here linked list recording the original tag and the temp name for later body capture. Here is what the rewrite looks like for grep foo <<EOF:

```
source line as lexed:       grep foo << EOF

what yylex()/heredoc() give to the parser:
                            grep foo <  /tmp/here0a3f.0001
                                     ^
                                     rtype = HERE (not READ)

global ‘here‘ list after compile():
  +--------+     +--------+
  | h->tag | --> | "EOF"  |
  | h->name| --> | "/tmp/here0a3f.0001"
  | h->next|---> nil
  +--------+
```

⟨*function* psubst 160⟩≡ (235b)

```
void
psubst(io *f, uchar *s)
{
    int savec, n;
    uchar *t, *u;
    Rune r;
    word *star;

    while(*s){
        if(*s != '$'){  /* copy plain text rune */
            if(*s < Runeself)
                pchr(f, *s++);
            else{
                n = chartorune(&r, (char *)s);
                while(n-- > 0)
                    pchr(f, *s++);
            }
        }else{   /* $something -- perform substitution */
            t = ++s;
            if(*t == '$')
                pchr(f, *t++);
            else{
                while(*t && idchr(*t))
                    t++;
```

```
                savec = *t;
                *t = '\0';
                n = 0;
                for(u = s; *u && '0' <= *u && *u <= '9'; u++)
                    n = n*10 + *u - '0';
                if(n && *u == '\0'){
                    star = vlook("*")->val;
                    if(star && 1 <= n && n <= count(star)){
                        while(--n)
                            star = star->next;
                        pstr(f, star->word);
                    }
                }else
                    pstrs(f, vlook((char *)s)->val);
                *t = savec;
                if(savec == '^')
                    t++;
            }
            s = t;
        }
    }
}
```

⟨*function* `pstrs` 161a⟩≡ (235b)

```
void
pstrs(io *f, word *a)
{
    if(a){
        while(a->next && a->next->word){
            pstr(f, a->word);
            pchr(f, ' ');
            a = a->next;
        }
        pstr(f, a->word);
    }
}
```

Uses `pchr()` 188a and `pstr()` 191a.

14.4.6 Read-write redirections: <> <file>

The <> operator opens a file for both reading and writing (`ORDWR`), useful for in-place modification of files.

⟨*constant* `RDWR` 161b⟩≡ (223b)

```
#define RDWR 7
```

⟨`yylex()` *in switch when redirection character, if read/write redirect* 161c⟩≡ (62a)

```
else if (nextis('>')){
    t->rtype = RDWR;
    *w++=c;
}
```

Uses `RDWR` 161b and `nextis()` 53c.

⟨`outcode()` *when REDIR case, switch redirection type cases* 161d⟩+≡ (91d) ◁94a

```
case RDWR:
    emitf(Xrdwr);
    break;
```

⟨*function* Xrdwr 162a⟩≡ (230b)

```
void
Xrdwr(void)
{
    char *file;
    int f;

    switch(count(runq->argv->words)){
    default:
        Xerror1("<> requires singleton\n");
        return;
    case 0:
        Xerror1("<> requires file\n");
        return;
    case 1:
        break;
    }
    file = runq->argv->words->word;
    if((f = open(file, ORDWR))<0){
        pfmt(err, "%s: ", file);
        Xerror("can't open");
        return;
    }
    pushredir(ROPEN, f, runq->code[runq->pc].i);
    runq->pc++;
    poplist();
}
```

Uses ROPEN 91c, Xerror() 82a, Xerror1() 82c, count() 37a, err 179b, pfmt() 190b, poplist() 77c, pushredir() 90f, and runq 35a.

14.4.7 General redirections: >[2] <file>

The basic redirections > and < always use file descriptors 1 and 0 respectively. The bracket syntax >[2] or <[3] lets you redirect any file descriptor. For example, `echo error >[2] /tmp/log` redirects stderr.

The lexer handles the bracket syntax inline: after seeing > or <, if a [follows, it reads the file descriptor number. If it then sees =, the redirection becomes a DUP (file descriptor duplication or closing) rather than a file redirection.

⟨yylex() *in switch when redirection character, if bracket after* 162b⟩≡ (60d)

```
if(nextis('[')){
    *w++='[';
    c = advance();
    *w++=c;
    if(c<'0' || '9'<c){
    RedirErr:
        *w = 0;
        yyerror(t->type==PIPE?"pipe syntax"
                :"redirection syntax");
        return EOF;
    }
    t->fd0 = 0;
    do{
        t->fd0 = t->fd0*10 + c-'0';
        *w++=c;
        c = advance();
    }while('0'<=c && c<='9');

    if(c=='='){
```

```
            *w++='=';
            if(t->type==REDIR)
                // change the token type
                t->type = DUP;
            c = advance();
            if('0'<=c && c<='9'){
                t->rtype = DUPFD;
                t->fd1 = t->fd0;
                t->fd0 = 0;
                do{
                    t->fd0 = t->fd0*10+c-'0';
                    *w++=c;
                    c = advance();
                }while('0'<=c && c<='9');
            }
            else{
                if(t->type==PIPE)
                    goto RedirErr;
                t->rtype = CLOSE;
            }
        }
        if(c!=']'
        || t->type==DUP && (t->rtype==HERE || t->rtype==APPEND))
            goto RedirErr;
        *w++=']';
    }
```

Uses APPEND 61d, CLOSE 163a, DUP, DUPFD 163d, EOF 50a, HERE 157a, PIPE, REDIR, advance() 53b, nextis() 53c, and yyerror() 179e.

⟨*constant* CLOSE 163a⟩≡ (223b)

```
#define CLOSE 6
```

14.4.8 Advanced dup: >[<fd0>=<fd1>], <>[<fd0>=<fd1>] , <[<fd0>=<fd1>]

The DUP construct ties two file descriptors together with Plan 9's dup system call—which, unlike UNIX, takes two arguments, dup(from, to), subsuming what UNIX splits off into dup2. >[1=2] makes fd 1 point to the same destination as fd 2 (so stdout goes to stderr). >[1=] (with nothing after the equals) closes fd 1. This is cleaner than the Bourne shell's 2>&1 syntax.

Like other redirections, the implementation uses the two-phase approach: Xdup pushes a RDUP record onto the redir stack (undone by Xpopredir afterward), and the actual dup happens in doredir() when a child process is forked.

⟨*token declarations* 163b⟩+≡ (65a) ◁155a

```
%token DUP
```

⟨*redir rule other cases* 163c⟩≡ (70b)

```
|   DUP
```

⟨*constant* DUPFD 163d⟩≡ (223b)

```
#define DUPFD 5
```

⟨outcode() *cases* 163e⟩+≡ (76b) ◁156b

```
case DUP:
    if(t->rtype==DUPFD){
        emitf(Xdup);
        emiti(t->fd0);
        emiti(t->fd1);
    }
```

```
    else{
        emitf(Xclose);
        emiti(t->fd0);
    }
    outcode(c1, eflag);
    emitf(Xpopredir);
    break;
```

⟨*function* `Xdup` 164a⟩≡ (230b)

```
  void
  Xdup(void)
  {
      pushredir(RDUP, runq->code[runq->pc].i, runq->code[runq->pc+1].i);
      runq->pc+=2;
  }
```

Uses `RDUP` 164c, `pushredir()` 90f, and `runq` 35a.

⟨`codefree()` *in loop over code* `cp`*, switch bytecode cases* 164b⟩+≡ (34b) ◁106b

```
  else if(p->f==Xdup || p->f==Xpipefd)
          p+=2;
```

Uses `Xdup()` 164a and `Xpipefd()` 154b.

⟨*constant* `RDUP` 164c⟩≡ (225)

```
  #define RDUP 2   /* dup2(from, to); */
```

⟨*constant* `RCLOSE` 164d⟩≡ (225)

```
  #define RCLOSE 3   /* close(from); */
```

⟨*function* `Xclose` 164e⟩≡ (230b)

```
  void
  Xclose(void)
  {
      pushredir(RCLOSE, runq->code[runq->pc].i, 0);
      runq->pc++;
  }
```

Uses `RCLOSE` 164d, `pushredir()` 90f, and `runq` 35a.

⟨`doredir()` *switch redir type cases* 164f⟩+≡ (91a) ◁91b 164g▷

```
  case RDUP:
      dup(rp->from, rp->to);
      break;
```

Uses `RDUP` 164c.

⟨`doredir()` *switch redir type cases* 164g⟩+≡ (91a) ◁164f

```
  case RCLOSE:
      close(rp->from);
      break;
```

Uses `RCLOSE` 164d.

14.4.9 Advanced pipes: `|[<fd>]` , `|[<fd0>=<fd1>]`

Just as general redirections extend `>` and `<` with bracket syntax, pipes can also specify which file descriptors to connect. `cmd1 |[2] cmd2` pipes fd 2 of `cmd1` (stderr) to `cmd2`'s stdin.

There is no separate code for it: the `Xpipe()`[97c] bytecode we already saw is parameterized by two file descriptors, `lfd` and `rfd`, which it reads as operands and uses for the `pushredir()`[90f] on each side. A plain `|` just compiles to `lfd` 1 and `rfd` 0; the bracket syntax makes the lexer fill `t->fd0`/`t->fd1` with other numbers, which `outcode()`[76b] then emits in their place.

14.5 Advanced builtins

Here are the remaining builtins left over from Chapter 9: less central than `cd` or `.`, but still useful. `exec` replaces the shell process, `whatis` inspects variables and functions, `rfork` manipulates namespaces, `flag` queries command-line flags, and `shift` modifies `$*`.

14.5.1 % exec

The `exec` builtin replaces the current shell process with the given command (no fork). Its code was in fact already shown, back with the simple-command machinery: `execexec()`[83b] is the very function that runs a plain command after the `fork`, and the `exec` builtin simply calls it *without* forking first.

14.5.2 % whatis

`execwhatis()`[165] is the shell's *introspection* tool. Given a name, it checks (in order): is it a variable? a function? a builtin? an executable on `$path`? It then prints the definition in a form that can be fed back to `rc`—variables as `x=value`, functions as `fn name {body}`, builtins as `builtin name`.

⟨*function* `execwhatis` 165⟩≡ (237b)

```
void
execwhatis(void){   /* mildly wrong -- should fork before writing */
    word *a, *b, *path;
    var *v;
    struct Builtin *bp;
    char *file;
    struct Io out[1];
    int found, sep;
    a = runq->argv->words->next;
    if(a==0){
        Xerror1("Usage: whatis name ...");
        return;
    }
    setstatus("");
    memset(out, 0, sizeof out);
    out->fd = mapfd(1);
    out->bufp = out->buf;
    out->ebuf = &out->buf[NBUF];
    out->strp = nil;
    for(;a;a = a->next){
        v = vlook(a->word);
        if(v->val){
            pfmt(out, "%s=", a->word);
            if(v->val->next==0)
                pfmt(out, "%q\n", v->val->word);
            else{
                sep='(';
                for(b = v->val;b && b->word;b = b->next){
                    pfmt(out, "%c%q", sep, b->word);
                    sep=' ';
                }
                pfmt(out, ")\n");
            }
            found = 1;
        }
        else
            found = 0;
        v = gvlook(a->word);
```

```
        if(v->fn)
            pfmt(out, "fn %q %s\n", v->name, v->fn[v->pc-1].s);
        else{
            for(bp = builtins;bp->name;bp++)
                if(strcmp(a->word, bp->name)==0){
                    pfmt(out, "builtin %s\n", a->word);
                    break;
                }
            if(!bp->name){
                for(path = searchpath(a->word); path;
                    path = path->next){
                    if(path->word[0] != '\0')
                        file = appfile(path->word, a->word);
                    else
                        file = strdup(a->word);
                    if(Executable(file)){
                        pfmt(out, "%s\n", file);
                        free(file);
                        break;
                    }
                    free(file);
                }
                if(!path && !found){
                    pfmt(err, "%s: not found\n", a->word);
                    setstatus("not found");
                }
            }
        }
    }
    poplist();
    flush(err);
}
```

The subtle `mapfd()` helper is needed because `whatis` writes to fd 1, but fd 1 might have been redirected; `mapfd` walks the redir stack to find where fd 1 actually points.

⟨*function* `mapfd` 166a⟩≡ (237b)

```
int
mapfd(int fd)
{
    redir *rp;
    for(rp = runq->redir;rp;rp = rp->next){
        switch(rp->type){
        case RCLOSE:
            if(rp->from==fd)
                fd=-1;
            break;
        case RDUP:
        case ROPEN:
            if(rp->to==fd)
                fd = rp->from;
            break;
        }
    }
    return fd;
}
```

Uses `RCLOSE` 164d, `RDUP` 164c, `ROPEN` 91c, and `runq` 35a.

⟨*function* `Executable` 166b⟩≡ (237b)

```
bool
```

```
Executable(char *file)
{
    Dir *statbuf;
    bool ret;

    statbuf = dirstat(file);
    if(statbuf == nil)
        return false;
    ret = ((statbuf->mode&0111)!=0 && (statbuf->mode&DMDIR)==0);
    free(statbuf);
    return ret;
}
```

14.5.3 % rfork

The `rfork` builtin exposes Plan 9's `rfork(2)` system call, which selectively shares or unshares resources between parent and child (see the KERNEL book [Pad14]). The flags control which namespaces to copy or clear: `n`/`N` for the mount namespace, `e`/`E` for environment, `s` for note group, `f`/`F` for file descriptors. This is the Plan 9 equivalent of Linux containers and namespaces—but from 1992.

The environment flag is particularly interesting. As mentioned in Chapter 10, all `rc` variables are automatically exported to `/env/`—there is no `export` command like in `bash`. But this raises the question: how do you *prevent* a variable from being visible to children? The answer is `rfork`: `rfork e` creates a *copy* of the current `/env/`, so subsequent changes in either the parent or the child are invisible to the other; `rfork E` creates an entirely *empty* `/env/`, starting with a clean slate. Note that the lowercase flags (`n`, `e`, `f`) copy the resource, while the uppercase flags (`N`, `E`, `F`) clear it entirely. When called with no arguments, `rfork` defaults to `RFENVG|RFNAMEG|RFNOTEG`, creating new environment, mount namespace, and note groups all at once.

⟨*function* `execnewpgrp` 167⟩≡ (237b)

```
void
execnewpgrp(void)
{
    int arg;
    char *s;

    switch(count(runq->argv->words)){
    case 1:
        arg = RFENVG|RFNAMEG|RFNOTEG;
        break;
    case 2:
        arg = 0;
        for(s = runq->argv->words->next->word;*s;s++) switch(*s){
        default:
            goto Usage;
        case 'n':
            arg|=RFNAMEG;  break;
        case 'N':
            arg|=RFCNAMEG;
            break;
        //pad: commented for goken
        //case 'm':
        //    arg|=RFNOMNT;  break;
        case 'e':
            arg|=RFENVG;   break;
        case 'E':
            arg|=RFCENVG;  break;
        case 's':
            arg|=RFNOTEG;  break;
```

```
            case 'f':
                arg|=RFFDG;    break;
            case 'F':
                arg|=RFCFDG;   break;
            }
            break;
        default:
        Usage:
            pfmt(err, "Usage: %s [fnesFNEm]\n", runq->argv->words->word);
            setstatus("rfork usage");
            poplist();
            return;
        }
        if(rfork(arg)==-1){
            pfmt(err, "rc: %s failed\n", runq->argv->words->word);
            setstatus("rfork failed");
        }
        else
            setstatus("");
        poplist();
    }
```

Uses `count()` 37a, `err` 179b, `pfmt()` 190b, `poplist()` 77c, `runq` 35a, and `setstatus()` 78a.

14.5.4 % flag

The `flag` builtin queries and sets `rc`'s own command-line flags at runtime. You invoke it as a command, with the flag letter and an optional `+` or `-`: typing `flag e +` enables `-e` (exit on error), `flag e -` disables it, and `flag e` with no sign tests whether it is currently set, reporting the answer through the exit status.

This builtin is not just an interactive convenience: `rcmain` itself uses it to read the flags `rc` was launched with and branch on them. The `if(flag p)` and `if(flag i)` tests in Section 11.5 are exactly this—`rc` inspecting its own `-p` and `-i` command-line flags from within a script written in `rc`.

⟨*function* `execflag` 168⟩≡ (237b)

```
  void
  execflag(void)
  {
      char *letter, *val;

      switch(count(runq->argv->words)){
      case 2:
          setstatus(flag[(uchar)runq->argv->words->next->word[0]]?"":"flag not set");
          break;
      case 3:
          letter = runq->argv->words->next->word;
          val = runq->argv->words->next->next->word;
          if(strlen(letter)==1){
              if(strcmp(val, "+")==0){
                  flag[(uchar)letter[0]] = flagset;
                  break;
              }
              if(strcmp(val, "-")==0){
                  flag[(uchar)letter[0]] = 0;
                  break;
              }
          }
      default:
          Xerror1("Usage: flag [letter] [+-]");
          return;
```

```
    }
    poplist();
}
```

Uses `Xerror1()` 82c, `count()` 37a, `flag` 43a, `flagset` 43c, `poplist()` 77c, `runq` 35a, and `setstatus()` 78a.

14.5.5 % shift

`shift` removes the first `n` elements (default 1) from `$*`, the positional parameters. It modifies the variable in place by walking the linked list and freeing the removed nodes.

⟨*function* `execshift` 169⟩≡ (237b)

```
void
execshift(void)
{
    int n;
    word *a;
    var *star;

    switch(count(runq->argv->words)){
    default:
        pfmt(err, "Usage: shift [n]\n");
        setstatus("shift usage");
        poplist();
        return;
    case 2:
        n = atoi(runq->argv->words->next->word);
        break;
    case 1:
        n = 1;
        break;
    }
    star = vlook("*");
    for(;n && star->val;--n){
        a = star->val->next;
        efree(star->val->word);
        efree((char *)star->val);
        star->val = a;
        star->changed = true;
    }
    setstatus("");
    poplist();
}
```

Chapter 15

Conclusion

You now know how the Plan 9 shell `rc` works, and more generally how many shells work. Despite being only about 5700 lines of C (plus a small Yacc grammar), `rc` implements a complete programming language: a hand-written lexer, a Yacc parser, a bytecode compiler, and a bytecode interpreter—the same architecture found in much larger language implementations like Python or Java. Along the way, you have seen how `rc` uses only a handful of system calls—`rfork()`, `exec()`, `wait()`, `pipe()`, `open()`, `close()`, `dup()`—to implement pipes, redirections, subshells, and background jobs. The beauty of the UNIX process model is that these few primitives are enough to build a surprisingly powerful command-line environment.

15.1 Patterns and techniques

These techniques apply far beyond shells:

- *Bytecode dispatch*: representing each operation as a function pointer and looping with `(*pc++)()` is how CPython's `ceval.c` and the JVM work. It decouples the description of an action from its execution—useful in command queues, event systems, and undo/redo frameworks.
- *Transparent interposition*: I/O redirection via `dup()` lets a program's input and output be rewired without its knowledge. The same idea—inserting a layer between producer and consumer transparently—is the basis of middleware, proxy servers, and the decorator pattern.
- *Uniform data representation*: every value in `rc` is a list of strings, just as every Lisp value is an S-expression. A single universal type eliminates conversion boilerplate and makes composition trivial. Taken one step further—so that a program's own *code* is just another value of that universal type—the property is called *homoiconicity*; Lisp (code as S-expressions) and Prolog (programs are ordinary terms, built and queried at runtime) are the classic examples. `rc` stops short of that: its values are string lists, but its programs are not.

15.2 Connections to other books

A shell sits at the intersection of many topics covered in other Principia Softwarica books:

- The KERNEL book [Pad14] explains what lies behind the system calls you have seen throughout this book: `rfork()`, `exec()`, `wait()`, `chdir()`, and the `/env` and `/dev/cons` device files. Understanding the kernel will deepen your understanding of how pipes are implemented at the OS level, how the `#!` shebang mechanism works, or how notes (signals) are delivered.

- The Compiler book [Pad16c] uses the same compilation pipeline as `rc`—lexing, parsing, code generation—but for the C language and targeting real machine code rather than a bytecode interpreter. Having seen `rc`'s simpler compiler first should make the C compiler easier to follow.
- The Generators book [?] (Lex and Yacc) explains what the `yacc`-generated parser in `rc` actually does behind the scenes: the shift-reduce algorithm, the parse tables, and the interaction between `yylex()` and `yyparse()`.
- The Builder book [Pad16b] describes `mk`, the Plan 9 build system, which relies heavily on `rc` for executing recipes. Features like `-e` (exit on error) and `-c` (command from string) exist largely because `mk` needs them.
- The Utilities book [Pad25] covers the small text-processing programs—`grep`, `sed`, `awk`, and friends—that `rc` scripts lean on so heavily. The shell itself does almost no string manipulation: it splits words, expands variables, globs filenames, and does glob-style pattern matching with (the same `*`, `?`, `[]` wildcards, not full regular expressions). Anything more—substitution, field extraction, real regular expressions—is delegated to these external commands. Seeing how they work explains what most of the power in a typical `rc` script actually comes from.

15.3 Missing features?

`rc` is deliberately minimal. Many features that other shells provide are instead handled by other programs in Plan 9, for instance the Plan 9 window manager `rio`:

- *Command-line editing and history:* Under `rio`, you can scroll back in the terminal window and copy-paste previous commands. There is no need for a readline library like in `bash`.
- *Filename completion:* Provided by `rio`'s terminal, not by `rc` itself. This is possible because `rc` publishes its current directory to the window: the `cd` builtin writes the new path to `/dev/wdir` when running interactively (see Section 9.2), so `rio` always knows which directory to complete filenames against.
- *Job control (`C-z`, `bg`, `fg`):* Less necessary when you have multiple windows. The `&` operator and `wait $apid` cover most use cases in `rc`.
- *Aliases:* Not needed because `fn` (functions) serve the same purpose and are stored persistently in the environment.
- *String manipulation*: `bash` provide expansion operators for substring extraction (`${var:offset:length}`), pattern replacement (`${var//pat/rep}`), and case conversion. `rc`'s position is that these operations belong in dedicated tools like `sed` and `awk`, not in the shell itself—keeping the shell language small and letting each tool do one thing well.

This separation of concerns—keeping the shell simple and letting the terminal handle interactive features—is a characteristic Plan 9 design decision. It contrasts with shells like `bash` or `zsh`, which include tens of thousands of lines of code for features that `rc` delegates elsewhere.

15.4 Beyond the Plan 9 shell

The UNIX shell world has evolved considerably since `rc` was written. Here are some of the features found in modern shells:

- *Rich data types*: `rc` has one data type: lists of strings. `bash` (version 4+) adds indexed arrays, associative arrays, and arithmetic expressions. PowerShell takes a radically different approach, passing structured .NET objects through pipelines instead of text. Nushell follows a similar philosophy with typed, tabular data—an interesting departure from the UNIX tradition of plain-text pipelines.

- *Interactive features*: beyond the readline and history that `rio` provides, modern shells add programmable completion (context-aware suggestions for flags, filenames, git branches), syntax highlighting as you type (`fish`, `zsh` with plugins), and autosuggestions from history (`fish`). `zsh` has entire plugin ecosystems (oh-my-zsh).

- *POSIX compliance*: `rc` deliberately ignores the POSIX shell standard, which gives it a cleaner syntax but means scripts are not portable. Most Linux distributions require `/bin/sh` scripts to work with any POSIX-compliant shell (`dash`, `bash --posix`), and POSIX compliance remains important for portable system scripts.

At the end of the day, the core model remains the one that `rc` shares with the Bourne shell: fork, exec, wait, pipes, and file descriptors. `rc` captures this essential engine in 5700 lines of C. `bash`'s 170 000+ lines add convenience, compatibility, and interactive polish—but the fundamental architecture is the same one you have studied in this book.

Appendix A

Debugging

`rc` has several built-in debugging aids. The `-x` flag traces each simple command before execution (printing the command and its arguments). The `-v` flag echoes each input line before parsing. This appendix presents the code behind these flags, as well as the AST pretty-printer used to serialize function bodies into the `/env/` filesystem.

A.1 AST dumper

⟨*global* `nl` 173a⟩≡ (235a)
```
char nl='\n';  /* change to semicolon for bourne-proofing */
```
Uses `nl` 173a.

`pcmd` is the inverse of the parser: it walks an AST and prints it back as `rc` source text. This is what lets `rc` store a function in `/env` as readable text, and what `whatis` uses to echo a function body. The recursion rides on a custom `%t` format verb, which `pfmt`—`rc`'s own small `printf` replacement, defined later in this appendix rather than borrowed from Plan 9's `print`/`fmt` library—dispatches straight back into `pcmd` for each child node.

⟨*function* `pcmd` 173b⟩≡ (235a)
```
/// (simplemung | fnstr) -> pfmt %t -> <>
void
pcmd(io *f, tree *t)
{
    if(t==nil)
        return;
    assert(f != nil);

    switch(t->type){
    case '$': pfmt(f, "$%t", c0); break;
    case '"': pfmt(f, "$\"%t", c0); break;
    case '&': pfmt(f, "%t&", c0); break;
    case '^': pfmt(f, "%t^%t", c0, c1); break;
    case '`': pfmt(f, "`%t%t", c0, c1); break;
    case ANDAND:pfmt(f, "%t && %t", c0, c1); break;
    case OROR: pfmt(f, "%t || %t", c0, c1); break;
    case BANG: pfmt(f, "! %t", c0); break;
    case BRACE: pfmt(f, "{%t}", c0); break;
    case COUNT: pfmt(f, "$#%t", c0); break;
    case FN: pfmt(f, "fn %t %t", c0, c1); break;
    case IF: pfmt(f, "if%t%t", c0, c1); break;
    case NOT: pfmt(f, "if not %t", c0); break;
    case PCMD:
    case PAREN: pfmt(f, "(%t)", c0); break;
    case SUB: pfmt(f, "$%t(%t)", c0, c1); break;
    case SIMPLE: pfmt(f, "%t", c0); break;
```

```
case SUBSHELL: pfmt(f, "@ %t", c0); break;
case SWITCH: pfmt(f, "switch %t %t", c0, c1); break;
case TWIDDLE: pfmt(f, "~ %t %t", c0, c1); break;
case WHILE:  pfmt(f, "while %t%t", c0, c1); break;

case ARGLIST:
    if(c0==nil)
        pfmt(f, "%t", c1);
    else if(c1==nil)
        pfmt(f, "%t", c0);
    else
        pfmt(f, "%t %t", c0, c1);
    break;
case ';':
    if(c0){
        if(c1)
            pfmt(f, "%t%c%t", c0, nl, c1);
        else pfmt(f, "%t", c0);
    }
    else pfmt(f, "%t", c1);
    break;
case WORDS:
    if(c0)
        pfmt(f, "%t ", c0);
    pfmt(f, "%t", c1);
    break;
case FOR:
    pfmt(f, "for(%t", c0);
    if(c1)
        pfmt(f, " in %t", c1);
    pfmt(f, ")%t", c2);
    break;
case WORD:
    if(t->quoted)
        pfmt(f, "%Q", t->str);
    else pdeglob(f, t->str);
    break;
case DUP:
    if(t->rtype==DUPFD)
        pfmt(f, ">[%d=%d]", t->fd1, t->fd0); /* yes, fd1, then fd0; read lex.c */
    else
        pfmt(f, ">[%d=]", t->fd0);
    pfmt(f, "%t", c1);
    break;
case PIPEFD:
case REDIR:
    switch(t->rtype){
    case HERE:
        pchr(f, '<');
    case READ:
    case RDWR:
        pchr(f, '<');
        if(t->rtype==RDWR)
            pchr(f, '>');
        if(t->fd0!=0)
            pfmt(f, "[%d]", t->fd0);
        break;
    case APPEND:
        pchr(f, '>');
    case WRITE:
```

```
            pchr(f, '>');
            if(t->fd0!=1)
                pfmt(f, "[%d]", t->fd0);
            break;
        }
        pfmt(f, "%t", c0);
        if(c1)
            pfmt(f, " %t", c1);
        break;
    case '=':
        pfmt(f, "%t=%t", c0, c1);
        if(c2)
            pfmt(f, " %t", c2);
        break;
    case PIPE:
        pfmt(f, "%t|", c0);
        if(t->fd1==0){
            if(t->fd0!=1)
                pfmt(f, "[%d]", t->fd0);
        }
        else pfmt(f, "[%d=%d]", t->fd0, t->fd1);
        pfmt(f, "%t", c1);
        break;

    default:
        pfmt(f, "bad cmd %d %p %p %p", t->type, c0, c1, c2);
        break;
    }
}
```

Uses ANDAND, APPEND 61d, ARGLIST, BANG, BRACE, COUNT, DUP, DUPFD 163d, FN, FOR, HERE 157a, IF, NOT, OROR, PAREN, PCMD, PIPE, PIPEFD, RDWR 161b, READ 61c, REDIR, SIMPLE, SUB, SUBSHELL, SWITCH, TWIDDLE, WHILE, WORD, WORDS, WRITE 61b, c0-5 175a, c1-6 175b, c2-7 175c, nl 173a, pchr() 188a, pdeglob() 175d, and pfmt() 190b.

These c0/c1/c2 child-accessor macros are the same ones outcode()[76b] uses in the code generator; they are redefined here only because pcmd lives in a separate file. A shared header would let both sides factor them out.

⟨*constant* c0 175a⟩≡ (235a)
```
#define c0 t->child[0]
```

⟨*constant* c1 175b⟩≡ (235a)
```
#define c1 t->child[1]
```

⟨*constant* c2 175c⟩≡ (235a)
```
#define c2 t->child[2]
```

⟨*function* pdeglob 175d⟩≡ (235a)
```
/// pcmd -> <>
void
pdeglob(io *f, char *s)
{
    while(*s){
        if(*s==GLOB)
            s++;
        pchr(f, *s++);
    }
}
```

Uses GLOB 132 and pchr() 188a.

A.2 Bytecode generator trace: `rc -r`

The `-r` flag traces the interpreter: before running each bytecode it prints the process id, the code array, the program counter, the bytecode's name, and its arguments—the output we already saw in Section 2.5.3. The `r` echoes the `runq`, the running thread it dumps on every cycle.

⟨`main()` *debug runq in interpreter loop* 176a⟩≡ (46c)

```
if(flag['r'])
    pfnc(err, runq);
```

Uses `err` 179b, `flag` 43a, `pfnc()` 177a, and `runq` 35a.

⟨*global* `fname` 176b⟩≡ (236b)

```
struct{
    void (*f)(void);
    char *name;
} fname[] = {
    Xappend, "Xappend",
    Xassign, "Xassign",
    Xasync, "Xasync",
    Xbackq, "Xbackq",
    Xbang, "Xbang",
    Xcase, "Xcase",
    Xclose, "Xclose",
    Xconc, "Xconc",
    Xcount, "Xcount",
    Xdelfn, "Xdelfn",
    Xdelhere, "Xdelhere",
    Xdol, "Xdol",
    Xdup, "Xdup",
    Xeflag, "Xeflag",
    (void (*)(void))Xerror, "Xerror",
    Xexit, "Xexit",
    Xfalse, "Xfalse",
    Xfn, "Xfn",
    Xfor, "Xfor",
    Xglob, "Xglob",
    Xif, "Xif",
    Xifnot, "Xifnot",
    Xjump, "Xjump",
    Xlocal, "Xlocal",
    Xmark, "Xmark",
    Xmatch, "Xmatch",
    Xpipe, "Xpipe",
    Xpipefd, "Xpipefd",
    Xpipewait, "Xpipewait",
    Xpopm, "Xpopm",
    Xpopredir, "Xpopredir",
    Xqdol, "Xqdol",
    Xrdcmds, "Xrdcmds",
    Xrdfn, "Xrdfn",
    Xrdwr, "Xrdwr",
    Xread, "Xread",
    Xreturn, "Xreturn",
    Xsimple, "Xsimple",
    Xsub, "Xsub",
    Xsubshell, "Xsubshell",
    Xtrue, "Xtrue",
    Xunlocal, "Xunlocal",
    Xwastrue, "Xwastrue",
    Xword, "Xword",
```

```
    Xwrite, "Xwrite",
    0
};
```

Uses Xappend() 94b, Xassign() 109c, Xasync() 99b, Xbackq() 151c, Xbang() 88d, Xcase() 105a, Xclose() 164e, Xconc() 113b, Xdelfn() 107b, Xdelhere() 159b, Xdup() 164a, Xeflag() 149d, Xerror() 82a, Xexit() 96b, Xfalse() 88b, Xfn() 107a, Xfor() 102b, Xglob() 133d, Xif() 100b, Xifnot() 100f, Xjump() 101g, Xlocal() 110a, Xmark() 77b, Xmatch() 89a, Xpipe() 97c, Xpipefd() 154b, Xpipewait() 98d, Xpopm() 104a, Xpopredir() 92c, Xqdol() 156c, Xrdcmds() 47a, Xrdfn() 127, Xrdwr() 162a, Xread() 93b, Xreturn() 96c, Xsimple() 81, Xsub() 112b, Xsubshell() 155d, Xtrue() 88a, Xunlocal() 110b, Xwastrue() 100d, Xword() 77a, Xwrite() 92b, and __anon_struct_4 176b.

⟨*function* pfnc 177a⟩≡ (236b)

```
void
pfnc(io *fd, thread *t)
{
    int i;
    void (*fn)(void) = t->code[t->pc].f;
    list *a;

    pfmt(fd, "pid %d cycle %p %d ", getpid(), t->code, t->pc);
    for (i = 0; fname[i].f; i++)
        if (fname[i].f == fn) {
            pstr(fd, fname[i].name);
            break;
        }
    if (!fname[i].f)
        pfmt(fd, "%p", fn);
    for (a = t->argv; a; a = a->next)
        pfmt(fd, " (%v)", a->words);
    pchr(fd, '\n');
    flush(fd);
}
```

Uses flush() 187b, fname 176b, pchr() 188a, pfmt() 190b, and pstr() 191a.

A.3 Printing subprocesses status: `rc -s`

The `-s` flag makes `rc` print `$status` after any command that leaves a non-true status, a convenience when watching a script's progress.

⟨Xrdcmds() *print status if -s* 177b⟩≡ (47a)

```
if(flag['s'] && !truestatus())
    pfmt(err, "status=%v\n", vlook("status")->val);
```

Uses err 179b, flag 43a, pfmt() 190b, truestatus() 78c, and vlook() 39c.

A.4 Printing commands: `rc -x`

The `-x` flag echoes each simple command, after expansion, just before it runs—the shell's equivalent of `bash`'s `set -x` tracing.

⟨Xsimple() *if -x* 177c⟩≡ (81)

```
if(flag['x'])
    pfmt(err, "%v\n", p->argv->words); /* wrong, should do redirs */
```

Uses err 179b, flag 43a, and pfmt() 190b.

A.5 Printing characters: `rc -v` and `rc -V`

The `-v` and `-V` flags echo input characters as they are read: `-v` for the user's own typing, `-V` for everything, including the characters pulled from sourced files like `rcmain`.

⟨`getnext()` *after read, if not EOF but verbose mode, print character read* 178⟩≡ (49)

```
else
    if(flag['V'] || ndot>=2 && flag['v'])
        pchr(err, c);
```

Uses `err` 179b, `flag` 43a, `ndot` 120a, and `pchr()` 188a.

Appendix B

Error Management

`rc` has three levels of error response. Syntax and parse errors (reported by `yyerror()`) are recoverable: `rc` prints the error, discards the current input line, and returns to the prompt. Runtime errors (reported by `Xerror()`) set the exit status but continue execution. Fatal errors (reported by `panic()`) print a message and call `abort()`—these indicate bugs in `rc` itself, not in the user's script.

⟨*constant* `ERRMAX` 179a⟩≡ (223b)
```
#define ERRMAX 128
```

`err` is the buffered output stream for diagnostics; it is set once in `main()`[42b], to `openfd(STDERR)`, wrapping file descriptor 2.

⟨*global* `err` 179b⟩≡ (239b)
```
io *err;
```

⟨*global* `nerror` 179c⟩≡ (239b)
```
int nerror;  /* number of errors encountered during compilation */
```

⟨`Xrdcmds()` *flush errors and reset error count* 179d⟩≡ (47a)
```
flush(err);
nerror = 0;
```
Uses `err` 179b, `flush()` 187b, and `nerror` 179c.

⟨*function* `yyerror` 179e⟩≡ (234a)
```
void
yyerror(char *m)
{
    pfmt(err, "rc: ");
    if(runq->cmdfile && !runq->iflag)
        pfmt(err, "%s:%d: ", runq->cmdfile, runq->lineno);
    else if(runq->cmdfile)
        pfmt(err, "%s: ", runq->cmdfile);
    else if(!runq->iflag)
        pfmt(err, "line %d: ", runq->lineno);

    if(tok[0] && tok[0]!='\n')
        pfmt(err, "token %q: ", tok);
    pfmt(err, "%s\n", m);
    flush(err);
    lastword = false;
    lastdol = false;

    while(lastc!='\n' && lastc!=EOF)
        advance();

    nerror++;
```

```
    setvar("status", newword(m, (word *)nil));
}
```

Uses EOF 50a, advance() 53b, err 179b, flush() 187b, lastc 180a, lastdol 60b, lastword 58c, nerror 179c, newword() 36c, pfmt() 190b, runq 35a, setvar() 39b, and tok 56c.

⟨*global* lastc 180a⟩≡ (228)

```
int lastc;
```

⟨advance() *save future in lastc* 180b⟩≡ (53b)

```
lastc = future;
```

Uses future 52e and lastc 180a.

⟨*function* panic 180c⟩≡ (239b)

```
void
panic(char *s, int n)
{
    pfmt(err, "rc: ");
    pfmt(err, s, n);
    pchr(err, '\n');
    flush(err);
    Abort();
}
```

Uses Abort() 180d, err 179b, flush() 187b, pchr() 188a, and pfmt() 190b.

⟨*function* Abort 180d⟩≡ (239b)

```
void
Abort(void)
{
    pfmt(err, "aborting\n");
    flush(err);
    Exit("aborting", __LOC__);
}
```

Uses err 179b, flush() 187b, and pfmt() 190b.

⟨*function* Exit 180e⟩≡ (240b)

```
void
Exit(char *stat, char* _LOC)
{
    USED(_LOC);
    Updenv();
    setstatus(stat);
    exits(truestatus() ? "" : getstatus());
}
```

Uses getstatus() 78b, setstatus() 78a, and truestatus() 78c.

Appendix C

Utilities

This appendix collects the utility code that `rc` uses throughout: memory allocation wrappers (`emalloc`, `erealloc`), the buffered I/O system (the `io` structure and its read/write functions), command-line flag parsing (`getflags`), and string conversion helpers. None of it is specific to a shell—it is the support layer almost any C program needs—which is why it is gathered here in an appendix rather than woven through the main chapters.

C.1 Memory management

⟨*function* `emalloc` 181a⟩≡ (236c)

```
void *
emalloc(long n)
{
    void *p = Malloc(n);

    if(p==nil)
        panic("Can't malloc %d bytes", n);
    return p;
}
```

Uses `panic()` 180c.

Even this generic wrapper reaches into `rc`'s own machinery: on failure it calls `panic()` (Chapter B) rather than returning `nil`, so an out-of-memory condition becomes one of the fatal errors described there.

⟨*function* `Malloc` 181b⟩≡ (240b)

```
void*
Malloc(ulong n)
{
    return mallocz(n, 1);
}
```

⟨*function* `efree` 181c⟩≡ (236c)

```
void
efree(void *p)
{
    if(p)
        free(p);
    else pfmt(err, "free 0\n");
}
```

Uses `err` 179b and `pfmt()` 190b.

C.2 Command-line arguments

`getflags` parses the command line against a *specification string* that names every accepted flag at once—more declarative than Plan 9's usual `ARGBEGIN`/`ARGEND` macros, which you wrap a hand-written loop around. `rc` itself calls it as

```
getflags(argc, argv, "SsrdiIlxepvVc:1m:1[command]", 1);
```

where a bare letter is a boolean flag and a letter followed by `:1` takes one argument. This is the same engine the standalone `aux/getflags` program (Section E.4) later exposes to shell scripts.

⟨*global* `cmdname` 182a⟩≡ (227a)
```
char *cmdname;
```

⟨*global* `flagarg` 182b⟩≡ (227a)
```
static char *flagarg="";
```
Uses `flagarg-16` 182b.

⟨*global* `reason` 182c⟩≡ (227a)
```
static int reason;
```

⟨*constant* `RESET` 182d⟩≡ (227a)
```
#define RESET 1
```

⟨*constant* `FEWARGS` 182e⟩≡ (227a)
```
#define FEWARGS 2
```

⟨*constant* `FLAGSYN` 182f⟩≡ (227a)
```
#define FLAGSYN 3
```

⟨*constant* `BADFLAG` 182g⟩≡ (227a)
```
#define BADFLAG 4
```

⟨*global* `badflag` 182h⟩≡ (227a)
```
static int badflag;
```

⟨*function* `getflags` 182i⟩≡ (227a)
```
int
getflags(int argc, char *argv[], char *flags, int stop)
{
    char *s;
    int i, j, c, count;
    flagarg = flags;
    if(cmdname==0)
        cmdname = argv[0];

    i = 1;
    while(i!=argc){
        if(argv[i][0] != '-' || argv[i][1] == '\0'){
            if(stop)  /* always true in rc */
                return argc;
            i++;
            continue;
        }
        s = argv[i]+1;
        while(*s){
            c=*s++;
            count = scanflag(c, flags);
            if(count==-1)
                return -1;
```

```
            if(flag[c]){ reason = RESET; badflag = c; return -1; }
            if(count==0){
                flag[c] = flagset;
                if(*s=='\0'){
                    for(j = i+1;j<=argc;j++)
                        argv[j-1] = argv[j];
                    --argc;
                }
            }
            else{
                if(*s=='\0'){
                    for(j = i+1;j<=argc;j++)
                        argv[j-1] = argv[j];
                    --argc;
                    s = argv[i];
                }
                if(argc-i<count){
                    reason = FEWARGS;
                    badflag = c;
                    return -1;
                }
                reverse(argv+i, argv+argc);
                reverse(argv+i, argv+argc-count);
                reverse(argv+argc-count+1, argv+argc);
                argc-=count;
                flag[c] = argv+argc+1;
                flag[c][0] = s;
                s="";
            }
        }
    }
    return argc;
}
```

Uses FEWARGS-19 182e, RESET-18 182d, badflag-22 182h, cmdname 182a, flag 43a, flagarg-16 182b, flagset 43c, reason-17 182c, and reverse() 183a.

⟨*function* reverse 183a⟩≡ (227a)

```
static void
reverse(char **p, char **q)
{
    char *t;
    for(;p<q;p++,--q){ t=*p; *p=*q; *q = t; }
}
```

⟨*function* scanflag 183b⟩≡ (227a)

```
static int
scanflag(int c, char *f)
{
    int fc, count;
    if(0<=c && c<NFLAG)
        while(*f){
            if(*f==' '){
                f++;
                continue;
            }
            fc=*f++;
            if(*f==':'){
                f++;
                if(*f<'0' || '9'<*f){ reason = FLAGSYN; return -1; }
                count = 0;
```

```
                while('0'<=*f && *f<='9') count = count*10+*f++-'0';
            }
            else
                count = 0;
            if(*f=='['){
                do{
                    f++;
                    if(*f=='\0'){ reason = FLAGSYN; return -1; }
                }while(*f!=']');
                f++;
            }
            if(c==fc)
                return count;
        }
    reason = BADFLAG;
    badflag = c;
    return -1;
}
```

⟨*function* usage 184⟩≡ (227a)

```
void
usage(char *tail)
{
    char *s, *t, c;
    int count, nflag = 0;
    switch(reason){
    case RESET:
        errs("Flag -");
        errc(badflag);
        errs(": set twice\n");
        break;
    case FEWARGS:
        errs("Flag -");
        errc(badflag);
        errs(": too few arguments\n");
        break;
    case FLAGSYN:
        errs("Bad argument to getflags!\n");
        break;
    case BADFLAG:
        errs("Illegal flag -");
        errc(badflag);
        errc('\n');
        break;
    }
    errs("Usage: ");
    errs(cmdname);
    for(s = flagarg;*s;){
        c=*s;
        if(*s++==' ')
            continue;
        if(*s==':'){
            s++;
            count = 0;
            while('0'<=*s && *s<='9') count = count*10+*s++-'0';
        }
        else count = 0;
        if(count==0){
            if(nflag==0)
                errs(" [-");
```

```
            nflag++;
            errc(c);
        }
        if(*s==’[’){
            s++;
            while(*s!=’]’ && *s!=’\0’) s++;
            if(*s==’]’)
                s++;
        }
    }
    if(nflag)
        errs("]");
    for(s = flagarg;*s;){
        c=*s;
        if(*s++==’ ’)
            continue;
        if(*s==’:’){
            s++;
            count = 0;
            while(’0’<=*s && *s<=’9’) count = count*10+*s++-’0’;
        }
        else count = 0;
        if(count!=0){
            errs(" [-");
            errc(c);
            if(*s==’[’){
                s++;
                t = s;
                while(*s!=’]’ && *s!=’\0’) s++;
                errs(" ");
                errn(t, s-t);
                if(*s==’]’)
                    s++;
            }
            else
                while(count--) errs(" arg");
            errs("]");
        }
        else if(*s==’[’){
            s++;
            while(*s!=’]’ && *s!=’\0’) s++;
            if(*s==’]’)
                s++;
        }
    }
    if(tail){
        errs(" ");
        errs(tail);
    }
    errs("\n");
    Exit("bad flags", "getflags.c");
}
```

⟨*function* errn 185⟩≡ (227a)

```
static void
errn(char *s, int count)
{
    while(count){ errc(*s++); --count; }
}
```

Uses errc() 186e.

⟨*function* `errs` 186a⟩≡ (227a)
```
static void
errs(char *s)
{
    while(*s) errc(*s++);
}
```
Uses `errc()` 186e.

⟨*constant NBUF (rc/getflags.c)* 186b⟩≡ (227a)
```
#define NBUF 80
```

⟨*global* `buf` 186c⟩≡ (227a)
```
static char buf[NBUF];
```
Uses `NBUF-23` 186b.

⟨*global* `bufp` 186d⟩≡ (227a)
```
static char *bufp = buf;
```
Uses `buf-24` 186c and `bufp-25` 186d.

⟨*function* `errc` 186e⟩≡ (227a)
```
static void
errc(int c)
{
    *bufp++=c;
    if(bufp==&buf[NBUF] || c=='\n'){
        write(2, buf, bufp-buf);
        bufp = buf;
    }
}
```
Uses `NBUF-23` 186b, `buf-24` 186c, and `bufp-25` 186d.

C.3 Buffered I/O

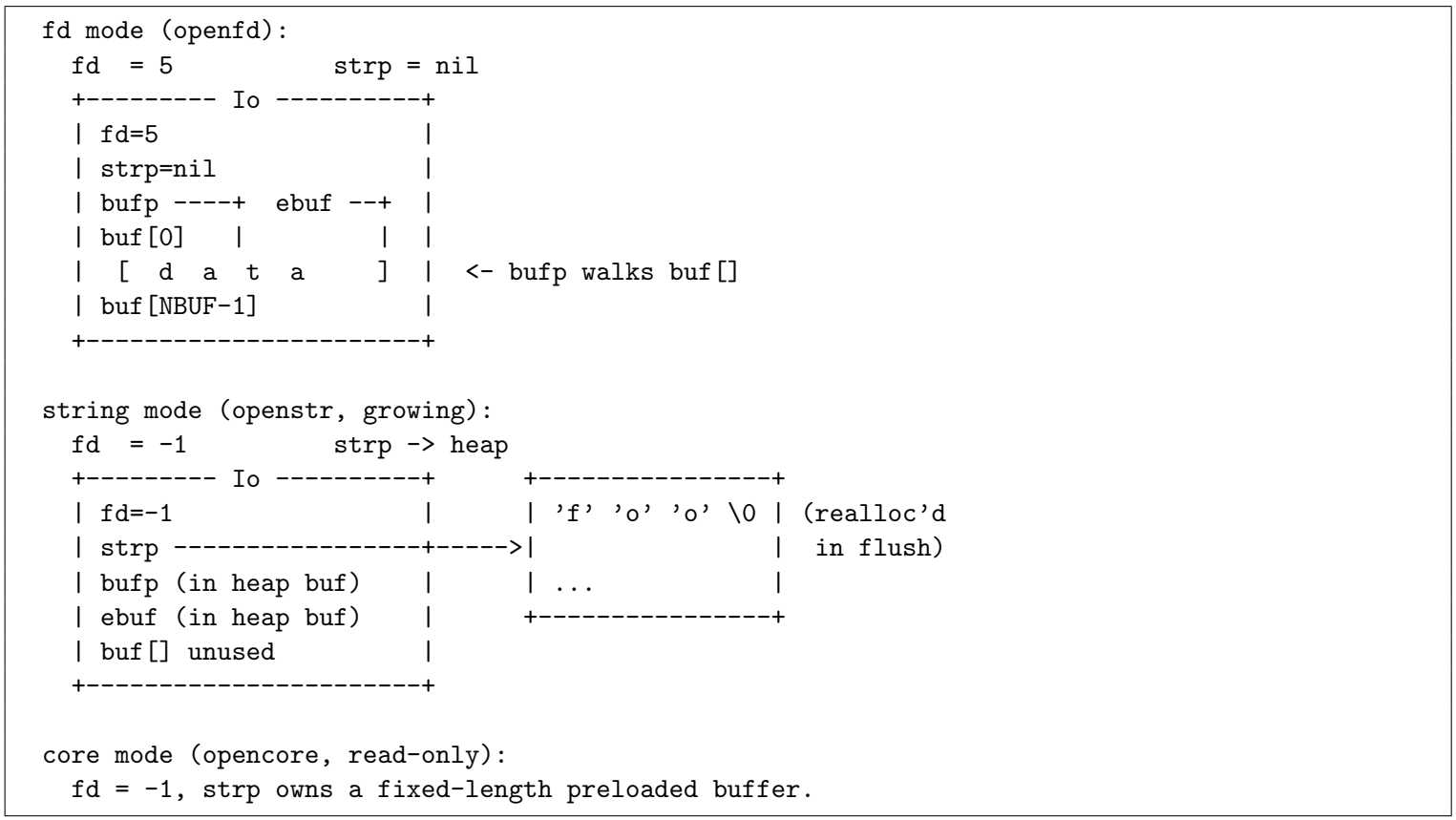
`Io` is `rc`'s own buffered-I/O type: a file descriptor plus a fixed buffer and a cursor. Plan 9 already ships such a thing in `libbio` (the `Biobuf`), but `rc` rolls its own to stay portable—it builds on UNIX too—and free of that dependency, at the cost of re-implementing a few hundred lines. The same motive recurs for the format and string-conversion helpers below.

⟨*struct* `Io` 186f⟩≡ (223a)
```
struct Io {
    fdt fd;
    byte *bufp, *ebuf, *strp;
    byte buf[NBUF];
};
```
Uses `NBUF` 186g.

⟨*constant* `NBUF` 186g⟩≡ (223a)
```
#define NBUF 512
```

⟨*function* `openfd` 186h⟩≡ (227b)
```
io*
openfd(fdt fd)
{
    io *f = new(struct Io);
    f->fd = fd;
    f->bufp = f->ebuf = f->buf;
    f->strp = nil;
    return f;
}
```
Uses `Io` 186f.

The `Io` struct serves three different roles depending on which fields are populated, which is why it looks strangely overloaded. In fd mode (used by `openfd()`[186h]), `fd` is a real file descriptor, `strp` is `nil`, and the inline `buf` array is the read/write window. In string mode (used by `openstr()`[189b]), `fd` is -1, `strp` owns a heap-grown buffer, and `bufp`/`ebuf` walk that buffer instead of `buf`. In core mode (used by `opencore()`[189c]), `fd` is also -1 but `strp` points to a preloaded read-only buffer of known length. The three modes coexist without a tag because `flush()` and `emptybuf()` branch on `f->strp != nil` and `f->fd == -1`.



The unused inline `buf` in string and core modes costs one `NBUF` per `Io` object, but avoids a second allocation in the common fd case.

⟨*enum* `MiscConstants` 187a⟩≡ (227b)
```
enum { Stralloc = 100, };
```

⟨*function* `flush` 187b⟩≡ (227b)
```
void
flush(io *f)
{
    int n;

    if(f->strp){
        n = f->ebuf - f->strp;
        f->strp = realloc(f->strp, n+Stralloc+1);
        if(f->strp==0)
            panic("Can't realloc %d bytes in flush!", n+Stralloc+1);
        f->bufp = f->strp + n;
        f->ebuf = f->bufp + Stralloc;
        memset(f->bufp, '\0', Stralloc+1);
    }
    else{
        n = f->bufp-f->buf;
        if(n && write(f->fd, f->buf, n) != n){
```

```
            write(2, "Write error\n", 12);
            if(ntrap)
                dotrap();
        }
        f->bufp = f->buf;
        f->ebuf = f->buf+NBUF;
    }
}
```

Uses NBUF 186g, Stralloc-1 187a, dotrap() 145b, ntrap 144c, and panic() 180c.

⟨*function* pchr 188a⟩≡ (238b)

```
void
pchr(io *b, int c)
{
    if(b->bufp==b->ebuf)
        fullbuf(b, c);
    else *b->bufp++=c;
}
```

Uses fullbuf() 188b.

⟨*function* fullbuf 188b⟩≡ (227b)

```
int
fullbuf(io *f, int c)
{
    flush(f);
    return *f->bufp++=c;
}
```

Uses flush() 187b.

`rchr` reads a single byte from a buffer, refilling it from the file descriptor when empty. It is the bottom of the input stack: every character the lexer (`yylex()`[55a]) consumes ultimately comes through here.

⟨*function* rchr 188c⟩≡ (227b)

```
int
rchr(io *b)
{
    if(b->bufp==b->ebuf)
        return emptybuf(b);
    return *b->bufp++;
}
```

Uses emptybuf() 188d.

⟨*function* emptybuf 188d⟩≡ (227b)

```
int
emptybuf(io *f)
{
    int n;
    if(f->fd==-1 || (n = read(f->fd, f->buf, NBUF))<=0) return EOF;
    f->bufp = f->buf;
    f->ebuf = f->buf + n;
    return *f->bufp++;
}
```

Uses EOF 50a and NBUF 186g.

⟨*function* rutf 188e⟩≡ (227b)

```
int
rutf(io *b, char *buf, Rune *r)
{
    int n, i, c;
```

```
    c = rchr(b);
    if(c == EOF)
        return EOF;
    *buf = c;
    if(c < Runesync){
        *r = c;
        return 1;
    }
    for(i = 1; (c = rchr(b)) != EOF; ){
        buf[i++] = c;
        buf[i] = 0;
        if(fullrune(buf, i)){
            n = chartorune(r, buf);
            b->bufp -= i - n; /* push back unconsumed bytes */
            assert(b->fd == -1 || b->bufp > b->buf);
            return n;
        }
    }
    /* at eof */
    b->bufp -= i - 1;   /* consume 1 byte */
    *r = Runeerror;
    return runetochar(buf, r);
}
```

Uses EOF 50a and rchr() 188c.

⟨*function* closeio 189a⟩≡ (227b)

```
void
closeio(io *io)
{
    if(io->fd>=0)
        close(io->fd);
    if(io->strp)
        efree(io->strp);
    efree(io);
}
```

Uses efree() 181c.

⟨*function* openstr 189b⟩≡ (227b)

```
/// ??? -> <>
io*
openstr(void)
{
    io *f = new(struct Io);

    f->fd = -1;
    f->bufp = f->strp = emalloc(Stralloc+1);
    f->ebuf = f->bufp + Stralloc;
    memset(f->bufp, '\0', Stralloc+1);
    return f;
}
```

Uses Io 186f, Stralloc-1 187a, and emalloc() 181a.

⟨*function* opencore 189c⟩≡ (227b)

```
/// ??? -> <>
/*
 * Open a corebuffer to read.  EOF occurs after reading len
 * characters from buf.
 */
```

```
io*
opencore(char *s, int len)
{
    io *f = new(struct Io);
    uchar *buf = emalloc(len);

    f->fd = -1 /*open("/dev/null", 0)*/;
    f->bufp = f->strp = buf;
    f->ebuf = buf+len;
    Memcpy(buf, s, len);
    return f;
}
```

Uses `Io` 186f, `Memcpy()` 194b, and `emalloc()` 181a.

C.4 Format

`pfmt` is a small `printf`-style formatter with verbs specific to `rc`—`%t` for an AST node, `%v` for a word list, `%q` for a quoted word. Plan 9's `fmt` library could host these through `fmtinstall`, but a self-contained formatter avoids that dependency, for the same portability reason as `Io` above.

⟨*global* `pfmtnest` 190a⟩≡ (238b)

```
int pfmtnest = 0;
```

Uses `pfmtnest` 190a.

⟨*function* `pfmt` 190b⟩≡ (238b)

```
void
pfmt(io *f, char *fmt, ...)
{
    va_list ap;
    char err[ERRMAX];

    va_start(ap, fmt);
    pfmtnest++;
    for(;*fmt;fmt++) {
        if(*fmt!='%') {
            pchr(f, *fmt);
            continue;
        }
        if(*++fmt == '\0')  /* "blah%"? */
            break;
        switch(*fmt){
        case 'c':
            pchr(f, va_arg(ap, int));
            break;
        case 'd':
            pdec(f, va_arg(ap, int));
            break;
        case 'o':
            poct(f, va_arg(ap, unsigned int));
            break;
        case 'p':
            pptr(f, va_arg(ap, void*));
            break;
        case 'Q':
            pquo(f, va_arg(ap, char *));
            break;
        case 'q':
```

```
            pwrd(f, va_arg(ap, char *));
            break;
        case 's':
            pstr(f, va_arg(ap, char *));
            break;

        case 'r':
            errstr(err, sizeof err); pstr(f, err);
            break;

        // rc specific, TODO LP split here
        case 't':
            pcmd(f, va_arg(ap, struct Tree *));
            break;
        case 'v':
            pval(f, va_arg(ap, struct Word *));
            break;

        default:
            pchr(f, *fmt);
            break;
        }
    }
    va_end(ap);
    if(--pfmtnest==0)
        flush(f);
}
```

Uses `err` 179b, `flush()` 187b, `pchr()` 188a, `pcmd()` 173b, `pdec()` 191b, `pfmtnest` 190a, `poct()` 192a, `pptr()` 192d, `pquo()` 192b, `pstr()` 191a, `pval()` 192e, and `pwrd()` 192c.

⟨*function* `pstr` 191a⟩≡ (238b)

```
void
pstr(io *f, char *s)
{
    if(s==0)
        s="(null)";
    while(*s) pchr(f, *s++);
}
```

Uses `pchr()` 188a.

⟨*function* `pdec` 191b⟩≡ (238b)

```
void
pdec(io *f, int n)
{
    if(n<0){
        n=-n;
        if(n>=0){
            pchr(f, '-');
            pdec(f, n);
            return;
        }
        /* n is two's complement minimum integer */
        n = 1-n;
        pchr(f, '-');
        pdec(f, n/10);
        pchr(f, n%10+'1');
        return;
    }
    if(n>9)
        pdec(f, n/10);
```

```
    pchr(f, n%10+'0');
}
```

Uses pchr() 188a and pdec() 191b.

⟨*function* poct 192a⟩≡ (238b)

```
void
poct(io *f, unsigned int n)
{
    if(n>7)
        poct(f, n>>3);
    pchr(f, (n&7)+'0');
}
```

Uses pchr() 188a and poct() 192a.

⟨*function* pquo 192b⟩≡ (238b)

```
void
pquo(io *f, char *s)
{
    pchr(f, '\'');
    for(;*s;s++)
        if(*s=='\'')
            pfmt(f, "''");
        else pchr(f, *s);
    pchr(f, '\'');
}
```

Uses pchr() 188a and pfmt() 190b.

⟨*function* pwrd 192c⟩≡ (238b)

```
void
pwrd(io *f, char *s)
{
    char *t;
    for(t = s;*t;t++) if(*t >= 0 && needsrcquote(*t)) break;
    if(t==s || *t)
        pquo(f, s);
    else pstr(f, s);
}
```

Uses needsrcquote() 241, pquo() 192b, and pstr() 191a.

⟨*function* pptr 192d⟩≡ (238b)

```
void
pptr(io *f, void *v)
{
    int n;
    uintptr p;

    p = (uintptr)v;
    if(sizeof(uintptr) == sizeof(uvlong) && p>>32)
        for(n = 60;n>=32;n-=4) pchr(f, "0123456789ABCDEF"[(p>>n)&0xF]);

    for(n = 28;n>=0;n-=4) pchr(f, "0123456789ABCDEF"[(p>>n)&0xF]);
}
```

Uses pchr() 188a.

⟨*function* pval 192e⟩≡ (238b)

```
void
pval(io *f, word *a)
{
    if(a){
```

```
        while(a->next && a->next->word){
            pwrd(f, (char *)a->word);
            pchr(f, ' ');
            a = a->next;
        }
        pwrd(f, (char *)a->word);
    }
}
```

Uses `pchr()` 188a and `pwrd()` 192c.

C.5 String conversions

Small integer-to-string and string-to-integer helpers. Like the I/O and format code, they duplicate what a C library already offers, again traded for keeping `rc` self-contained and portable.

⟨*global* `bp` 193a⟩≡ (236c)

```
char *bp;
```

⟨*function* `inttoascii` 193b⟩≡ (236c)

```
void
inttoascii(char *s, long n)
{
    bp = s;
    iacvt(n);
    *bp='\0';
}
```

Uses `bp` 193a and `iacvt()` 193c.

⟨*function* `iacvt` 193c⟩≡ (236c)

```
static void
iacvt(int n)
{
    if(n<0){
        *bp++='-';
        n=-n; /* doesn't work for n==-inf */
    }
    if(n/10)
        iacvt(n/10);
    *bp++=n%10+'0';
}
```

Uses `bp` 193a and `iacvt()` 193c.

C.6 Plan 9 and UNIX compatibility layer

⟨*function* `Write` 193d⟩≡ (236c)

⟨*function* `Read` 193e⟩≡ (236c)

⟨*function* `Seek` 193f⟩≡ (236c)

⟨*function* `Creat` 193g⟩≡ (236c)

```
int
Creat(char *file)
{
    return create(file, 1, 0666L);
}
```

⟨*function* `Dup` 194a⟩≡ (236c)

⟨*function* `Memcpy` 194b⟩≡ (236c)

```
void
Memcpy(void *a, void *b, long n)
{
    memmove(a, b, n);
}
```

Appendix D

Examples of `rc` scripts

We have already read two real `rc` scripts in passing: `boot.rc` (Section 2.3), which brings up a Plan 9 machine, and `/rc/lib/rcmain` (Section 11.5), which finishes `rc`'s own startup. Both were short and used only a handful of constructs. The login `profile` script and the few shell-adjacent programs collected here show more of the language and its ecosystem in practice.

D.1 `/usr/glenda/lib/profile`

This is Glenda's (the default Plan 9 user) login profile, sourced by `rcmain` when `rc` is invoked with the `-l` (login shell) flag.

⟨`rcmain.rc` *if* `-l`*, source user's profile* 195a⟩≡ (148b 130d)
```
if(flag l && /bin/test -r $home/lib/profile) . $home/lib/profile
```

It uses `bind` to overlay the user's personal `bin` directories onto `/bin` (so user scripts and binaries are found automatically), sets up the font and mail filesystem (`upas/fs`), and then dispatches based on `$service`: on a terminal, it starts the window manager `rio`; on a CPU server, it connects to the terminal's devices via `/mnt/term`.

⟨*/usr/glenda/lib/profile* 195b⟩≡
```
bind -a $home/bin/rc /bin
bind -a $home/bin/$cputype /bin
bind -c tmp /tmp
if(! syscall create /tmp/xxx 1 0666 >[2]/dev/null)
        ramfs   # in case we're running off a cd
font = /lib/font/bit/pelm/euro.9.font
upas/fs
fn cd { builtin cd $* && awd }  # for acme
switch($service){
case terminal
        plumber
        echo -n accelerated > '#m/mousectl'
        echo -n 'res 3' > '#m/mousectl'
        prompt=('term% ' '      ')
        fn term%{ $* }
        exec rio -i riostart
case cpu
        if (test -e /mnt/term/mnt/wsys) { # rio already running
                bind -a /mnt/term/mnt/wsys /dev
                if(test -w /dev/label)
                        echo -n $sysname > /dev/label
        }
        bind /mnt/term/dev/cons /dev/cons
        bind /mnt/term/dev/consctl /dev/consctl
```

```
        bind -a /mnt/term/dev /dev
        prompt=('cpu% ' '       ')
        fn cpu%{ $* }
        news
        if (! test -e /mnt/term/mnt/wsys) {     # cpu call from drawterm
                font=/lib/font/bit/pelm/latin1.8.font
                exec rio
        }
case con
        prompt=('cpu% ' '       ')
        news
}
```

D.2 /rc/bin/man

⟨*/rc/bin/man* 196⟩≡

```
#!/bin/rc
# man - print manual pages
rfork e

. /docs/man/fonts

cmd=n
sec=()
S=/docs/man
d=0

fn roff {
        preproc=()
        postproc=cat
        x='{doctype $2}
        if (~ $1 t) {
                if(~ $x *grap*)
                        preproc=($preproc grap)
                if(~ $x *pic*)
                        preproc=($preproc pic)
                Nflag=-Tutf
        }
        if not {
                Nflag='-N'
                Lflag='-rL1000i'
                # setting L changes page length to infinity (sed script removes empty lines)
                if (grep -s '^\.(2C|sp *[0-9]*\.)' $2)
                        postproc=col
        }
        if(~ $x *eqn*)
                preproc=($preproc eqn)
        if(~ $x *tbl*)
                preproc=($preproc tbl)
        {echo -n $FONTS; cat $2 </dev/null} |
                switch($#preproc) {
                case 0
                        troff $Nflag $Lflag -$MAN
                case 1
                        $preproc | troff $Nflag $Lflag -$MAN
                case 2
                        $preproc(1) | $preproc(2) | troff $Nflag $Lflag -$MAN
                case 3
```

```
                        $preproc(1) | $preproc(2) | $preproc(3) |
                                troff $Nflag $Lflag -$MAN
                case *
                        $preproc(1) | $preproc(2) | $preproc(3) |
                                $preproc(4) | troff $Nflag $Lflag -$MAN
                } | $postproc
}

fn page {
        if(test -d /mnt/wsys/acme)
                /bin/page -w
        if not
                /bin/page
}


search=yes
while(~ $d 0) {
        if(~ $#* 0) {
                echo 'Usage: man [-bntpPSw] [0-9] [0-9] ... name1 name2 ...' >[1=2]
                exit
        }
        if(test -d $S/$1){
                sec=($sec $1)
                shift
        }
        if not
                switch($1) {
                case -b ; cmd=b ; shift
                case -n ; cmd=n ; shift
                case -P ; cmd=P ; shift
                case -p ; cmd=p ; shift
                case -S ; search=no ; shift
                case -t ; cmd=t ; shift
                case -w ; cmd=w ; shift
                case * ; d=1
                }
}
if(~ $#sec 0) {
        sec=`{ls -pd $S/[0-9]* }
}
ix=$S/$sec/INDEX
if(~ $#* 1) pat='^'^$1^' '
if not pat='^('^`{echo $* | sed 's/ /|/g'}^') '
fils=()
if(~ $search yes)
for(i in $S/$sec){
        if(/bin/test -f $i/INDEX){
                try=`{grep -i $pat $i/INDEX | sed 's/^[^ ]* //' | sort -u}
                if(! ~ $#try 0)
                        fils=($fils $i/$try)
        }
}
# bug: should also do following loop if not all pages found
if(~ $#fils 0) {
        # nothing in INDEX. try for file of given name
        for(i) {
                if(~ $i intro) i=0intro
                for(n in $sec) {
                        try=`{echo $S/$n/$i | tr A-Z a-z}
```

```
                        if (/bin/test -f $try)
                                fils=($fils $try)
                }
        }
        if(~ $#fils 0) {
                echo 'man: no manual page' >[1=2]
                exit 'no man'
        }
}
for(i in $fils) {
        if(! /bin/test -f $i)
                echo need $i >[1=2]
        if not {
                switch($cmd) {
                case w
                        echo $i

                case t
                        roff t $i

                case p
                        roff t $i | grep -v '^x X html' | proof

                case P
                        roff t $i | page

                case n
                        roff n $i | sed '
                                ${
                                        /^$/p
                                }
                                //N
                                /^\n$/D'

                case b
                        x=`{echo $i | sed 's;/sys/man/(.*)/(.*);\1 \2;'}
                        if(~ $x(2) 0intro) x=($x(1) intro)
                        roff n $i | sed '
                                ${
                                        /^$/p
                                }
                                //N
                                /^\n$/D' |
                        plumb -i -d edit -a 'action=showdata filename=/man/'$x(2)^'('$x(1)^')'
                }
        }
}
```

Appendix E

Shell Companion Utilities

This appendix presents the source code of a few small programs that are closely tied to the shell. They are too simple to deserve their own book, but they are used so frequently in shell scripts and interactive sessions that understanding their internals is worthwhile. Several of them—`echo`, `test`, `read`—are built into `bash` and most other shells; Plan 9 keeps them as separate external programs, true to its preference for small composable tools over a fat shell. For more general Plan 9 utilities, see UTILITIES book [Pad25].

E.1 `echo`

`echo` concatenates its arguments separated by spaces, followed by a newline. The `-n` flag suppresses the trailing newline. Note that it builds the entire output in a single buffer and writes it with one `write()` call, so the output is atomic—important when multiple processes write to the same file descriptor.

⟨*function* `main`*(echo)* 199⟩≡ (252a)

```
void
main(int argc, char *argv[])
{
    bool nflag;
    int i, len;
    char *buf, *p;

    nflag = false;
    if(argc > 1 && strcmp(argv[1], "-n") == 0)
        nflag = true;

    len = 1;
    for(i = 1+nflag; i < argc; i++)
        len += strlen(argv[i])+1;

    buf = malloc(len);
    if(buf == 0)
        exits("no memory");

    p = buf;
    for(i = 1+nflag; i < argc; i++){
        strcpy(p, argv[i]);
        p += strlen(p);
        if(i < argc-1)
            *p++ = ' ';
    }

    if(!nflag)
        *p++ = '\n';
```

```
    if(write(STDOUT, buf, p-buf) < 0){
        fprint(STDERR, "echo: write error: %r\n");
        exits("write error");
    }

    exits((char *)nil);
}
```

E.2 `read`

`read` reads one line from standard input and prints it. In Plan 9 it is an external program, not a shell builtin: you capture a line with the command substitution `x=‘{read}`, where `bash` would write the built-in `read x`. Keeping it outside the shell is the same minimalist choice `rc` makes for `test` and `echo`.

⟨*function* `main`*(read)* 200⟩≡ (252b)

```
void
main(int argc, char *argv[])
{
    int i, fd;
    char *s;

    ARGBEGIN{
    case 'm':
        multi = 1;
        break;
    case 'n':
        s = ARGF();
        if(s){
            nlines = atoi(s);
            break;
        }
        /* fall through */
    default:
        fprint(2, "usage: read [-m] [-n nlines] [files...]\n");
        exits("usage");
    }ARGEND

    if(argc == 0)
        lines(0, "<stdin>");
    else
        for(i=0; i<argc; i++){
            fd = open(argv[i], OREAD);
            if(fd < 0){
                fprint(2, "read: can't open %s: %r\n", argv[i]);
                exits("open");
            }
            lines(fd, argv[i]);
            close(fd);
        }

    exits(status);
}
```

E.3 `test`

`test` evaluates conditional expressions, used heavily in shell scripts (e.g., `if(test -f foo) ...`). It is an

external program, not a builtin—unlike in `bash` where `test` and `[` are built in. The implementation is a small recursive-descent parser with three precedence levels: `-o` (or), `-a` (and), and `!` (not), followed by the primary tests (`-f`, `-d`, `-r`, `-eq`, etc.).

⟨*function* `EQ` 201a⟩≡ (253)
```
#define EQ(a,b) ((tmp=a)==0?0:(strcmp(tmp,b)==0))
```

⟨*global* `ap` 201b⟩≡ (253)
```
int ap;
```

⟨*global* `ac` 201c⟩≡ (253)
```
int ac;
```

⟨*global* `av` 201d⟩≡ (253)
```
char **av;
```

⟨*global* `tmp` 201e⟩≡ (253)
```
static char *tmp;
```

⟨*function* `main`*(test)* 201f⟩≡ (253)
```
void
main(int argc, char *argv[])
{
    int r;
    char *c;

    ac = argc; av = argv; ap = 1;
    if(EQ(argv[0],"[")) {
        if(!EQ(argv[--ac],"]"))
            synbad("] missing","");
    }
    argv[ac] = 0;
    if (ac<=1)
        exits("usage");
    r = e();
    /*
     * nice idea but short-circuit -o and -a operators may have
     * not consumed their right-hand sides.
     */
    if(false && (c = nxtarg(1)) != nil)
        synbad("unexpected operator/operand: ", c);
    exits(r? nil : "false");
}
```

⟨*function* `nxtarg` 201g⟩≡ (253)
```
char *
nxtarg(int mt)
{
    if(ap>=ac){
        if(mt){
            ap++;
            return(0);
        }
        synbad("argument expected","");
    }
    return(av[ap++]);
}
```

⟨*function* nxtintarg 202a⟩≡ (253)

```
bool
nxtintarg(int *pans)
{
    if(ap<ac && isint(av[ap], pans)){
        ap++;
        return true;
    }
    return false;
}
```

⟨*function* e 202b⟩≡ (253)

```
int
e(void)
{
    int p1;

    p1 = e1();
    if (EQ(nxtarg(1), "-o"))
        return(p1 || e());
    ap--;
    return(p1);
}
```

⟨*function* e1 202c⟩≡ (253)

```
int
e1(void)
{
    int p1;

    p1 = e2();
    if (EQ(nxtarg(1), "-a"))
        return (p1 && e1());
    ap--;
    return(p1);
}
```

⟨*function* e2 202d⟩≡ (253)

```
int
e2(void)
{
    if (EQ(nxtarg(0), "!"))
        return(!e2());
    ap--;
    return(e3());
}
```

⟨*function* e3 202e⟩≡ (253)

```
bool
e3(void)
{
    int p1, int1, int2;
    char *a, *p2;

    a = nxtarg(0);
    if(EQ(a, "(")) {
        p1 = e();
        if(!EQ(nxtarg(0), ")"))
            synbad(") expected","");
        return(p1);
```

```
}

if(EQ(a, "-A"))
    return(hasmode(nxtarg(0), DMAPPEND));

if(EQ(a, "-L"))
    return(hasmode(nxtarg(0), DMEXCL));

if(EQ(a, "-T"))
    return(hasmode(nxtarg(0), DMTMP));

if(EQ(a, "-f"))
    return(isreg(nxtarg(0)));

if(EQ(a, "-d"))
    return(isdir(nxtarg(0)));

if(EQ(a, "-r"))
    return(tio(nxtarg(0), 4));

if(EQ(a, "-w"))
    return(tio(nxtarg(0), 2));

if(EQ(a, "-x"))
    return(tio(nxtarg(0), 1));

if(EQ(a, "-e"))
    return(tio(nxtarg(0), 0));

if(EQ(a, "-c"))
    return false;

if(EQ(a, "-b"))
    return false;

if(EQ(a, "-u"))
    return false;

if(EQ(a, "-g"))
    return false;

if(EQ(a, "-s"))
    return(fsizep(nxtarg(0)));

if(EQ(a, "-t"))
    if(ap>=ac)
        return(isatty(1));
    else if(nxtintarg(&int1))
        return(isatty(int1));
    else
        synbad("not a valid file descriptor number ", "");

if(EQ(a, "-n"))
    return(!EQ(nxtarg(0), ""));
if(EQ(a, "-z"))
    return(EQ(nxtarg(0), ""));

p2 = nxtarg(1);
if (p2==0)
    return(!EQ(a,""));
```

```
    if(EQ(p2, "="))
        return(EQ(nxtarg(0), a));

    if(EQ(p2, "!="))
        return(!EQ(nxtarg(0), a));

    if(EQ(p2, "-older"))
        return(isolder(nxtarg(0), a));

    if(EQ(p2, "-ot"))
        return(isolderthan(nxtarg(0), a));

    if(EQ(p2, "-nt"))
        return(isnewerthan(nxtarg(0), a));

    if(!isint(a, &int1))
        synbad("unexpected operator/operand: ", p2);

    if(nxtintarg(&int2)){
        if(EQ(p2, "-eq"))
            return(int1==int2);
        if(EQ(p2, "-ne"))
            return(int1!=int2);
        if(EQ(p2, "-gt"))
            return(int1>int2);
        if(EQ(p2, "-lt"))
            return(int1<int2);
        if(EQ(p2, "-ge"))
            return(int1>=int2);
        if(EQ(p2, "-le"))
            return(int1<=int2);
    }

    synbad("unknown operator ",p2);
    return false;  /* to shut ken up */
}
```

⟨*function* tio 204a⟩≡ (253)

```
bool
tio(char *a, int f)
{
    return access (a, f) >= 0;
}
```

⟨*function* hasmode 204b⟩≡ (253)

```
/*
 * note that the name strings pointed to by Dir members are
 * allocated with the Dir itself (by the same call to malloc),
 * but are not included in sizeof(Dir), so copying a Dir won't
 * copy the strings it points to.
 */
bool
hasmode(char *f, ulong m)
{
    bool r;
    Dir *dir;

    dir = dirstat(f);
    if (dir == nil)
        return false;
```

```
    r = (dir->mode & m) != 0;
    free(dir);
    return r;
}
```

⟨*function* isdir 205a⟩≡ (253)

```
bool
isdir(char *f)
{
    return hasmode(f, DMDIR);
}
```

⟨*function* isreg 205b⟩≡ (253)

```
bool
isreg(char *f)
{
    bool r;
    Dir *dir;

    dir = dirstat(f);
    if (dir == nil)
        return false;
    r = (dir->mode & DMDIR) == 0;
    free(dir);
    return r;
}
```

⟨*function* isatty 205c⟩≡ (253)

```
int
isatty(int fd)
{
    int r;
    Dir *d1, *d2;

    d1 = dirfstat(fd);
    d2 = dirstat("/dev/cons");
    if (d1 == nil || d2 == nil)
        r = 0;
    else
        r = d1->type == d2->type && d1->dev == d2->dev &&
            d1->qid.path == d2->qid.path;
    free(d1);
    free(d2);
    return r;
}
```

⟨*function* fsizep 205d⟩≡ (253)

```
bool
fsizep(char *f)
{
    bool r;
    Dir *dir;

    dir = dirstat(f);
    if (dir == nil)
        return false;
    r = dir->length > 0;
    free(dir);
    return r;
}
```

⟨*function* synbad 206a⟩≡ (253)

```
void
synbad(char *s1, char *s2)
{
    int len;

    write(2, "test: ", 6);
    if ((len = strlen(s1)) != 0)
        write(2, s1, len);
    if ((len = strlen(s2)) != 0)
        write(2, s2, len);
    write(2, "\n", 1);
    exits("bad syntax");
}
```

⟨*function* isint 206b⟩≡ (253)

```
int
isint(char *s, int *pans)
{
    char *ep;

    *pans = strtol(s, &ep, 0);
    return (*ep == 0);
}
```

⟨*function* isolder 206c⟩≡ (253)

```
bool
isolder(char *pin, char *f)
{
    int r, rel;
    ulong n, m;
    char *p = pin;
    Dir *dir;

    dir = dirstat(f);
    if (dir == nil)
        return false;

    /* parse time */
    n = 0;
    rel = 0;
    while(*p){
        m = strtoul(p, &p, 0);
        switch(*p){
        case 0:
            n = m;
            break;
        case 'y':
            m *= 12;
            /* fall through */
        case 'M':
            m *= 30;
            /* fall through */
        case 'd':
            m *= 24;
            /* fall through */
        case 'h':
            m *= 60;
            /* fall through */
        case 'm':
```

```
            m *= 60;
            /* fall through */
        case 's':
            n += m;
            p++;
            rel = 1;
            break;
        default:
            synbad("bad time syntax, ", pin);
        }
    }
    if (!rel)
        m = n;
    else{
        m = time(0);
        if (n > m)  /* before epoch? */
            m = 0;
        else
            m -= n;
    }
    r = dir->mtime < m;
    free(dir);
    return r;
}
```

⟨*function* isolderthan 207a⟩≡ (253)

```
int
isolderthan(char *a, char *b)
{
    int r;
    Dir *ad, *bd;

    ad = dirstat(a);
    bd = dirstat(b);
    if (ad == nil || bd == nil)
        r = 0;
    else
        r = ad->mtime > bd->mtime;
    free(ad);
    free(bd);
    return r;
}
```

⟨*function* isnewerthan 207b⟩≡ (253)

```
int
isnewerthan(char *a, char *b)
{
    int r;
    Dir *ad, *bd;

    ad = dirstat(a);
    bd = dirstat(b);
    if (ad == nil || bd == nil)
        r = 0;
    else
        r = ad->mtime < bd->mtime;
    free(ad);
    free(bd);
    return r;
}
```

E.4 aux/getflags

`getflags` parses command-line flags according to a specification string, letting shell scripts do option handling the same disciplined way C programs do with `getflags()`. Many Plan 9 commands rely on it, and Git book [?]'s `git9` extended it.

⟨*function* `main`*(getflags)* 208⟩≡ (254)

```
void
main(int argc, char *argv[])
{
    char *flags, *p, *s, *e, buf[512];
    int i, n;
    Rune r;
    Fmt fmt;

    quotefmtinstall();
    argv0 = argv[0];    /* for sysfatal */

    flags = getenv("flagfmt");
    if(flags == nil){
        fprint(2, "$flagfmt not set\n");
        print("exit 'missing flagfmt'");
        exits(0);
    }

    fmtfdinit(&fmt, 1, buf, sizeof buf);
    for(p=skipspace(flags); *p; p=nextarg(p)){
        s = e = nil;
        n = chartorune(&r, p);
        if(p[n] == ':'){
            s = p + n + 1;
            e = argname(s);
        }
        if(s != e)
            fmtprint(&fmt, "%.*s=()\n", (int)(e - s), s);
        else
            fmtprint(&fmt, "flag%C=()\n", r);
    }
    ARGBEGIN{
    default:
        if((p = findarg(flags, ARGC())) == nil)
            usage();
        p += runelen(ARGC());
        s = p + 1;
        e = p + 1;
        if(*p == ':' && (e = argname(s)) != s)
            p = e;
        if(*p == ',' || *p == 0){
            if(s != e)
                fmtprint(&fmt, "%.*s=1\n", (int)(e - s), s);
            else
                fmtprint(&fmt, "flag%C=1\n", ARGC());
            break;
        }
        n = countargs(p);
        if(s != e)
            fmtprint(&fmt, "%.*s=(", (int)(e - s), s);
        else
            fmtprint(&fmt, "flag%C=(", ARGC());
        for(i=0; i<n; i++)
```

```
            fmtprint(&fmt, "%s%q", i ? " " : "", EARGF(usage()));
        fmtprint(&fmt, ")\n");
    }ARGEND

    fmtprint(&fmt, "*=(");
    for(i=0; i<argc; i++)
        fmtprint(&fmt, "%s%q", i ? " " : "", argv[i]);
    fmtprint(&fmt, ")\n");
    fmtprint(&fmt, "status=''\n");
    fmtfdflush(&fmt);
    exits(0);
}
```

E.5 aux/usage

`usage` is the companion to `getflags`: it prints a standard "usage: ..." diagnostic and exits non-zero, giving scripts a uniform way to complain about bad arguments.

⟨*function* `main`*(usage)* 209⟩≡ (256)

```
void
main(void)
{
    Fmt fmt;
    char buf[512];
    char *argv0, *args, *flags, *p, *p0;
    int single;
    Rune r;

    argv0 = getenv("0");
    if((p = strrchr(argv0, '/')) != nil)
        argv0 = p+1;
    flags = getenv("flagfmt");
    args = getenv("args");

    if(argv0 == nil){
        fprint(2, "aux/usage: $0 not set\n");
        exits("$0");
    }
    if(flags == nil)
        flags = "";
    if(args == nil)
        args = "";

    fmtfdinit(&fmt, 2, buf, sizeof buf);
    fmtprint(&fmt, "usage: %s", argv0);
    if(flags[0]){
        single = 0;
        for(p=flags; *p; ){
            while(isspace(*p))
                p++;
            p += chartorune(&r, p);
            if(*p == ':')
                while(*p && *p != ',' && !isspace(*p))
                    p++;
            if(*p == ',' || *p == 0){
                if(!single){
                    fmtprint(&fmt, " [-");
                    single = 1;
```

```
                }
                fmtprint(&fmt, "%C", r);
                if(*p == ',')
                    p++;
                continue;
            }
            while(isspace(*p))
                p++;
            if(single){
                fmtprint(&fmt, "]");
                single = 0;
            }
            p0 = p;
            p = strchr(p0, ',');
            if(p == nil)
                p = "";
            else
                *p++ = 0;
            fmtprint(&fmt, " [-%C %s]", r, p0);
        }
        if(single)
            fmtprint(&fmt, "]");
    }
    if(args)
        fmtprint(&fmt, " %s", args);
    fmtprint(&fmt, "\n");
    fmtfdflush(&fmt);
    exits("usage");
}
```

Appendix F

sh, the simple shell

This is a minimal shell, useful during early stages of porting Plan 9 to a new architecture when `rc` is not yet available. It supports basic command execution, pipes, redirections, and backgrounding, but has no variables, functions, or control flow. Comparing it to `rc` helps appreciate how much complexity the full shell adds.

⟨*constant* MAXLINE 211a⟩≡ (250)
```
#define MAXLINE 200  /* maximum line length */
```

⟨*constant* WORD 211b⟩≡ (250)
```
#define WORD 256  /* token code for words */
```

⟨*constant* EOF 211c⟩≡ (250)
```
#define EOF -1   /* token code for end of file */
```

⟨*function* ispunct 211d⟩≡ (250)
```
#define ispunct(c)  (c=='|' || c=='&' || c==';' || c=='<' || \
                 c=='>' || c=='(' || c==')' || c=='\n')
```

⟨*function* isspace 211e⟩≡ (250)
```
#define isspace(c)  (c==' ' || c=='\t')
```

⟨*function* execute 211f⟩≡ (250)
```
#define execute(np)  (ignored = (np? (*(np)->op)(np) : 0))
```

⟨*struct* Node 211g⟩≡ (250)
```
struct Node {   /* parse tree node */
    int (*op)(Node *);  /* operator function */
    Node *args[2];  /* argument nodes */
    char *argv[100]; /* argument pointers */
    char *io[3];  /* i/o redirection */
};
```

⟨*global* nodes 211h⟩≡ (250)
```
Node nodes[25];  /* node pool */
```

⟨*global* nfree 211i⟩≡ (250)
```
Node *nfree;   /* next available node */
```

⟨*global* strspace 211j⟩≡ (250)
```
char strspace[10*MAXLINE]; /* string storage */
```

⟨*global* sfree 211k⟩≡ (250)
```
char *sfree;   /* next free character in strspace */
```

⟨*global* t 211l⟩≡ (250)
```
int t;   /* current token code */
```

⟨*global* token 212a⟩≡ (250)
```
char  *token;   /* current token text (in strspace) */
```

⟨*global* putback 212b⟩≡ (250)
```
int putback = 0;  /* lookahead */
```

⟨*global* status 212c⟩≡ (250)
```
char status[256];  /* exit status of most recent command */
```

⟨*global* cflag 212d⟩≡ (250)
```
int cflag = 0;  /* command is argument to sh */
```

⟨*global* tflag 212e⟩≡ (250)
```
int tflag = 0;  /* read only one line */
```

⟨*global* interactive 212f⟩≡ (250)
```
bool interactive = false; /* prompt */
```

⟨*global* cflagp 212g⟩≡ (250)
```
char *cflagp;  /* command line for cflag */
```

⟨*global* path 212h⟩≡ (250)
```
char *path[] ={"/bin", 0};
```

⟨*global* ignored 212i⟩≡ (250)
```
int ignored;
```

⟨*function* main*(sh)* 212j⟩≡ (250)
```
void
main(int argc, char *argv[])
{
    Node *np;

    if(argc>1 && strcmp(argv[1], "-t")==0)
        tflag++;
    else if(argc>2 && strcmp(argv[1], "-c")==0){
        cflag++;
        cflagp = argv[2];
    }else if(argc>1){
        close(0);
        if(open(argv[1], 0) != 0){
            error(": can't open", argv[1]);
            exits("argument");
        }
    }else
        interactive = true;
    for(;;){
        if(interactive)
            fprint(2, "%d$ ", getpid());
        nfree = nodes;
        sfree = strspace;
        if((t=gettoken()) == EOF)
            break;
        if(t != '\n')
            if(np = list())
                execute(np);
            else
                error("syntax error", "");
        while(t!=EOF && t!='\n') /* flush syntax errors */
            t = gettoken();
    }
    exits(status);
}
```

⟨*function* alloc 213a⟩≡ (250)

```
/* alloc - allocate for op and return a node */
Node*
alloc(int (*op)(Node *))
{
    if(nfree < nodes+sizeof(nodes)){
        nfree->op = op;
        nfree->args[0] = nfree->args[1] = 0;
        nfree->argv[0] = nfree->argv[1] = 0;
        nfree->io[0] = nfree->io[1] = nfree->io[2] = 0;
        return nfree++;
    }
    error("node storage overflow", "");
    exits("node storage overflow");
    return nil;
}
```

⟨*function* builtin 213b⟩≡ (250)

```
/* builtin - check np for builtin command and, if found, execute it */
bool
builtin(Node *np)
{
    int n = 0;
    char name[MAXLINE];
    Waitmsg *wmsg;

    if(np->argv[1])
        n = strtoul(np->argv[1], 0, 0);
    if(strcmp(np->argv[0], "cd") == 0){
        if(chdir(np->argv[1]? np->argv[1] : "/") == -1)
            error(": bad directory", np->argv[0]);
        return true;
    }else if(strcmp(np->argv[0], "exit") == 0)
        exits(np->argv[1]? np->argv[1] : status);
    else if(strcmp(np->argv[0], "bind") == 0){
        if(np->argv[1]==0 || np->argv[2]==0)
            error("usage: bind new old", "");
        else if(bind(np->argv[1], np->argv[2], 0)==-1)
            error("bind failed", "");
        return true;
//#ifdef asdf
// }else if(strcmp(np->argv[0], "unmount") == 0){
//  if(np->argv[1] == 0)
//   error("usage: unmount [new] old", "");
//  else if(np->argv[2] == 0){
//   if(unmount((char *)0, np->argv[1]) == -1)
//    error("unmount:", "");
//  }else if(unmount(np->argv[1], np->argv[2]) == -1)
//   error("unmount", "");
//  return true;
//#endif
    }else if(strcmp(np->argv[0], "wait") == 0){
        while((wmsg = wait()) != nil){
            strncpy(status, wmsg->msg, sizeof(status)-1);
            if(n && wmsg->pid==n){
                n = 0;
                free(wmsg);
                break;
            }
            free(wmsg);
```

```
        }
        if(n)
            error("wait error", "");
        return true;
    }else if(strcmp(np->argv[0], "rfork") == 0){
        char *p;
        int mask;

        p = np->argv[1];
        if(p == 0 || *p == 0)
            p = "ens";
        mask = 0;

        while(*p)
            switch(*p++){
            case 'n': mask |= RFNAMEG; break;
            case 'N': mask |= RFCNAMEG; break;
            case 'e': mask |= RFENVG; break;
            case 'E': mask |= RFCENVG; break;
            case 's': mask |= RFNOTEG; break;
            case 'f': mask |= RFFDG; break;
            case 'F': mask |= RFCFDG; break;
            case 'm': mask |= RFNOMNT; break;
            default: error(np->argv[1], "bad rfork flag");
            }
        rfork(mask);

        return OK_1;
    }else if(strcmp(np->argv[0], "exec") == 0){
        redirect(np);
        if(np->argv[1] == (char *) 0)
            return 1;
        exec(np->argv[1], &np->argv[1]);
        n = np->argv[1][0];
        if(n!='/' && n!='#' && (n!='.' || np->argv[1][1]!='/'))
            for(n = 0; path[n]; n++){
                sprint(name, "%s/%s", path[n], np->argv[1]);
                exec(name, &np->argv[1]);
            }
        error(": not found", np->argv[1]);
        return true;
    }
    // no builtin found
    return false;
}
```

⟨*function* command 214⟩≡ (250)

```
/* command - ( list ) [ ( < | > | >> ) word ]* | simple */
Node*
command(void)
{
    Node *np;

    if(t != '(')
        return simple();
    np = alloc(xsubshell);
    t = gettoken();
    if((np->args[0]=list())==0 || t!=')')
        return nil;
    while((t=gettoken())=='<' || t=='>')
```

```
        if(!setio(np))
            return nil;
    return np;
}
```

⟨*function* getch 215a⟩≡ (250)

```
/* getch - get next, possibly pushed back, input character */
int
getch(void)
{
    unsigned char c;
    static int done=0;

    if(putback){
        c = putback;
        putback = 0;
    }else if(tflag){
        if(done || read(0, &c, 1)!=1){
            done = 1;
            return EOF;
        }
        if(c == '\n')
            done = 1;
    }else if(cflag){
        if(done)
            return EOF;
        if((c=*cflagp++) == 0){
            done = 1;
            c = '\n';
        }
    }else if(read(0, &c, 1) != 1)
        return EOF;
    return c;
}
```

⟨*function* gettoken 215b⟩≡ (250)

```
/* gettoken - get next token into string space, return token code */
int
gettoken(void)
{
    int c;

    while((c = getch()) != EOF)
        if(!isspace(c))
            break;
    if(c==EOF || ispunct(c))
        return c;
    token = sfree;
    do{
        if(sfree >= strspace+sizeof(strspace) - 1){
            error("string storage overflow", "");
            exits("string storage overflow");
        }
        *sfree++ = c;
    }while((c=getch()) != EOF && !ispunct(c) && !isspace(c));
    *sfree++ = 0;
    putback = c;
    return WORD;
}
```

⟨*function* list 216a⟩≡ (250)

```
/* list - pipeline ( ( ; | & ) pipeline )* [ ; | & ]  (not LL(1), but ok) */
Node*
list(void)
{
    Node *np, *np1;

    np = alloc(0);
    if((np->args[1]=pipeline()) == 0)
        return nil;
    while(t==';' || t=='&'){
        np->op = (t==';')? xwait : xnowait;
        t = gettoken();
        if(t==')' || t=='\n') /* tests ~first(pipeline) */
            break;
        np1 = alloc(0);
        np1->args[0] = np;
        if((np1->args[1]=pipeline()) == 0)
            return nil;
        np = np1;
    }
    if(np->op == 0)
        np->op = xwait;
    return np;
}
```

⟨*function* error 216b⟩≡ (250)

```
/* error - print error message s, prefixed by t */
void
error(char *s, char *t)
{
    char buf[256];

    fprint(2, "%s%s", t, s);
    errstr(buf, sizeof buf);
    fprint(2, ": %s\n", buf);
}
```

⟨*function* pipeline 216c⟩≡ (250)

```
/* pipeline - command ( | command )* */
Node*
pipeline(void)
{
    Node *np, *np1;

    if((np=command()) == 0)
        return nil;
    while(t == '|'){
        np1 = alloc(xpipeline);
        np1->args[0] = np;
        t = gettoken();
        if((np1->args[1]=command()) == 0)
            return nil;
        np = np1;
    }
    return np;
}
```

⟨*function* redirect 216d⟩≡ (250)

```
/* redirect - redirect i/o according to np->io[] values */
```

```
void
redirect(Node *np)
{
    int fd;

    if(np->io[0]){
        if((fd = open(np->io[0], 0)) < 0){
            error(": can't open", np->io[0]);
            exits("open");
        }
        dup(fd, 0);
        close(fd);
    }
    if(np->io[1]){
        if((fd = create(np->io[1], 1, 0666L)) < 0){
            error(": can't create", np->io[1]);
            exits("create");
        }
        dup(fd, 1);
        close(fd);
    }
    if(np->io[2]){
        if((fd = open(np->io[2], 1)) < 0 && (fd = create(np->io[2], 1, 0666L)) < 0){
            error(": can't write", np->io[2]);
            exits("write");
        }
        dup(fd, 1);
        close(fd);
        seek(1, 0, 2);
    }
}
```

⟨*function* setio 217a⟩≡ (250)

```
/* setio - ( < | > | >> ) word; fill in np->io[] */
error0
setio(Node *np)
{
    if(t == '<'){
        t = gettoken();
        np->io[0] = token;
    }else if(t == '>'){
        t = gettoken();
        if(t == '>'){
            t = gettoken();
            np->io[2] = token;
            }else
            np->io[1] = token;
    }else
        return ERROR_0;
    if(t != WORD)
        return ERROR_0;
    return OK_1;
}
```

⟨*function* simple 217b⟩≡ (250)

```
/* simple - word ( [ < | > | >> ] word )* */
Node*
simple(void)
{
    Node *np;
```

```
    int n = 1;

    if(t != WORD)
        return nil;
    np = alloc(xsimple);
    np->argv[0] = token;
    while((t = gettoken())==WORD || t=='<' || t=='>')
        if(t == WORD)
            np->argv[n++] = token;
        else if(!setio(np))
            return nil;
    np->argv[n] = 0;
    return np;
}
```

⟨*function* xpipeline 218a⟩≡ (250)

```
/* xpipeline - execute cmd | cmd */
int
xpipeline(Node *np)
{
    int pid, fd[2];

    if(pipe(fd) < 0){
        error("can't create pipe", "");
        return 0;
    }
    if((pid=fork()) == 0){ /* left side; redirect stdout */
        dup(fd[1], 1);
        close(fd[0]);
        close(fd[1]);
        execute(np->args[0]);
        exits(status);
    }else if(pid == -1){
        error("can't create process", "");
        return 0;
    }
    if((pid=fork()) == 0){ /* right side; redirect stdin */
        dup(fd[0], 0);
        close(fd[0]);
        close(fd[1]);
        pid = execute(np->args[1]); /*BUG: this is wrong sometimes*/
        if(pid > 0)
            while(waitpid()!=pid)
                ;
        exits(nil);
    }else if(pid == -1){
        error("can't create process", "");
        return 0;
    }
    close(fd[0]); /* avoid using up fd's */
    close(fd[1]);
    return pid;
}
```

⟨*function* xsimple 218b⟩≡ (250)

```
/* xsimple - execute a simple command */
int
xsimple(Node *np)
{
    char name[MAXLINE];
```

```
    int pid, i;

    if(builtin(np))
        return 0;

    if(pid = fork()){
        if(pid == -1)
            error(": can't create process", np->argv[0]);
        return pid;
    }
    redirect(np); /* child process */
    exec(np->argv[0], &np->argv[0]);
    i = np->argv[0][0];

    if(i!='/' && i!='#' && (i!='.' || np->argv[0][1]!='/'))
        for(i = 0; path[i]; i++){
            sprint(name, "%s/%s", path[i], np->argv[0]);
            exec(name, &np->argv[0]);
        }
    error(": not found", np->argv[0]);
    exits("not found");
    return -1;  // suppress compiler warnings
}
```

⟨*function* xsubshell 219a⟩≡ (250)

```
/* xsubshell - execute (cmd) */
int
xsubshell(Node *np)
{
    int pid;

    if(pid = fork()){
        if(pid == -1)
            error("can't create process", "");
        return pid;
    }
    redirect(np); /* child process */
    execute(np->args[0]);
    exits(status);
    return -1;  // suppress compiler warnings
}
```

⟨*function* xnowait 219b⟩≡ (250)

```
/* xnowait - execute cmd & */
int
xnowait(Node *np)
{
    int pid;

    execute(np->args[0]);
    pid = execute(np->args[1]);
    if(interactive)
        fprint(2, "%d\n", pid);
    return 0;
}
```

⟨*function* xwait 219c⟩≡ (250)

```
/* xwait - execute cmd ; */
int xwait(Node *np)
{
```

```
    int pid;
    Waitmsg *wmsg;

    execute(np->args[0]);
    pid = execute(np->args[1]);
    if(pid > 0){
        while((wmsg = wait()) != nil){
            if(wmsg->pid == pid)
                break;
            free(wmsg);
        }
        if(wmsg == nil)
            error("wait error", "");
        else {
            strncpy(status, wmsg->msg, sizeof(status)-1);
            free(wmsg);
        }
    }
    return 0;
}
```

Appendix G

Extra Code

This appendix collects the code chunks the preceding chapters referred to but did not show inline: header boilerplate, forward declarations, and the small uninteresting helpers whose definitions would only have interrupted the narrative. Nothing here is new—it is the remainder of `rc`'s source, included so that the literate program remains, as it must, the whole program and not just its interesting parts.

G.1 rc/

G.1.1 rc/fns.h

⟨rc/fns.h 221⟩≡

```
// for var.c, words.c, tree.c: see rc.h
// see also io.h and getflags.h

// input.c
int getnext(void);
int advance(void);
int nextc(void);
bool nextis(int c);
void pprompt(void);
void kinit(void);
tree *token(char*, int);
tree *klook(char*);

// lex.c
int yylex(void);
//@Scheck: used in syn.y
void skipnl(void);
int idchr(int);

// syn.y
//@Scheck: defined in syn.y and y.tab.c
int yyparse(void);

// executils.c
void start(code*, int, var*);
void pushlist(void);
void poplist(void);
void pushword(char*);
void popword(void);
void pushredir(int, int, int);
```

```
// status.c
char *getstatus(void);
void setstatus(char*);
char* concstatus(char *s, char *t);
bool truestatus(void);

// path.c
word* searchpath(char*);

// env.c
void Updenv(void);
void Vinit(void);

// processes.c
int Waitfor(int, bool);
void Execute(word*,  word*);

// code.c
error0 compile(tree*);
code *codecopy(code*);
void codefree(code*);
void cleanhere(char*);

// trap.c
void dotrap(void);
void Trapinit(void);
bool Eintr(void);
void Noerror(void);

// here.c
void readhere(void);
tree *heredoc(tree*);

// glob.c
void deglob(void*);
void globlist(void);
bool match(void*, void*, int);

// utils.c
#define new(type) ((type *)emalloc(sizeof(type)))
void *emalloc(long);
void efree(void *);
void Memcpy(void*, void*, long);
int Creat(char*);
int Opendir(char*);
int Readdir(int, void*, int);
void Closedir(int);
void inttoascii(char*, long);

// error.c
void panic(char*, int);
void yyerror(char*);
void Exit(char*, char*);
```

G.1.2 rc/getflags.h

⟨rc/getflags.h 222⟩≡
  ⟨*constant* NFLAG 43b⟩

```
extern char **flag[NFLAG];
extern char *flagset[];

int getflags(int, char*[], char*, int);
void usage(char*);
```

G.1.3 rc/io.h

⟨rc/io.h 223a⟩≡

```
⟨constant EOF (rc/io.h) 50a⟩
⟨constant NBUF 186g⟩

⟨struct Io 186f⟩

extern io *err;

// io.c
io *openfd(fdt);
io *openstr(void);
io *opencore(char *, int);
void closeio(io*);
void flush(io*);

int rchr(io*);
int rutf(io*, char*, Rune*);

// fmt.c
void pchr(io*, int);
void pstr(io*, char*);
void pfmt(io*, char*, ...);

// pcmd.c
void pcmd(io*, tree*);
// pfnc.c
void pfnc(io*, thread*);
```

G.1.4 rc/rc.h

⟨rc/rc.h 223b⟩≡

```
/*
 * Assume plan 9 by default; if Unix is defined, assume unix.
 * Please don't litter the code with ifdefs.  The five below should be enough.
 */
#ifndef Unix
/* plan 9 */
#include <u.h>
#include <libc.h>

// could be in trap.c
⟨constant NSIG 143a⟩
⟨constant SIGINT 144a⟩
⟨constant SIGQUIT 144b⟩

//???
//#define fcntl(fd, op, arg) /* unix compatibility */
//#define F_SETFD
//#define FD_CLOEXEC
```

```
//TODO:
#define __LOC__ "NO__LOC__INFO"

#else
#include "unix.h"

// magic incantation for cpp (found by chatGPT)
#define STR2(x) #x
#define STR(x) STR2(x)
#define __LOC__ __FILE__ ":" STR(__LINE__)

#endif

//#ifndef YYPREFIX
//#ifndef PAREN
//#include "x.tab.h"
//pad: better like that, otherwise get some "redefined FOR macro" error
//#endif
//#endif

//#ifndef Unix
//#pragma incomplete word
//#pragma incomplete io
//#endif


// MiscPlan9 is back in plan9.c

#ifndef ERRMAX
⟨constant ERRMAX 179a⟩
#endif

⟨constant YYMAXDEPTH ??⟩

// forward decls
typedef struct Tree tree;
typedef struct Word word;
typedef struct Io io;
typedef union Code code;
typedef struct Var var;
typedef struct List list;
typedef struct Redir redir;
typedef struct Thread thread;
typedef struct Builtin builtin;

⟨struct Tree 31a⟩

// tree.c
tree *newtree(void);
//@Scheck: useful, for syn.y, and not just for tree.c
tree *tree1(int, tree*);
tree *tree2(int, tree*, tree*);
//@Scheck: useful, for syn.y, and not just for tree.c
tree *tree3(int, tree*, tree*, tree*);
tree *mung1(tree*, tree*);
tree *mung2(tree*, tree*, tree*);
tree *mung3(tree*, tree*, tree*, tree*);
tree *epimung(tree*, tree*);
tree *simplemung(tree*);
void freenodes(void);
```

⟨*struct* Code 33c⟩

⟨*constant* APPEND 61d⟩
⟨*constant* WRITE 61b⟩
⟨*constant* READ 61c⟩
⟨*constant* HERE 157a⟩
⟨*constant* DUPFD 163d⟩
⟨*constant* CLOSE 163a⟩
⟨*constant* RDWR 161b⟩

⟨*struct* Word 36b⟩

```
// words.c
word *newword(char *, word *);
word *copywords(word *, word *);
word* copynwords(word *a, word *tail, int n);
void freelist(word *w);
void freewords(word*);
int count(word*);
```

⟨*struct* Var 38c⟩

```
// var.c
var *vlook(char*);
var *gvlook(char*);
var *newvar(char*, var*);
void setvar(char*, word*);
```

⟨*constant* NVAR 39d⟩
```
extern var *gvar[NVAR];  /* hash for globals */
```

⟨*struct* Here 157c⟩

⟨*constant* GLOB 132⟩

⟨*constant* PRD 97a⟩
⟨*constant* PWR 97b⟩

```
// globals.c
extern int mypid;
extern char *promptstr;
extern int ndot;
// input.c
extern bool doprompt;
extern bool inquote;
extern bool incomm;
extern int lastc;
// lex.c
extern bool lastword;
// error.c
extern int nerror; /* number of errors encountered during compilation */
```

Uses Builtin 115a, Code 33c, Io 186f, List 38a, Redir 90c, Thread 34d, Tree, Var 38c, and Word 36b.

G.1.5 rc/exec.h

⟨rc/exec.h 225⟩≡
```
/*
 * Definitions used in the interpreter
```

```
 */
extern void Xappend(void), Xasync(void), Xbackq(void), Xbang(void), Xclose(void);
extern void Xconc(void), Xcount(void), Xdelfn(void), Xdol(void), Xqdol(void), Xdup(void);
extern void Xexit(void), Xfalse(void), Xfn(void), Xfor(void), Xglob(void);
extern void Xjump(void), Xmark(void), Xmatch(void), Xpipe(void), Xread(void);
extern void Xrdwr(void);
extern void Xrdfn(void), Xreturn(void), Xsubshell(void);
extern void Xtrue(void), Xword(void), Xwrite(void), Xpipefd(void), Xcase(void);
extern void Xlocal(void), Xunlocal(void), Xassign(void), Xsimple(void), Xpopm(void);
extern void Xrdcmds(void), Xwastrue(void), Xif(void), Xifnot(void), Xpipewait(void);
extern void Xdelhere(void), Xpopredir(void), Xsub(void), Xeflag(void), Xsettrue(void);
extern void Xerror(char*);
extern void Xerror1(char*);
```

⟨*struct* `Redir` 90c⟩

⟨*constant* `NSTATUS` 98c⟩
⟨*constant* `ROPEN` 91c⟩
⟨*constant* `RDUP` 164c⟩
⟨*constant* `RCLOSE` 164d⟩

⟨*struct* `List` 38a⟩

⟨*struct* `Thread` 34d⟩

⟨*struct* `Builtin` 115a⟩

```
// globals.c
extern thread *runq;
extern code *codebuf;  /* compiler output */
extern bool eflagok;  /* kludge flag so that -e doesn't exit in startup */
// path.c
extern word nullpath;
// trap.c
extern int ntrap;  /* number of outstanding traps */
// builtins.c
extern struct Builtin builtins[];

// simple.c
void execexec(void);
void execcmds(io *);
// processes.c
int  execforkexec(void);
```

G.1.6 rc/globals.c

⟨`rc/globals.c` 226⟩≡
⟨*includes* 25⟩

```
// was in rc.h
```
⟨*global* `mypid` 130a⟩
⟨*global* `promptstr` 51d⟩
⟨*global* `ndot` 120a⟩

```
// was in exec.h
```
⟨*global* `runq` 35a⟩
⟨*global* `codebuf` 33b⟩
⟨*global* `eflagok` 149b⟩

G.1.7 rc/getflags.c

⟨rc/getflags.c 227a⟩≡
```
#include <u.h>
#include <libc.h>
#include "getflags.h"

extern void Exit(char*, char*);


static void reverse(char**, char**);
static int scanflag(int, char*);
static void errn(char*, int);
static void errs(char*);
static void errc(int);
```

⟨*global* flagset 43c⟩
⟨*global* flag 43a⟩
⟨*global* cmdname 182a⟩
⟨*global* flagarg 182b⟩
⟨*global* reason 182c⟩

⟨*constant* RESET 182d⟩
⟨*constant* FEWARGS 182e⟩
⟨*constant* FLAGSYN 182f⟩
⟨*constant* BADFLAG 182g⟩

⟨*global* badflag 182h⟩

⟨*function* getflags 182i⟩

⟨*function* reverse 183a⟩

⟨*function* scanflag 183b⟩

⟨*function* usage 184⟩

⟨*function* errn 185⟩

⟨*function* errs 186a⟩
⟨*constant NBUF (rc/getflags.c)* 186b⟩
⟨*global* buf 186c⟩
⟨*global* bufp 186d⟩

⟨*function* errc 186e⟩

G.1.8 rc/io.c

⟨rc/io.c 227b⟩≡
⟨*includes* 25⟩

⟨*enum* MiscConstants 187a⟩

```
// forward decls
int emptybuf(io*);
int fullbuf(io*, int);
```

⟨*function* rchr 188c⟩

⟨*function* `rutf` 188e⟩

⟨*function* `fullbuf` 188b⟩

⟨*function* `flush` 187b⟩

⟨*function* `openfd` 186h⟩

⟨*function* `openstr` 189b⟩

⟨*function* `opencore` 189c⟩

⟨*function* `closeio` 189a⟩

⟨*function* `emptybuf` 188d⟩

G.1.9 `rc/input.c`

⟨`rc/input.c` 228⟩≡

```
⟨includes 25⟩
#include "y.tab.h"

// was in lex.c

⟨global future 52e⟩
⟨global doprompt 51c⟩
⟨global inquote 58a⟩
⟨global incomm 56e⟩

// was in rc.h
⟨global lastc 180a⟩

// forward decl
int getnext(void);

⟨function nextc 53a⟩
⟨function advance 53b⟩
⟨function getnext 49⟩

⟨function pprompt 52a⟩


⟨function nextis 53c⟩


extern unsigned hash(char *as, int n);

// was in tree.c

⟨function token 59b⟩

// was in var.c

⟨constant NKW 63d⟩
⟨struct Kw 63b⟩
⟨global kw 63c⟩

⟨function kenter 64a⟩
```

⟨*function* `kinit` 63e⟩

⟨*function* `klook` 63a⟩

G.1.10 rc/var.c

⟨`rc/var.c` 229a⟩≡
⟨*includes* 25⟩

```
// was in rc.h
```
⟨*global* `gvar` 39e⟩

⟨*function* `hash` 40a⟩

⟨*function* `gvlook` 39f⟩

⟨*function* `vlook` 39c⟩

⟨*function* `setvar` 39b⟩

⟨*function* `newvar` 40b⟩

G.1.11 rc/env.c

⟨`rc/env.c` 229b⟩≡
⟨*includes* 25⟩

```
// the Vxxx are back in plan9.c
```

G.1.12 rc/glob.c

⟨`rc/glob.c` 229c⟩≡
⟨*includes* 25⟩

```
bool matchfn(void*, void*);

// NDIR and Globsize are back in plan9.c
extern int Globsize(char *p);
```

⟨*global* `globname` 134d⟩
⟨*global* `globv` 134a⟩
⟨*function* `deglob` 137a⟩

⟨*function* `globcmp` 137b⟩

⟨*function* `globsort` 136⟩
⟨*function* `globdir` 135b⟩
⟨*function* `glob` 135a⟩

⟨*function* `equtf` 150a⟩

⟨*function* `nextutf` 150b⟩

⟨*function* `unicode` 150c⟩

⟨*function* `matchfn` 138a⟩

⟨*function* `match` 138b⟩

⟨*function* `globlist1` 134c⟩

⟨*function* `globlist` 134b⟩

G.1.13 `rc/executils.c`

⟨`rc/executils.c` 230a⟩≡
⟨*includes* 25⟩

⟨*global* `argv0` 45c⟩

⟨*function* `start` 45b⟩

⟨*function* `pushword` 46b⟩

⟨*function* `popword` 77d⟩

⟨*function* `pushlist` 46a⟩

⟨*function* `poplist` 77c⟩

⟨*function* `pushredir` 90f⟩

⟨*function* `Xerror` 82a⟩

⟨*function* `Xerror1` 82c⟩

⟨*function* `turfredir` 94d⟩

⟨*function* `Xpopredir` 92c⟩

⟨*function* `Xreturn` 96c⟩

G.1.14 `rc/exec.c`

⟨`rc/exec.c` 230b⟩≡
⟨*includes* 25⟩

```
/*
 * Opcode routines
 * Arguments on stack (...)
 * Arguments in line [...]
 * Code in line with jump around {...}
 *
 * Xappend(file)[fd]   open file to append
 * Xassign(name, val)   assign val to name
 * Xasync{... Xexit}   make thread for {}, no wait
 * Xbackq(split){... Xreturn}   make thread for {}, push stdout
 * Xbang    complement condition
 * Xcase(pat, value){...}  exec code on match, leave (value) on
 *      stack
```

```
 * Xclose[i]    close file descriptor
 * Xconc(left, right)   concatenate, push results
 * Xcount(name)    push var count
 * Xdelfn(name)    delete function definition
 * Xdelhere
 * Xdol(name)    get variable value
 * Xdup[i j]    dup file descriptor
 * Xeflag
 * Xerror
 * Xexit    rc exits with status
 * Xfalse{...}    execute {} if false
 * Xfn(name){... Xreturn}   define function
 * Xfor(var, list){... Xreturn}  for loop
 * Xglob
 * Xif
 * Xifnot
 * Xjump[addr]    goto
 * Xlocal(name, val)   create local variable, assign value
 * Xmark    mark stack
 * Xmatch(pat, str)   match pattern, set status
 * Xpipe[i j]{... Xreturn}{... Xreturn} construct a pipe between 2 new threads,
 *      wait for both
 * Xpipefd[type]{... Xreturn}  connect {} to pipe (input or output,
 *      depending on type), push /dev/fd/??
 * Xpipewait
 * Xpopm(value)    pop value from stack
 * Xpopredir
 * Xrdcmds
 * Xrdfn
 * Xrdwr(file)[fd]   open file for reading and writing
 * Xread(file)[fd]   open file to read
 * Xqdol(name)    concatenate variable components
 * Xreturn    kill thread
 * Xsimple(args)   run command and wait
 * Xsub
 * Xsubshell{... Xexit}   execute {} in a subshell and wait
 * Xtrue{...}    execute {} if true
 * Xunlocal    delete local variable
 * Xwastrue
 * Xword[string]   push string
 * Xwrite(file)[fd]   open file to write
 */
```

⟨*function* `Xappend` 94b⟩

⟨*function* `Xsettrue` 101f⟩

⟨*function* `Xbang` 88d⟩

⟨*function* `Xclose` 164e⟩

⟨*function* `Xdup` 164a⟩

⟨*function* `Xeflag` 149d⟩

⟨*function* `Xexit` 96b⟩

⟨*function* `Xfalse` 88b⟩
⟨*global* `ifnot` 100c⟩

⟨*function* `Xifnot` 100f⟩

⟨*function* `Xjump` 101g⟩

⟨*function* `Xmark` 77b⟩

⟨*function* `Xpopm` 104a⟩

⟨*function* `Xread` 93b⟩

⟨*function* `Xrdwr` 162a⟩

⟨*function* `Xtrue` 88a⟩

⟨*function* `Xif` 100b⟩

⟨*function* `Xwastrue` 100d⟩

⟨*function* `Xword` 77a⟩

⟨*function* `Xwrite` 92b⟩

⟨*function* `list2str` 89b⟩

⟨*function* `Xmatch` 89a⟩

⟨*function* `Xcase` 105a⟩

⟨*function* `conclist` 114⟩

⟨*function* `Xconc` 113b⟩

⟨*function* `Xassign` 109c⟩

⟨*function* `Xdol` 110d⟩

⟨*function* `Xqdol` 156c⟩

⟨*function* `subwords` 112c⟩

⟨*function* `Xsub` 112b⟩

⟨*function* `Xcount` 111b⟩

⟨*function* `Xlocal` 110a⟩

⟨*function* `Xunlocal` 110b⟩

⟨*function* `Xfn` 107a⟩

⟨*function* `Xdelfn` 107b⟩

⟨*function* `Xpipewait` 98d⟩

⟨*function* `Xrdcmds` 47a⟩

⟨*function* `Xdelhere` 159b⟩

⟨*function* `Xfor` 102b⟩

⟨*function* `Xglob` 133d⟩

⟨*global* `envdir` 125⟩

```
// Xrdfn is back in plan9.c
```

G.1.15 rc/processes.c

⟨`rc/processes.c` 233a⟩≡

```
⟨includes 25⟩
#include <str.h>

// Fdprefix, delwaitpid(), havewaitpid() Waitfor() are back in plan9.c
extern char *Fdprefix;
```

⟨*global* `waitpids` 79a⟩
⟨*global* `nwaitpids` 79b⟩
⟨*function* `addwaitpid` 79c⟩
⟨*function* `clearwaitpids` 79f⟩

⟨*function* `mkargv` 85a⟩

⟨*function* `Execute` 84c⟩

```
// was in havefork.c
```

⟨*function* `Xasync` 99b⟩

⟨*function* `Xpipe` 97c⟩

⟨*function* `Xbackq` 151c⟩

⟨*function* `Xpipefd` 154b⟩

⟨*function* `Xsubshell` 155d⟩

⟨*function* `execforkexec` 83a⟩

G.1.16 rc/tree.c

⟨`rc/tree.c` 233b⟩≡

```
⟨includes 25⟩
#include "y.tab.h"
```

⟨*global* `treenodes` 31c⟩
⟨*function* `newtree` 32a⟩

⟨*function* `freenodes` 32b⟩

⟨*function* `tree1` 32c⟩

⟨*function* `tree2` 32d⟩

⟨*function* `tree3` 32e⟩

⟨*function* `mung1` 69e⟩

⟨*function* mung2 69g⟩

⟨*function* mung3 71d⟩

⟨*function* epimung 70e⟩
⟨*function* simplemung 67a⟩

G.1.17 rc/lex.c

⟨rc/lex.c 234a⟩≡
```
⟨includes 25⟩
#include "y.tab.h"

⟨constant NTOK 56d⟩

// was used by subr.c
⟨global lastdol 60b⟩
⟨global lastword 58c⟩
// was in rc.h
⟨global tok 56c⟩

⟨function wordchr 55b⟩
⟨function idchr 56a⟩

⟨function yyerror 179e⟩

⟨function skipwhite 57a⟩
⟨function skipnl 57b⟩

⟨function addtok 133b⟩
⟨function addutf 149e⟩

⟨function yylex 55a⟩
```

G.1.18 rc/trap.c

⟨rc/trap.c 234b⟩≡
```
⟨includes 25⟩

// NSIG, SIGINT, SIGQUIT are back in rc.h (under an ifdef for plan9)

// was in exec.h
⟨global ntrap 144c⟩
⟨global trap 144d⟩

// signame, syssigname, interrputed are back in plan9.c
// notifyf(), Trapinit(), Eintr(), Noerror() are back in plan9.c
extern char *signame[];

// generic part independent of plan9
⟨function dotrap 145b⟩
```

G.1.19 rc/simple.c

⟨rc/simple.c 234c⟩≡
```
/*
 * Maybe 'simple' is a misnomer.
```

```
 */
⟨includes 25⟩

// forward decls
void execfunc(var*);

⟨function exitnext 86b⟩

⟨function Xsimple 81⟩

⟨function doredir 91a⟩

⟨function execexec 83b⟩

⟨function execfunc 108c⟩

⟨global rdcmds 121c⟩

⟨function execcmds 122a⟩
```

G.1.20 rc/pcmd.c

```
⟨rc/pcmd.c 235a⟩≡
  ⟨includes 25⟩
  #include "y.tab.h"

  ⟨global nl 173a⟩
  ⟨constant c0 175a⟩
  ⟨constant c1 175b⟩
  ⟨constant c2 175c⟩

  ⟨function pdeglob 175d⟩

  ⟨function pcmd 173b⟩
```

G.1.21 rc/here.c

```
⟨rc/here.c 235b⟩≡
  ⟨includes 25⟩
  #include "y.tab.h"

  ⟨global here 157d⟩
  ⟨global ehere 157e⟩
  ⟨global ser 159d⟩
  ⟨global tmp (rc/here.c) 159e⟩
  ⟨global hex 159f⟩

  void psubst(io*, uchar*);
  void pstrs(io*, word*);

  ⟨function hexnum 159g⟩

  ⟨function heredoc 159h⟩

  ⟨constant NLINE 158a⟩

  ⟨function readhere 158c⟩
```

⟨*function* `psubst` 160⟩

⟨*function* `pstrs` 161a⟩

G.1.22 `rc/code.c`

⟨`rc/code.c` 236a⟩≡
⟨*includes* 25⟩
```
#include "y.tab.h"
```

⟨*constant c0 (rc/code.c)* 80a⟩
⟨*constant c1 (rc/code.c)* 80b⟩
⟨*constant c2 (rc/code.c)* 80c⟩
⟨*global* `codep` 75a⟩
⟨*global* `ncode` 75b⟩
⟨*function* `emitf` 75d⟩
⟨*function* `emiti` 75c⟩
⟨*function* `emits` 75e⟩

```
// forward decls
void outcode(tree*, bool);
void codeswitch(tree*, bool);
bool iscase(tree*);
```

⟨*function* `morecode` 75f⟩

⟨*function* `stuffdot` 87d⟩

⟨*function* `compile` 75g⟩

⟨*function* `cleanhere` 159a⟩

⟨*function* `fnstr` 106c⟩

⟨*function* `outcode` 76b⟩
⟨*function* `codeswitch` 103b⟩

⟨*function* `iscase` 104b⟩

⟨*function* `codecopy` 34a⟩

⟨*function* `codefree` 34b⟩

G.1.23 `rc/pfnc.c`

⟨`rc/pfnc.c` 236b⟩≡
⟨*includes* 25⟩

⟨*global* `fname` 176b⟩

⟨*function* `pfnc` 177a⟩

G.1.24 `rc/utils.c`

⟨`rc/utils.c` 236c⟩≡
⟨*includes* 25⟩

```
// Opendir, Closedir and so on are back in plan9.c
```

⟨*function* `Unlink` 159c⟩
⟨*function* `Write` 193d⟩
⟨*function* `Read` 193e⟩
⟨*function* `Seek` 193f⟩

⟨*function* `Creat` 193g⟩

⟨*function* `Dup` 194a⟩

⟨*function* `Memcpy` 194b⟩

```
// back in plan9.c
extern void* Malloc(ulong n);
```
⟨*function* `emalloc` 181a⟩

⟨*function* `efree` 181c⟩

⟨*global* `bp` 193a⟩

⟨*function* `iacvt` 193c⟩

⟨*function* `inttoascii` 193b⟩

G.1.25 `rc/status.c`

⟨`rc/status.c` 237a⟩≡
⟨*includes* 25⟩

⟨*function* `setstatus` 78a⟩

⟨*function* `getstatus` 78b⟩

⟨*function* `truestatus` 78c⟩

⟨*function* `concstatus` 98f⟩

G.1.26 `rc/builtins.c`

⟨`rc/builtins.c` 237b⟩≡
⟨*includes* 25⟩

⟨*function* `dochdir` 117b⟩

⟨*function* `appfile` 117a⟩

⟨*function* `mapfd` 166a⟩

⟨*function* `execcd` 116c⟩

⟨*function* `execexit` 118c⟩

⟨*function* `execeval` 121b⟩

⟨*function* `execflag` 168⟩

```
// was in plan9.c
```

⟨*function* `Executable` 166b⟩

⟨*function* `execwhatis` 165⟩

⟨*function* `execshift` 169⟩

⟨*global* `dotcmds` 120b⟩

⟨*function* `execdot` 119⟩

⟨*function* `execwait` 122b⟩

⟨*function* `execnewpgrp` 167⟩

⟨*global* `rdfns` 126a⟩

```
extern fdt envdir;
```

⟨*function* `execfinit` 126b⟩

⟨*global* `builtins` 115b⟩

G.1.27 `rc/path.c`

⟨`rc/path.c` 238a⟩≡
⟨*includes* 25⟩

⟨*global* `nullpath` 84b⟩

⟨*function* `searchpath` 84a⟩

G.1.28 `rc/fmt.c`

⟨`rc/fmt.c` 238b⟩≡
⟨*includes* 25⟩

⟨*global* `pfmtnest` 190a⟩

```
// forward decls
void pdec(io*, int);
void poct(io*, unsigned int);
void pptr(io*, void*);
void pval(io*, word*);
void pquo(io*, char*);
void pwrd(io*, char*);

// in io.c
```

```
int fullbuf(io*, int);
```

⟨*function* `pfmt` 190b⟩

⟨*function* `pchr` 188a⟩

⟨*function* `pquo` 192b⟩

⟨*function* `pwrd` 192c⟩

⟨*function* `pptr` 192d⟩

⟨*function* `pstr` 191a⟩

⟨*function* `pdec` 191b⟩

⟨*function* `poct` 192a⟩

⟨*function* `pval` 192e⟩

```
// _efgfmt is back in plan9.c
```

G.1.29 `rc/words.c`

⟨`rc/words.c` 239a⟩≡
⟨*includes* 25⟩

⟨*function* `newword` 36c⟩

⟨*function* `freewords` 37b⟩

⟨*function* `count` 37a⟩

⟨*function* `copynwords` 37c⟩

⟨*function* `copywords` 37d⟩

⟨*function* `freelist` 38b⟩

G.1.30 `rc/error.c`

⟨`rc/error.c` 239b⟩≡
⟨*includes* 25⟩

```
// was in rc.h
```
⟨*global* `nerror` 179c⟩
⟨*global* `err` 179b⟩

```
// forward decls
void Abort(void);
```

⟨*function* `panic` 180c⟩

```
// Exit is back in plan9.c
```

⟨*function* `Abort` 180d⟩

G.1.31 rc/main.c

⟨rc/main.c 240a⟩≡
```
⟨includes 25⟩

// Rcmain and Isatty are back in plan9.c
extern char* Rcmain;
extern bool Isatty(fdt fd);

⟨function main 42b⟩
```

G.1.32 rc/plan9.c

⟨rc/plan9.c 240b⟩≡
```
/*
 * Plan 9 versions of system-specific functions
 * By convention, exported routines herein have names beginning with an
 * upper case letter.
 */
⟨includes 25⟩

// could be in main.c
⟨global Rcmain 129b⟩
⟨function Isatty(plan9.c) 43g⟩

// could be in rc.h
⟨enum MiscPlan9 124a⟩

// defined in trap.c
extern int trap[NSIG];
// defined in exec.c
extern fdt envdir;

// could be in env.c
⟨function Vinit 124d⟩
⟨function addenv(plan9.c) 124c⟩
⟨function updenvlocal 123b⟩
⟨function Updenv 123a⟩

// could be in glob.c
⟨constant NDIR 139a⟩
⟨function Globsize 139b⟩

// could be in processes.c
⟨global Fdprefix 154a⟩

// in processes.c
extern int *waitpids;
extern int nwaitpids;

⟨function delwaitpid 79e⟩
⟨function havewaitpid 79g⟩
⟨function Waitfor(plan9.c) 85b⟩

// could be in trap.c
⟨global signame(plan9.c) 146f⟩
⟨global syssigname(plan9.c) 143b⟩
⟨global interrupted 146a⟩
⟨function notifyf 144f⟩
```

⟨*function* Trapinit 144e⟩
⟨*function* Eintr 146b⟩
⟨*function* Noerror 146d⟩

```
// could be in fmt.c
```
⟨*function* _efgfmt ??⟩

```
// could be in error.c
```
⟨*function* Exit 180e⟩

```
// could be in exec.c
```
⟨*function* Xrdfn 127⟩

```
// could be in utils.c
```
⟨*function* Malloc 181b⟩

```
// could be in utils.c
```
⟨*constant* NFD 140a⟩
⟨*struct* DirEntryWrapper 140b⟩
⟨*global* dir 140c⟩
⟨*function* trimdirs 140e⟩
⟨*function* Readdir 141a⟩
⟨*function* Opendir 140d⟩
⟨*function* Closedir 141b⟩

G.1.33 rc/unix.c

⟨rc/unix.c 241⟩≡

```
/*
 * Unix versions of system-specific functions
 * By convention, exported routines herein have names beginning with an
 * upper case letter.
 */
// to avoid conflict for wait(), waitpid() signatures
#define NOPLAN9DEFINES
#include "rc.h"
#include "io.h"
#include "exec.h"
#include "getflags.h"

//TODO? use plan9port errstr instead?
#include <errno.h>

// system-specific globals defined here but used in other files
char *Rcmain = "/usr/lib/rcmain";
char *Fdprefix = "/dev/fd/";

//*****************************************************************************
// Environment
//*****************************************************************************

#define SEP '\1'

word*
enval(char *s)
{
  char *t, c;
  word *v;
  for(t = s; *t && *t!=SEP; t++);
```

```
  c=*t;
  *t='\0';
  v = newword(s, c=='\0'?(word *)0:enval(t+1));
  *t = c;
  return v;
}

// set in Vinit()
char **environp;

void
Vinit(void)
{
  // see Unix environ(7) man page
  extern char **environ;
  char *s;
  char **env = environ;
  word* wd;
  environp = env;
  for(; *env; env++){
    for(s=*env; *s && *s!='(' && *s!='='; s++);
    switch(*s){
    case '\0':
      pfmt(err, "environment %q?\n", *env);
      break;
    case '=':
      *s='\0';
      wd = enval(s+1);
      setvar(*env, wd);
      //TODO: should generalize this in main.c so it also applies for Plan9
      if(strcmp(*env, "RCMAIN") == 0) {
        Rcmain = strdup(wd->word);
      }
      *s='=';
      break;
    case '(': /* ignore functions for now */
      break;
    }
  }
}

//TODO: should be set in execfinit
char **envp = nil;

void
Xrdfn(void)
{
  char *s;
  int len;
  for(;*envp;envp++){
    for(s=*envp;*s && *s!='(' && *s!='=';s++);
    switch(*s){
    case '\0':
      pfmt(err, "environment %q?\n", *envp);
      break;
    case '=': /* ignore variables */
      break;
    case '(':  /* Bourne again */
      s=*envp+3;
      envp++;
```

```
      len = strlen(s);
      s[len]='\n';
      execcmds(opencore(s, len+1));
      s[len]='\0';
      return;
    }
  }
  Xreturn();
}
//
//union code rdfns[4];
//
//void
//execfinit(void)
//{
// static int first = 1;
// if(first){
//  rdfns[0].i = 1;
//  rdfns[1].f = Xrdfn;
//  rdfns[2].f = Xjump;
//  rdfns[3].i = 1;
//  first = 0;
// }
// Xpopm();
// envp = environp;
// start(rdfns, 1, runq->local);
//}
//
//int
//cmpenv(const void *aa, const void *ab)
//{
// char **a = aa, **b = ab;
//
// return strcmp(*a, *b);
//}
//
//char **
//mkenv(void)
//{
// char **env, **ep, *p, *q;
// struct var **h, *v;
// struct word *a;
// int nvar = 0, nchr = 0, sep;
//
// /*
//  * Slightly kludgy loops look at locals then globals.
//  * locals no longer exist - geoff
//  */
// for(h = gvar-1; h != &gvar[NVAR]; h++)
// for(v = h >= gvar? *h: runq->local; v ;v = v->next){
//  if((v==vlook(v->name)) && v->val){
//   nvar++;
//   nchr+=strlen(v->name)+1;
//   for(a = v->val;a;a = a->next)
//    nchr+=strlen(a->word)+1;
//  }
//  if(v->fn){
//   nvar++;
//   nchr+=strlen(v->name)+strlen(v->fn[v->pc-1].s)+8;
//  }
```

```
// }
// env = (char **)emalloc((nvar+1)*sizeof(char *)+nchr);
// ep = env;
// p = (char *)&env[nvar+1];
// for(h = gvar-1; h != &gvar[NVAR]; h++)
// for(v = h >= gvar? *h: runq->local;v;v = v->next){
//  if((v==vlook(v->name)) && v->val){
//   *ep++=p;
//   q = v->name;
//   while(*q) *p++=*q++;
//   sep='=';
//   for(a = v->val;a;a = a->next){
//    *p++=sep;
//    sep = SEP;
//    q = a->word;
//    while(*q) *p++=*q++;
//   }
//   *p++='\0';
//  }
//  if(v->fn){
//   *ep++=p;
//   *p++='#'; *p++='('; *p++=')'; /* to fool Bourne */
//   *p++='f'; *p++='n'; *p++=' ';
//   q = v->name;
//   while(*q) *p++=*q++;
//   *p++=' ';
//   q = v->fn[v->pc-1].s;
//   while(*q) *p++=*q++;
//   *p++='\0';
//  }
// }
// *ep = 0;
// qsort((void *)env, nvar, sizeof ep[0], cmpenv);
// return env;
//}

void
Updenv(void)
{
}

//******************************************************************************
// Signals/notes and Waitfor()
//******************************************************************************

char *sigmsg[] = {
/*  0 normal  */ 0,
/*  1 SIGHUP  */ "Hangup",
/*  2 SIGINT  */ 0,
/*  3 SIGQUIT */ "Quit",
/*  4 SIGILL  */ "Illegal instruction",
/*  5 SIGTRAP */ "Trace/BPT trap",
/*  6 SIGIOT  */ "abort",
/*  7 SIGEMT  */ "EMT trap",
/*  8 SIGFPE  */ "Floating exception",
/*  9 SIGKILL */ "Killed",
/* 10 SIGBUS  */ "Bus error",
/* 11 SIGSEGV */ "Memory fault",
/* 12 SIGSYS  */ "Bad system call",
/* 13 SIGPIPE */ 0,
```

```
/* 14 SIGALRM */ "Alarm call",
/* 15 SIGTERM */ "Terminated",
/* 16 unused  */ "signal 16",
/* 17 SIGSTOP */ "Process stopped",
/* 18 unused  */ "signal 18",
/* 19 SIGCONT */ "Process continued",
/* 20 SIGCHLD */ "Child death",
};

//pad: Note that plan9 Waitfor(), that was in processes.c (now in plan9.c), was
// compiling correctly also under Unix and I originally used it and rc
// was partially working. However, when called from mk (via MKSHELL), I had
// weird bugs like the $status was containing weird strings and so
// failing mk even if the command was run correctly.
// Anyway, simpler to uncomment and use the Waitfor() below.
void
Waitfor(int pid, bool persist)
{
 int wpid, sig;
 thread *p;
 int wstat;
 char wstatstr[12];

 for(;;){
  errno = 0;
  wpid = wait(&wstat);
  if(errno==EINTR && persist)
   continue;
  if(wpid==-1)
   break;
  sig = wstat&0177;
  if(sig==0177){
   pfmt(err, "trace: ");
   sig = (wstat>>8)&0177;
  }
  if(sig>(sizeof sigmsg/sizeof sigmsg[0]) || sigmsg[sig]){
   if(pid!=wpid)
    pfmt(err, "%d: ", wpid);
   if(sig<=(sizeof sigmsg/sizeof sigmsg[0]))
    pfmt(err, "%s", sigmsg[sig]);
   else if(sig==0177) pfmt(err, "stopped by ptrace");
   else pfmt(err, "signal %d", sig);
   if(wstat&0200)pfmt(err, " -- core dumped");
   pfmt(err, "\n");
  }
  wstat = sig?sig+1000:(wstat>>8)&0xFF;
  if(wpid==pid){
   inttoascii(wstatstr, wstat);
   setstatus(wstatstr);
   break;
  }
  else{
   for(p = runq->ret;p;p = p->ret)
    if(p->pid==wpid){
     p->pid=-1;
     inttoascii(p->status, wstat);
     break;
    }
  }
 }
```

```
}

char *signame[] = {
  "sigexit", "sighup", "sigint", "sigquit",
  "sigill", "sigtrap", "sigiot", "sigemt",
  "sigfpe", "sigkill", "sigbus", "sigsegv",
  "sigsys", "sigpipe", "sigalrm", "sigterm",
  "sig16", "sigstop", "sigtstp", "sigcont",
  "sigchld", "sigttin", "sigttou", "sigtint",
  "sigxcpu", "sigxfsz", "sig26", "sig27",
  "sig28", "sig29", "sig30", "sig31",
  0,
};

// defined in trap.c
extern int trap[NSIG];

void
gettrap(int sig)
{
  signal(sig, gettrap);
  trap[sig]++;
  ntrap++;
  if(ntrap>=NSIG){
    pfmt(err, "rc: Too many traps (trap %d), dumping core\n", sig);
    signal(SIGABRT, (void (*)())0);
    kill(getpid(), SIGABRT);
  }
}

void
Trapinit(void)
{
  int i;
  void (*sig)();

  if(1 || flag['d']){ /* wrong!!! */
    sig = signal(SIGINT, gettrap);
    if(sig==SIG_IGN)
      signal(SIGINT, SIG_IGN);
  }
  else{
    for(i = 1;i<=NSIG;i++) if(i!=SIGCHLD){
 sig = signal(i, gettrap);
 if(sig==SIG_IGN)
   signal(i, SIG_IGN);
      }
  }
}

//*****************************************************************************
// Errno
//*****************************************************************************

bool
Eintr(void){
  return errno==EINTR;
}

void
```

```
Noerror(void)
{
  errno = 0;
}

//*****************************************************************************
// Directories
//*****************************************************************************

#define NDIR 14  /* should get this from param.h */
int
Globsize(char *p)
{
  int isglob = 0, globlen = NDIR+1;
  for(;*p;p++){
    if(*p==GLOB){
      p++;
      if(*p!=GLOB)
 isglob++;
      globlen+=*p=='*'?NDIR:1;
    }
    else
      globlen++;
  }
  return isglob?globlen:0;
}

#include <sys/types.h>
#include <dirent.h>

#define NDIRLIST 50

DIR *dirlist[NDIRLIST];

Opendir(name)
     char *name;
{
  DIR **dp;
  for(dp = dirlist;dp!=&dirlist[NDIRLIST];dp++)
    if(*dp==0){
      *dp = opendir(name);
      return *dp?dp-dirlist:-1;
    }
  return -1;
}

int
Readdir(int f, char *p, int onlydirs)
{
  struct dirent *dp = readdir(dirlist[f]);

  if(dp==0)
    return 0;
  strcpy(p, dp->d_name);
  return 1;
}

void
Closedir(int f)
{
```

```
  closedir(dirlist[f]);
  dirlist[f] = 0;
}

//*****************************************************************************
// Misc
//*****************************************************************************

//char **
//mkargv(a)
//register struct word *a;
//{
// char **argv = (char **)emalloc((count(a)+2)*sizeof(char *));
// char **argp = argv+1; /* leave one at front for runcoms */
//
// for(;a;a = a->next)
//  *argp++=a->word;
// *argp = 0;
// return argv;
//}
//

//void
//Execute(struct word *args, struct word *path)
//{
// char *msg="not found";
// int txtbusy = 0;
// char **env = mkenv();
// char **argv = mkargv(args);
// char file[512];
//
// for(;path;path = path->next){
//  strcpy(file, path->word);
//  if(file[0])
//   strcat(file, "/");
//  strcat(file, argv[1]);
//ReExec:
//  execve(file, argv+1, env);
//  switch(errno){
//  case ENOEXEC:
//   pfmt(err, "%s: Bourne again\n", argv[1]);
//   argv[0]="sh";
//   argv[1] = strdup(file);
//   execve("/bin/sh", argv, env);
//   goto Bad;
//  case ETXTBSY:
//   if(++txtbusy!=5){
//    sleep(txtbusy);
//    goto ReExec;
//   }
//   msg="text busy"; goto Bad;
//  case EACCES:
//   msg="no access";
//   break;
//  case ENOMEM:
//   msg="not enough memory"; goto Bad;
//  case E2BIG:
//   msg="too big"; goto Bad;
//  }
// }
```

```
//Bad:
// pfmt(err, "%s: %s\n", argv[1], msg);
// efree((char *)env);
// efree((char *)argv);
//}


/*
 * Wrong:  should go through components of a|b|c and return the maximum.
 */
void
Exit(char *stat, char* loc)
{
  int n = 0;
  USED(loc);
  //if(flag['s'])
  //  fprint(STDERR, "Exit from %s: %s\n", loc, stat);

  while(*stat){
    if(*stat!='|'){
      if(*stat<'0' || '9'<*stat)
 exit(1);
      else n = n*10+*stat-'0';
    }
    stat++;
  }
  exit(n);
}

bool Isatty(fdt fd){
  return isatty(fd);
}

//void
//execumask(void)  /* wrong -- should fork before writing */
//{
// int m;
// struct io out[1];
// switch(count(runq->argv->words)){
// default:
//  pfmt(err, "Usage: umask [umask]\n");
//  setstatus("umask usage");
//  poplist();
//  return;
// case 2:
//  umask(octal(runq->argv->words->next->word));
//  break;
// case 1:
//  umask(m = umask(0));
//  out->fd = mapfd(1);
//  out->bufp = out->buf;
//  out->ebuf=&out->buf[NBUF];
//  out->strp = 0;
//  pfmt(out, "%o\n", m);
//  break;
// }
// setstatus("");
// poplist();
//}
```

```
void*
Malloc(unsigned long n)
{
  return (void *)malloc(n);
}

int
needsrcquote(int c)
{
  if(c <= ' ')
    return 1;
  if(strchr("‘^#*[]=|\\?${}()’<>&;", c))
    return 1;
  return 0;
}

//void
//delwaitpid(int pid)
//{
// int r, w;
//
// for(r=w=0; r<nwaitpids; r++)
//  if(waitpids[r] != pid)
//   waitpids[w++] = waitpids[r];
// nwaitpids = w;
//}

//int
//havewaitpid(int pid)
//{
// int i;
//
// for(i=0; i<nwaitpids; i++)
//  if(waitpids[i] == pid)
//   return 1;
// return 0;
//
//}
```

Uses GLOB 132, NDIR-14 241, NDIRLIST-15 241, dirlist 241, envp 241, err 179b, flag 43a, gettrap() 241, inttoascii() 193b, ntrap 144c, pfmt() 190b, runq 35a, setstatus() 78a, sigmsg 241, and trap 144d.

G.2 sh/

G.2.1 sh/sh.c

⟨sh/sh.c 250⟩≡

```
/* sh - simple shell - great for early stages of porting */
#include <u.h>
#include <libc.h>

⟨constant MAXLINE 211a⟩
⟨constant WORD 211b⟩
⟨constant EOF 211c⟩
⟨function ispunct 211d⟩
⟨function isspace 211e⟩
⟨function execute 211f⟩
```

```
typedef struct Node Node;
```
⟨*struct* Node 211g⟩

⟨*global* nodes 211h⟩
⟨*global* nfree 211i⟩
⟨*global* strspace 211j⟩
⟨*global* sfree 211k⟩
⟨*global* t 211l⟩
⟨*global* token 212a⟩
⟨*global* putback 212b⟩
⟨*global* status 212c⟩
⟨*global* cflag 212d⟩
⟨*global* tflag 212e⟩
⟨*global* interactive 212f⟩
⟨*global* cflagp 212g⟩
⟨*global* path 212h⟩
⟨*global* ignored 212i⟩

```
// lexer
int gettoken(void);
int getch(void);
// parser (recursive descent)
Node *list(void);
Node *command(void);
Node *simple(void);
Node *pipeline(void);
int setio(Node *np);
// interpreter
// see execute macro above
int builtin(Node *np);


void redirect(Node *np);

int xpipeline(Node *np);
int xsimple(Node *np);
int xsubshell(Node *np);
int xnowait(Node *np);
int xwait(Node *np);

// error management
void error(char *s, char *t);
// memory management
Node *alloc(int (*op)(Node *));
```

⟨*function* main*(sh)* 212j⟩

⟨*function* alloc 213a⟩

⟨*function* builtin 213b⟩

⟨*function* command 214⟩

⟨*function* getch 215a⟩

⟨*function* gettoken 215b⟩

⟨*function* list 216a⟩

⟨*function* `error` 216b⟩

⟨*function* `pipeline` 216c⟩

⟨*function* `redirect` 216d⟩

⟨*function* `setio` 217a⟩

⟨*function* `simple` 217b⟩

⟨*function* `xpipeline` 218a⟩

⟨*function* `xsimple` 218b⟩

⟨*function* `xsubshell` 219a⟩

⟨*function* `xnowait` 219b⟩

⟨*function* `xwait` 219c⟩

G.3 misc/

G.3.1 misc/echo.c

⟨`misc/echo.c` 252a⟩≡

```
#include <u.h>
#include <libc.h>
```

⟨*function* `main`*(echo)* 199⟩

G.3.2 misc/read.c

⟨`misc/read.c` 252b⟩≡

```
#include <u.h>
#include <libc.h>

int multi;
int nlines;
char    *status = nil;

int
line(int fd, char *file)
{
    char c;
    int m, n, nalloc;
    char *buf;

    nalloc = 0;
    buf = nil;
    for(m=0; ; ){
        n = read(fd, &c, 1);
        if(n < 0){
            fprint(2, "read: error reading %s: %r\n", file);
            exits("read error");
        }
        if(n == 0){
            if(m == 0)
```

```
                status = "eof";
            break;
        }
        if(m == nalloc){
            nalloc += 1024;
            buf = realloc(buf, nalloc);
            if(buf == nil){
                fprint(2, "read: malloc error: %r\n");
                exits("malloc");
            }
        }
        buf[m++] = c;
        if(c == '\n')
            break;
    }
    if(m > 0)
        write(1, buf, m);
    free(buf);
    return m;
}

void
lines(int fd, char *file)
{
    do{
        if(line(fd, file) == 0)
            break;
    }while(multi || --nlines>0);
}
```

⟨*function* `main` *(read)* 200⟩

G.3.3 misc/test.c

⟨`misc/test.c` 253⟩≡

```
/*
 * POSIX standard
 * test expression
 * [ expression ]
 *
 * Plan 9 additions:
 * -A file exists and is append-only
 * -L file exists and is exclusive-use
 * -T file exists and is temporary
 */

#include <u.h>
#include <libc.h>
```

⟨*function* `EQ` 201a⟩

⟨*global* `ap` 201b⟩
⟨*global* `ac` 201c⟩
⟨*global* `av` 201d⟩
⟨*global* `tmp` 201e⟩

```
void synbad(char *, char *);
int fsizep(char *);
int isdir(char *);
```

```
int isreg(char *);
int isatty(int);
int isint(char *, int *);
int isolder(char *, char *);
int isolderthan(char *, char *);
int isnewerthan(char *, char *);
int hasmode(char *, ulong);
int tio(char *, int);
int e(void), e1(void), e2(void), e3(void);
char *nxtarg(int);
```

⟨*function* `main`*(test)* 201f⟩

⟨*function* `nxtarg` 201g⟩

⟨*function* `nxtintarg` 202a⟩

⟨*function* `e` 202b⟩

⟨*function* `e1` 202c⟩
⟨*function* `e2` 202d⟩
⟨*function* `e3` 202e⟩

⟨*function* `tio` 204a⟩

⟨*function* `hasmode` 204b⟩

⟨*function* `isdir` 205a⟩
⟨*function* `isreg` 205b⟩

⟨*function* `isatty` 205c⟩

⟨*function* `fsizep` 205d⟩

⟨*function* `synbad` 206a⟩

⟨*function* `isint` 206b⟩

⟨*function* `isolder` 206c⟩
⟨*function* `isolderthan` 207a⟩
⟨*function* `isnewerthan` 207b⟩

G.4 aux/

G.4.1 aux/getflags.c

⟨`aux/getflags.c` 254⟩≡
```
#include <u.h>
#include <libc.h>
#include <ctype.h>

void
usage(void)
{
    print("status=usage\n");
    exits(0);
}
```

```
char*
skipspace(char *p)
{
    while(isspace(*p))
        p++;
    return p;
}

char*
nextarg(char *p)
{
    char *s;

    s = strchr(p, ',');
    if(s == nil)
        return p+strlen(p); /* to \0 */
    while(*s == ',' || isspace(*s))
        s++;
    return s;
}

char*
findarg(char *flags, Rune r)
{
    char *p;
    Rune rr;

    for(p=skipspace(flags); *p; p=nextarg(p)){
        chartorune(&rr, p);
        if(rr == r)
            return p;
    }
    return nil;
}

char*
argname(char *p)
{
    Rune r;
    int n;

    while(1){
        n = chartorune(&r, p);
        if(!isalpharune(r) && !isdigitrune(r))
            break;
        p += n;
    }
    return p;
}

int
countargs(char *p)
{
    int n;

    n = 1;
    for(p=skipspace(p); *p && *p != ','; p++)
        if(isspace(*p) && !isspace(*(p-1)))
            n++;
    return n;
```

```
}
```

⟨*function* `main`*(getflags)* 208⟩

G.4.2 aux/usage.c

⟨`aux/usage.c` 256⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
```

⟨*function* `main`*(usage)* 209⟩

Glossary

LOC = Lines Of Code
CLI = Command-Line Interface
GUI = Graphical User Interface
IDE = Integrated Development Environment
REPL = Read-Eval-Print Loop
AST = Abstract Syntax Tree
PID = Process IDentifier
EOF = End Of File
ASCII = American Standard Code for Information Interchange
UTF = Unicode Transformation Format

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[BK89] Morris Bolsky and David Korn. *The KornShell Command and Programming Language*. Prentice Hall, 1989. cited page(s) 10

[BKM86] Jon Bentley, Donald Knuth, and Doug McIlroy. Programming pearls: A literate program. *Communication of the ACM*, 29(6):471–483, June 1986. The end of the article contains a one-line shell script solving a problem that required Donald Knuth a hundred lines of Pascal code. cited page(s) 8

[BLM92] Doug Brown, John Levine, and Tony Mason. *Lex and Yacc*. O'Reilly, 1992. cited page(s) 12

[Bou79] Stephen R. Bourne. An introduction to the unix shell. In *Unix Programmer's Manual Vol 2a*, 1979. Also available at `shell/docs/sh.pdf`. cited page(s) 9, 12

[Bou15] Stephen R. Bourne. Early days of unix and design of sh. In *BSDCan - The BSD Conference*, 2015. `https://www.bsdcan.org/2015/schedule/attachments/306_srbBSDCan2015.pdf`, also in `shells/docs/early-days-unix-sh-bourne.pdf`. cited page(s) 12

[Duf90] Tom Duff. Rc – a shell for plan 9 and unix. In *Unix Research System: Papers (Vol 2) 10th ed.* Saunders College, 1990. cited page(s) 9, 19

[Duf00] Tom Duff. Rc – the plan 9 shell. Technical report, Bell Labs, 2000. Also available at `shells/docs/rc.pdf`. cited page(s) 12

[Joh79] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual Vol 2b*, 1979. Also available at `generators/docs/yacc.pdf`. cited page(s) 9

[Joy86] William Joy. An introduction to the C shell. In *UNIX Programmer's Supplementary Documents 1*, 1986. Available at `https://docs.freebsd.org/44doc/usd/04.csh/paper.pdf`. cited page(s) 10

[Kid05] Oliver Kiddle. *From Bash to Z Shell*. Apress, 2005. cited page(s) 10, 12

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 13

[Kor94] David G. Korn. ksh - an extensible high level language. In *Proceedings of the USENIX 1994 Very High Level Languages Symposium*, 1994. cited page(s) 10

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984. cited page(s) 12

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 12

[Mic08] Randal K. Michael. *Master UNIX Shell Scripting*. Wiley, 2008. cited page(s) 12

[New95] Cameron Newham. *Learning the bash Shell*. O'Reilly, 1995. cited page(s) 9, 12

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 13

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 10, 12, 16, 18, 21, 25, 78, 123, 142, 167, 170

[Pad15] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 87

[Pad16a] Yoann Padioleau. *Principia Softwarica: Lex and Yacc*. 2016. cited page(s) 30, 65

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Build System mk*. 2016. cited page(s) 148, 171

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 12, 25, 171

[Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 20

[Pad16e] Yoann Padioleau. *Principia Softwarica: The Plan 9 Windowing System rio*. 2016. cited page(s) 9, 14, 118, 142, 148

[Pad25] Yoann Padioleau. *Principia Softwarica: The Plan 9 Utilities*. 2025. cited page(s) 171, 199

[Pou00] Louis Pouzin. The origin of the shell, 2000. http://multicians.org/shell.html. cited page(s) 14, 17

[Ram94] Chet Ramey. Bash, the bourne-again shell, 1994. Available in bash/doc/rose94.pdf. cited page(s) 9

[Roc04] Marc J. Rochkind. *Advanced UNIX Programming*. Addison-Wesley, 2004. cited page(s) 8, 12

[Ste94] Richard W. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1994. cited page(s) 12