Principia Softwarica: The Plan 9 Shell `rc`

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Tom Duff

April 28, 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). 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 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 (e.g., programming language design, compilation, interpretation, system programming, etc.) 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?
- 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-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 6700 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`, partly because `rc` is inspired by the syntax of C with its curly braces (`sh`, instead, is using multiple keywords inspired by Algol).

`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 [Pad16c]).

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 first 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., `begin/end`, `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. 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://minnie.tuhs.org/cgi-bin/utree.pl?file=V1/sh.s
[4] http://minnie.tuhs.org/cgi-bin/utree.pl?file=V6/usr/source/s2/sh.c
[5] Also known as globbing.
[6] http://minnie.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, here-documents, 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).

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 (6700 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). You can also run `rc` on Linux, macOS, or Windows through plan9port[13] or Goken9cc[14], where it is compiled natively using gcc or clang. 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
6   % rc -help
7   Usage: rc [-SsrdiIlxepvV] [-c arg] [-m command] [file [arg...]]
8   % rc
9   % prompt='$ '
10  $ ls /
11  bin
```

[10] https://tiswww.case.edu/php/chet/readline/rltop.html
[11] http://www.zsh.org/
[12] See https://github.com/robbyrussell/oh-my-zsh for a large repository of such contributions.
[13] https://9fans.github.io/plan9port/
[14] https://github.com/aryx/goken9cc

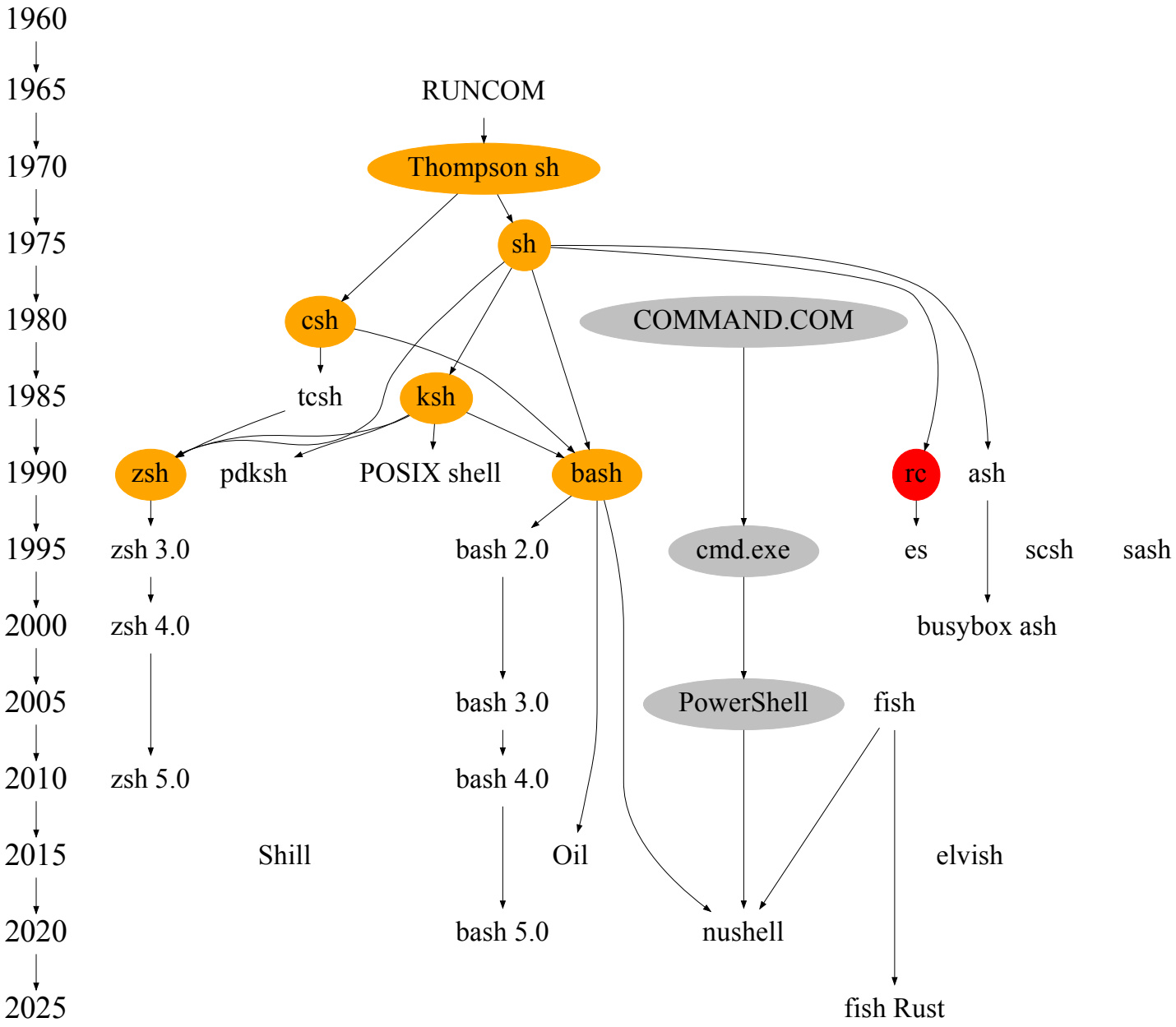


Figure 1.1: Shells timeline

```
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 8.7.3 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.

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 a pipe, a redirection, 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 sytem 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 [Pad16b]).

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`.

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 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

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* (historically a physical device like a DEC VT100, today a *terminal emulator* like iTerm2, Windows Terminal, or Alacritty) handles character I/O: it displays characters on a screen, reads keystrokes, and sends them to whatever program is attached. 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 and writes to file descriptor 1.

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 *login* 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`). On a modern laptop, 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 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.

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`.

The tension between being an interactive command interpreter and a full programming language has shaped many design choices in shells. Ken Thompson's original V6 shell was purely interactive—it could not be scripted. The Bourne shell added scripting, but the need to keep the minimalist command-line syntax (no quotes, no types, newlines as terminators) forced compromises: variables need the `$` prefix to distinguish them from bare-word arguments, newlines must sometimes be "skipped" after binary operators so that multi-line statements work, and values are always strings.

In `rc`, the basis is even cleaner: 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`.

There are no booleans: control flow is based on the exit status of the command you run. The only comparison operator is `~` (pattern matching on strings), and for numerical tests, the external program `test` is used.

A shell is also a kind of universal read-eval-print loop (REPL): you call programs (like functions), pass arguments (like parameters), use environment variables (like globals), and get results back (via exit status and stdout). The core loop is simply: read a line, parse it, fork, exec, wait, repeat.

2.1.3 From scripting to programming language

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.

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. `rc` sits at the minimalist end of this spectrum: it is a shell and nothing more, deliberately leaving the "real programming" to C and the composition to pipes.

2.1.4 An interactive interpreter

The previous subsections covered 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, themed prompts, syntax highlighting—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 50 thousand lines, `zsh` over a hundred thousand, 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, no syntax highlighting. Reading a command is just a `read()` from file descriptor 0 (Section 4.8 shows the exact code). If you want line editing, you wrap `rc` in `rlwrap`, or you run it inside an `acme` window where the editor itself provides editing, search, and history through its normal commands. The shell stays around 7 thousand lines; the interactive features stay in the tools that already do them well.

This split reflects the broader Plan 9 attitude that text editors are the right place to edit text, including command lines. In `rio`, every command lives in a window where you can scroll back through previous output and re-execute old commands by selecting them with the mouse. In `acme`, commands and their output share the same buffer as your source code, and "running a command" is just executing a line from the buffer. Neither approach needs the shell to know what a cursor is. The price is that running `rc` bare in a vt100 terminal gives a pretty stark experience— which is fine if you live in `acme`, less fine if you do not.

2.1.5 The terminal renaissance

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 GPU-accelerated with ligatures and inline images (Kitty, Alacritty, Warp, Windows Terminal). TUI frameworks (charm.sh's Bubbletea, Python's Textual and Rich) are creating a new generation of rich terminal applications. AI coding agents like Claude Code live in the terminal and generate shell pipelines as their primary mode of operation. And 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 Essential shell features

The following sections describe the concrete features of a shell, from running commands through pipes and redirections to variables and builtins.

2.2.1 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 equivalent C program:

⟨`lsroot.c` 16⟩≡

```
#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*.

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

[2] You can even sometimes avoid to specify fully all the arguments by using shortcuts, for example, wildcards as explained in Chapter 13.

- *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.

⟨`minishell.c` 17⟩≡

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

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

  for(;;) {
    pwrite(1, "$ ", 2, 0LL);
    n = pread(0, buf, sizeof(buf)-1, 0LL);
    if(n <= 0)
      break;
    buf[n-1] = '\0';  /* strip newline */
    args[0] = buf;
    args[1] = nil;
    if(rfork(RFPROC|RFFDG) == 0) {
      exec(buf, args);
      exits("exec failed");
    }
    await(buf, sizeof(buf));
  }
  exits(0);
}
```

2.2.2 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`.

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

[4] `rc` allows to redirect not just the standard input and output, as explained in Section 14.4.4.

2.2.3 Pipes

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 18⟩≡

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

  for(;;) {
    pwrite(1, "$ ", 2, 0LL);
    n = pread(0, buf, sizeof(buf)-1, 0LL);
    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, 1);
        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], 1);
        close(pfd[1]);
```

```
        runcmd(buf);
      }
      if(rfork(RFPROC|RFFDG) == 0) {
        close(pfd[1]);
        dup(pfd[0], 0);
        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(0);
}
```

Notice how little code it takes: the key insight is that `fork` 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.

2.2.4 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".

Scripts allow to automate easily repetitive tasks. Moreover, because scripts are commands themselves, they can also be combined, leading to more forms of components-oriented programming.

2.2.5 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 (or filename expansion) 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 (typically by linking with `setargv.obj`). Programs that forget to do so simply receive `*.c` as a literal string—which is why porting UNIX tools to Windows often involves debugging "no such file" errors on patterns that worked everywhere else.

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 13 shows how `rc` implements its three-operator subset—the patterns are tagged with an in-band GLOB byte during lexing, then expanded to a `Word` list during execution.

2.2.6 Quoting

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 rule depends on a global variable `IFS` that controls which characters split words. Forgetting to quote a variable that might contain spaces is the source of an enormous class of UNIX shell bugs, and explains the defensive idiom of writing `"$@"` instead of `$@` in scripts.

`rc` eliminates almost all of this. 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. 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`.

2.2.7 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 be addressed by its `pid` and controlled through `/proc/<pid>/` (read state, send notes, debug with `acid`) just as well as through a shell-private job number; `rfork` gives finer control over what is shared with the child than a single `&`; and `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`.

2.3 `rc` command-line interface

I just described the main features of a shell. 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[5]. 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 12.

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.

2.4 `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 while I will 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

/boot/echo booooooooting...

path=(/bin /boot)

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

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

```
# 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
...

exec /boot/rc -m /boot/rcmain -i
```

2.4.1 `#!`

The first line of `boot.rc` is `#!/boot/rc -m /boot/rcmain`. This is a *shebang* line: when the kernel encounters the `#!` marker at the start of a file being `exec()`-ed, it runs the specified interpreter with the file as argument instead of treating the file as a binary.

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.

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 as a script.

2.4.2 Initialization script

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 D.1 and Chapter 12 for details).

2.4.3 Variables

The `boot.rc` script above shows a variable assignment: `path=(/bin /boot)`. In `rc`, variables hold *lists* 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.

2.4.4 `$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.

2.4.5 Comments

Comments in `rc` start with `#` and extend to the end of the line. This is why the `#!` shebang works as described above: the line is simply a comment from `rc`'s perspective.

2.4.6 Quoting and escaping

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 6.5).

2.4.7 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`[6]. 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.

2.4.8 Builtins

The last line of `boot.rc` is `exec /boot/rc -m /boot/rcmain -i`. `exec` 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 `exec` builtin replaces the current shell process with the specified command. Here it replaces the boot shell with an interactive instance of `rc` (the `-i` flag). Without `exec`, the boot shell would fork a child `rc`, wait for it to finish, and then continue—but there is nothing left to do, so the extra process would just waste a process slot until the interactive shell exits. See Chapter 9 for the full list of builtins.

2.4.9 Other features

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

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

2.5 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`[36a] `Var`[38b]	
execution data structures	3	`exec.h`	`Thread`[34b] `List`[38a]	
AST helpers	3	`tree.c`	`newtree()`[31e] `tree1()`[32b] `tree2()`[32c]	
globals	3	`globals.c`	`codebuf`[33b] `runq`[34c]	
list of strings (words)	3	`words.c`	`newword()`[36b] `count()`[36c] `freewords()`[37b]	
shell variables	3	`var.c`	`setvar()`[39a] `vlook()`[39b] `gvar`[39d] `gvlook()`[40a]	
function prototypes	3	`fns.h`		
entry point	4	`main.c`	`main()`X	
execution utilities	4	`executils.c`	`start()`[44a] `pushlist()`[45a] `pushword()`[45b]	
character input and prompt	5	`input.c`	`getnext()`[48] `pprompt()`[50h] `nextc()`[52b]	
lexer	6	`lex.c`	`yylex()`[53a]	
parser	7	`syn.y`	`yyparse()`[62b]	
bytecode generation	8	`code.c`	`emitf()`[71d] `compile()`[71g] `outcode()`[72e]	
process status	8	`status.c`	`setstatus()`[73f] `getstatus()`[74a]	
simple command bytecode	8	`simple.c`	`Xsimple()`[77a] `execexec()`[79a] `doredir()`[86]	
`$path` management	8	`path.c`	`searchpath()`[79c]	
process-related bytecodes	8	`processes.c`	`execforkexec()`[78b] `Xpipe()`[92c] `Xasync()`[94b]	
other bytecodes	8	`exec.c`	`Xtrue()`[83b] `Xjump()`[97b] `Xfn()`[102a]	
shell builtins	9	`builtins.c`	`execcd()`[110c] `execexit()`[112c] `execdot()`[112d]	
shell environment	10	`env.c`	`addenv()`[118b] `Vinit()`[118c]	
signal management	11	`trap.c`	`dotrap()`[123a] `Trapinit()`[122c]	
initialization (bootstrap)	12	`main.c`	`Rcmain`[126b] `dotcmds`[114b]	
wildcard matching	13	`glob.c`	`glob()`[131b] `match()`[134b]	
here documents	14	`here.c`	`readhere()`[142a] `heredoc()`[144a]	
AST dumper	A	`pcmd.c`	`pcmd()`[162e]	
bytecode dumper	A	`pfnc.c`	`pfnc()`[165b]	
error management	B	`error.c`	`panic()`[168c]	
memory management	C	`utils.c`	`emalloc()`[169b] `efree()`[169d]	
command-line flags	C	`getflags.c`	`getflags()`[170i] `usage()`[172]	
buffered IO	C	`io.c`	`openfd()`[175b] `rchr()`[176c]	
pretty printer	C	`fmt.c`	`pfmt()`[178b]	
Total				6800

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 (`Tree`, `Code`, `Word`, `Var`), `exec.h` defines the execution data structures (`Thread`, `List`, `Redir`[85b]), `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⟩≡ (225 224 223 222 221 220 219 218 215 214 213 212b 211b)

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

2.6 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.

2.6.1 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.

2.6.2 `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()`[53a]). Then, `rc` parses those tokens (with `yyparse()`[62b]) 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 [Pad16b]), 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`[34b] data structure and in a queue `runq`[34c]. 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()`[92c], 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()`[46a]) that then triggers the compiler. I will now explain briefly the control flow of `rc`, starting from the top of Figure 2.3.

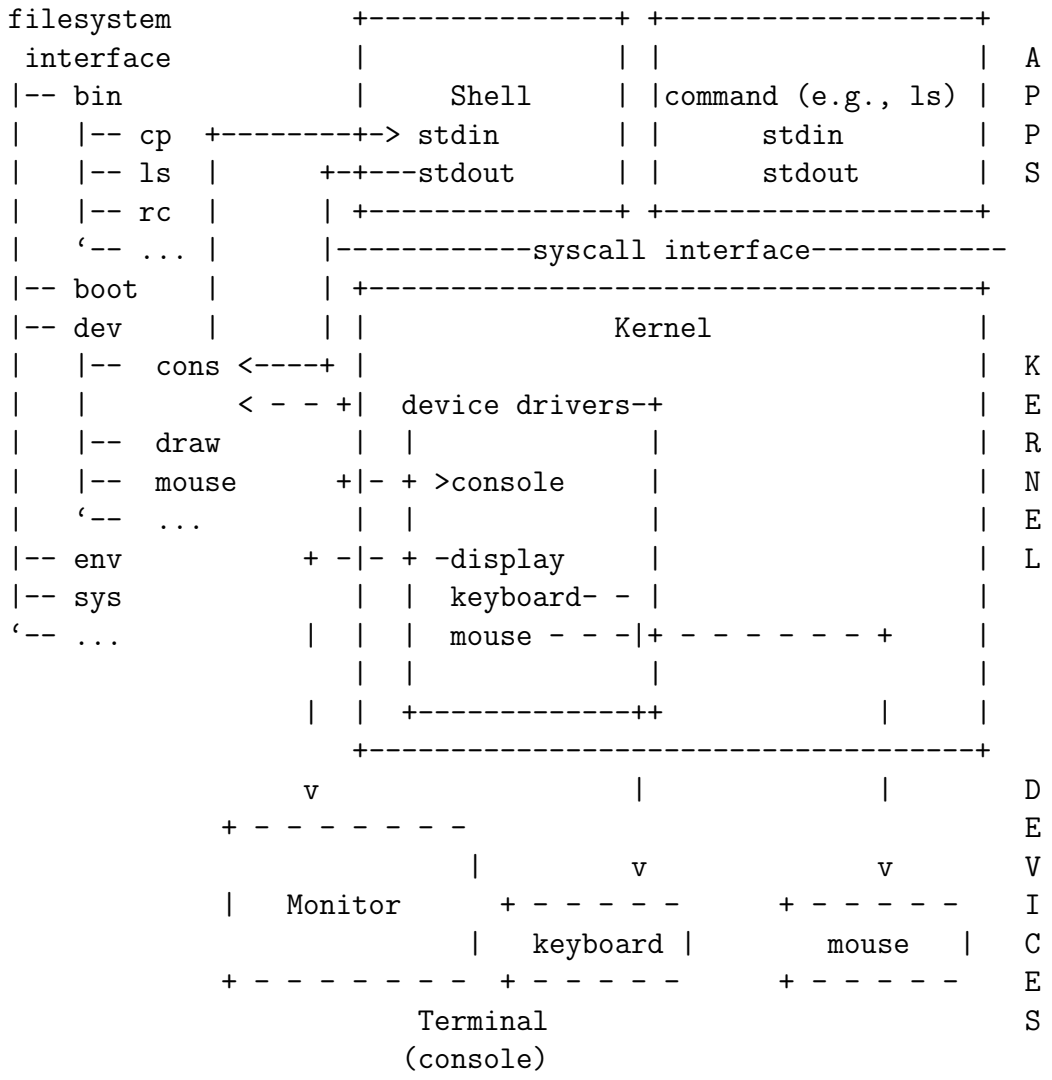


Figure 2.1: Components of the command-line user interface under Plan 9.

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

Figure 2.2: Data flow diagram of `rc`.

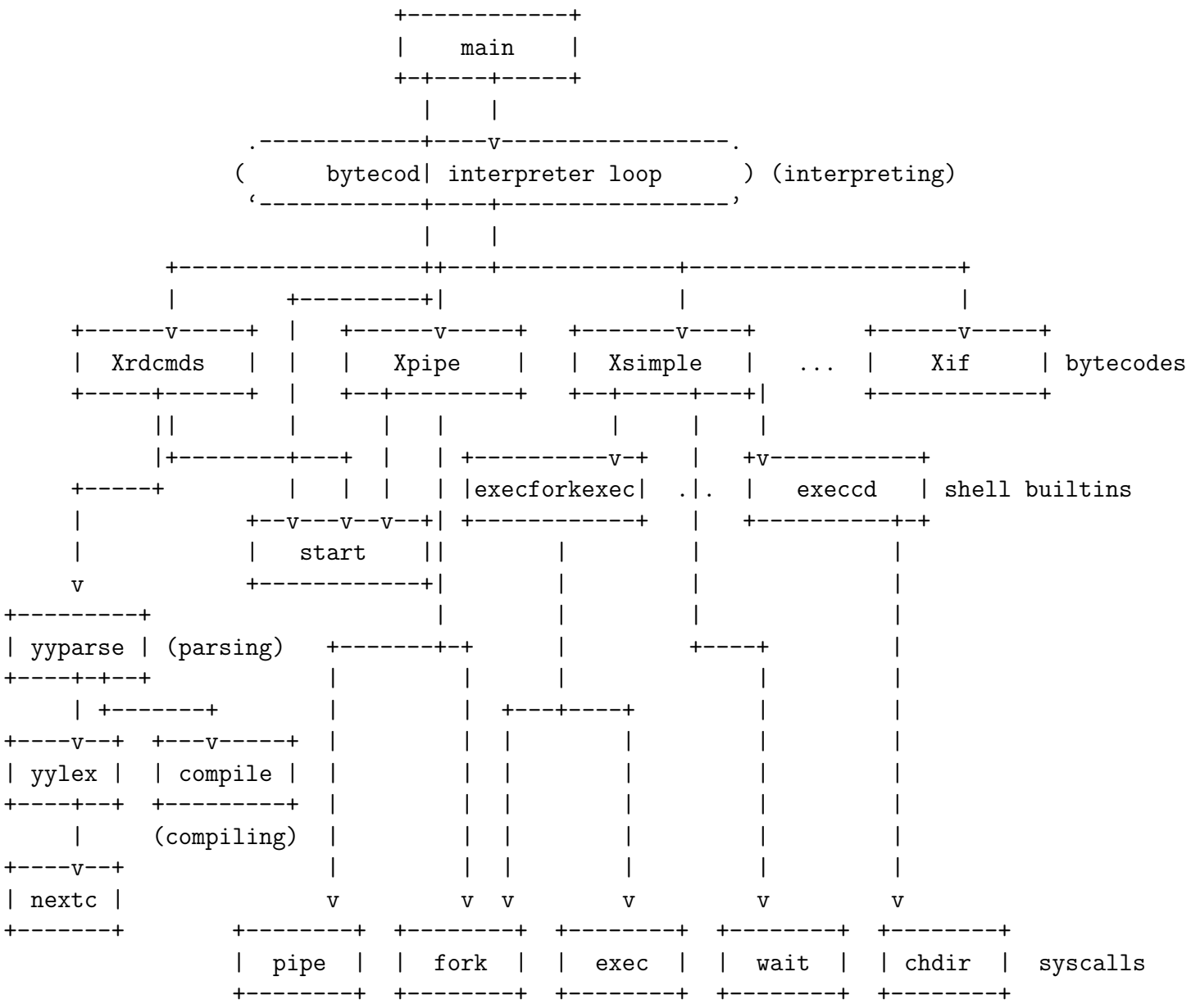


Figure 2.3: Control flow diagram of `rc`.

After some basic initializations, `main()X` sets its local `bootstrap`[41a] to contain the initial set of bytecodes to execute. It then calls `start()`[44a] 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`[34b] to the content of `bootstrap`. After `start()` returns, `main()X` 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()`[52b] 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()`[71g] 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 `codebuf` as a parameter. `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`[34d] field). Moreover, `compile()` adds the bytecode `Xreturn()`[91d] at the end of the series of bytecodes deriving from your command. Thus, after `main()X` 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()`[95b]). 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()`[77a], 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.

2.6.3 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()`[46a], `Xsimple()`[77a], and `Xreturn()`[91d]. I will explain `Xmark()`[73a] and `Xword()`[72f] later in this document.

2.7 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()X`, the initialization of `rc`, the bytecode interpreter loop, and the function `Xrdcmds()`[46a]. The following chapters will describe the main components of the compiler 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()`[110c] for `cd`). Chapter 10 contains the code to manage the environment of the processes launched from the shell, and Chapter 11 the code to manage the signals sent to the processes (also known as *notes* under Plan 9). Chapter 12 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 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). Finally, Appendix D presents examples of `rc` scripts.

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, which is itself defined by a list of words.

3.1 `Token`

As mentioned in the previous chapter, 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()`[53a] (described in Chapter 6) is responsible for this transformation.

The token kinds in `rc` are declared below using the `yacc` syntax `%token` (see Compiler book [Pad16b]). 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()`[162e]).

⟨*token declarations* 30⟩≡ (62a) 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()`[62b] (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.

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.

⟨*struct* `Tree` 31a⟩≡ (208c)
```
struct Tree {

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

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

    ⟨Tree redirection and pipe specific fields 57a⟩
    ⟨Tree word specific fields 59a⟩

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

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⟩+≡ (62a) ◁30 68c▷
```
/* 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` through the `Tree.next` field below. This makes it easy to free all nodes at once (via `freenodes()`[32a]) after the tree has been compiled to bytecodes in `Xrdcmds()`[46a], instead of having to walk the tree recursively.

⟨*global* `treenodes` 31c⟩≡ (218b)
```
// list<ref_own<Tree>> (next = Tree.next)
tree *treenodes;
```

⟨`Tree` *extra fields* 31d⟩≡ (31a)
```
tree *next;
```

The functions below are used by the parser to build AST nodes. `newtree()`[31e] allocates a fresh node and links it into `treenodes`. The convenience wrappers `tree1()`[32b], `tree2()`[32c], and `tree3()`[32d] create a node with a given type and one, two, or three children respectively.

⟨*function* `newtree` 31e⟩≡ (218b)
```
/*
 * 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 32a⟩≡ (218b)

```
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() 169d and treenodes 31c.

⟨*function* tree1 32b⟩≡ (218b)

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

Uses tree3() 32d.

⟨*function* tree2 32c⟩≡ (218b)

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

Uses tree3() 32d.

⟨*function* tree3 32d⟩≡ (218b)

```
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() 31e.

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⟩≡ (32d)
```
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()`[71g] (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⟩≡ (211b)
```
// 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`[77a], `Xpipe`[92c]), an integer `i` (used for jump offsets or file descriptor numbers), or a string `s` (used for inline string arguments like in `Xword`[72f]). The first element of any code vector is always a reference count integer, managed by `codecopy()`[33d] and `codefree()`[33e] below. This reference counting is needed because bytecode vectors can be shared, for instance when a function is defined with `fn`.

⟨*struct* `Code` 33c⟩≡ (208c)
```
/*
 * 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;
    char *s;
};
```

⟨*function* `codecopy` 33d⟩≡ (221)
```
code*
codecopy(code *cp)
{
    cp[0].i++;
    return cp;
}
```

⟨*function* `codefree` 33e⟩≡ (221)
```
void
codefree(code *cp)
{
    code *p;
    // check ref count
    if(--cp[0].i != 0)
        return;
```

```
    for(p = cp+1; p->f; p++){
        ⟨codefree() in loop over code cp, switch bytecode cases 34a⟩
    }
    efree((char *)cp);
}
```

Uses efree() 169d.

⟨codefree() *in loop over code* cp*, switch bytecode cases* 34a⟩≡ (33e) 76e▷

```
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() 89c, Xasync() 94b, Xbackq() 150c, Xcase() 100b, Xclose() 148d, Xfalse() 83c, Xfor() 98a, Xjump() 97b, Xrdwr() 145e, Xread() 89a, Xsubshell() 140d, Xtrue() 83b, and Xwrite() 88a.

3.4 Thread and runqeue

Once the bytecodes are compiled, rc needs a data structure to keep track of the current execution state. This is the role of the Thread 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()[92c] or Xasync()[94b] needs to run commands concurrently, it creates actual OS processes (with fork()), but within the interpreter itself, threads are managed as a stack.

The global runq[34c] (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()[91d]), 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()[46a].

⟨*struct* Thread 34b⟩≡ (211a)

```
struct Thread {
    union Code *code;  /* code for this thread */
    int pc;   /* code[pc] is the next instruction */

    ⟨Thread other fields 35a⟩

    // Extra
    ⟨Thread extra fields 34d⟩
};
```

Uses Code 33c.

⟨*global* runq 34c⟩≡ (211b)

```
// stack<ref_own<Thread>> (next = Thread.ret)
thread *runq;
```

⟨Thread *extra fields* 34d⟩≡ (34b)

```
thread *ret;  /* who continues when this finishes */
```

The most important field of a thread is argv, the argument stack. It is a list of lists of words: bytecodes like Xmark()[73a] push a new empty list on this stack, and Xword()[72f] pushes a word onto the current list. When a bytecode like Xsimple()[77a] needs to run a command, it pops the top list from argv and uses it as the command and its arguments.

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):

```
after Xmark                 after Xword "/tmp"

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

after Xword "-l"            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 | wc` 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.

⟨`Thread` *other fields* 35a⟩≡ (34b) 35b▷
```
// list<list<ref_own<word>>> (next = List.next)
struct List *argv;  /* argument stack */
```
Uses `List` 38a.

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()`[39b], `rc` first searches the current thread's locals before falling back to the global variable table.

⟨`Thread` *other fields* 35b⟩+≡ (34b) ◁35a 46b▷
```
// list<ref_own<Var>> (next = Var.next)
struct Var *local;  /* list of local variables */
```
Uses `Var` 38b.

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` 36a⟩≡ (208c)
```
/*
 * word lists are in correct order,
 * i.e. word0->word1->word2->word3->nil
 */
struct Word {
    char *word;

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

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

⟨*function* `newword` 36b⟩≡ (224b)
```
word*
newword(char *wd, word *next)
{
    word *p = new(struct Word);
    p->word = strdup(wd);
    p->next = next;
    return p;
}
```
Uses `Word` 36a.

⟨*function* `count` 36c⟩≡ (224b)
```
int
count(word *w)
{
    int n;
    for(n = 0;w;n++)
        w = w->next;
    return n;
}
```

`copynwords()`[36d] 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` 36d⟩≡ (224b)
```
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() 36b.

⟨*function* copywords 37a⟩≡ (224b)

```
/*
 * 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() 36b.

⟨*function* freewords 37b⟩≡ (224b)

```
void
freewords(word *w)
{
    word *nw;
    while(w){
        efree(w->word);
        nw = w->next;
        efree((char *)w);
        w = nw;
    }
}
```

Uses efree() 169d.

⟨*function* freelist 37c⟩≡ (224b)

```
void
freelist(word *w)
{
    word *nw;
    while(w){
        nw = w->next;
        efree(w->word);
        efree((char *)w);
        w = nw;
    }
}
```

Uses efree() 169d.

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()`[73a] and `Xword()`[72f] 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⟩≡ (211a)
```
struct List {
    // list<ref_own<Word>> (next = Word.next)
    word *words;

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

3.6 Variables

I already mentioned local variables in `Thread.local` 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).

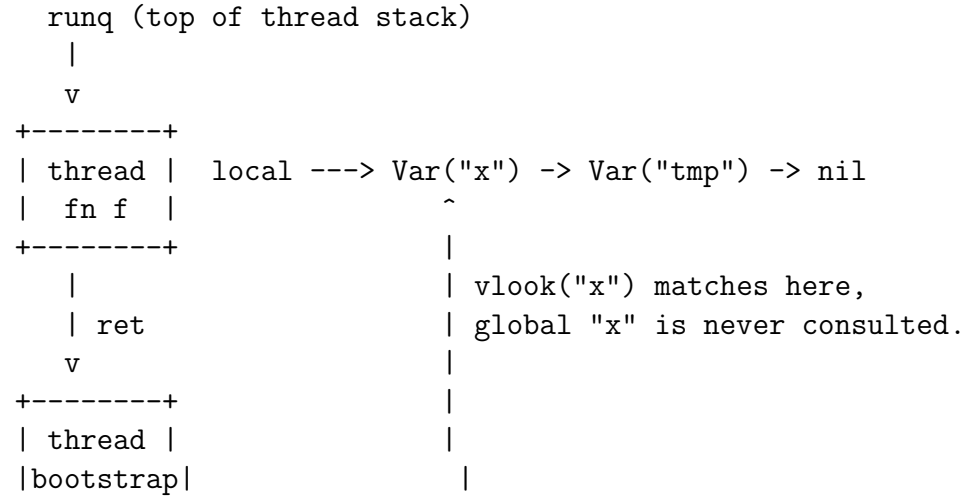
A `Var` has a `name` (the variable name as a string), a `val` (the variable value as a list of words, since in `rc` all variables can hold multiple values), and a `next` pointer for chaining in hash buckets or local lists.

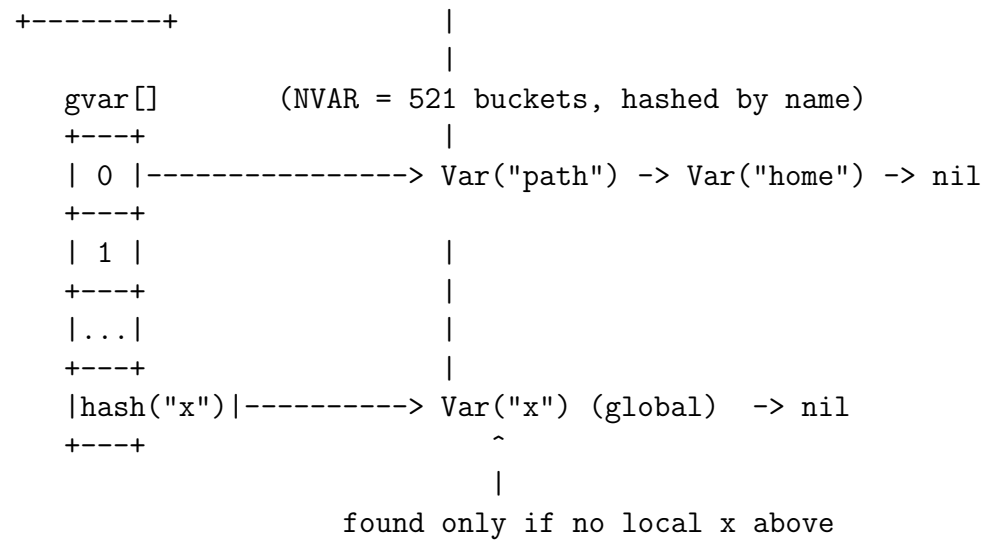
⟨*struct* `Var` 38b⟩≡ (208c)
```
struct Var {
    // key
    char *name;  /* ascii name */
    // value
    word *val;  /* value */

    ⟨Var other fields 101a⟩
    // Extra
    ⟨Var extra fields 38c⟩
};
```

⟨`Var` *extra fields* 38c⟩≡ (38b)
```
var *next;  /* next on hash or local list */
```

The main interface to the variable system consists of `setvar()`[39a] to assign a value, and `vlook()`[39b] 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()`[40a].

Pictorially, a lookup for `x` inside a function call where `x` has been declared local walks the current thread's `local` chain first, and only on miss does it hash into `gvar`:





The lookup order matters because `for x in ...` and `fn f x=42; ...` introduce names that must shadow any pre-existing global `x` for the duration of the block, and then disappear when the block exits. Making `vlook` search `local` first—and making `Xlocal` prepend rather than mutate—gives shadowing and restoration for free: when the thread pops, its `local` list is discarded and the global `x` is visible again. There is no separate "scope chain" structure: the chain is just `Thread.local` walked from the top of `runq`.

⟨*function* `setvar` 39a⟩≡ (214a)
```
void
setvar(char *name, word *val)
{
    struct Var *v = vlook(name);
    freewords(v->val);
    v->val = val;
    v->changed = true;
}
```
Uses `Var` 38b, `freewords()` 37b, and `vlook()` 39b.

⟨*function* `vlook` 39b⟩≡ (214a)
```
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()` 40a and `runq` 34c.

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` 39c⟩≡ (208c)
```
#define NVAR 521
```

⟨*global* `gvar` 39d⟩≡ (214a)
```
// map<string, ref_own<Var>> (next = Var.next in bucket list)
var *gvar[NVAR];  /* hash for globals */
```
Uses `NVAR` 39c.

⟨*function* gvlook 40a⟩≡ (214a)

```
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 39c, gvar 39d, hash() 40b, and newvar() 40c.

⟨*function* hash 40b⟩≡ (214a)

```
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 40c⟩≡ (214a)

```
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;
}
```

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()X`, the entry point of `rc`.

4.1 Overview

The `main()X` 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 here a simplified version of the bootstrap; the actual bootstrap code (which sources `rcmain`) is described in Chapter 12.

⟨`main()` *locals* 41a⟩≡ (41b) ?? ▷
```
code bootstrap[17];
```

⟨*function main (rc/exec.c)* 41b⟩≡ (225a)
```
void main(int argc, char *argv[])
{
    ⟨main() locals 41a⟩

    ⟨main() argc argv processing, modify flags 42b⟩

    ⟨main() initialisation 43a⟩
    ⟨main() initialize boostrap 126a⟩
    ⟨main() initialize runq with bootstrap code 43d⟩
    ⟨main() initialize runq->argv 44c⟩

    ⟨main() interpreter loop 45c⟩
}
```

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` 41c⟩≡ (212a)
```
// map<char, option<array<string>>>
char **flag[NFLAG];
```
Uses `NFLAG` 41d.

⟨*constant* `NFLAG` 41d⟩≡ (208a)
```
#define NFLAG 128
```

⟨*global* `flagset` 42a⟩≡ (212a)
```
char *flagset[] = {"<flag>"};
```

⟨`main()` *argc argv processing, modify flags* 42b⟩≡ (41b) 42c▷
```
argc = getflags(argc, argv, "SsrdiIlxepvVc:1m:1[command]", 1);
if(argc==-1)
    usage("[file [arg ...]]");
```
Uses `getflags()` 170i.

`getflags()`[170i] 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 the kernel starts `rc` as a login shell, 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 this and sets `flag['l']` accordingly, so that `rc` will later source the user's profile (`$home/lib/profile`).

⟨`main()` *argc argv processing, modify flags* 42c⟩+≡ (41b) ◁42b 42d▷
```
if(argv[0][0]=='-')
    flag['l'] = flagset;
```
Uses `flag` 41c and `flagset` 42a.

4.2.2 Interactive mode: `rc -i`

Interactive mode determines whether `rc` will display a prompt and handle 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 (useful when piping commands into `rc` from another program).

⟨`main()` *argc argv processing, modify flags* 42d⟩+≡ (41b) ◁42c
```
if(flag['I'])
    flag['i'] = nil;
else
    if(flag['i']==nil && argc==1 && Isatty(STDIN))
       flag['i'] = flagset;
```
Uses `flag` 41c and `flagset` 42a.

Under Plan 9, there is no `isatty()` system call. Instead, `Isatty()`[42e] checks whether the file descriptor corresponds to `/dev/cons` (the console device) by inspecting its path with `fd2path()`.

⟨*function* `Isatty` 42e⟩≡ (225b)
```
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()`[175b]). Then it initializes three subsystems: `kinit()`[60e] registers the shell keywords (`if`, `for`, etc.) so the lexer can recognize them, `Trapinit()`[122c] sets up the signal handler (see Chapter 11), and `Vinit()`[118c] reads the process environment (from `/env`) and populates the global variable table (see Chapter 10).

⟨`main()` *initialisation* 43a⟩≡ (41b) 43b▷
```
err = openfd(STDERR);
```
Uses `err` 167b and `openfd()` 175b.

⟨`main()` *initialisation* 43b⟩+≡ (41b) ◁43a 126d▷
```
kinit();    // initialize keywords
Trapinit(); // notify() function setup
Vinit();    // read environment variables and add them in gvar
```
Uses `Vinit()` 118c and `kinit()` 60e.

4.4 Bootstrapping bytecodes (simplified)

As 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`[46a], 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 12): it sources the `rcmain` initialization script, which sets up the default `$path` and runs the user's profile.

⟨`main()` *initialize* `boostrap` *(simplified)* 43c⟩≡
```
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()`[44a] creates a new thread and pushes it on top of `runq`[34c]. 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* 43d⟩≡ (41b) 46c▷
```
start(bootstrap, 1, (var *)nil);
```
Uses `start()` 44a.

⟨*function* start 44a⟩≡ (215a)
```
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 85a⟩

    // add_stack(runq, p)
    p->ret = runq;
    runq = p;
}
```
Uses Thread 34b, codecopy() 33d, and runq 34c.

4.6 Setting runq->argv

Now that runq[34c] is set, main()X populates its argument stack with the command-line arguments. These arguments are pushed in reverse order because pushword()[45b] 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 44b⟩≡ (215a)
```
/*
 * Start executing the given code at the given pc with the given redirection
 */
char *argv0="rc";
```
Uses argv0 44b.

⟨main() *initialize* runq->argv 44c⟩≡ (41b)
```
/* prime bootstrap argv */
pushlist();
argv0 = strdup(argv[0]);
for(i = argc-1; i!=0; --i)
    pushword(argv[i]);
```
Uses argv0 44b, pushlist() 45a, and pushword() 45b.

For example, after rc myscript.rc foo bar is initialized, 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 bootstrap code will then assign this word list to `\$*` and source `rcmain`, which sets up `\$path` and runs the script.

`pushlist()`[45a] 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()`[73a] and `Xword()`[72f].

⟨*function* `pushlist` 45a⟩≡ (215a)
```
void
pushlist(void)
{
    list *p = new(list);

    // add_list(p, runq->argv)
    p->next = runq->argv;
    p->words = nil;
    runq->argv = p;
}
```
Uses `runq` 34c.

⟨*function* `pushword` 45b⟩≡ (215a)
```
void
pushword(char *wd)
{
    if(runq->argv==nil)
        panic("pushword but no argv!", 0);
    runq->argv->words = newword(wd, runq->argv->words);
}
```
Uses `newword()` 36b, `panic()` 168c, and `runq` 34c.

4.7 Bytecode interpreter loop

The heart of `rc` is this infinite loop. 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`[72f]), it can read them at `runq->code[runq->pc]` and advance `pc` itself. Between bytecodes, the loop also checks for pending signals (traps).

⟨`main()` *interpreter loop* 45c⟩≡ (41b)
```
for(;;){
    ⟨main() debug runq in interpreter loop 164b⟩

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

    ⟨main() handing trap if necessary in interpreter loop 122e⟩
}
```
Uses `runq` 34c.

4.8 Reading commands: `Xrdcmds()`

`Xrdcmds()`[46a] is the bytecode that implements the read-eval loop of `rc`. It calls `yyparse()`[62b] to read and compile one command line, then calls `start()`[44a] to push a new thread executing the compiled bytecodes. The key subtlety is the `--p->pc` before `start()`: this makes the bootstrap thread re-execute `Xrdcmds()` when the command thread finishes (via `Xreturn()`[91d]). This is how `rc` loops forever reading commands without an

explicit loop in the bootstrap bytecodes themselves. Note that `yyparse()` internally calls `compile()`[71g], which modifies the global `codebuf`[33b]. So when `Xrdcmds()` calls `start(codebuf, ...)`, it starts a thread on the freshly compiled bytecodes.

⟨*function* `Xrdcmds` 46a⟩≡ (215b)
```
void
Xrdcmds(void)
{
    struct Thread *p = runq;
    bool error;
    ⟨Xrdcmds() other locals 50e⟩

    ⟨Xrdcmds() flush errors and reset error count 167d⟩
    ⟨Xrdcmds() print status if -s 166a⟩
    ⟨Xrdcmds() set promptstr if interactive mode 50f⟩

    ⟨Xrdcmds() calls Noerror() before yyparse() 123d⟩
    // read one cmd line, compiles it, and modifies codebuf global
    error = yyparse();

    ⟨Xrdcmds() if yyparse() returned an error 46d⟩
    else{
        ⟨Xrdcmds() reset ntrap 124⟩
        --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` 34b, `codebuf` 33b, `freenodes()` 32a, `runq` 34c, `start()` 44a, and `yyparse()`.

⟨`Thread` *other fields* 46b⟩+≡ (34b) ◁35b 49c▷
```
struct Io *cmdfd; /* file descriptor for Xrdcmd */
char *cmdfile;  /* file name in Xrdcmd */
bool iflag;  /* interactive? */
```
Uses `Io` 175a.

Each thread also carries input-related fields: `cmdfd` 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 and `-c` flag create new threads that read from different sources (a script file, a string), and each thread needs its own input state.

⟨`main()` *initialize* `runq` *with bootstrap code* 46c⟩+≡ (41b) ◁43d
```
runq->cmdfd = openfd(STDIN); // reading from stdin
runq->cmdfile = "<stdin>";
runq->iflag = flag['i']? true : false;// interactive mode; will print a prompt
```
Uses `flag` 41c, `openfd()` 175b, and `runq` 34c.

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* 46d⟩≡ (46a)
```
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()` 91d, `closeio()` 177a, `efree()` 169d, `err` 167b, and `pchr()` 176a.

Chapter 5

Input

Recall the main flow from Chapter 4: `main()X` → `Xrdcmds()`[46a] → `yyparse()`[62b] → `yylex()`[53a]. 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()`[48] reads a raw character and handles escaped newlines (for multi-line commands) and prompt display. `nextc()`[52b] adds a one-character lookahead on top of `getnext()`, and `advance()`[52c] consumes that lookahead.

5.2 Reading a character: `getnext()`

`getnext()`[48] reads one character from the current thread's input stream (`runq->cmdfd`). Besides the actual read (via `rchr()`[176c], 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` 48⟩≡ (213)

```
/*
 * read a character from the input stream
 */
int
getnext(void)
{
    int c;
    ⟨getnext() other locals 50a⟩
```

```
    ⟨getnext() peekc handling 50b⟩
    ⟨getnext() return if already at EOF 49d⟩
    ⟨getnext() prompt management before reading the character 50g⟩

    c = rchr(runq->cmdfd);

    ⟨getnext() handle backslash 49e⟩
    ⟨getnext() prompt management after the character is read 51b⟩
    ⟨getnext() if character read is EOF 49b⟩
    ⟨getnext() if not EOF but verbose mode, print character read 166c⟩

    return c;
}
```

Uses rchr() 176c and runq 34c.

5.2.1 End-of-file management

When rchr()[176c] returns EOF, getnext()[48] 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()[46a] to decide whether to close the input and return to the parent thread (see Chapter 4).

⟨*constant EOF (rc/io.h)* 49a⟩≡ (208b)
```
#define EOF (-1)
```

⟨getnext() *if character read is EOF* 49b⟩≡ (48)
```
if(c==EOF)
    runq->eof = true;
```

Uses EOF 49a and runq 34c.

⟨Thread *other fields* 49c⟩+≡ (34b) ◁46b 51a▷
```
bool eof;  /* is cmdfd at eof? */
```

⟨getnext() *return if already at EOF* 49d⟩≡ (48)
```
if(runq->eof)
    return EOF;
```

Uses EOF 49a and runq 34c.

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. getnext()[48] 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() *handle backslash* 49e⟩≡ (48)
```
if(!inquote && c=='\\'){

    c = rchr(runq->cmdfd);

    if(c=='\n' && !incomm){  /* don't continue a comment */
        ⟨getnext() when backslash and newline, set doprompt 51c⟩
        c=' ';
    }
    else{
        peekc = c;
        c='\\';
    }
}
```

Uses incomm 54e, inquote 58c, rchr() 176c, and runq 34c.

⟨getnext() *other locals* 50a⟩≡ (48)
```
static int peekc = EOF;
```
Uses EOF 49a.

⟨getnext() *peekc handling* 50b⟩≡ (48)
```
if(peekc!=EOF){
    c = peekc;
    peekc = EOF;
    return c;
}
```
Uses EOF 49a.

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 displayed just before reading a character, not at the end of executing a command. This is done through the global doprompt flag, which is set to true after each newline (or EOF) and reset to false after the prompt is printed.

⟨*global* doprompt 50c⟩≡ (213)
```
bool doprompt = true;
```
Uses doprompt 50c.

⟨*global* promptstr 50d⟩≡ (211b)
```
char *promptstr;
```

⟨Xrdcmds() *other locals* 50e⟩≡ (46a)
```
word *prompt;
```

⟨Xrdcmds() *set promptstr if interactive mode* 50f⟩≡ (46a)
```
if(runq->iflag){
    prompt = vlook("prompt")->val;
    if(prompt)
        promptstr = prompt->word;
    else
        promptstr="% ";
}
```
Uses promptstr 50d, runq 34c, and vlook() 39b.

⟨getnext() *prompt management before reading the character* 50g⟩≡ (48)
```
if(doprompt)
    // pprompt() internally set doprompt back to false at the end
    pprompt();
```
Uses doprompt 50c and pprompt() 50h.

The subtlety in pprompt()[50h] 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()[46a] before each new command.

⟨*function* pprompt 50h⟩≡ (213)
```
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` 50c, `err` 167b, `flush()` 175d, `promptstr` 50d, `pstr()` 179a, `runq` 34c, and `vlook()` 39b.

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* 51a⟩+≡ (34b) ◁49c 85c▷

```
int lineno;   /* linenumber */
```

⟨`getnext()` *prompt management after the character is read* 51b⟩≡ (48)

```
doprompt = doprompt || c=='\n' || c==EOF;
```

Uses `EOF` 49a and `doprompt` 50c.

⟨`getnext()` *when backslash and newline, set doprompt* 51c⟩≡ (49e)

```
doprompt = true;
```

Uses `doprompt` 50c.

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()`[52b] provides this lookahead by caching the next character in the global `future`. Multiple calls to `nextc()` return the same character until `advance()`[52c] is called to consume it. The helper `nextis()`[52d] combines both: it checks whether the next character matches, and consumes it if so.

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

```
|-----------------------|
|   yylex()             |  wants "next non-space token char"
|-----------------------|
          |
          v
|-----------------------|
|   nextc / advance     |  slot: future    (lexer level)
|-----------------------|
          |
          v
|-----------------------|
|   getnext()           |  slot: peekc     (char-stream level)
|-----------------------|
          |
```

```
          v
  |-----------------------|
  |   rchr(runq->cmdfd)   |  raw bufio read
  |-----------------------|
```

`peekc` is internal to `getnext()`[48] 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()`[53a] 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.

⟨*global* `future` 52a⟩≡ (213)
```
int future = EOF;
```
Uses `EOF` 49a and `future` 52a.

⟨*function* `nextc` 52b⟩≡ (213)
```
/*
 * Look ahead in the input stream
 */
int
nextc(void)
{
    if(future==EOF)
        future = getnext();
    return future;
}
```
Uses `EOF` 49a, `future` 52a, and `getnext()` 48.

⟨*function* `advance` 52c⟩≡ (213)
```
/*
 * Consume the lookahead character.
 */
int
advance(void)
{
    int c = nextc();
    ⟨advance() save future in lastc 168b⟩
    future = EOF;
    return c;
}
```
Uses `EOF` 49a, `future` 52a, and `nextc()` 52b.

⟨*function* `nextis` 52d⟩≡ (213)
```
bool
nextis(int c)
{
    if(nextc()==c){
        advance();
        return true;
    }
    return false;
}
```
Uses `advance()` 52c and `nextc()` 52b.

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()`[53a], called by the `yacc`-generated parser whenever it needs the next token.

6.1 `yylex()`

The structure of `yylex()`[53a] 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. There are a few lexing subtleties worth noting: newlines are sometimes skipped (via `skipnl()`[55a]) 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).

⟨*function* `yylex` 53a⟩≡ (219a)
```
//@Scheck: called from yyparse()
int yylex(void)
{
    int c;
    ⟨yylex() other locals 56d⟩

    ⟨yylex() hack for SUB 61b⟩
    ⟨yylex() initialisations 56f⟩

    skipwhite();

    switch(c = advance()){
    ⟨yylex() switch c cases 54b⟩
    }
    // else
    ⟨yylex() if c is not a word character 58b⟩
    // else
    ⟨yylex() if c is a word character 59c⟩
}
```
Uses `advance()` 52c and `skipwhite()` 54f.

⟨*function* `wordchr` 53b⟩≡ (219a)
```
int
wordchr(int c)
{
    return !strchr("\n \t#;&|^$=‘’{}()<>", c) && c!=EOF;
}
```
Uses `EOF` 49a.

⟨*function* `idchr` 54a⟩≡ (219a)
```
int
idchr(int c)
{
    /*
     * Formerly:
     * return 'a'<=c && c<='z' || 'A'<=c && c<='Z' || '0'<=c && c<='9'
     * || c=='_' || c=='*';
     */
    return c > ' ' && !strchr("!\"#$%&'()+,-./:;<=>?@[\\]^`{|}~", c);
}
```

⟨`yylex()` *switch c cases* 54b⟩≡ (53a) 55b▷
```
case EOF:
    lastdol = false;
    strcpy(tok, "EOF");
    return EOF;
```
Uses `EOF` 49a, `lastdol` 56b, and `tok` 54c.

⟨*global* `tok` 54c⟩≡ (219a)
```
char tok[NTOK + UTFmax];
```
Uses `NTOK-10` 54d.

⟨*constant* `NTOK` 54d⟩≡ (219a)
```
#define NTOK 8192  /* maximum bytes in a word (token) */
```

6.2 Spaces and comments ('#')

`skipwhite()`[54f] skips spaces, tabs, and comments. In `rc`, comments start with `#` and extend to the end of the line. Note that the `incomm` global is needed by `getnext()`[48]: inside a comment, a backslash-newline should *not* be treated as a line continuation.

⟨*global* `incomm` 54e⟩≡ (213)
```
bool incomm;
```

⟨*function* `skipwhite` 54f⟩≡ (219a)
```
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;
                }
                advance();
            }
        }
        if(c==' ' || c=='\t')
            advance();
        else
            return;
```

```
    }
}
```

Uses EOF 49a, advance() 52c, incomm 54e, and nextc() 52b.

6.3 Newlines

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()[55a] 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 55a⟩≡ (219a)

```
void
skipnl(void)
{
    int c;
    for(;;){
        skipwhite();
        c = nextc();
        if(c!='\n')
            return;
        // consume the newline
        advance();
    }
}
```

Uses advance() 52c, nextc() 52b, and skipwhite() 54f.

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

Each operator is handled as a case in the main switch of yylex()[53a]. For multi-character operators like && or ||, the lexer uses nextis()[52d] to check the second character. Note that after binary operators like &&, ||, and |, the lexer calls skipnl()[55a] 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 6.7).

⟨yylex() *switch c cases* 55b⟩+≡ (53a) ◁54b 56a▷

```
case '&':
    lastdol = false;
    if(nextis('&')){
        skipnl();
        strcpy(tok, "&&");
        return ANDAND;
    }
    strcpy(tok, "&");
    return '&';
```

Uses ANDAND, lastdol 56b, nextis() 52d, skipnl() 55a, and tok 54c.

⟨yylex() *switch c cases* 56a⟩+≡ (53a) ◁55b 56c▷
```
case '$':
    lastdol = true;
    if(nextis('#')){
        strcpy(tok, "$#");
        return COUNT;
    }
    if(nextis('"')){
        strcpy(tok, "$\"");
        return '"';
    }
    strcpy(tok, "$");
    return '$';
```
Uses COUNT, lastdol 56b, nextis() 52d, and tok 54c.

⟨*global* lastdol 56b⟩≡ (219a)
```
bool lastdol; /* was the last token read '$' or '$#' or '"'? */
```

⟨yylex() *switch c cases* 56c⟩+≡ (53a) ◁56a 58f▷
```
case '|':
    lastdol = false;
    if(nextis('|')){
        skipnl();
        strcpy(tok, "||");
        return OROR;
    }
    // FALLTHROUGH
case '<':
case '>':
    lastdol = false;
    ⟨yylex() in switch when redirection character 56g⟩
```
Uses OROR, lastdol 56b, nextis() 52d, skipnl() 55a, and tok 54c.

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 a Tree node right here (rather than in the parser), because the bracket syntax >[2=1] requires parsing digits and = signs that would be awkward to express in the yacc grammar.

⟨yylex() *other locals* 56d⟩≡ (53a) 56e▷
```
char *w = tok;
```
Uses tok 54c.

⟨yylex() *other locals* 56e⟩+≡ (53a) ◁56d 61a▷
```
struct Tree *t;
```
Uses Tree.

⟨yylex() *initialisations* 56f⟩≡ (53a) 58d▷
```
yylval.tree = nil;
```
Uses yylval.

⟨yylex() *in switch when redirection character* 56g⟩≡ (56c)
```
/*
 * 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 57e⟩
}
⟨yylex() in switch when redirection character, if bracket after 146⟩
*w='\0';
yylval.tree = t;
if(t->type==PIPE)
    skipnl();
return t->type;
```

Uses PIPE, newtree() 31e, skipnl() 55a, and yylval.

⟨Tree *redirection and pipe specific fields* 57a⟩≡ (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 */
```

⟨*constant* WRITE 57b⟩≡ (208c)

```
#define WRITE 2
```

⟨*constant* READ 57c⟩≡ (208c)

```
#define READ 3
```

⟨*constant* APPEND 57d⟩≡ (208c)

```
#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 14.4.4.

⟨yylex() *in switch when redirection character, switch c cases* 57e⟩≡ (56g) 57f▷

```
case '|':
    t->type = PIPE;
    t->fd0 = 1; // left fd of pipe (stdout of left cmd)
    t->fd1 = 0; // right fd of pipe (stdin of right cmd)
    break;
```

Uses PIPE.

⟨yylex() *in switch when redirection character, switch c cases* 57f⟩+≡ (56g) ◁57e 58a▷

```
case '>':
    t->type = REDIR;
    if(nextis('>')){
        t->rtype = APPEND;
        *w++=c;
    }
    else
        t->rtype = WRITE;
    t->fd0 = 1;
    break;
```

Uses APPEND 57d, REDIR, WRITE 57b, and nextis() 52d.

⟨yylex() *in switch when redirection character, switch c cases* 58a⟩+≡ (56g) ◁57f
```
case '<':
    t->type = REDIR;
    ⟨yylex() in switch when redirection character, if here document 141b⟩
    ⟨yylex() in switch when redirection character, if read/write redirect 145c⟩
    else
        t->rtype = READ;
    t->fd0 = 0;
    break;
```
Uses READ 57c and REDIR.

⟨yylex() *if c is not a word character* 58b⟩≡ (53a)
```
if(!wordchr(c)){
    lastdol = false;
    tok[0] = c;
    tok[1]='\0';
    return c;
}
```
Uses lastdol 56b, tok 54c, and wordchr() 53b.

6.5 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 flag is needed by getnext()[48] because inside a quoted string, backslash-newline should *not* be treated as a line continuation. The lastword flag tracks whether the previous token was a word, which the grammar uses to decide whether adjacent tokens should be concatenated (e.g., a^b).

⟨*global* inquote 58c⟩≡ (213)
```
bool inquote;
```

⟨yylex() *initialisations* 58d⟩+≡ (53a) ◁56f
```
inquote = false;
```
Uses inquote 58c.

⟨*global* lastword 58e⟩≡ (219a)
```
// used also by syn.y
bool lastword; /* was the last token read a word or compound word terminator? */
```

⟨yylex() *switch c cases* 58f⟩+≡ (53a) ◁56c
```
case '\'':
    inquote = true;
    lastword = true;
    lastdol = false;
    for(;;){
        c = advance();
        if(c==EOF)
            break;
        if(c=='\''){
            if(nextc()!='\'')
                break;
            advance();
        }
        w = addutf(w, c);
    }
    if(w != nil)
        *w='\0';
```

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

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

Uses EOF 49a, WORD, addutf() 139b, advance() 52c, inquote 58c, lastdol 56b, lastword 58e, nextc() 52b, tok 54c, token() 59b, and yylval.

⟨Tree *word specific fields* 59a⟩≡ (31a)

```
  bool quoted;
```

⟨*function* token 59b⟩≡ (213)

```
  tree*
  token(char *str, int type)
  {
        tree *t = newtree();

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

Uses newtree() 31e.

6.6 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. Once the word is assembled, klook()[60a] checks whether it matches a keyword. Keywords are stored in a small hash table initialized by kinit()[60e] at startup. If the word is a keyword, klook() returns the appropriate token type; otherwise it returns WORD.

⟨yylex() *if c is a word character* 59c⟩≡ (53a)

```
  for(;;){
        ⟨yylex() when c is a word character, if glob character 129b⟩
        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() 139b, advance() 52c, idchr() 54a, klook() 60a, lastdol 56b, lastword 58e, nextc() 52b, tok 54c, wordchr() 53b, and yylval.

⟨*function* klook 60a⟩≡ (213)

```
tree*
klook(char *name)
{
    struct Kw *p;
    tree *t = token(name, WORD);

    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 60b, NKW-8 60d, WORD, hash() 40b, kw 60c, and token() 59b.

⟨*struct* Kw 60b⟩≡ (213)

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

    struct Kw *next;
};
```

Uses Kw 60b.

⟨*global* kw 60c⟩≡ (213)

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

Uses Kw 60b and NKW-8 60d.

⟨*constant* NKW 60d⟩≡ (213)

```
#define NKW 30
```

⟨*function* kinit 60e⟩≡ (213)

```
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() 60f.

⟨*function* kenter 60f⟩≡ (213)

```
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 60b, NKW-8 60d, hash() 40b, and kw 60c.

6.7 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* 61a⟩+≡ (53a) ◁56e

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

Uses nextc() 52b.

⟨yylex() *hack for SUB* 61b⟩≡ (53a)

```
/*
 * 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;
    }
    if(wordchr(d) || d=='\'' || d=='`' || d=='$' || d=='"'){
        strcpy(tok, "^");
        return '^';
    }
}
```

Uses SUB, advance() 52c, lastword 58e, tok 54c, and wordchr() 53b.

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 abstract syntax tree.

7.1 Overview

The parser of `rc` is written using `yacc` (see COMPILER book [Pad16b] for an introduction to `yacc`). 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()`[71g] 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.

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* 62a⟩≡
```
%{
#include "rc.h"
#include "fns.h"
%}

%union {
 struct Tree *tree;
};

⟨token declarations 30⟩
⟨priority and associativity declarations 62c⟩
⟨type declarations 63a⟩

%%
⟨grammar 63b⟩
```

⟨*function* `yyparse` 62b⟩≡
```
... generated code from syn.y by yacc ...
```

⟨*priority and associativity declarations* 62c⟩≡ (62a)
```
/* 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 63a⟩≡ (62a) ?? ▷
```
%type<tree> line cmd simple word comword
```

⟨*grammar* 63b⟩≡ (62a)
```
rc:
    /*empty*/       { return ERROR_1;}
|   line '\n'       { return !compile($1);}
```

⟨*line rule* 64a⟩
⟨*cmd rule* 64b⟩
⟨*simple rule* 64c⟩

⟨*word rule* 64f⟩
⟨*comword rule* 64e⟩

⟨*other rules* 64d⟩

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.

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()`[65a] 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"
```

A slightly more interesting example is `ls -l | wc -l`: 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 the two children on it:

```
          PIPE (fd0=1, fd1=0)
          /    \
      SIMPLE    SIMPLE
        |         |
```

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

Two subtleties fall out of this shape. First, the `SIMPLE` pass is not strictly needed for correctness but makes code generation uniform: `outcode()`[72e] can handle any command as "push a mark, walk the `ARGLIST` chain emitting `Xword`s, then `Xsimple`". Without the `SIMPLE` wrapper, the generator would need a separate entry-point for bare `ARGLIST` subtrees at the top. Second, the `PIPE` tree is built by the lexer rather than the parser because the tree needs `fd0` and `fd1` fields that depend on the concrete bracket syntax (`>[2=1]`, `|[2]`, etc.)—encoding that in the `yacc` grammar would require many more tokens, so the lexer returns a pre-built `Tree` instead.

⟨*line rule* 64a⟩≡ (63b)
```
line:
    cmd
```
⟨*line rule other cases* 66a⟩

⟨*cmd rule* 64b⟩≡ (63b)
```
cmd:
    /*empty*/          {$$=nil;}
|   simple             {$$=simplemung($1);}
```
⟨*cmd rule other cases* 66c⟩

⟨*simple rule* 64c⟩≡ (63b)
```
simple:
    first
|   simple word        {$$=tree2(ARGLIST, $1, $2);}
```
⟨*simple rule other cases* 67c⟩

⟨*other rules* 64d⟩≡ (63b) 64g▷
```
first:
    comword
|   first '^' word     {$$=tree2('^', $1, $3);}
```

⟨*comword rule* 64e⟩≡ (63b)
```
comword:
    WORD
```
⟨*comword rule other cases* 69d⟩

⟨*word rule* 64f⟩≡ (63b)
```
word:
    comword
|   word '^' word      {$$=tree2('^', $1, $3);}
|   keyword            {lastword=true; $1->type=WORD;}
```

⟨*other rules* 64g⟩+≡ (63b) ◁64d 64h▷
```
keyword: FOR|IN|WHILE|IF|NOT|TWIDDLE|BANG|SUBSHELL|SWITCH|FN
```

⟨*other rules* 64h⟩+≡ (63b) ◁64g 66b▷
```
words:
    /*empty*/      {$$=(struct Tree*)nil;}
|   words word      {$$=tree2(WORDS, $1, $2);}
```

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

⟨*function* `simplemung` 65a⟩≡ (218b)
```
/*
 * 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 65b⟩
    return t;
}
```
Uses `Io` 175a, `SIMPLE`, `closeio()` 177a, `openstr()` 177b, `pfmt()` 178b, and `tree1()` 32b.

⟨`simplemung()` *percolate redirections up to the root* 65b⟩≡ (65a)
```
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.

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).

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* 66a⟩≡ (64a)
```
|   cmdsa line      {$$=tree2(';', $1, $2);}
```

⟨*other rules* 66b⟩+≡ (63b) ◁64h 67d▷
```
cmdsa:
    cmd ';'
|   cmd '&'         {$$=tree1('&', $1);}
```

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

⟨*cmd rule other cases* 66c⟩≡ (64b) 66d▷
```
|   cmd ANDAND cmd      {$$=tree2(ANDAND, $1, $3);}
|   cmd ORORR cmd       {$$=tree2(ORORR, $1, $3);}
```

⟨*cmd rule other cases* 66d⟩+≡ (64b) ◁66c 66g▷
```
|   BANG cmd            {$$=mung1($1, $2);}
```

⟨*function* `mung1` 66e⟩≡ (218b)
```
//@Scheck: used by syn.y
tree* mung1(tree *t, tree *c0)
{
    t->child[0] = c0;
    return t;
}
```

⟨*function* `mung3` 66f⟩≡ (218b)
```
//@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;
}
```

7.3.3 String matching ('~')

The `~` (twiddle) operator is the fundamental comparison operator in `rc`. Since all values in `rc` are strings (or lists 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 `[ ]`).

⟨*cmd rule other cases* 66g⟩+≡ (64b) ◁66d 67b▷
```
|   TWIDDLE word words  {$$=mung2($1, $2, $3);}
```

[1] The `*` here is interpreted by the twiddle operator itself using `match()`[134b] (see Chapter 13), not by the shell's globbing mechanism which expands filenames.

⟨*function* mung2 67a⟩≡ (218b)
```
//@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* 67b⟩+≡ (64b) ◁66g 67e▷
```
|   cmd PIPE cmd        {$$=mung2($2, $1, $3);}
```

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()` function later percolates them to the root of the AST subtree.

⟨*simple rule other cases* 67c⟩≡ (64c)
```
|   simple redir        {$$=tree2(ARGLIST, $1, $2);}
```

⟨*other rules* 67d⟩+≡ (63b) ◁66b 67f▷
```
redir:
    REDIR word      {$$=mung1($1, $1->rtype==HERE ? heredoc($2) : $2);}
```
⟨*redir rule other cases* 147c⟩

⟨*cmd rule other cases* 67e⟩+≡ (64b) ◁67b 67h▷
```
|   brace epilog        {$$=epimung($1, $2);}
```

⟨*other rules* 67f⟩+≡ (63b) ◁67d 68b▷
```
epilog:
    /*empty*/           {$$=nil;}
|   redir epilog        {$$=mung2($1, $1->child[0], $2);}
```

⟨*function* epimung 67g⟩≡ (218b)
```
//@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* 67h⟩+≡ (64b) ◁67e 68a▷
```
|   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. 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. `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. 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* 68a⟩+≡ (64b) ◁67h 69a▷

```
  |   IF paren {skipnl();} cmd  {$$=mung2($1, $2, $4);}
  |   IF NOT   {skipnl();} cmd  {$$=mung1($2, $4);}

  |   WHILE paren {skipnl();} cmd    {$$=mung2($1, $2, $4);}
  |   SWITCH word {skipnl();} brace  {$$=tree2(SWITCH, $2, $4);}

   /*
    * 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);}
```

⟨*other rules* 68b⟩+≡ (63b) ◁67f 68d▷

```
  paren:  '(' body ')'        {$$=tree1(PCMD, $2);}
```

⟨*token declarations* 68c⟩+≡ (62a) ◁31b 140a▷

```
  %token PCMD
```

⟨*other rules* 68d⟩+≡ (63b) ◁68b 68e▷

```
  brace:  '{' body '}'        {$$=tree1(BRACE, $2);}
```

⟨*other rules* 68e⟩+≡ (63b) ◁68d 69c▷

```
  body:
      cmd
  |   cmdsan body     {$$=tree2(';', $1, $2);}

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

7.5 Functions (fn)

Function definition in rc uses `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 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* 69a⟩+≡ (64b) ◁68a 69b▷
```
  |   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). The `%prec BANG` directive 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.

⟨*cmd rule other cases* 69b⟩+≡ (64b) ◁69a 140b▷
```
  |   assign cmd                    %prec BANG
        {$$=mung3($1, $1->child[0], $1->child[1], $2);}
```

⟨*other rules* 69c⟩+≡ (63b) ◁68e
```
  assign: first '=' word      {$$=tree2('=', $1, $3);}
```

⟨*comword rule other cases* 69d⟩≡ (64e) 69e▷
```
  |   '$' word        {$$=tree1('$', $2);}
  |   COUNT word      {$$=tree1(COUNT, $2);}
  |   '$' word SUB words ')'  {$$=tree2(SUB, $2, $4);}
```

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* 69e⟩+≡ (64e) ◁69d 148g▷
```
  |   '(' words ')'   {$$=tree1(PAREN, $2);}
```

Chapter 8

Bytecode Generation and Interpretation

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

8.1 Bytecode vs tree-walking interpretation

Why bytecode at all? Most shells—`bash`, `dash`, `zsh`, `fish`—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, in the same spirit as a small Lisp, Forth, or PostScript implementation. The compiled form lives in a `codebuf` of `code` cells, each holding either an opcode function pointer or an inline argument (a string, an integer, or a chained-buffer pointer). 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. 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.

`csh` also uses a bytecode-like representation, but its opcodes are closer to a tokenized command list. `rc`'s opcodes go further: each one is a C function that operates on a per-thread argument stack, the same way Forth or PostScript does. The result is that the C side stays small (one `for` loop dispatching to the next opcode) while the language side stays expressive.

8.2 Overview

Recall the main flow: `main()X` → `Xrdcmds()`[46a] → `yyparse()`[62b] → `compile()`[71g]. 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()X`.

8.2.1 `emitxxx()`

The emit functions append elements to `codebuf`[33b], growing it as needed. There are three variants: `emitf()`[71d] for function pointers (the bytecodes), `emiti()`[71c] for integers (jump offsets, file descriptors), and `emits()`[71e] for

strings (inline arguments like the string after `Xword`[72f]). `codep` is the current write position in the buffer, and `ncode` is the allocated capacity.

⟨*global* `codep` 71a⟩≡ (221)
```
// idx in codebuf
int codep;
```

⟨*global* `ncode` 71b⟩≡ (221)
```
int ncode;
```

⟨*function* `emiti` 71c⟩≡ (221)
```
#define emiti(x) ((codep!=ncode || morecode()), codebuf[codep].i = (x), codep++)
```

⟨*function* `emitf` 71d⟩≡ (221)
```
#define emitf(x) ((codep!=ncode || morecode()), codebuf[codep].f = (x), codep++)
```

⟨*function* `emits` 71e⟩≡ (221)
```
#define emits(x) ((codep!=ncode || morecode()), codebuf[codep].s = (x), codep++)
```

⟨*function* `morecode` 71f⟩≡ (221)
```
//@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` 71b, and `panic()` 168c.

8.2.2 `compile()`

`compile()`[71g] allocates a fresh code buffer, then calls `outcode()`[72e] to recursively walk the AST and emit bytecodes. It appends `Xreturn`[91d] 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` 71g⟩≡ (221)
```
//@Scheck: called from syn.y
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 72a⟩
    ⟨compile() after outcode and error management, read heredoc 141g⟩

    emitf(Xreturn);
    emitf(nil);

    return OK_1;
}
```

Uses `Xreturn()` 91d, `codebuf` 33b, `codep` 71a, `emalloc()` 169b, `emitf-65` 71d, `emiti-66` 71c, `flag` 41c, and `ncode` 71b.

⟨compile() *check nerror* 72a⟩≡ (71g)
```
if(nerror){
    efree((char *)codebuf);
    return ERROR_0;
}
```
Uses codebuf 33b, efree() 169d, and nerror 167c.

8.2.3 outcode()

outcode()[72e] 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): when set, an Xeflag[139a] bytecode is emitted after certain commands to check if they failed and exit accordingly.

⟨*constant c0 (rc/code.c)* 72b⟩≡ (221)
```
#define c0 t->child[0]
```

⟨*constant c1 (rc/code.c)* 72c⟩≡ (221)
```
#define c1 t->child[1]
```

⟨*constant c2 (rc/code.c)* 72d⟩≡ (221)
```
#define c2 t->child[2]
```

⟨*function* outcode 72e⟩≡ (221)
```
void
outcode(tree *t, bool eflag)
{
    ⟨outcode() locals 82c⟩

    if(t==nil)
        return;

    ⟨outcode() set iflast before switch 96g⟩
    switch(t->type){
    ⟨outcode() cases 76a⟩
    default:
        pfmt(err, "bad type %d in outcode\n", t->type);
        break;
    }
    ⟨outcode() set iflast after switch 96f⟩
}
```

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[73a] to start a new word list on argv, emit a series of Xword[72f] to push words onto it, then emit a bytecode (like Xsimple[77a]) that consumes the list. For example, ls -l /tmp compiles to: Xmark, Xword "ls", Xword "-l", Xword "/tmp", Xsimple.

⟨*function* Xword 72f⟩≡ (215b)
```
void
Xword(void)
{
    pushword(runq->code[runq->pc++].s);
}
```
Uses pushword() 45b and runq 34c.

⟨*function* `Xmark` 73a⟩≡ (215b)
```
void
Xmark(void)
{
    pushlist();
}
```
Uses `pushlist()` 45a.

⟨*function* `poplist` 73b⟩≡ (215a)
```
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()` 169d, `freelist()` 37c, `panic()` 168c, and `runq` 34c.

⟨*function* `popword` 73c⟩≡ (215a)
```
void
popword(void)
{
    word *p;
    ⟨popword() sanity check argv 73d⟩
    p = runq->argv->words;
    ⟨popword() sanity check argv words 73e⟩
    runq->argv->words = p->next;
    efree(p->word);
    efree((char *)p);
}
```
Uses `efree()` 169d and `runq` 34c.

⟨`popword()` *sanity check argv* 73d⟩≡ (73c)
```
if(runq->argv==nil)
    panic("popword but no argv!", 0);
```
Uses `panic()` 168c and `runq` 34c.

⟨`popword()` *sanity check argv words* 73e⟩≡ (73c)
```
if(p==nil)
    panic("popword but no word!", 0);
```
Uses `panic()` 168c.

8.2.5 Process status management

The exit status of the last command is stored in the `$status` variable. `setstatus()`[73f] sets it, and `getstatus()`[74a] and `truestatus()`[74b] read it. In `rc`, a status is a string (not an integer like in the Bourne shell). An empty string or the string `"0"` means success; 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` 73f⟩≡ (222c)
```
void
setstatus(char *s)
{
    setvar("status", newword(s, (word *)nil));
}
```
Uses `newword()` 36b and `setvar()` 39a.

⟨*function* `getstatus` 74a⟩≡ (222c)
```
char*
getstatus(void)
{
    var *status = vlook("status");
    return status->val ? status->val->word : "";
}
```
Uses `vlook()` 39b.

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. 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).

⟨*function* `truestatus` 74b⟩≡ (222c)
```
bool
truestatus(void)
{
    char *s;
    for(s = getstatus();*s;s++)
        if(*s!='|' && *s!='0')
            return false;
    return true;
}
```
Uses `getstatus()` 74a.

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) and for proper cleanup when `rc` exits.

⟨*global* `waitpids` 74c⟩≡ (218a)
```
// growing_array<pid> (but really a list)
int *waitpids;
```

⟨*global* `nwaitpids` 74d⟩≡ (218a)
```
int nwaitpids;
```

⟨*function* `addwaitpid` 74e⟩≡ (218a)
```
void
addwaitpid(int pid)
{
    ⟨addwaitpid() grow waitpids if needed 75a⟩
    waitpids[nwaitpids++] = pid;
}
```
Uses `nwaitpids` 74d and `waitpids` 74c.

⟨addwaitpid() *grow* waitpids *if needed* 75a⟩≡ (74e)

```
waitpids = realloc(waitpids, (nwaitpids+1)*sizeof waitpids[0]);
if(waitpids == nil)
    panic("Can't realloc %d waitpids", nwaitpids+1);
```

Uses nwaitpids 74d, panic() 168c, and waitpids 74c.

⟨*function* delwaitpid 75b⟩≡ (225b)

```
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 74d and waitpids 74c.

⟨*function* clearwaitpids 75c⟩≡ (218a)

```
void
clearwaitpids(void)
{
    nwaitpids = 0;
}
```

Uses nwaitpids 74d.

⟨*function* havewaitpid 75d⟩≡ (225b)

```
bool
havewaitpid(int pid)
{
    int i;

    for(i=0; i<nwaitpids; i++)
        if(waitpids[i] == pid)
            return true;
    return false;
}
```

Uses nwaitpids 74d and waitpids 74c.

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.

8.3.1 Bytecode generation

A SIMPLE command compiles to: Xmark[73a] (start a new word list), followed by the bytecodes for the arguments (which push words onto the stack), followed by Xsimple[77a] (which pops the word list and executes the command).

Note that `ARGLIST` and `WORDS` emit their children in reverse order: since `Xword`[72f] prepends to the word list, emitting the last word first ensures the list ends up in the correct order.

⟨`outcode()` *cases* 76a⟩≡ (72e) 76b▷
```
case SIMPLE:
    emitf(Xmark);
    outcode(c0, eflag); // the arguments and argv0
    emitf(Xsimple);
    ⟨outcode() emit Xeflag after Xsimple 138b⟩
    break;
```

⟨`outcode()` *cases* 76b⟩+≡ (72e) ◁76a 76c▷
```
case ARGLIST:
    outcode(c1, eflag);
    outcode(c0, eflag);
    break;
```

⟨`outcode()` *cases* 76c⟩+≡ (72e) ◁76b 76d▷
```
case WORDS:
    outcode(c1, eflag);
    outcode(c0, eflag);
    break;
```

⟨`outcode()` *cases* 76d⟩+≡ (72e) ◁76c 82b▷
```
case WORD:
    emitf(Xword);
    emits(strdup(t->str));
    break;
```

⟨`codefree()` *in loop over code* `cp`*, switch bytecode cases* 76e⟩+≡ (33e) ◁34a 91c▷
```
else if(p->f==Xword || p->f==Xdelhere)
        efree((++p)->s);
```
Uses `Xdelhere()` 142c, `Xword()` 72f, and `efree()` 169d.

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"
```

Note how the arguments are emitted in reverse order (`ARGLIST` emits `c1` before `c0`), so that after `Xword` prepends each word to the list, `argv` ends up in the correct order: `echo`, `hello`, `world`.

8.3.2 `Xsimple()`

`Xsimple()`[77a] 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()`[78b] and waits for it to finish.

The `while(Waitfor(...) < 0)` loop 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.

⟨*function* `Xsimple` 77a⟩≡ (220a)
```
void
Xsimple(void)
{
    word *a;
    thread *p = runq;
    int pid;
    ⟨Xsimple() other locals 103a⟩

    ⟨Xsimple() initializations, globlist() 129e⟩

    a = runq->argv->words;
    ⟨Xsimple() sanity check a 77b⟩

    ⟨Xsimple() if -x 166b⟩

    ⟨Xsimple() if argv0 is a function 103b⟩
    else{
        ⟨Xsimple() if argv0 is a builtin 110a⟩
        ⟨Xsimple() if exitnext() 81b⟩
        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()` 77c, `err` 167b, `execforkexec()` 78b, `flush()` 175d, `poplist()` 73b, and `runq` 34c.

⟨`Xsimple()` *sanity check a* 77b⟩≡ (77a)
```
if(a==nil){
    Xerror1("empty argument list");
    return;
}
```
Uses `Xerror1()` 78a.

⟨*function* `Xerror` 77c⟩≡ (215a)
```
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() 91d, argv0 44b, err 167b, flush() 175d, pfmt() 178b, runq 34c, and setstatus() 73f.

⟨*function* Xerror1 78a⟩≡ (215a)

```
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() 91d, argv0 44b, err 167b, flush() 175d, pfmt() 178b, runq 34c, and setstatus() 73f.

8.3.3 Fork

execforkexec()[78b] 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()[79a] 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. The pushword("exec") before execexec() is a trick: execexec() is also used as the exec builtin, where exec appears as the first word in argv. Here we add it artificially so execexec() can popword() it uniformly.

⟨*function* execforkexec 78b⟩≡ (218a)

```
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() 74e, clearwaitpids() 75c, execexec() 79a, and pushword() 45b.

8.3.4 Exec

⟨*function* execexec 79a⟩≡ (220a)

```
void
execexec(void)
{
    popword(); /* "exec" */
    ⟨execexec() sanity check arguments 79b⟩
    ⟨execexec() perform the redirections 84c⟩

    Execute(runq->argv->words, searchpath(runq->argv->words->word));
    // should not be reached! unless command did not exist
    poplist();
}
```

Uses Execute() 80a, poplist() 73b, popword() 73c, runq 34c, and searchpath() 79c.

⟨execexec() *sanity check arguments* 79b⟩≡ (79a)

```
if(runq->argv->words==nil){
    Xerror1("empty argument list");
    return;
}
```

Uses Xerror1() 78a and runq 34c.

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()[79c] returns the list of directories in $path to try. Execute()[80a] 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 79c⟩≡ (223)

```
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 79d and vlook() 39b.

⟨*global* nullpath 79d⟩≡ (223)

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

Uses Word 36a.

⟨*function* Execute 80a⟩≡ (218a)

```
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()` 169d, `err` 167b, `mkargv()` 80b, and `pfmt()` 178b.

⟨*function* mkargv 80b⟩≡ (218a)

```
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() 36c and emalloc() 169b.

8.3.6 Wait

Back in the parent process, Waitfor()[81a] 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 81a⟩≡ (225b)

```
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 93e⟩
    }

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

Uses delwaitpid() 75b, havewaitpid() 75d, and setstatus() 73f.

8.3.7 Fork optimization

If the next bytecode after Xsimple[77a] is Xexit[94c] (possibly preceded by Xpopredir[88b] calls), the shell is about to exit anyway, so forking a child and waiting for it is pointless. In that case, Xsimple calls execexec()[79a] 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()* 81b⟩≡ (77a)

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

Uses Xexit() 94c, execexec() 79a, exitnext() 82a, and pushword() 45b.

⟨*function* `exitnext` 82a⟩≡ (220a)
```
/*
 * 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()` 94c, `Xpopredir()` 88b, and `runq` 34c.

8.4 Operators

8.4.1 Basic sequence

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

⟨`outcode()` *cases* 82b⟩+≡ (72e) ◁76d 82d▷
```
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`[83b] followed by a placeholder jump offset, then the bytecodes of `cmd2`. After emitting `cmd2`, `stuffdot()`[82e] patches the placeholder with the actual address, so `Xtrue` can jump past `cmd2` if the status is false. `Xtrue` and `Xfalse`[83c] are symmetric: `Xtrue` jumps if the status is *false* (skip the second command), and `Xfalse` jumps if the status is *true*.

⟨`outcode()` *locals* 82c⟩≡ (72e) 91a▷
```
int p;
```

⟨`outcode()` *cases* 82d⟩+≡ (72e) ◁82b 83a▷
```
case ANDAND:
    outcode(c0, false);
    emitf(Xtrue);
    p = emiti(0);
    outcode(c1, eflag);
    stuffdot(p);
    break;
```

⟨*function* `stuffdot` 82e⟩≡ (221)
```
void
stuffdot(int a)
{
    if(a<0 || codep<=a)
        panic("Bad address %d in stuffdot", a);
    codebuf[a].i = codep;
}
```
Uses `codebuf` 33b, `codep` 71a, and `panic()` 168c.

⟨outcode() *cases* 83a⟩+≡ (72e) ◁82d 83d▷
```
  case OROR:
      outcode(c0, false);
      emitf(Xfalse);
      p = emiti(0);
      outcode(c1, eflag);
      stuffdot(p);
      break;
```

⟨*function* Xtrue 83b⟩≡ (215b)
```
  void
  Xtrue(void)
  {
      if(truestatus()) runq->pc++;
      else runq->pc = runq->code[runq->pc].i;
  }
```
Uses runq 34c and truestatus() 74b.

⟨*function* Xfalse 83c⟩≡ (215b)
```
  void
  Xfalse(void)
  {
      if(truestatus()) runq->pc = runq->code[runq->pc].i;
      else runq->pc++;
  }
```
Uses runq 34c and truestatus() 74b.

⟨outcode() *cases* 83d⟩+≡ (72e) ◁83a 83f▷
```
  case BANG:
      outcode(c0, eflag);
      emitf(Xbang);
      break;
```

⟨*function* Xbang 83e⟩≡ (215b)
```
  void
  Xbang(void)
  {
      setstatus(truestatus()? "false" : "");
  }
```
Uses setstatus() 73f and truestatus() 74b.

8.4.3 String matching

The ~ operator pushes two word lists onto argv (the subject and the patterns), then calls Xmatch[84a]. Two Xmark[73a] 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. Xmatch()[84a] converts the subject words to a single string (joining with spaces), then tries each pattern word against it using match()[134b] (described in Chapter 13).

⟨outcode() *cases* 83f⟩+≡ (72e) ◁83d 87c▷
```
  case TWIDDLE:
      emitf(Xmark);
      outcode(c1, eflag);
      emitf(Xmark);
      outcode(c0, eflag);
      emitf(Xmatch);
      if(eflag)
          emitf(Xeflag);
      break;
```

⟨*function* Xmatch 84a⟩≡ (215b)

```
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() 169d, list2str() 84b, poplist() 73b, runq 34c, and setstatus() 73f.

⟨*function* list2str 84b⟩≡ (215b)

```
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';
    else s[-1]='\0';
    return value;
}
```

Uses emalloc() 169b.

8.4.4 Redirection

Redirections in rc are handled in two phases. When the interpreter encounters a redirection bytecode (e.g., Xwrite[88a]), 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()[79a] (inside the forked child process), just before exec(): doredir()[86] 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* 84c⟩≡ (79a)

```
doredir(runq->redir);
```

Uses doredir() 86 and runq 34c.

⟨`start()` *set redir* 85a⟩≡ (44a)
```
p->redir = p->startredir = runq ? runq->redir : nil;
```
Uses `runq` 34c.

`Redir`

The `Redir` structure records a pending redirection: a `type` (e.g., `ROPEN` for a regular file redirection), a `from` file descriptor (the one that was opened), and a `to` file descriptor (the one that should be replaced, such as 1 for stdout). The thread's `redir` field points to a linked list of these records, built up as the interpreter encounters redirection bytecodes. The `startredir` field in `start()`[44a] 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.

⟨*struct* `Redir` 85b⟩≡ (211a)
```
struct Redir {
    // enum<redirection_kind_bis>
    char type; /* what to do */

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

    // Extra
    ⟨Redir extra fields 85d⟩
};
```

⟨`Thread` *other fields* 85c⟩+≡ (34b) ◁51a 90a▷
```
// list<ref_own<Redir>> (next = Redir.next)
struct Redir *redir; /* redirection stack */
```
Uses `Redir` 85b.

⟨`Redir` *extra fields* 85d⟩≡ (85b)
```
struct Redir *next;  /* what else to do (reverse order) */
```
Uses `Redir` 85b.

⟨*function* `pushredir` 85e⟩≡ (215a)
```
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` 34c.

`doredir()`

Because `pushredir()`[85e] 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()`[86] 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. Concretely, for `cmd > foo >[2=1]` the stack and recursion unroll as follows. At parse/eval time each `pushredir` prepends, so the newest redirection sits at the head. Then `doredir()` recurses to the tail before touching any descriptor, producing a call chain that applies the oldest entry first:

```
runq->redir (after both Xwrite / Xdup2 have pushed):

    head                                          tail
     |                                             |
     v                                             v
  +--------------+      +--------------+      +---+
  | >[2=1]       | ---> | > foo        | ---> |nil|
  | pushed last  |      | pushed first |      +---+
  +--------------+      +--------------+

doredir() recursion (left-to-right in source order):

  doredir(>[2=1])
    doredir(>foo)
      doredir(nil)     <- returns
      apply: dup(fd_foo, 1); close(fd_foo)
    apply: dup(1, 2)   <- now fd 2 points where foo went
```

The same recursion is what makes the `startredir` "inheritance point" work on the rewind path: `turfredir()`[90b] iterates `while(runq->redir != runq->startredir)`, popping (via `Xpopredir`[88b]) only the records pushed since this thread started, stopping exactly at the frontier it inherited from its parent.

⟨*function* `doredir` 86⟩≡ (220a)

```
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 87a⟩
        }
    }
}
```

Uses `doredir()` 86.

The `ROPEN` case is the file descriptor plumbing itself. It is worth tracing concretely what happens for `ls > out`. At parse time `Xwrite`[88a] calls `Creat("out")` and gets back some arbitrary free fd (say 3), and pushes `ROPENfrom=3, to=1`. In the child, `doredir()` then maps fd 3 onto fd 1 (stdout) via `dup`, and closes the original fd 3 so `ls` inherits a clean descriptor table:

```
before Xwrite (in parent, after fork):      after doredir (in child):
  fd 0 -> /dev/cons  (stdin)                  fd 0 -> /dev/cons
  fd 1 -> /dev/cons  (stdout)                 fd 1 -> out  *** dup'd
  fd 2 -> /dev/cons  (stderr)                 fd 2 -> /dev/cons
  fd 3 -> (free)                              fd 3 -> (closed)
```

```
after Xwrite (Creat'd "out"):
  fd 0 -> /dev/cons
  fd 1 -> /dev/cons
  fd 2 -> /dev/cons
  fd 3 -> out            Redir{from=3,to=1}
```

The `if(rp->from != rp->to)` guard handles the rare case where `Creat()` 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.

⟨`doredir()` *switch redir type cases* 87a⟩≡ (86) 148e▷

```
case ROPEN:
    if(rp->from != rp->to){
        dup(rp->from, rp->to);
        close(rp->from);
    }
    break;
```

Uses `ROPEN` 87b.

⟨*constant* `ROPEN` 87b⟩≡ (211a)

```
/*
 * redir types
 */
#define ROPEN 1   /* dup2(from, to); close(from); */
```

Bytecode generation

The bytecode for a `REDIR` node first evaluates the filename (with globbing), then emits the appropriate redirection bytecode (e.g., `Xwrite`[88a] for `>`), followed by the file descriptor number. After the redirected command (`c1`) finishes, `Xpopredir`[88b] 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* 87c⟩+≡ (72e) ◁83f 91b▷

```
case REDIR:
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xglob);

    switch(t->rtype){
    ⟨outcode() when REDIR case, switch redirection type cases 87d⟩
    }
    emiti(t->fd0);
    outcode(c1, eflag);
    emitf(Xpopredir);
    break;
```

`Xwrite()`

`Xwrite()`[88a] handles the `>` redirection at runtime. It pops the filename from the argv stack, opens (or creates) the file with `Creat()`[181f], 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()`[86].

⟨`outcode()` *when REDIR case, switch redirection type cases* 87d⟩≡ (87c) 88c▷

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

⟨*function* Xwrite 88a⟩≡ (215b)

```
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()` 181f, `ROPEN` 87b, `Xerror()` 77c, `Xerror1()` 78a, `count()` 36c, `err` 167b, `pfmt()` 178b, `poplist()` 73b, `pushredir()` 85e, and `runq` 34c.

`Xpopredir()`[88b] 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 88b⟩≡ (215a)

```
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` 87b, `Redir` 85b, `efree()` 169d, `panic()` 168c, and `runq` 34c.

`Xread()`

⟨`outcode()` *when REDIR case, switch redirection type cases* 88c⟩+≡ (87c) ◁87d 89b▷

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

⟨*function* Xread 89a⟩≡ (215b)

```
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 87b, Xerror() 77c, Xerror1() 78a, count() 36c, err 167b, pfmt() 178b, poplist() 73b, pushredir() 85e, and runq 34c.

Xappend()

⟨outcode() *when REDIR case, switch redirection type cases* 89b⟩+≡ (87c) ◁88c 145d▷

```
case APPEND:
    emitf(Xappend);
    break;
```

⟨*function* Xappend 89c⟩≡ (215b)

```
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() 181f, ROPEN 87b, Xerror() 77c, Xerror1() 78a, count() 36c, err 167b, pfmt() 178b, poplist() 73b, pushredir() 85e, and runq 34c.

Closing redirection opened files

⟨Thread *other fields* 90a⟩+≡ (34b) ◁85c 93a▷

```
struct Redir *startredir; /* redir inheritance point */
```

Uses Redir 85b.

⟨*function* turfredir 90b⟩≡ (215a)

```
void
turfredir(void)
{
    while(runq->redir != runq->startredir)
        Xpopredir();
}
```

Uses Xpopredir() 88b and runq 34c.

⟨Xreturn() *pop the redirections from this thread* 90c⟩≡ (91d)

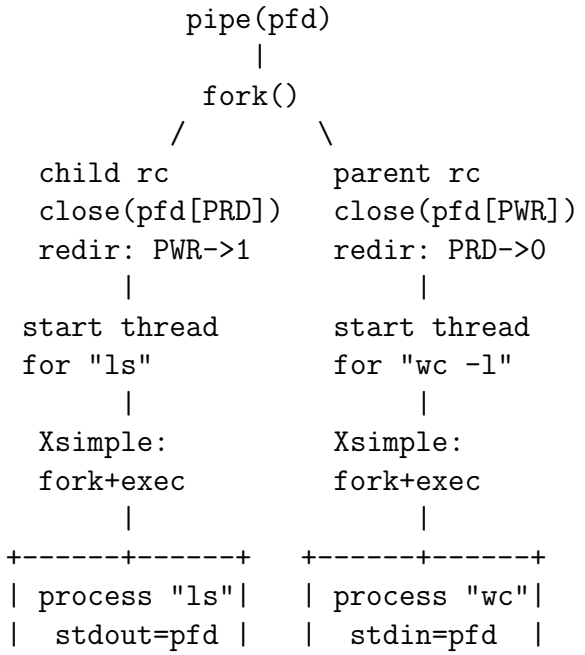
```
turfredir();
```

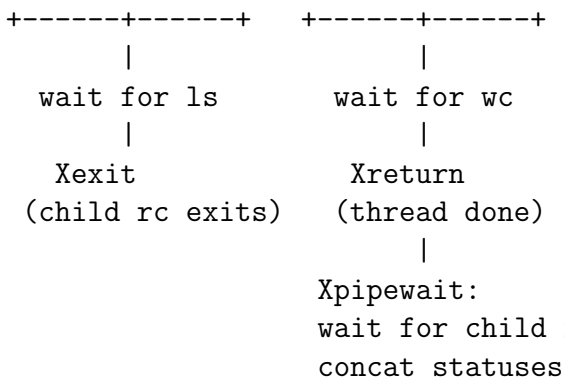
Uses turfredir() 90b.

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`[93d].

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





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"`.

⟨`outcode()` *locals* 91a⟩+≡ (72e) ◁82c 103d▷

```
int q;
```

⟨`outcode()` *cases* 91b⟩+≡ (72e) ◁87c 94a▷

```
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;
```

⟨`codefree()` *in loop over code* `cp`, *switch bytecode cases* 91c⟩+≡ (33e) ◁76e 101c▷

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

Uses `Xpipe()` 92c.

⟨*function* `Xreturn` 91d⟩≡ (215a)

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

    ⟨Xreturn() pop the redirections from this thread 90c⟩
    // 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 34b, codefree() 33e, efree() 169d, getstatus() 74a, poplist() 73b, and runq 34c.

⟨*constant* PRD 92a⟩≡ (208c)

```
/*
 * 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 92b⟩≡ (208c)

```
#define PWR 1
```

⟨*function* Xpipe 92c⟩≡ (218a)

```
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 92a, PWR 92b, ROPEN 87b, Thread 34b, Xerror() 77c, addwaitpid() 74e, clearwaitpids() 75c, pushredir() 85e, runq 34c, and start() 44a.

⟨Thread *other fields* 93a⟩+≡ (34b) ◁90a 93b▷
```
// option<int> (None = -1)
int pid;  /* process for Xpipewait to wait for */
```

⟨Thread *other fields* 93b⟩+≡ (34b) ◁93a 96e▷
```
char status[NSTATUS]; /* status for Xpipewait */
```
Uses NSTATUS 93c.

⟨*constant* NSTATUS 93c⟩≡ (211a)
```
#define NSTATUS ERRMAX   /* length of status (from plan 9) */
```

⟨*function* Xpipewait 93d⟩≡ (215b)
```
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 93c, concstatus() 93f, getstatus() 74a, runq 34c, and setstatus() 73f.

⟨Waitfor() *in while loop, if wait returns another pid* 93e⟩≡ (81a)
```
// 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 34c.

⟨*function* concstatus 93f⟩≡ (222c)
```
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 93c.

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. 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* 94a⟩+≡ (72e) ◁91b 95a▷

```
case '&':
    emitf(Xasync);
    p = emiti(0);
    outcode(c0, eflag);
    emitf(Xexit);
    stuffdot(p);
    break;
```

⟨*function* `Xasync` 94b⟩≡ (218a)

```
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, 0);
        // 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` 87b, `Xerror()` 77c, `addwaitpid()` 74e, `clearwaitpids()` 75c, `inttoascii()` 181a, `newword()` 36b, `pushredir()` 85e, `runq` 34c, `setvar()` 39a, and `start()` 44a.

⟨*function* `Xexit` 94c⟩≡ (215b)

```
void
Xexit(void)
{
    struct Var *trapreq;
    struct Word *starval;
    static bool beenhere = false;
```

```
    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;
        }
    }
    Exit(getstatus(), __LOC__);
}
```

Uses Var 38b, Word 36a, copywords() 37a, getstatus() 74a, mypid 127a, newvar() 40c, runq 34c, start() 44a, and vlook() 39b.

8.5 Control flow statements

8.5.1 if

The if compilation uses the same forward-jump technique as &&: emit the condition, then Xif[95b] with a jump offset to skip the body if the condition is false. The Xwastrue[96a] 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[96c] 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* 95a⟩+≡ (72e) ◁94a 96b▷

```
case IF:
    outcode(c0, false);
    emitf(Xif);
    p = emiti(0);
    outcode(c1, eflag);
    emitf(Xwastrue);
    stuffdot(p);
    break;
```

⟨*function* Xif 95b⟩≡ (215b)

```
void
Xif(void)
{
    ifnot = true;
    if(truestatus())
        runq->pc++;
    else
        runq->pc = runq->code[runq->pc].i;
}
```

Uses ifnot 95c, runq 34c, and truestatus() 74b.

⟨*global* ifnot 95c⟩≡ (215b)

```
bool ifnot;  /* dynamic if not flag */
```

⟨*function* `Xwastrue` 96a⟩≡ (215b)
```
void
Xwastrue(void)
{
    ifnot = false;
}
```
Uses `ifnot` 95c.

⟨`outcode()` *cases* 96b⟩+≡ (72e) ◁95a 96h▷
```
case NOT:
    ⟨outcode() when NOT, sanity check last command was an if 96d⟩
    emitf(Xifnot);
    p = emiti(0);
    outcode(c0, eflag);
    stuffdot(p);
    break;
```

⟨*function* `Xifnot` 96c⟩≡ (215b)
```
void
Xifnot(void)
{
    if(ifnot)
        runq->pc++;
    else
        runq->pc = runq->code[runq->pc].i;
}
```
Uses `ifnot` 95c and `runq` 34c.

⟨`outcode()` *when NOT, sanity check last command was an if* 96d⟩≡ (96b)
```
if(!runq->iflast)
    yyerror("‘if not’ does not follow ‘if(...)’");
```

⟨`Thread` *other fields* 96e⟩+≡ (34b) ◁93b
```
bool iflast;  /* static ‘if not’ checking */
```

⟨`outcode()` *set iflast after switch* 96f⟩≡ (72e)
```
if(t->type!=NOT && t->type!=';')
    runq->iflast = t->type==IF;
else
   if(c0)
       runq->iflast = c0->type==IF;
```

⟨`outcode()` *set iflast before switch* 96g⟩≡ (72e)
```
if(t->type!=NOT && t->type!=';')
    runq->iflast = false;
```

8.5.2 `while`

The `while` loop compiles to: the condition bytecodes, `Xtrue`[83b] (jump out if false), the body bytecodes, `Xjump`[97b] (jump back to the condition). If the condition is empty (e.g., `while()`), an `Xsettrue`[97a] bytecode is emitted to make it loop forever.

⟨`outcode()` *cases* 96h⟩+≡ (72e) ◁96b 97c▷
```
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` 97a⟩≡ (215b)

```
  void
  Xsettrue(void)
  {
      setstatus("");
  }
```

Uses `setstatus()` 73f.

⟨*function* `Xjump` 97b⟩≡ (215b)

```
  void
  Xjump(void)
  {
      runq->pc = runq->code[runq->pc].i;
  }
```

Uses `runq` 34c.

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`[104c]), and then loops with `Xfor`[98a]. `Xfor()`[98a] 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`[105a] removes the local binding.

⟨`outcode()` *cases* 97c⟩+≡ (72e) ◁96h 98b▷

```
  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;
```

⟨*function* `Xfor` 98a⟩≡ (215b)

```
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()` 37c, `poplist()` 73b, and `runq` 34c.

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`[100b], which uses `match()`[134b] (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. The comment in `codeswitch()`[98c] below shows the layout.

⟨`outcode()` *cases* 98b⟩+≡ (72e) ◁97c 100c▷

```
case SWITCH:
    codeswitch(t, eflag);
    break;
```

⟨*function* `codeswitch` 98c⟩≡ (221)

```
/*
 * 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() 100b, Xjump() 97b, Xmark() 73a, Xpopm() 99, c0-62 72b, c1-63 72c, emitf-65 71d, emiti-66 71c, iscase() 100a, stuffdot() 82e, and yyerror() 167e.

⟨*function* Xpopm 99⟩≡ (215b)

```
void
Xpopm(void)
{
    poplist();
}
```

Uses poplist() 73b.

⟨*function* iscase 100a⟩≡ (221)
```
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 c0-62 72b.

⟨*function* Xcase 100b⟩≡ (215b)
```
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() 169d, list2str() 84b, poplist() 73b, and runq 34c.

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* 100c⟩+≡ (72e) ◁98b 100d▷
```
case PAREN:
    outcode(c0, eflag);
    break;
```

⟨outcode() *cases* 100d⟩+≡ (72e) ◁100c 101b▷
```
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* 101a⟩≡ (38b) 118a▷
```
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`[102a] stores the current code vector (via `codecopy`[33d]) 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`[102b] deletes the function by setting `v->fn` to `nil`.

⟨`outcode()` *cases* 101b⟩+≡ (72e) ◁100d 104a▷
```
case FN:
    emitf(Xmark);
    outcode(c0, eflag);
    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* 101c⟩+≡ (33e) ◁91c 148a▷
```
else if(p->f==Xfn){
          efree(p[2].s);
          p+=2;
      }
```
Uses `Xfn()` 102a and `efree()` 169d.

⟨*function* `fnstr` 101d⟩≡ (221)
```
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()` 177a, `nl` 162a, `openstr()` 177b, and `pfmt()` 178b.

The key move in `Xfn()`[102a] 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 in `outcode()` and patched by `stuffdot(p)` once the body had been emitted; at run time `Xfn` uses it to *skip over* the body (`runq->pc = end`) so the definition itself is a no-op for the enclosing script. When the function is later invoked, `execfunc()`[103c] calls `start(func->fn, func->pc, ...)`, which creates a new thread that resumes right after the `Xfn` and runs until the matching `Xreturn`. Because several variables can alias the same code vector (e.g. `fn a b c { ... }` defines three functions with one body), the vector is refcounted and `codecopy`/`codefree()` bump the count on entry and drop it on redefinition.

⟨*function* `Xfn` 102a⟩≡ (215b)

```
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()` 33d, `codefree()` 33e, `globlist()` 129f, `gvlook()` 40a, `poplist()` 73b, and `runq` 34c.

⟨*function* `Xdelfn` 102b⟩≡ (215b)

```
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() 33e, gvlook() 40a, poplist() 73b, and runq 34c.

8.6.2 Function uses (<foo>(...))

When Xsimple()[77a] detects that the command name matches a function, it calls execfunc()[103c]. 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* 103a⟩≡ (77a) 109c▷

```
var *v;
```

⟨Xsimple() *if argv0 is a function* 103b⟩≡ (77a)

```
v = gvlook(a->word);
if(v->fn)
    execfunc(v);
```

Uses execfunc() 103c and gvlook() 40a.

⟨*function* execfunc 103c⟩≡ (220a)

```
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() 40c, poplist() 73b, popword() 73c, runq 34c, and start() 44a.

8.7 Variables

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[104c]/Xunlocal[105a]); when standalone (e.g., x=1), it is global (using Xassign[104b]). 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* 103d⟩+≡ (72e) ◁91a

```
tree *tt;
```

⟨outcode() *cases* 104a⟩+≡ (72e) ◁101b 105b▷

```
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 104b⟩≡ (215b)

```
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() 78a, count() 36c, deglob() 131c, freewords() 37b, globlist() 129f, poplist() 73b, runq 34c, and vlook() 39b.

⟨*function* Xlocal 104c⟩≡ (215b)

```
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() 78a, count() 36c, deglob() 131c, globlist() 129f, newvar() 40c, poplist() 73b, and runq 34c.

⟨*function* Xunlocal 105a⟩≡ (215b)

```
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() 169d, freewords() 37b, panic() 168c, runq 34c, and vlook() 39b.

8.7.2 Variable uses ($<x>)

Xdol()[105c] (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.

⟨outcode() *cases* 105b⟩+≡ (72e) ◁104a 106a▷

```
case '$': //$
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xdol);
    break;
```

⟨*function* Xdol 105c⟩≡ (215b)

```
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;
    }
```

⟨outcode() *cases* 106a⟩+≡ (72e) ◁105b 107a▷

```
    case COUNT:
        emitf(Xmark);
        outcode(c0, eflag);
        emitf(Xcount);
        break;
```

⟨*function* Xcount 106b⟩≡ (215b)

```
    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);
    }
```

Subscripting (`$x(2)` or `$x(2-5)`) extracts elements from a list variable by index or range. `subwords()`[107c] 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()`[36d] 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* 107a⟩+≡ (72e) ◁106a 108a▷
```
case SUB:
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xmark);
    outcode(c1, eflag);
    emitf(Xsub);
    break;
```

⟨*function* `Xsub` 107b⟩≡ (215b)
```
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()` 78a, `count()` 36c, `deglob()` 131c, `poplist()` 73b, `runq` 34c, and `vlook()` 39b.

⟨*function* `subwords` 107c⟩≡ (215b)
```
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);
}
```

⟨outcode() *cases* 108a⟩+≡ (72e) ◁107a 140c▷

```
case '^':
    emitf(Xmark);
    outcode(c1, eflag);
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xconc);
    break;
```

⟨*function* Xconc 108b⟩≡ (215b)

```
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() 78a, conclist() 108c, count() 36c, poplist() 73b, and runq 34c.

⟨*function* conclist 108c⟩≡ (215b)

```
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() 108c, efree() 169d, emalloc() 169b, and newword() 36b.

8.7.3 Special variables

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()`[77a] detects that the command name matches a builtin, it calls the corresponding function directly. 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.

⟨*struct* `Builtin` 109a⟩≡ (211a)
```
struct Builtin {
    char *name;
    void (*fnc)(void);
};
```

⟨*global* `builtins` 109b⟩≡ (222d)
```
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()` 110c, `execdot()` 112d, `execeval()` 114e, `execexec()` 79a, `execexit()` 112c, `execfinit()` 157b, `execflag()` 156a, `execnewpgrp()` 155, and `execwait()` 115c.

⟨`Xsimple()` *other locals* 109c⟩+≡ (77a) ◁103a
```
struct Builtin *bp;
```
Uses `Builtin` 109a.

⟨Xsimple() *if argv0 is a builtin* 110a⟩≡ (77a)
```
⟨Xsimple() if argv0 is the builtin keyword 110b⟩
for(bp = builtins;bp->name;bp++)
    if(strcmp(a->word, bp->name)==0){
        (*bp->fnc)();
        return;
    }
```
Uses builtins 109b.

⟨Xsimple() *if argv0 is the builtin keyword* 110b⟩≡ (110a)
```
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() 36c, err 167b, pfmt() 178b, poplist() 73b, popword() 73c, and setstatus() 73f.

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. Under rio, cd also writes the new directory to /dev/wdir so the window manager can track it.

⟨*function* execcd 110c⟩≡ (222d)
```
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()` 111a, `count()` 36c, `dochdir()` 111b, `err` 167b, `nullpath` 79d, `pfmt()` 178b, `poplist()` 73b, `runq` 34c, `setstatus()` 73f, and `vlook()` 39b.

⟨*function* `appfile` 111a⟩≡ (222d)

```
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()` 169b.

⟨*function* `dochdir` 111b⟩≡ (222d)

```
errorneg1
dochdir(char *word)
{
    ⟨dochdir() locals 112a⟩

    // the actual syscall
    if(chdir(word)<0)
        return ERROR_NEG1;

    ⟨dochdir() adjust /dev/wdir if run under rio 112b⟩
    return OK_1;
}
```

In Plan 9, the window manager `rio` (see WINDOWS book [Pad16c]) 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* 112a⟩≡ (111b)
```
/* report to /dev/wdir if it exists and we're interactive */
static fdt wdirfd = -2;
```

⟨`dochdir()` *adjust* `/dev/wdir` *if run under* `rio` 112b⟩≡ (111b)
```
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` 41c.

9.3 $ exit

⟨*function* `execexit` 112c⟩≡ (222d)
```
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()` 94c, `count()` 36c, `err` 167b, `pfmt()` 178b, `runq` 34c, and `setstatus()` 73f.

9.4 $.

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()`[46a] will read commands from that file instead of the terminal.

⟨*function* `execdot` 112d⟩≡ (222d)
```
void
execdot(void)
{
    thread *p = runq;
    bool iflag = false;
    fdt fd;
    list *av;
    char *zero; // new stdin
```

```
char *file;
word *path;
⟨execdot() other locals 114c⟩

⟨execdot() if first 114d⟩
⟨execdot() if not first execution 138d⟩

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 148c, Var 38b, Xerror() 77c, Xerror1() 78a, appfile() 111a, dotcmds 114b, efree() 169d, err 167b, ndot 114a, openfd() 175b, pfmt() 178b, popword() 73c, pushlist() 45a, pushredir() 85e, pushword() 45b, runq 34c, searchpath() 79c, setstatus() 73f, and start() 44a.

⟨*global* ndot 114a⟩≡ (211b)

```
    /*
     * How many dot commands have we executed?
     * Used to ensure that -v flag doesn't print rcmain.
     */
    int ndot;
```

⟨*global* dotcmds 114b⟩≡ (222d)

```
    union Code dotcmds[14];
```

Uses Code 33c.

⟨execdot() *other locals* 114c⟩≡ (112d)

```
    static bool first = true;
```

⟨execdot() *if first* 114d⟩≡ (112d)

```
    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() 104c, Xmark() 73a, Xrdcmds() 46a, Xreturn() 91d, Xunlocal() 105a, Xword() 72f, and dotcmds 114b.

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). It joins the arguments into a single string, then creates a new thread reading from that string (via opencore()[177c], which creates a buffered IO from memory).

⟨*function* execeval 114e⟩≡ (222d)

```
    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() 78a, count() 36c, eflagok 138c, efree() 169d, emalloc() 169b, execcmds() 115b, opencore() 177c, poplist() 73b, and runq 34c.

⟨*global* rdcmds 115a⟩≡ (220a)

```
union Code rdcmds[4];
```

Uses Code 33c.

⟨*function* execcmds 115b⟩≡ (220a)

```
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() 46a, Xreturn() 91d, rdcmds 115a, runq 34c, and start() 44a.

9.6 $ wait

⟨*function* execwait 115c⟩≡ (222d)

```
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()` 78a, `count()` 36c, `poplist()` 73b, and `runq` 34c.

Chapter 10

Environment

Under Plan 9, the environment is not passed via the `envp` parameter of `main()` as in UNIX. Instead, each environment variable is stored as a file under `/env/`: the variable `$path` is stored in `/env/path`. This is a more elegant design because child processes automatically inherit the environment through the namespace, and changes are visible to all processes sharing the same namespace. `rc` must synchronize its internal variable table with these files in both directions: `Updenv()`[117a] writes changed variables to `/env/`, and `Vinit()`[118c] reads variables from `/env/` at startup.

10.1 `Updenv()`

`Updenv()`[117a] 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` 117a⟩≡ (225b)
```
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` 39c, `addenv()` 118b, `gvar` 39d, `runq` 34c, and `updenvlocal()` 117b.

⟨*function* `updenvlocal` 117b⟩≡ (225b)
```
void
updenvlocal(var *v)
{
    if(v){
        updenvlocal(v->next);
        addenv(v);
    }
}
```
Uses `addenv()` 118b and `updenvlocal()` 117b.

⟨*enum* `MiscPlan9` 117c⟩≡ (225b)
```
enum {
    Maxenvname = 256, /* undocumented limit */
};
```

⟨Var *other fields* 118a⟩+≡ (38b) ◁101a
```
bool changed;
```

⟨*function* addenv 118b⟩≡ (225b)
```
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()` 181f, `Maxenvname-2` 117c, `closeio()` 177a, `err` 167b, `openfd()` 175b, and `pfmt()` 178b.

10.2 `Vinit()`

`Vinit()`[118c] 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()`[39a] for each variable. It then resets the `changed` flag so that `Updenv()`[117a] 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 118c⟩≡ (225b)
```
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` 117c, `efree()` 169d, `emalloc()` 169b, `err` 167b, `newword()` 36b, `pfmt()` 178b, `setvar()` 39a, and `vlook()` 39b.

⟨*global* `envdir` 119⟩≡ (215b)

```
fdt envdir;
```

Chapter 11

Signals

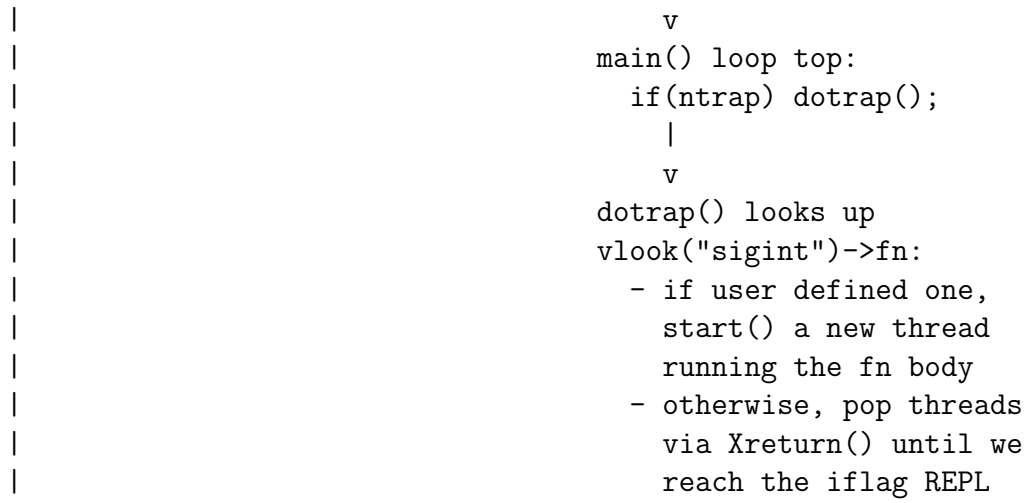
Signal handling is critical for a shell because the shell sits between the user and every running program. When the user presses `^C`, the interrupt signal is sent to all processes in the foreground group—including the shell itself. If the shell simply died along with the interrupted command, the user 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 Kernel book [Pad14] for how the kernel delivers notes to processes.

11.1 Overview

Under Plan 9, signals are called *notes*. When a note is received (e.g., an interrupt from `^C`), the kernel calls the handler registered by `Trapinit()`[122c]. 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`) and the main interpreter loop checks this counter after each bytecode (see the `handing trap if necessary` chunk in `main()X`). 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:

```
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
  |                                       |
```



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`[104b] 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`[123a] 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()`[46a]. 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` 121a⟩≡ (208c)
```
#define NSIG 32
```

⟨*constant* `SIGINT` 121b⟩≡ (208c)
```
#define SIGINT 2
```

⟨*constant* `SIGQUIT` 121c⟩≡ (208c)
```
#define SIGQUIT 3
```

⟨*global* `syssigname` 121d⟩≡ (225b)
```
char *syssigname[] = {
    "exit",  /* can't happen */
    "hangup",
    "interrupt",
    "quit",  /* can't happen */
    "alarm",
    "kill",
    "sys: fp: ",
    "term",
    0
};
```

⟨*global* `signame` 121e⟩≡ (225b)
```
char *signame[] = {
    "sigexit",
    "sighup",
    "sigint",
    "sigquit",
    "sigalrm",
    "sigkill",
    "sigfpe",
```

```
    "sigterm",
    0
};
```

⟨*global* `ntrap` 122a⟩≡ (219b)
```
int ntrap;    /* number of outstanding traps */
```

⟨*global* `trap` 122b⟩≡ (219b)
```
int trap[NSIG];    /* number of outstanding traps per type */
```

11.2 Registering handlers: `Trapinit()`

⟨*function* `Trapinit` 122c⟩≡ (225b)
```
void
Trapinit(void)
{
    notify(notifyf);
}
```
Uses `notifyf()` 122d.

⟨*function* `notifyf` 122d⟩≡ (225b)
```
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` 167b, `interrupted` 123b, `ntrap` 122a, `pfmt()` 178b, `syssigname` 121d, and `trap` 122b.

11.3 Trap dispatch: `dotrap()`

⟨`main()` *handing trap if necessary in interpreter loop* 122e⟩≡ (45c)
```
if(ntrap)
    dotrap();
```
Uses `dotrap()` 123a and `ntrap` 122a.

⟨*function* `dotrap` 123a⟩≡ (219b)

```
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__);
        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;
        }
        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` 38b, `Word` 36a, `Xreturn()` 91d, `copywords()` 37a, `getstatus()` 74a, `mypid` 127a, `newvar()` 40c, `ntrap` 122a, `runq` 34c, `start()` 44a, `trap` 122b, and `vlook()` 39b.

⟨*global* `interrupted` 123b⟩≡ (225b)

```
bool interrupted = false;
```

Uses `interrupted` 123b.

⟨*function* `Eintr` 123c⟩≡ (225b)

```
bool
Eintr(void)
{
    return interrupted;
}
```

Uses `interrupted` 123b.

⟨`Xrdcmds()` *calls Noerror() before yyparse()* 123d⟩≡ (46a)

```
Noerror();
```

⟨*function* `Noerror` 123e⟩≡ (225b)

```
void
Noerror(void)
```

```
{
    interrupted = false;
}
```

Uses `interrupted` 123b.

⟨`Xrdcmds()` *reset ntrap* 124⟩≡ (46a)

```
ntrap = 0; /* avoid double-interrupts during blocked writes */
```

Uses `ntrap` 122a.

Chapter 12

Initialization

In Chapter 4, we showed a simplified version of `main()` 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.

12.1 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, which is the source of the famous "my `PATH` works in xterm but not in tmux" class of bug. `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` (system-wide, sourced by the bootstrap bytecodes shown below). For login shells, `rcmain` then sources `$home/lib/prc` for user customization. That is the entire ladder—two files instead of six, with the same logic regardless of how the shell was started. The price is that there is no separate "interactive only" file: if you want a setting in interactive shells only, you test `$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.

12.2 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. 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`.

⟨main() *initialize* boostrap 126a⟩≡ (41b)

```
memset(bootstrap, 0, sizeof bootstrap);

i = 0;
bootstrap[i++].i=1;
// runq->argv is populated with the arguments to rc
// we just need to add '*=(argv)'
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()` 104b, `Xexit()` 94c, `Xmark()` 73a, `Xsimple()` 77a, and `Xword()` 72f.

12.3 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.

⟨*global* Rcmain 126b⟩≡ (225b)

```
char *Rcmain = "/rc/lib/rcmain";
```

⟨main() *locals* 126c⟩+≡ (41b) ◁?? 127b▷

```
char *rcmain;
```

⟨main() *initialisation* 126d⟩+≡ (41b) ◁43b 127c▷

```
rcmain = flag['m'] ? flag['m'][0] : Rcmain;
```

Uses `flag` 41c.

12.4 Actual environment

Before entering the interpreter loop, `rc` also sets up a few built-in variables: `$pid` (the shell's own process id, useful for creating unique temporary filenames) and `$rcname` (the name `rc` was invoked as, i.e., `argv[0]`).

⟨*global* `mypid` 127a⟩≡ (211b)
```
int mypid;
```

⟨`main()` *locals* 127b⟩+≡ (41b) ◁126c
```
char num[12];
```

⟨`main()` *initialisation* 127c⟩+≡ (41b) ◁126d 127d▷
```
mypid = getpid();
inttoascii(num, mypid);
setvar("pid", newword(num, (word *)nil));
```
Uses `inttoascii()` 181a, `mypid` 127a, `newword()` 36b, and `setvar()` 39a.

⟨`main()` *initialisation* 127d⟩+≡ (41b) ◁127c 138a▷
```
setvar("rcname", newword(argv[0], (word *)nil));
```
Uses `newword()` 36b and `setvar()` 39a.

Chapter 13

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. In the original Unix shell, globbing was actually handled by a separate external program called `glob`. In MS-DOS, the opposite approach was taken: wildcards are passed unexpanded to each program, which must interpret them itself. This is generally worse (every program must reimplement the logic), but it does allow commands like `ren *.txt *.bak` to have a meaning—the program sees both patterns and can pair them up, something that is impossible when the shell expands both arguments independently.

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`[130b] instructions at the right points. And at runtime, the actual matching is done by `globlist()`[129f], which walks directories and filters filenames against the pattern using `matchfn()`[134a].

13.1 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()`[131c] 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` with a `word` field of length 4, 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` must skip over the marker, and `deglob` must strip it once matching is done.

⟨*constant* `GLOB` 129a⟩≡ (208c)
```
/*
 * 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* 129b⟩≡ (59c)
```
if(c=='*' || c=='[' || c=='?' || c==GLOB)
    w = addtok(w, GLOB);
```
Uses `GLOB` 129a and `addtok()` 129c.

⟨*function* `addtok` 129c⟩≡ (219a)
```
char*
addtok(char *p, int val)
{
    if(p==nil)
        return nil;
    if(p >= &tok[NTOK]){
        *p = '\0';
        yyerror("token buffer too short");
        return nil;
    }
    *p++=val;
    return p;
}
```
Uses `NTOK-10` 54d, `tok` 54c, and `yyerror()` 167e.

13.2 Expanding globbing characters

The expansion happens at runtime, not at compile time. The `Xglob`[130b] 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.

⟨*global* `globv` 129d⟩≡ (214c)
```
struct Word *globv;
```
Uses `Word` 36a.

⟨`Xsimple()` *initializations,* `globlist()` 129e⟩≡ (77a)
```
globlist();
```
Uses `globlist()` 129f.

⟨*function* `globlist` 129f⟩≡ (214c)
```
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() 130a, globv 129d, poplist() 73b, pushlist() 45a, and runq 34c.

⟨*function* globlist1 130a⟩≡ (214c)

```
void
globlist1(word *gl)
{
    if(gl){
        globlist1(gl->next);
        glob(gl->word);
    }
}
```

Uses glob() 131b and globlist1() 130a.

⟨*function* Xglob 130b⟩≡ (215b)

```
void
Xglob(void)
{
    globlist();
}
```

Uses globlist() 129f.

13.3 glob()

glob()[131b] takes a pattern string and pushes all matching filenames onto globv. If no files match, the original pattern (with GLOB markers removed) is kept as-is. globdir()[132c] 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()[132b].

Walking through ls src/*.c makes the recursion concrete. After lexing, the argument is the byte string src/\001*.c. globdir[132c] 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` 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.

⟨*global* `globname` 131a⟩≡ (214c)
```
char *globname;
```

⟨*function* `glob` 131b⟩≡ (214c)
```
/*
 * 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;
    }
    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()` 131c, `efree()` 169d, `emalloc()` 169b, `globdir()` 132c, `globname` 131a, `globsort()` 132b, `globv` 129d, and `newword()` 36b.

⟨*function* `deglob` 131c⟩≡ (214c)
```
/*
 * 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 129a.

⟨*function* globcmp 132a⟩≡ (214c)

```
int
globcmp(const void *s, const void *t)
{
    return strcmp(*(char**)s, *(char**)t);
}
```

⟨*function* globsort 132b⟩≡ (214c)

```
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() 169d, emalloc() 169b, and globcmp() 132a.

⟨*function* globdir 132c⟩≡ (214c)

```
/*
 * 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;
    int 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 129a, Opendir() 136d, globdir() 132c, globname 131a, globv 129d, matchfn() 134a, and newword() 36b.

⟨*function* equtf 133a⟩≡ (214c)

```
/*
 * 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 133b⟩≡ (214c)

```
/*
 * 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 133c⟩≡ (214c)

```
/*
 * Convert the utf code at *p to a unicode value
 */
int
unicode(uchar *p)
{
    Rune r;

    chartorune(&r, (char*)p);
    return r;
}
```

13.4 match()

`matchfn()`[134a] is the entry point called from `glob()`[131b]. 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.

The actual matching is done by `match()`[134b], 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()`[132c], so `match()` stops at slashes rather than consuming them.

⟨*function* `matchfn` 134a⟩≡ (214c)

```
/*
 * 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, '/');
}
```

⟨*function* `match` 134b⟩≡ (214c)

```
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 135a⟩≡ (225b)

```
#define NDIR 256  /* shoud be a better way */
```

⟨*function* Globsize 135b⟩≡ (225b)

```
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 129a and NDIR-3 135a.

⟨*constant* NFD 136a⟩≡ (225b)
```
#define NFD 50
```

⟨*struct* DirEntryWrapper 136b⟩≡ (225b)
```
struct DirEntryWrapper {
    Dir *dbuf;
    int i;
    int n;
};
```

⟨*global* dir 136c⟩≡ (225b)
```
struct DirEntryWrapper dir[NFD];
```
Uses DirEntryWrapper 136b and NFD-4 136a.

⟨*function* Opendir 136d⟩≡ (225b)
```
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 136a and dir 136c.

⟨*function* trimdirs 136e⟩≡ (225b)
```
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 136f⟩≡ (225b)
```
/*
 * 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 136a, dir 136c, and trimdirs() 136e.

⟨*function* Closedir 137⟩≡ (225b)

```
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 136a and dir 136c.

Chapter 14

Advanced Topics

In this chapter I present features 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, general redirections—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. 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 Build book) 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* 138a⟩+≡ (41b) ◁127d
```
setvar("cflag", flag['c']? newword(flag['c'][0], (word *)nil) : (word *)nil);
```
Uses `flag` 41c, `newword()` 36b, and `setvar()` 39a.

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 for build systems: `mk` (see the Build book) 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.

Note that `-e` is a command-line flag, not a `set` option, so you cannot enable it from within a script. 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* 138b⟩≡ (76a)
```
if(eflag)
    emitf(Xeflag);
```

⟨*global* `eflagok` 138c⟩≡ (211b)
```
bool eflagok; /* kludge flag so that -e doesn't exit in startup */
```

⟨`execdot()` *if not first execution* 138d⟩≡ (112d)
```
else
    eflagok = true;
```
Uses `eflagok` 138c.

⟨*function* Xeflag 139a⟩≡ (215b)

```
void
Xeflag(void)
{
    if(eflagok && !truestatus())
        Xexit();
}
```

Uses Xexit() 94c, eflagok 138c, and truestatus() 74b.

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.

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()[139b] does exactly this during lexing: after addtok stores the first byte, it reads continuation bytes by checking the leading-bit pattern.

⟨*function* addutf 139b⟩≡ (219a)

```
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;

    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() 129c and advance() 52c.

14.4 Advanced constructs

14.4.1 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. Braces {...} group commands but execute them in the current process—so {cd /tmp; ls} would change the parent's directory. With @{cd /tmp; ls}, the directory change is confined to the subshell.

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* 140a⟩+≡ (62a) ◁68c 147b▷
```
%token SUBSHELL /** @ */
```

⟨*cmd rule other cases* 140b⟩+≡ (64b) ◁69b
```
|   SUBSHELL cmd        {$$=mung1($1, $2);}
```

⟨`outcode()` *cases* 140c⟩+≡ (72e) ◁108a 147e▷
```
case SUBSHELL:
    emitf(Xsubshell);
    p = emiti(0);
    outcode(c0, eflag);
    emitf(Xexit);
    stuffdot(p);
    if(eflag)
        emitf(Xeflag);
    break;
```

⟨*function* `Xsubshell` 140d⟩≡ (218a)
```
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()` 77c, `addwaitpid()` 74e, `clearwaitpids()` 75c, `runq` 34c, and `start()` 44a.

14.4.2 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. 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. A here document feeds the data as a raw byte stream to stdin, which is exactly what programs like `grep` or `sed` expect.

The implementation is clever: during parsing, `heredoc()` creates a temporary file (in `/tmp`) and records the tag and filename in a linked list. After the entire command line is parsed and compiled, `readhere()` goes back and reads the here-document bodies from the input, writing them to those temporary files. This two-phase approach is necessary because a command line can have multiple here documents (e.g., `cmd <<A <<B`), and the bodies appear after the full command line.

If the here-tag was not quoted, variable substitution is performed via `psubst()`[144b]; otherwise the text is copied literally. The temporary files are cleaned up by emitting `Xdelhere` bytecodes that `remove()` them after use.

⟨*constant* `HERE` 141a⟩≡ (208c)
```
#define HERE 4
```

⟨`yylex()` *in switch when redirection character, if here document* 141b⟩≡ (58a)
```
if(nextis(c)){
    t->rtype = HERE;
    *w++=c;
}
```
Uses `HERE` 141a and `nextis()` 52d.

⟨*struct* `Here` 141c⟩≡ (208c)
```
struct Here {
    tree *tag;
    char *name;
    struct Here *next;
};
```
Uses `Here` 141c.

⟨*global* `here` 141d⟩≡ (220c)
```
struct Here *here;
```
Uses `Here` 141c.

⟨*global* `ehere` 141e⟩≡ (220c)
```
struct Here **ehere;
```
Uses `Here` 141c.

⟨*constant* `NLINE` 141f⟩≡ (220c)
```
/*
 * 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* 141g⟩≡ (71g)
```
readhere();
```
Uses `readhere()` 142a.

⟨*function* `readhere` 142a⟩≡ (220c)

```
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()` 181f, `EOF` 49a, `Here` 141c, `NLINE-9` 141f, `cleanhere()` 142b, `closeio()` 177a, `doprompt` 50c, `efree()` 169d, `flush()` 175d, `here` 141d, `openfd()` 175b, `pchr()` 176a, `pprompt()` 50h, `pstr()` 179a, `rchr()` 176c, `runq` 34c, and `yyerror()` 167e.

⟨*function* `cleanhere` 142b⟩≡ (221)

```
void
cleanhere(char *f)
{
    emitf(Xdelhere);
    emits(strdup(f));
}
```

Uses `Xdelhere()` 142c, `emitf-65` 71d, and `emits-67` 71e.

⟨*function* `Xdelhere` 142c⟩≡ (215b)

```
void
Xdelhere(void)
{
    remove(runq->code[runq->pc++].s);
}
```

Uses runq 34c.

⟨*function* Unlink 143a⟩≡ (222b)

⟨*global* ser 143b⟩≡ (220c)

```
int ser = 0;
```

Uses ser 143b.

⟨*global tmp (rc/here.c)* 143c⟩≡ (220c)

```
char tmp[] = "/tmp/here0000.0000";
```

Uses tmp 143c.

⟨*global* hex 143d⟩≡ (220c)

```
char hex[] = "0123456789abcdef";
```

Uses hex 143d.

⟨*function* hexnum 143e⟩≡ (220c)

```
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 143d.

`heredoc()`[144a] 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
  +--------+
```

After `compile()`[71g] finishes emitting bytecode for the whole line, `readhere()`[142a] 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`[89a] of the temp file, and a matching `Xdelhere` bytecode deletes it afterwards.

⟨*function* `heredoc` 144a⟩≡ (220c)
```
//@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` 141c, `WORD`, `ehere` 141e, `here` 141d, `hexnum()` 143e, `ser` 143b, `tmp` 143c, `token()` 59b, and `yyerror()` 167e.

⟨*function* `psubst` 144b⟩≡ (220c)
```
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` 145a⟩≡ (220c)

```
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()` 176a and `pstr()` 179a.

14.4.3 Read-write redirections: <> <file>

The <> operator opens a file for both reading and writing (`ORDWR`), useful for in-place modification of files.

⟨*constant* `RDWR` 145b⟩≡ (208c)

```
#define RDWR 7
```

⟨`yylex()` *in switch when redirection character, if read/write redirect* 145c⟩≡ (58a)

```
else if (nextis('>')){
    t->rtype = RDWR;
    *w++=c;
}
```

Uses `RDWR` 145b and `nextis()` 52d.

⟨`outcode()` *when REDIR case, switch redirection type cases* 145d⟩+≡ (87c) ◁89b

```
case RDWR:
    emitf(Xrdwr);
    break;
```

⟨*function* `Xrdwr` 145e⟩≡ (215b)

```
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 87b, Xerror() 77c, Xerror1() 78a, count() 36c, err 167b, pfmt() 178b, poplist() 73b, pushredir() 85e, and runq 34c.

14.4.4 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* 146⟩≡ (56g)

```
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 57d, CLOSE 147a, DUP, DUPFD 147d, EOF 49a, HERE 141a, PIPE, REDIR, advance() 52c, nextis() 52d, and yyerror() 167e.

⟨*constant* CLOSE 147a⟩≡ (208c)

```
#define CLOSE 6
```

14.4.5 Advanced dup: >[<fd0>=<fd1>], <>[<fd0>=<fd1>] , <[<fd0>=<fd1>]

The DUP construct performs dup2 between file descriptors. >[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 dup2 happens in doredir() when a child process is forked.

⟨*token declarations* 147b⟩+≡ (62a) ◁140a 148h▷

```
%token DUP
```

⟨*redir rule other cases* 147c⟩≡ (67d)

```
|   DUP
```

⟨*constant* DUPFD 147d⟩≡ (208c)

```
#define DUPFD 5
```

⟨outcode() *cases* 147e⟩+≡ (72e) ◁140c 149a▷

```
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 147f⟩≡ (215b)

```
void
Xdup(void)
{
    pushredir(RDUP, runq->code[runq->pc].i, runq->code[runq->pc+1].i);
    runq->pc+=2;
}
```

Uses RDUP 148b, pushredir() 85e, and runq 34c.

⟨codefree() *in loop over code* cp*, switch bytecode cases* 148a⟩+≡ (33e) ◁101c
```
else if(p->f==Xdup || p->f==Xpipefd)
        p+=2;
```
Uses Xdup() 147f and Xpipefd() 149c.

⟨*constant* RDUP 148b⟩≡ (211a)
```
#define RDUP 2   /* dup2(from, to); */
```

⟨*constant* RCLOSE 148c⟩≡ (211a)
```
#define RCLOSE 3   /* close(from); */
```

⟨*function* Xclose 148d⟩≡ (215b)
```
void
Xclose(void)
{
    pushredir(RCLOSE, runq->code[runq->pc].i, 0);
    runq->pc++;
}
```
Uses RCLOSE 148c, pushredir() 85e, and runq 34c.

⟨doredir() *switch redir type cases* 148e⟩+≡ (86) ◁87a 148f▷
```
case RDUP:
    dup(rp->from, rp->to);
    break;
```
Uses RDUP 148b.

⟨doredir() *switch redir type cases* 148f⟩+≡ (86) ◁148e
```
case RCLOSE:
    close(rp->from);
    break;
```
Uses RCLOSE 148c.

14.4.6 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.

14.4.7 Command output as a file: '<{<cmd>}'

⟨*comword rule other cases* 148g⟩+≡ (64e) ◁69e 150a▷
```
|   REDIR brace     {$$=mung1($1, $2); $$->type=PIPEFD;}
```

Process substitution (called "command output as a file" in Plan 9) 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 appears in the comword rule 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* 148h⟩+≡ (62a) ◁147b
```
%token PIPEFD
```

⟨outcode() *cases* 149a⟩+≡ (72e) ◁147e 150b▷

```
case PIPEFD:
    emitf(Xpipefd);
    emiti(t->rtype);
    p = emiti(0);
    outcode(c0, eflag);
    emitf(Xexit);
    stuffdot(p);
    break;
```

⟨*global* Fdprefix 149b⟩≡ (225b)

```
char *Fdprefix = "/fd/";
```

⟨*function* Xpipefd 149c⟩≡ (218a)

```
void
Xpipefd(void)
{
    struct Thread *p = runq;
    int pc = p->pc, pid;
    char name[40];
    int pfd[2];
    int 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 92a, PWR 92b, READ 57c, ROPEN 87b, Thread 34b, Xerror() 77c, addwaitpid() 74e, clearwaitpids() 75c, inttoascii() 181a, pushredir() 85e, pushword() 45b, runq 34c, and start() 44a.

14.4.8 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.

The output is split into words using $ifs (inter-field separators, defaulting to space, tab, newline). 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* 150a⟩+≡ (64e) ◁148g 151a▷
```
|   ’‘’ brace       {$$=tree1(’‘’, $2);}
```

⟨outcode() *cases* 150b⟩+≡ (72e) ◁149a 151b▷
```
case ’‘’:
    emitf(Xbackq);
    p = emiti(0);
    outcode(c0, false);
    emitf(Xexit);
    stuffdot(p);
    break;
```

⟨*function* Xbackq 150c⟩≡ (218a)
```
/*
 * Who should wait for the exit from the fork?
 */
void
Xbackq(void)
{
    int n, pid;
    int pfd[2];
    char *stop;
    char utf[UTFmax+1];
    struct Io *f;
    var *ifs = vlook("ifs");
    word *v, *nextv;
    Rune r;
    //String *word;

    stop = ifs->val? ifs->val->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
        Exit("TODO", __LOC__);
        //pad: commented because use the string.h s_xxx library
        // that I didn’t want to port to goken
        //addwaitpid(pid);
        //close(pfd[PWR]);
```

```
        //f = openfd(pfd[PRD]);
        //word = s_new();
        //v = nil;
        ///* 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);
        //        }
        //}
        //if(s_len(word) > 0)
        //    v = newword(s_to_c(word), v);
        //s_free(word);
        //closeio(f);
        //Waitfor(pid, false);
        ///* 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` 175a, `PRD` 92a, `PWR` 92b, `ROPEN` 87b, `Xerror()` 77c, `clearwaitpids()` 75c, `pushredir()` 85e, `runq` 34c, `start()` 44a, and `vlook()` 39b.

14.4.9 Stringification of variables: `$"<foo>`

The `$"` operator converts a list variable into a single string with spaces between elements. If `$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 lists back into strings.

⟨*comword rule other cases* 151a⟩+≡ (64e) ◁150a

```
|   '"' word        {$$=tree1('"', $2);}
```

⟨`outcode()` *cases* 151b⟩+≡ (72e) ◁150b

```
case '"':
    emitf(Xmark);
    outcode(c0, eflag);
    emitf(Xqdol);
    break;
```

⟨*function* `Xqdol` 151c⟩≡ (215b)

```
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, " ");
            strcat(s, p->word);
        }
    }
    else
        s[0]='\0';
    pushword(s);
    efree(s);
}
```

Uses `Xerror1()` 78a, `count()` 36c, `deglob()` 131c, `efree()` 169d, `emalloc()` 169b, `poplist()` 73b, `pushword()` 45b, `runq` 34c, and `vlook()` 39b.

14.5 Advanced builtins

These builtins could in principle be external programs (like `echo` and `test` are), but they need access to shell internals: `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). It is called from `Xsimple()`[77a] when the first word of the command is `exec`.

14.5.2 `$ whatis`

`execwhatis()`[152] 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 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`.

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* `execwhatis` 152⟩≡ (222d)

```
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);
}
```

⟨*function* `mapfd` 154a⟩≡ (222d)

```
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` 148c, `RDUP` 148b, `ROPEN` 87b, and `runq` 34c.

⟨*function* `Executable` 154b⟩≡ (222d)

```
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. 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` 155⟩≡ (222d)

```
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()` 36c, `err` 167b, `pfmt()` 178b, `poplist()` 73b, `runq` 34c, and `setstatus()` 73f.

14.5.4 `$ flag`

The `flag` builtin queries and sets `rc`'s own command-line flags at runtime. `flag e +` enables `-e` (exit on error), `flag e -` disables it, and `flag e` alone tests whether it is set (via the exit status).

⟨*function* execflag 156a⟩≡ (222d)

```
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()` 78a, `count()` 36c, `flag` 41c, `flagset` 42a, `poplist()` 73b, `runq` 34c, and `setstatus()` 73f.

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 156b⟩≡ (222d)

```
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();
}
```

14.5.6 $ 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` 157a⟩≡ (222d)
```
union Code rdfns[4];
```
Uses `Code` 33c.

⟨*function* `execfinit` 157b⟩≡ (222d)
```
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` 157c⟩≡ (225b)
```
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` 117c, `Xreturn()` 91d, `envdir` 119, `execcmds()` 115b, and `openfd()` 175b.

Chapter 15

Conclusion

You now know how the Plan 9 shell `rc` works—from the lexer that splits input into tokens, through the Yacc parser that builds an AST, the bytecode compiler that emits function pointers, to the interpreter that executes them via the `fork/exec/wait` system call pattern—and more generally how many shells work.

Despite being only about 6700 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:

- *Compilation pipeline*: the lexer → parser → bytecode compiler → interpreter architecture is the same one used by Python, Lua, and most scripting languages. Any time you process a structured input (config files, query languages, templates), this four-stage pipeline is the standard approach.
- *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. Lisp does the same with S-expressions, Tcl with strings. A single universal type eliminates conversion boilerplate and makes composition trivial.

15.2 Connections to other books

A shell sits at the intersection of many topics covered in other books of Principia Softwarica:

- 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, and how signals (notes) are delivered.

- The COMPILER book [Pad16b] uses the same compilation pipeline as `rc`—lexing, parsing, bytecode 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 [Pad16a] 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.

15.3 Missing features

`rc` is deliberately minimal. Many features that other shells provide are instead handled by 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.
- *Filename completion:* Provided by `rio`'s terminal, not by `rc` itself.
- *Job control (`bg`, `fg`, `C-z`):* Less necessary when you have multiple windows. The `&` operator and `wait $apid` cover most use cases.
- *Aliases:* Not needed because `fn` (functions) serve the same purpose and are stored persistently in the environment.

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.
- *String manipulation*: `bash` and `zsh` provide parameter expansion operators for substring extraction (`${var:offs`
pattern replacement (`${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.
- *Process substitution*: `bash`'s `diff <(cmd1) <(cmd2)` uses `/dev/fd` to present command output as file arguments. `rc` has the backquote operator `` `{cmd} `` for command substitution into strings, and in Plan 9, the `<cmd` syntax provides equivalent functionality through the `/fd` file system.

- *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).

- *Error handling*: `rc`'s error model is straightforward: `$status` and `if not`. `bash` adds `set -e` (exit on error) and `set -o pipefail` (propagate pipe failures). The Oil shell (Oils) goes further with more principled error handling designed to avoid the well-known pitfalls of `set -e`.

- *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 6700 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` 162a⟩≡ (220b)
```
char nl='\n';  /* change to semicolon for bourne-proofing */
```
Uses `nl` 162a.

⟨*constant* `c0` 162b⟩≡ (220b)
```
#define c0 t->child[0]
```

⟨*constant* `c1` 162c⟩≡ (220b)
```
#define c1 t->child[1]
```

⟨*constant* `c2` 162d⟩≡ (220b)
```
#define c2 t->child[2]
```

⟨*function* `pcmd` 162e⟩≡ (220b)
```
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, "$\"%t", c0); break;
    case '&': pfmt(f, "%t&", c0); break;
    case '^': pfmt(f, "%t^%t", c0, c1); break;
    case '`': pfmt(f, "`%t", c0); break;
    case ANDAND:pfmt(f, "%t && %t", c0, c1); break;
    case OROR: pfmt(f, "%t || %t", c0, c1); break;
    case BANG: pfmt(f, "! %t", c0); break;
    case BRACE: pfmt(f, "{%t}", c0); break;
    case COUNT: pfmt(f, "$#%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 57d, ARGLIST, BANG, BRACE, COUNT, DUP, DUPFD 147d, FN, FOR, HERE 141a, IF, NOT, ORoR, PAREN, PCMD, PIPE, PIPEFD, RDWR 145b, READ 57c, REDIR, SIMPLE, SUB, SUBSHELL, SWITCH, TWIDDLE, WHILE, WORD, WORDS, WRITE 57b, c0-5 162b, c1-6 162c, c2-7 162d, nl 162a, pchr() 176a, pdeglob() 164a, and pfmt() 178b.

⟨*function* pdeglob 164a⟩≡ (220b)

```
void
pdeglob(io *f, char *s)
{
    while(*s){
        if(*s==GLOB)
            s++;
        pchr(f, *s++);
    }
}
```

Uses GLOB 129a and pchr() 176a.

A.2 Bytecode generator trace: `rc -r`

⟨main() *debug runq in interpreter loop* 164b⟩≡ (45c)

```
if(flag['r'])
    pfnc(err, runq);
```

Uses err 167b, flag 41c, pfnc() 165b, and runq 34c.

⟨*global* `fname` 165a⟩≡ (222a)

```
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()` 89c, `Xassign()` 104b, `Xasync()` 94b, `Xbackq()` 150c, `Xbang()` 83e, `Xcase()` 100b, `Xclose()` 148d, `Xconc()` 108b, `Xdelfn()` 102b, `Xdelhere()` 142c, `Xdup()` 147f, `Xeflag()` 139a, `Xerror()` 77c, `Xexit()` 94c, `Xfalse()` 83c, `Xfn()` 102a, `Xfor()` 98a, `Xglob()` 130b, `Xif()` 95b, `Xifnot()` 96c, `Xjump()` 97b, `Xlocal()` 104c, `Xmark()` 73a, `Xmatch()` 84a, `Xpipe()` 92c, `Xpipefd()` 149c, `Xpipewait()` 93d, `Xpopm()` 99, `Xpopredir()` 88b, `Xqdol()` 151c, `Xrdcmds()` 46a, `Xrdfn()` 157c, `Xrdwr()` 145e, `Xread()` 89a, `Xreturn()` 91d, `Xsimple()` 77a, `Xsub()` 107b, `Xsubshell()` 140d, `Xtrue()` 83b, `Xunlocal()` 105a, `Xwastrue()` 96a, `Xword()` 72f, `Xwrite()` 88a, and `__anon_struct_4` 165a.

⟨*function* `pfnc` 165b⟩≡ (222a)

```
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()` 175d, `fname` 165a, `pchr()` 176a, `pfmt()` 178b, and `pstr()` 179a.

A.3 Printing subprocesses status: `rc -s`

⟨`Xrdcmds()` *print status if -s* 166a⟩≡ (46a)

```
if(flag['s'] && !truestatus())
    pfmt(err, "status=%v\n", vlook("status")->val);
```

Uses `err` 167b, `flag` 41c, `pfmt()` 178b, `truestatus()` 74b, and `vlook()` 39b.

A.4 Printing commands: `rc -x`

⟨`Xsimple()` *if -x* 166b⟩≡ (77a)

```
if(flag['x'])
    pfmt(err, "%v\n", p->argv->words); /* wrong, should do redirs */
```

Uses `err` 167b, `flag` 41c, and `pfmt()` 178b.

A.5 Printing characters: `rc -v`

⟨`getnext()` *if not EOF but verbose mode, print character read* 166c⟩≡ (48)

```
else
    if(flag['V'] || ndot>=2 && flag['v'])
        pchr(err, c);
```

Uses `err` 167b, `flag` 41c, `ndot` 114a, and `pchr()` 176a.

A.6 Printing all characters: `rc -V`

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` 167a⟩≡ (208c)
```
#define ERRMAX 128
```

⟨*global* `err` 167b⟩≡ (224c)
```
io *err;
```

⟨*global* `nerror` 167c⟩≡ (224c)
```
int nerror;  /* number of errors encountered during compilation */
```

⟨`Xrdcmds()` *flush errors and reset error count* 167d⟩≡ (46a)
```
flush(err);
nerror = 0;
```
Uses `err` 167b, `flush()` 175d, and `nerror` 167c.

⟨*function* `yyerror` 167e⟩≡ (219a)
```
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` 49a, `advance()` 52c, `err` 167b, `flush()` 175d, `lastc` 168a, `lastdol` 56b, `lastword` 58e, `nerror` 167c, `newword()` 36b, `pfmt()` 178b, `runq` 34c, `setvar()` 39a, and `tok` 54c.

⟨*global* `lastc` 168a⟩≡ (213)
```
int lastc;
```

⟨`advance()` *save future in lastc* 168b⟩≡ (52c)
```
lastc = future;
```
Uses `future` 52a and `lastc` 168a.

⟨*function* `panic` 168c⟩≡ (224c)
```
void
panic(char *s, int n)
{
    pfmt(err, "rc: ");
    pfmt(err, s, n);
    pchr(err, '\n');
    flush(err);
    Abort();
}
```
Uses `Abort()` 168d, `err` 167b, `flush()` 175d, `pchr()` 176a, and `pfmt()` 178b.

⟨*function* `Abort` 168d⟩≡ (224c)
```
void
Abort(void)
{
    pfmt(err, "aborting\n");
    flush(err);
    Exit("aborting", __LOC__);
}
```
Uses `err` 167b, `flush()` 175d, and `pfmt()` 178b.

⟨*function* `Exit` 168e⟩≡ (225b)
```
void
Exit(char *stat, char* _LOC)
{
    USED(_LOC);
    Updenv();
    setstatus(stat);
    exits(truestatus() ? "" : getstatus());
}
```
Uses `getstatus()` 74a, `setstatus()` 73f, and `truestatus()` 74b.

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.

⟨*function* `_efgfmt` 169a⟩≡ (225b)
```
/* avoid loading any floating-point library code */
//@Scheck: weird, probably linker trick
int _efgfmt(Fmt *)
{
    return -1;
}
```

C.1 Memory management

⟨*function* `emalloc` 169b⟩≡ (222b)
```
void *
emalloc(long n)
{
    void *p = Malloc(n);

    if(p==nil)
        panic("Can't malloc %d bytes", n);
    return p;
}
```
Uses `panic()` 168c.

⟨*function* `Malloc` 169c⟩≡ (225b)
```
void*
Malloc(ulong n)
{
    return mallocz(n, 1);
}
```

⟨*function* `efree` 169d⟩≡ (222b)
```
void
efree(void *p)
{
    if(p)
        free(p);
    else pfmt(err, "free 0\n");
}
```
Uses `err` 167b and `pfmt()` 178b.

C.2 Command-line arguments

⟨*global* cmdname 170a⟩≡ (212a)
```
char *cmdname;
```

⟨*global* flagarg 170b⟩≡ (212a)
```
static char *flagarg="";
```
Uses flagarg-17 170b.

⟨*global* reason 170c⟩≡ (212a)
```
static int reason;
```

⟨*constant* RESET 170d⟩≡ (212a)
```
#define RESET 1
```

⟨*constant* FEWARGS 170e⟩≡ (212a)
```
#define FEWARGS 2
```

⟨*constant* FLAGSYN 170f⟩≡ (212a)
```
#define FLAGSYN 3
```

⟨*constant* BADFLAG 170g⟩≡ (212a)
```
#define BADFLAG 4
```

⟨*global* badflag 170h⟩≡ (212a)
```
static int badflag;
```

⟨*function* getflags 170i⟩≡ (212a)
```
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-20 170e, RESET-19 170d, badflag-23 170h, cmdname 170a, flag 41c, flagarg-17 170b, flagset 42a, reason-18 170c, and reverse() 171a.

⟨*function* reverse 171a⟩≡ (212a)

```
static void
reverse(char **p, char **q)
{
    char *t;
    for(;p<q;p++,--q){ t=*p; *p=*q; *q = t; }
}
```

⟨*function* scanflag 171b⟩≡ (212a)

```
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 172⟩≡ (212a)

```
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 173a⟩≡ (212a)

```
static void
errn(char *s, int count)
{
    while(count){ errc(*s++); --count; }
}
```

Uses errc() 174d.

⟨*function* errs 173b⟩≡ (212a)

```
static void
errs(char *s)
{
    while(*s) errc(*s++);
}
```

Uses errc() 174d.

⟨*constant NBUF (rc/getflags.c)* 174a⟩≡ (212a)
```
#define NBUF 80
```

⟨*global* buf 174b⟩≡ (212a)
```
static char buf[NBUF];
```
Uses NBUF-24 174a.

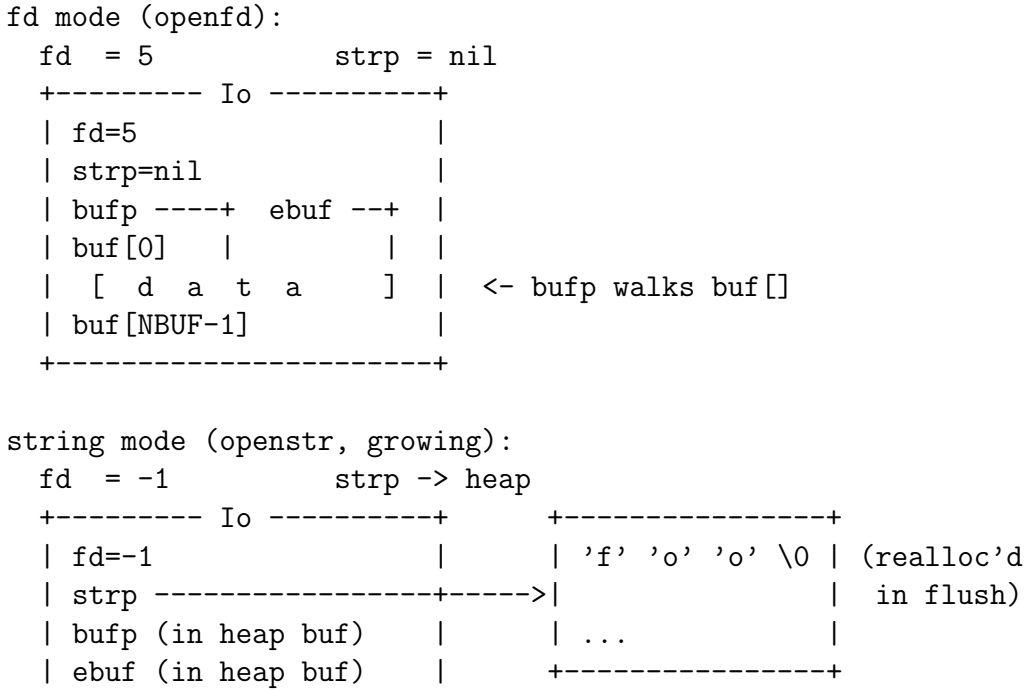
⟨*global* bufp 174c⟩≡ (212a)
```
static char *bufp = buf;
```
Uses buf-25 174b and bufp-26 174c.

⟨*function* errc 174d⟩≡ (212a)
```
static void
errc(int c)
{
    *bufp++=c;
    if(bufp==&buf[NBUF] || c=='\n'){
        write(2, buf, bufp-buf);
        bufp = buf;
    }
}
```
Uses NBUF-24 174a, buf-25 174b, and bufp-26 174c.

C.3 Buffered IO

⟨*constant* NBUF 174e⟩≡ (208b)
```
#define NBUF 512
```

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`), `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`), `fd` is -1, `strp` owns a heap-grown buffer, and `bufp`/`ebuf` walk that buffer instead of `buf`. In core mode (used by `opencore`), `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`.



```
    | buf[] unused          |
    +-----------------------+

  core mode (opencore, read-only):
    fd = -1, strp owns a fixed-length preloaded buffer.
```

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.

⟨*struct* `Io` 175a⟩≡ (208b)
```
struct Io {
    fdt fd;
    byte *bufp, *ebuf, *strp;
    byte buf[NBUF];
};
```
Uses `NBUF` 174e.

⟨*function* `openfd` 175b⟩≡ (212b)
```
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` 175a.

⟨*enum* `MiscConstants` 175c⟩≡ (212b)
```
enum { Stralloc = 100, };
```

⟨*function* `flush` 175d⟩≡ (212b)
```
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` 174e, `Stralloc-1` 175c, `dotrap()` 123a, `ntrap` 122a, and `panic()` 168c.

⟨*function* `pchr` 176a⟩≡ (224a)

```
void
pchr(io *b, int c)
{
    if(b->bufp==b->ebuf)
        fullbuf(b, c);
    else *b->bufp++=c;
}
```

Uses `fullbuf()` 176b.

⟨*function* `fullbuf` 176b⟩≡ (212b)

```
int
fullbuf(io *f, int c)
{
    flush(f);
    return *f->bufp++=c;
}
```

Uses `flush()` 175d.

⟨*function* `rchr` 176c⟩≡ (212b)

```
int
rchr(io *b)
{
    if(b->bufp==b->ebuf)
        return emptybuf(b);
    return *b->bufp++;
}
```

Uses `emptybuf()` 176d.

⟨*function* `emptybuf` 176d⟩≡ (212b)

```
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` 49a and `NBUF` 174e.

⟨*function* `rutf` 176e⟩≡ (212b)

```
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 49a and rchr() 176c.

⟨*function* closeio 177a⟩≡ (212b)

```
void
closeio(io *io)
{
    if(io->fd>=0)
        close(io->fd);
    if(io->strp)
        efree(io->strp);
    efree(io);
}
```

Uses efree() 169d.

⟨*function* openstr 177b⟩≡ (212b)

```
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 175a, Stralloc-1 175c, and emalloc() 169b.

⟨*function* opencore 177c⟩≡ (212b)

```
/*
 * 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 175a, Memcpy() 181h, and emalloc() 169b.

C.4 Format

⟨*global* pfmtnest 178a⟩≡ (224a)
```
int pfmtnest = 0;
```
Uses pfmtnest 178a.

⟨*function* pfmt 178b⟩≡ (224a)
```
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 167b, flush() 175d, pchr() 176a, pcmd() 162e, pdec() 179b, pfmtnest 178a, poct() 179c, pptr() 180b, pquo() 179d, pstr() 179a, pval() 180c, and pwrd() 180a.

⟨*function* pstr 179a⟩≡ (224a)

```
void
pstr(io *f, char *s)
{
    if(s==0)
        s="(null)";
    while(*s) pchr(f, *s++);
}
```

Uses pchr() 176a.

⟨*function* pdec 179b⟩≡ (224a)

```
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() 176a and pdec() 179b.

⟨*function* poct 179c⟩≡ (224a)

```
void
poct(io *f, unsigned int n)
{
    if(n>7)
        poct(f, n>>3);
    pchr(f, (n&7)+'0');
}
```

Uses pchr() 176a and poct() 179c.

⟨*function* pquo 179d⟩≡ (224a)

```
void
pquo(io *f, char *s)
{
    pchr(f, '\'');
    for(;*s;s++)
```

```
        if(*s=='\'')
            pfmt(f, "''");
        else pchr(f, *s);
    pchr(f, '\'');
}
```

Uses pchr() 176a and pfmt() 178b.

⟨*function* pwrd 180a⟩≡ (224a)

```
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() 226, pquo() 179d, and pstr() 179a.

⟨*function* pptr 180b⟩≡ (224a)

```
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() 176a.

⟨*function* pval 180c⟩≡ (224a)

```
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() 176a and pwrd() 180a.

C.5 String conversions

⟨*global* bp 180d⟩≡ (222b)

```
char *bp;
```

⟨*function* `inttoascii` 181a⟩≡ (222b)

```
void
inttoascii(char *s, long n)
{
    bp = s;
    iacvt(n);
    *bp='\0';
}
```

Uses `bp` 180d and `iacvt()` 181b.

⟨*function* `iacvt` 181b⟩≡ (222b)

```
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` 180d and `iacvt()` 181b.

C.6 Misc

⟨*function* `Write` 181c⟩≡ (222b)

⟨*function* `Read` 181d⟩≡ (222b)

⟨*function* `Seek` 181e⟩≡ (222b)

⟨*function* `Creat` 181f⟩≡ (222b)

```
int
Creat(char *file)
{
    return create(file, 1, 0666L);
}
```

⟨*function* `Dup` 181g⟩≡ (222b)

⟨*function* `Memcpy` 181h⟩≡ (222b)

```
void
Memcpy(void *a, void *b, long n)
{
    memmove(a, b, n);
}
```

Appendix D

Examples of `rc` scripts

D.1 `/rc/lib/rcmain`

This is the initialization script sourced by the bootstrap bytecodes (see Chapter 12). It sets up default values for `$home`, `$ifs`, `$prompt`, and `$path`, calls `finit` to import shell functions from the environment, 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.

```
# rcmain: Plan 9 version
if(~ $#home 0) home=/
if(~ $#ifs 0) ifs='
'

switch($#prompt){
case 0
 prompt=('% ' ' ')
case 1
 prompt=($prompt ' ')
}

if(~ $rcname ?.out) prompt=('broken! ' ' ')

if(flag p) path=/bin
if not{
 finit
 if(~ $#path 0) path=(. /bin)
}

fn sigexit

if(! ~ $#cflag 0){
 if(flag l && /bin/test -r $home/lib/profile) . $home/lib/profile
 status=''
 eval $cflag
}
if not if(flag i){
 if(flag l && /bin/test -r $home/lib/profile) . $home/lib/profile
 status=''
```

```
 if(! ~ $#* 0) . $*
 . -i '#d/0'
}
if not if(~ $#* 0) . '#d/0'
if not{
 status=''
 . $*
}
exit $status
```

D.2 /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 (the `flag l` test in `rcmain` above). 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`.

```
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.3 /rc/bin/man

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. 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)* 185⟩≡ (203d)

```
void
main(int argc, char *argv[])
{
    int nflag;
    int i, len;
    char *buf, *p;

    nflag = 0;
    if(argc > 1 && strcmp(argv[1], "-n") == 0)
        nflag = 1;

    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(1, buf, p-buf) < 0){
        fprint(2, "echo: write error: %r\n");
```

```
        exits("write error");
    }

    exits((char *)nil);
}
```

E.2 pwd

`pwd` simply calls `getwd()` and prints the result. In Plan 9, the current directory is a per-process kernel state, not an environment variable like `\$PWD` in some UNIX shells.

⟨*function main (misc/pwd.c)* 186a⟩≡ (203e)

```
/*
 * Print working (current) directory
 */
void
main(int argc, char *argv[])
{
    char pathname[512];

    USED(argc, argv);
    if(getwd(pathname, sizeof(pathname)) == 0) {
        fprint(2, "pwd: %r\n");
        exits("getwd");
    }
    print("%s\n", pathname);
    exits(nil);
}
```

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` 186b⟩≡ (203f)

```
#define EQ(a,b) ((tmp=a)==0?0:(strcmp(tmp,b)==0))
```

⟨*global* `ap` 186c⟩≡ (203f)

```
int ap;
```

⟨*global* `ac` 186d⟩≡ (203f)

```
int ac;
```

⟨*global* `av` 186e⟩≡ (203f)

```
char **av;
```

⟨*global* `tmp` 186f⟩≡ (203f)

```
static char *tmp;
```

⟨*function main (misc/test.c)* 187a⟩≡ (203f)
```
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 187b⟩≡ (203f)
```
char *
nxtarg(int mt)
{
    if(ap>=ac){
        if(mt){
            ap++;
            return(0);
        }
        synbad("argument expected","");
    }
    return(av[ap++]);
}
```

⟨*function* nxtintarg 187c⟩≡ (203f)
```
bool
nxtintarg(int *pans)
{
    if(ap<ac && isint(av[ap], pans)){
        ap++;
        return true;
    }
    return false;
}
```

⟨*function* e 187d⟩≡ (203f)
```
int
e(void)
{
    int p1;

    p1 = e1();
    if (EQ(nxtarg(1), "-o"))
        return(p1 || e());
    ap--;
```

```
    return(p1);
}
```

⟨*function* e1 188a⟩≡ (203f)

```
int
e1(void)
{
    int p1;

    p1 = e2();
    if (EQ(nxtarg(1), "-a"))
        return (p1 && e1());
    ap--;
    return(p1);
}
```

⟨*function* e2 188b⟩≡ (203f)

```
int
e2(void)
{
    if (EQ(nxtarg(0), "!"))
        return(!e2());
    ap--;
    return(e3());
}
```

⟨*function* e3 188c⟩≡ (203f)

```
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 190a⟩≡ (203f)

```
bool
tio(char *a, int f)
{
    return access (a, f) >= 0;
}
```

⟨*function* hasmode 190b⟩≡ (203f)

```
/*
 * 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 190c⟩≡ (203f)

```
bool
isdir(char *f)
{
    return hasmode(f, DMDIR);
}
```

⟨*function* isreg 190d⟩≡ (203f)

```
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` 191a⟩≡ (203f)

```
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` 191b⟩≡ (203f)

```
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` 191c⟩≡ (203f)

```
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` 191d⟩≡ (203f)

```
int
isint(char *s, int *pans)
{
    char *ep;
```

```
    *pans = strtol(s, &ep, 0);
    return (*ep == 0);
}
```

⟨*function* isolder 192⟩≡ (203f)

```
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` 193a⟩≡ (203f)

```
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` 193b⟩≡ (203f)

```
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 `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` 193c⟩≡ (204)

```
#define MAXLINE 200  /* maximum line length */
```

⟨*constant* `WORD` 193d⟩≡ (204)

```
#define WORD 256  /* token code for words */
```

⟨*constant* `EOF` 193e⟩≡ (204)

```
#define EOF -1   /* token code for end of file */
```

⟨*function* `ispunct` 193f⟩≡ (204)

```
#define ispunct(c)  (c=='|' || c=='&' || c==';' || c=='<' || \
                     c=='>' || c=='(' || c==')' || c=='\n')
```

⟨*function* isspace 194a⟩≡ (204)
```
#define isspace(c)  (c==' ' || c=='\t')
```

⟨*function* execute 194b⟩≡ (204)
```
#define execute(np)  (ignored = (np? (*(np)->op)(np) : 0))
```

⟨*struct* Node 194c⟩≡ (204)
```
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 194d⟩≡ (204)
```
Node nodes[25];  /* node pool */
```

⟨*global* nfree 194e⟩≡ (204)
```
Node *nfree;   /* next available node */
```

⟨*global* strspace 194f⟩≡ (204)
```
char strspace[10*MAXLINE]; /* string storage */
```

⟨*global* sfree 194g⟩≡ (204)
```
char *sfree;   /* next free character in strspace */
```

⟨*global* t 194h⟩≡ (204)
```
int t;   /* current token code */
```

⟨*global* token 194i⟩≡ (204)
```
char  *token;   /* current token text (in strspace) */
```

⟨*global* putback 194j⟩≡ (204)
```
int putback = 0;  /* lookahead */
```

⟨*global* status 194k⟩≡ (204)
```
char status[256];  /* exit status of most recent command */
```

⟨*global* cflag 194l⟩≡ (204)
```
int cflag = 0;  /* command is argument to sh */
```

⟨*global* tflag 194m⟩≡ (204)
```
int tflag = 0;  /* read only one line */
```

⟨*global* interactive 194n⟩≡ (204)
```
bool interactive = false; /* prompt */
```

⟨*global* cflagp 194o⟩≡ (204)
```
char *cflagp;  /* command line for cflag */
```

⟨*global* path 194p⟩≡ (204)
```
char *path[] ={"/bin", 0};
```

⟨*global* ignored 194q⟩≡ (204)
```
int ignored;
```

⟨*function main (sh/sh.c)* 195a⟩≡ (204)

```
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` 195b⟩≡ (204)

```
/* 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` 195c⟩≡ (204)

```
/* 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 197a⟩≡ (204)

```
/* 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 197b⟩≡ (204)

```
/* 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 198a⟩≡ (204)

```
/* 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 198b⟩≡ (204)

```
/* 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 199a⟩≡ (204)

```
/* 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 199b⟩≡ (204)

```
/* 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 199c⟩≡ (204)

```
/* 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 200a⟩≡ (204)

```
/* 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 200b⟩≡ (204)

```
/* 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 200c⟩≡ (204)

```
/* 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 201⟩≡ (204)

```
/* 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 202a⟩≡ (204)

```
/* 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 202b⟩≡ (204)

```
/* 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 202c⟩≡ (204)

```
/* 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 F

Extra Code

F.1 misc/

⟨rc/tests/ls_root.c 203a⟩≡

⟨rc/y.tab.h 203b⟩≡

⟨rc/y.tab.c 203c⟩≡

F.1.1 misc/echo.c

⟨misc/echo.c 203d⟩≡
```
#include <u.h>
#include <libc.h>

⟨function main(echo) 185⟩
```

F.1.2 misc/pwd.c

⟨misc/pwd.c 203e⟩≡
```
#include <u.h>
#include <libc.h>

⟨function main (misc/pwd.c) 186a⟩
```

F.1.3 misc/test.c

⟨misc/test.c 203f⟩≡
```
/*
 * 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 186b⟩

⟨*global* ap 186c⟩
⟨*global* ac 186d⟩
⟨*global* av 186e⟩
⟨*global* tmp 186f⟩

```
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 (misc/test.c)* 187a⟩

⟨*function* nxtarg 187b⟩

⟨*function* nxtintarg 187c⟩

⟨*function* e 187d⟩

⟨*function* e1 188a⟩
⟨*function* e2 188b⟩
⟨*function* e3 188c⟩

⟨*function* tio 190a⟩

⟨*function* hasmode 190b⟩

⟨*function* isdir 190c⟩
⟨*function* isreg 190d⟩

⟨*function* isatty 191a⟩

⟨*function* fsizep 191b⟩

⟨*function* synbad 191c⟩

⟨*function* isint 191d⟩

⟨*function* isolder 192⟩
⟨*function* isolderthan 193a⟩
⟨*function* isnewerthan 193b⟩

F.2 sh/

F.2.1 sh/sh.c

⟨sh/sh.c 204⟩≡

```
/* sh - simple shell - great for early stages of porting */
```

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

⟨constant MAXLINE 193c⟩
⟨constant WORD 193d⟩
⟨constant EOF 193e⟩
⟨function ispunct 193f⟩
⟨function isspace 194a⟩
⟨function execute 194b⟩

typedef struct Node Node;
⟨struct Node 194c⟩

⟨global nodes 194d⟩
⟨global nfree 194e⟩
⟨global strspace 194f⟩
⟨global sfree 194g⟩
⟨global t 194h⟩
⟨global token 194i⟩
⟨global putback 194j⟩
⟨global status 194k⟩
⟨global cflag 194l⟩
⟨global tflag 194m⟩
⟨global interactive 194n⟩
⟨global cflagp 194o⟩
⟨global path 194p⟩
⟨global ignored 194q⟩

// 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/sh.c) 195a⟩

⟨function alloc 195b⟩

⟨function builtin 195c⟩
```

⟨*function* `command` 197a⟩

⟨*function* `getch` 197b⟩

⟨*function* `gettoken` 198a⟩

⟨*function* `list` 198b⟩

⟨*function* `error` 199a⟩

⟨*function* `pipeline` 199b⟩

⟨*function* `redirect` 199c⟩

⟨*function* `setio` 200a⟩

⟨*function* `simple` 200b⟩

⟨*function* `xpipeline` 200c⟩

⟨*function* `xsimple` 201⟩

⟨*function* `xsubshell` 202a⟩

⟨*function* `xnowait` 202b⟩

⟨*function* `xwait` 202c⟩

F.3 rc/

F.3.1 rc/fns.h

⟨`rc/fns.h` 206⟩≡

```
// 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*);
```

F.3.2 rc/getflags.h

⟨rc/getflags.h 208a⟩≡
⟨*constant* NFLAG 41d⟩

```
extern char **flag[NFLAG];
extern char *flagset[];

int getflags(int, char*[], char*, int);
void usage(char*);
```

F.3.3 rc/io.h

⟨rc/io.h 208b⟩≡
⟨*constant EOF (rc/io.h)* 49a⟩
⟨*constant* NBUF 174e⟩

⟨*struct* Io 175a⟩

```
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*);
```

F.3.4 rc/rc.h

⟨rc/rc.h 208c⟩≡

```
/*
 * 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 121a⟩

⟨*constant* SIGINT 121b⟩
⟨*constant* SIGQUIT 121c⟩

```
//???
//#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 167a⟩
```
#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 57d⟩
⟨*constant* WRITE 57b⟩
⟨*constant* READ 57c⟩
⟨*constant* HERE 141a⟩
⟨*constant* DUPFD 147d⟩
⟨*constant* CLOSE 147a⟩
⟨*constant* RDWR 145b⟩

⟨*struct* Word 36a⟩

```
// 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 38b⟩

```
// var.c
var *vlook(char*);
var *gvlook(char*);
var *newvar(char*, var*);
void setvar(char*, word*);
```

⟨*constant* NVAR 39c⟩
```
extern var *gvar[NVAR];  /* hash for globals */
```

⟨*struct* Here 141c⟩

⟨*constant* GLOB 129a⟩

⟨*constant* PRD 92a⟩
⟨*constant* PWR 92b⟩

```
// 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 109a, Code 33c, Io 175a, List 38a, Redir 85b, Thread 34b, Tree, Var 38b, and Word 36a.

F.3.5 rc/exec.h

⟨rc/exec.h 211a⟩≡

```
/*
 * 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 85b⟩

⟨constant NSTATUS 93c⟩
⟨constant ROPEN 87b⟩
⟨constant RDUP 148b⟩
⟨constant RCLOSE 148c⟩

⟨struct List 38a⟩

⟨struct Thread 34b⟩

⟨struct Builtin 109a⟩

// 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);
```

F.3.6 rc/globals.c

⟨rc/globals.c 211b⟩≡

```
⟨includes 25⟩

// was in rc.h
⟨global mypid 127a⟩
```

⟨*global* `promptstr` 50d⟩
⟨*global* `ndot` 114a⟩

```
// was in exec.h
```
⟨*global* `runq` 34c⟩
⟨*global* `codebuf` 33b⟩
⟨*global* `eflagok` 138c⟩

F.3.7 rc/getflags.c

⟨`rc/getflags.c` 212a⟩≡
```
#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` 42a⟩
⟨*global* `flag` 41c⟩
⟨*global* `cmdname` 170a⟩
⟨*global* `flagarg` 170b⟩
⟨*global* `reason` 170c⟩

⟨*constant* `RESET` 170d⟩
⟨*constant* `FEWARGS` 170e⟩
⟨*constant* `FLAGSYN` 170f⟩
⟨*constant* `BADFLAG` 170g⟩

⟨*global* `badflag` 170h⟩

⟨*function* `getflags` 170i⟩

⟨*function* `reverse` 171a⟩

⟨*function* `scanflag` 171b⟩

⟨*function* `usage` 172⟩

⟨*function* `errn` 173a⟩

⟨*function* `errs` 173b⟩
⟨*constant NBUF (rc/getflags.c)* 174a⟩
⟨*global* `buf` 174b⟩
⟨*global* `bufp` 174c⟩

⟨*function* `errc` 174d⟩

F.3.8 rc/io.c

⟨`rc/io.c` 212b⟩≡
⟨*includes* 25⟩

⟨*enum* `MiscConstants` 175c⟩

```
// forward decls
int emptybuf(io*);
int fullbuf(io*, int);
```

⟨*function* `rchr` 176c⟩

⟨*function* `rutf` 176e⟩

⟨*function* `fullbuf` 176b⟩

⟨*function* `flush` 175d⟩

⟨*function* `openfd` 175b⟩

⟨*function* `openstr` 177b⟩

⟨*function* `opencore` 177c⟩

⟨*function* `closeio` 177a⟩

⟨*function* `emptybuf` 176d⟩

F.3.9 `rc/input.c`

⟨`rc/input.c` 213⟩≡

```
⟨includes 25⟩
#include "y.tab.h"

// was in lex.c

⟨global future 52a⟩
⟨global doprompt 50c⟩
⟨global inquote 58c⟩
⟨global incomm 54e⟩

// was in rc.h
⟨global lastc 168a⟩

// forward decl
int getnext(void);

⟨function nextc 52b⟩
⟨function advance 52c⟩
⟨function getnext 48⟩

⟨function pprompt 50h⟩


⟨function nextis 52d⟩


extern unsigned hash(char *as, int n);

// was in tree.c

⟨function token 59b⟩
```

```
// was in var.c

⟨constant NKW 60d⟩
⟨struct Kw 60b⟩
⟨global kw 60c⟩

⟨function kenter 60f⟩

⟨function kinit 60e⟩

⟨function klook 60a⟩
```

F.3.10 rc/var.c

⟨rc/var.c 214a⟩≡

```
⟨includes 25⟩

// was in rc.h
⟨global gvar 39d⟩

⟨function hash 40b⟩


⟨function gvlook 40a⟩

⟨function vlook 39b⟩

⟨function setvar 39a⟩

⟨function newvar 40c⟩
```

F.3.11 rc/env.c

⟨rc/env.c 214b⟩≡

```
⟨includes 25⟩

// the Vxxx are back in plan9.c
```

F.3.12 rc/glob.c

⟨rc/glob.c 214c⟩≡

```
⟨includes 25⟩

bool matchfn(void*, void*);

// NDIR and Globsize are back in plan9.c
extern int Globsize(char *p);

⟨global globname 131a⟩
⟨global globv 129d⟩
⟨function deglob 131c⟩

⟨function globcmp 132a⟩

⟨function globsort 132b⟩
⟨function globdir 132c⟩
```

⟨*function* `glob` 131b⟩

⟨*function* `equtf` 133a⟩

⟨*function* `nextutf` 133b⟩

⟨*function* `unicode` 133c⟩

⟨*function* `matchfn` 134a⟩

⟨*function* `match` 134b⟩

⟨*function* `globlist1` 130a⟩

⟨*function* `globlist` 129f⟩

F.3.13 `rc/executils.c`

⟨`rc/executils.c` 215a⟩≡
⟨*includes* 25⟩

⟨*global* `argv0` 44b⟩

⟨*function* `start` 44a⟩

⟨*function* `pushword` 45b⟩

⟨*function* `popword` 73c⟩

⟨*function* `pushlist` 45a⟩

⟨*function* `poplist` 73b⟩

⟨*function* `pushredir` 85e⟩

⟨*function* `Xerror` 77c⟩

⟨*function* `Xerror1` 78a⟩

⟨*function* `turfredir` 90b⟩

⟨*function* `Xpopredir` 88b⟩

⟨*function* `Xreturn` 91d⟩

F.3.14 `rc/exec.c`

⟨`rc/exec.c` 215b⟩≡
⟨*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{... 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` 89c⟩

⟨*function* `Xsettrue` 97a⟩

⟨*function* `Xbang` 83e⟩

⟨*function* `Xclose` 148d⟩

⟨*function* `Xdup` 147f⟩

⟨*function* `Xeflag` 139a⟩

⟨*function* `Xexit` 94c⟩

⟨*function* `Xfalse` 83c⟩
⟨*global* `ifnot` 95c⟩

⟨*function* `Xifnot` 96c⟩

⟨*function* `Xjump` 97b⟩

⟨*function* `Xmark` 73a⟩

⟨*function* `Xpopm` 99⟩

⟨*function* `Xread` 89a⟩

⟨*function* `Xrdwr` 145e⟩

⟨*function* `Xtrue` 83b⟩

⟨*function* `Xif` 95b⟩

⟨*function* `Xwastrue` 96a⟩

⟨*function* `Xword` 72f⟩

⟨*function* `Xwrite` 88a⟩

⟨*function* `list2str` 84b⟩

⟨*function* `Xmatch` 84a⟩

⟨*function* `Xcase` 100b⟩

⟨*function* `conclist` 108c⟩

⟨*function* `Xconc` 108b⟩

⟨*function* `Xassign` 104b⟩

⟨*function* `Xdol` 105c⟩

⟨*function* `Xqdol` 151c⟩

⟨*function* `subwords` 107c⟩

⟨*function* `Xsub` 107b⟩

⟨*function* `Xcount` 106b⟩

⟨*function* `Xlocal` 104c⟩

⟨*function* `Xunlocal` 105a⟩

⟨*function* `Xfn` 102a⟩

⟨*function* `Xdelfn` 102b⟩

⟨*function* Xpipewait 93d⟩

⟨*function* Xrdcmds 46a⟩

⟨*function* Xdelhere 142c⟩

⟨*function* Xfor 98a⟩

⟨*function* Xglob 130b⟩

⟨*global* envdir 119⟩

```
// Xrdfn is back in plan9.c
```

F.3.15 rc/processes.c

⟨rc/processes.c 218a⟩≡

⟨*includes* 25⟩

```
//#include <string.h>

// Fdprefix, delwaitpid(), havewaitpid() Waitfor() are back in plan9.c
extern char *Fdprefix;
```

⟨*global* waitpids 74c⟩
⟨*global* nwaitpids 74d⟩
⟨*function* addwaitpid 74e⟩
⟨*function* clearwaitpids 75c⟩

⟨*function* mkargv 80b⟩

⟨*function* Execute 80a⟩

```
// was in havefork.c
```

⟨*function* Xasync 94b⟩

⟨*function* Xpipe 92c⟩

⟨*function* Xbackq 150c⟩

⟨*function* Xpipefd 149c⟩

⟨*function* Xsubshell 140d⟩

⟨*function* execforkexec 78b⟩

F.3.16 rc/tree.c

⟨rc/tree.c 218b⟩≡

⟨*includes* 25⟩

```
#include "y.tab.h"
```

⟨*global* treenodes 31c⟩
⟨*function* newtree 31e⟩

⟨*function* freenodes 32a⟩

⟨*function* `tree1` 32b⟩

⟨*function* `tree2` 32c⟩

⟨*function* `tree3` 32d⟩

⟨*function* `mung1` 66e⟩

⟨*function* `mung2` 67a⟩

⟨*function* `mung3` 66f⟩

⟨*function* `epimung` 67g⟩
⟨*function* `simplemung` 65a⟩

F.3.17 rc/lex.c

⟨`rc/lex.c` 219a⟩≡
⟨*includes* 25⟩
```
#include "y.tab.h"
```

⟨*constant* `NTOK` 54d⟩

```
// was used by subr.c
```
⟨*global* `lastdol` 56b⟩
⟨*global* `lastword` 58e⟩
```
// was in rc.h
```
⟨*global* `tok` 54c⟩

⟨*function* `wordchr` 53b⟩

⟨*function* `idchr` 54a⟩

⟨*function* `yyerror` 167e⟩

⟨*function* `skipwhite` 54f⟩

⟨*function* `skipnl` 55a⟩

⟨*function* `addtok` 129c⟩

⟨*function* `addutf` 139b⟩

⟨*function* `yylex` 53a⟩

F.3.18 rc/trap.c

⟨`rc/trap.c` 219b⟩≡
⟨*includes* 25⟩

```
// NSIG, SIGINT, SIGQUIT are back in rc.h (under an ifdef for plan9)

// was in exec.h
```
⟨*global* `ntrap` 122a⟩
⟨*global* `trap` 122b⟩

```
// 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 123a⟩
```

F.3.19 rc/simple.c

⟨rc/simple.c 220a⟩≡

```
/*
 * Maybe 'simple' is a misnomer.
 */
⟨includes 25⟩

// forward decls
void execfunc(var*);

⟨function exitnext 82a⟩

⟨function Xsimple 77a⟩

⟨function doredir 86⟩

⟨function execexec 79a⟩

⟨function execfunc 103c⟩

⟨global rdcmds 115a⟩

⟨function execcmds 115b⟩
```

F.3.20 rc/pcmd.c

⟨rc/pcmd.c 220b⟩≡

```
⟨includes 25⟩
#include "y.tab.h"

⟨global nl 162a⟩
⟨constant c0 162b⟩
⟨constant c1 162c⟩
⟨constant c2 162d⟩

⟨function pdeglob 164a⟩

⟨function pcmd 162e⟩
```

F.3.21 rc/here.c

⟨rc/here.c 220c⟩≡

```
⟨includes 25⟩
#include "y.tab.h"

⟨global here 141d⟩
⟨global ehere 141e⟩
⟨global ser 143b⟩
```

⟨*global tmp (rc/here.c)* 143c⟩
⟨*global* `hex` 143d⟩

```
void psubst(io*, uchar*);
void pstrs(io*, word*);
```

⟨*function* `hexnum` 143e⟩

⟨*function* `heredoc` 144a⟩

⟨*constant* `NLINE` 141f⟩

⟨*function* `readhere` 142a⟩

⟨*function* `psubst` 144b⟩

⟨*function* `pstrs` 145a⟩

F.3.22 rc/code.c

⟨`rc/code.c` 221⟩≡
⟨*includes* 25⟩
```
#include "y.tab.h"
```

⟨*constant c0 (rc/code.c)* 72b⟩
⟨*constant c1 (rc/code.c)* 72c⟩
⟨*constant c2 (rc/code.c)* 72d⟩
⟨*global* `codep` 71a⟩
⟨*global* `ncode` 71b⟩
⟨*function* `emitf` 71d⟩
⟨*function* `emiti` 71c⟩
⟨*function* `emits` 71e⟩

```
// forward decls
void outcode(tree*, bool);
void codeswitch(tree*, bool);
bool iscase(tree*);
```

⟨*function* `morecode` 71f⟩

⟨*function* `stuffdot` 82e⟩

⟨*function* `compile` 71g⟩

⟨*function* `cleanhere` 142b⟩

⟨*function* `fnstr` 101d⟩

⟨*function* `outcode` 72e⟩
⟨*function* `codeswitch` 98c⟩

⟨*function* `iscase` 100a⟩

⟨*function* `codecopy` 33d⟩

⟨*function* `codefree` 33e⟩

F.3.23 rc/pfnc.c

⟨rc/pfnc.c 222a⟩≡
⟨*includes* 25⟩

⟨*global* fname 165a⟩

⟨*function* pfnc 165b⟩

F.3.24 rc/utils.c

⟨rc/utils.c 222b⟩≡
⟨*includes* 25⟩

```
// Opendir, Closedir and so on are back in plan9.c
```

⟨*function* Unlink 143a⟩
⟨*function* Write 181c⟩
⟨*function* Read 181d⟩
⟨*function* Seek 181e⟩

⟨*function* Creat 181f⟩

⟨*function* Dup 181g⟩

⟨*function* Memcpy 181h⟩

```
// back in plan9.c
extern void* Malloc(ulong n);
```
⟨*function* emalloc 169b⟩

⟨*function* efree 169d⟩

⟨*global* bp 180d⟩

⟨*function* iacvt 181b⟩

⟨*function* inttoascii 181a⟩

F.3.25 rc/status.c

⟨rc/status.c 222c⟩≡
⟨*includes* 25⟩

⟨*function* setstatus 73f⟩

⟨*function* getstatus 74a⟩

⟨*function* truestatus 74b⟩

⟨*function* concstatus 93f⟩

F.3.26 rc/builtins.c

⟨rc/builtins.c 222d⟩≡

⟨*includes* 25⟩

⟨*function* `dochdir` 111b⟩

⟨*function* `appfile` 111a⟩

⟨*function* `mapfd` 154a⟩

⟨*function* `execcd` 110c⟩

⟨*function* `execexit` 112c⟩

⟨*function* `execeval` 114e⟩

⟨*function* `execflag` 156a⟩

```
// was in plan9.c
```

⟨*function* `Executable` 154b⟩

⟨*function* `execwhatis` 152⟩

⟨*function* `execshift` 156b⟩

⟨*global* `dotcmds` 114b⟩

⟨*function* `execdot` 112d⟩

⟨*function* `execwait` 115c⟩

⟨*function* `execnewpgrp` 155⟩

⟨*global* `rdfns` 157a⟩

```
extern fdt envdir;
```

⟨*function* `execfinit` 157b⟩

⟨*global* `builtins` 109b⟩

F.3.27 `rc/path.c`

⟨`rc/path.c` 223⟩≡
⟨*includes* 25⟩

⟨*global* `nullpath` 79d⟩

⟨*function* `searchpath` 79c⟩

F.3.28 rc/fmt.c

⟨rc/fmt.c 224a⟩≡
⟨*includes* 25⟩

⟨*global* pfmtnest 178a⟩

```
// 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 178b⟩

⟨*function* pchr 176a⟩

⟨*function* pquo 179d⟩

⟨*function* pwrd 180a⟩

⟨*function* pptr 180b⟩

⟨*function* pstr 179a⟩

⟨*function* pdec 179b⟩

⟨*function* poct 179c⟩

⟨*function* pval 180c⟩

```
// _efgfmt is back in plan9.c
```

F.3.29 rc/words.c

⟨rc/words.c 224b⟩≡
⟨*includes* 25⟩

⟨*function* newword 36b⟩

⟨*function* freewords 37b⟩

⟨*function* count 36c⟩

⟨*function* copynwords 36d⟩

⟨*function* copywords 37a⟩

⟨*function* freelist 37c⟩

F.3.30 rc/error.c

⟨rc/error.c 224c⟩≡

⟨*includes* 25⟩

```
// was in rc.h
```
⟨*global* `nerror` 167c⟩
⟨*global* `err` 167b⟩

```
// forward decls
void Abort(void);
```

⟨*function* `panic` 168c⟩

```
// Exit is back in plan9.c
```

⟨*function* `Abort` 168d⟩

F.3.31 rc/main.c

⟨`rc/main.c` 225a⟩≡
⟨*includes* 25⟩

```
// Rcmain and Isatty are back in plan9.c
extern char* Rcmain;
extern bool Isatty(fdt fd);

/*
 * 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
 */
```
⟨*function main (rc/exec.c)* 41b⟩

F.3.32 rc/plan9.c

⟨`rc/plan9.c` 225b⟩≡

```
/*
 * 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` 126b⟩
⟨*function* `Isatty` 42e⟩

```
// could be in rc.h
```
⟨*enum* `MiscPlan9` 117c⟩

```
// defined in trap.c
extern int trap[NSIG];
// defined in exec.c
extern fdt envdir;

// could be in env.c
```
⟨*function* `Vinit` 118c⟩

```
⟨function addenv 118b⟩
⟨function updenvlocal 117b⟩
⟨function Updenv 117a⟩

// could be in glob.c
⟨constant NDIR 135a⟩
⟨function Globsize 135b⟩

// could be in processes.c
⟨global Fdprefix 149b⟩

// in processes.c
extern int *waitpids;
extern int nwaitpids;

⟨function delwaitpid 75b⟩
⟨function havewaitpid 75d⟩
⟨function Waitfor 81a⟩

// could be in trap.c
⟨global signame 121e⟩
⟨global syssigname 121d⟩
⟨global interrupted 123b⟩
⟨function notifyf 122d⟩
⟨function Trapinit 122c⟩
⟨function Eintr 123c⟩
⟨function Noerror 123e⟩

// could be in fmt.c
⟨function _efgfmt 169a⟩

// could be in error.c
⟨function Exit 168e⟩

// could be in exec.c
⟨function Xrdfn 157c⟩

// could be in utils.c
⟨function Malloc 169c⟩

// could be in utils.c
⟨constant NFD 136a⟩
⟨struct DirEntryWrapper 136b⟩
⟨global dir 136c⟩
⟨function trimdirs 136e⟩
⟨function Readdir 136f⟩
⟨function Opendir 136d⟩
⟨function Closedir 137⟩
```

F.3.33 rc/unix.c

```
⟨rc/unix.c 226⟩≡
  /*
   * 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 129a, NDIR-15 226, NDIRLIST-16 226, dirlist 226, envp 226, err 167b, flag 41c, gettrap() 226, inttoascii() 181a, ntrap 122a, pfmt() 178b, runq 34c, setstatus() 73f, sigmsg 226, and trap 122b.

Glossary

LOC = Lines Of Code
CLI = Command-Line Interface
GUI = Graphical User Interface
IDE = Integrated Development Environment

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

[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) 12

[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) 12

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 10, 12, 21, 25, 73, 120, 159

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Build System mk*. 2016. cited page(s) 160

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 12, 25, 30, 62, 160

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Windowing System rio*. 2016. cited page(s) 9, 112

[Pad25] Yoann Padioleau. *Principia Softwarica: The Plan 9 Utilities*. 2025. cited page(s) 185

[Pou00] Louis Pouzin. The origin of the shell, 2000. http://multicians.org/shell.html. cited page(s) 16, 19

[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