Principia Softwarica: The Plan 9 Shell `rc`

OCaml edition

version 0.1

Yoann Padioleau

yoann.padioleau@gmail.com

with code from

Tom Duff and Yoann Padioleau

March 13, 2026

The text and figures are Copyright © 2014–2026 Yoann Padioleau.
All rights reserved.

The source code is Copyright © 2021 Plan 9 Foundation.
MIT license.

Contents

1 Introduction **8**
1.1 Motivations . . . 8
1.2 The Plan 9 shell: `rc` . . . 9
1.3 Other shells . . . 9
1.4 Getting started . . . 11
1.5 Requirements . . . 11
1.6 About this document . . . 12
1.7 Copyright . . . 12
1.8 Acknowledgments . . . 12

2 Overview **13**
2.1 Essential shell features . . . 13
2.1.1 Running commands . . . 13
2.1.2 Redirections . . . 14
2.1.3 Pipes . . . 14
2.1.4 Storing commands in a script . . . 14
2.1.5 A shell as a scripting language . . . 15
2.2 `rc` command-line interface . . . 15
2.3 `boot.rc` . . . 16
2.3.1 `#!` . . . 16
2.3.2 Initialization script . . . 16
2.3.3 Variables . . . 17
2.3.4 `$path` . . . 17
2.3.5 Comments . . . 17
2.3.6 Quoting and escaping . . . 17
2.3.7 The environment . . . 17
2.3.8 Builtins . . . 18
2.3.9 Other features . . . 18
2.4 Code organization . . . 18
2.5 Software architecture . . . 18
2.5.1 The command-line user interface . . . 18
2.5.2 `rc`'s components . . . 18
2.5.3 Trace of a simple command: `ls /` . . . 22
2.6 Book structure . . . 22

3 Core Data Structures **24**
3.1 Tokens . . . 24
3.2 Abstract syntax tree (AST) . . . 25
3.3 Opcodes . . . 27
3.4 Words . . . 28

3.5 Variables 28
3.6 Functions 29
3.7 Threads and run queue 30
3.7.1 The stack 31
3.7.2 Local variables 32
3.7.3 Redirections 32
3.7.4 Wait status 33

4 `main()` 34
4.1 Overview 34
4.2 Command-line arguments processing 35
4.2.1 Interactive mode: `rc -i` 35
4.2.2 Login mode: `rc -l` 36
4.3 Bytecode interpreter loop 36
4.4 Reading commands: `O.REPL()` 38

5 Lexing 40
5.1 `lexfunc()` skeleton 40
5.2 `token()` skeleton 40
5.3 Spaces and comments (`#`) 42
5.4 Newlines and prompts 42
5.5 Operators (`;`, `&`, `|`, `<`, `>`, `$`, `&&`, `||`, . . .) 43
5.6 Quoted strings (`'...'`) 45
5.7 Keywords and words (`if`, `for`, `while`, `switch`, `fn`, . . .) 45

6 Parsing 47
6.1 `parse_line()` skeleton 47
6.2 Error location reporting 47
6.3 Grammar overview 48
6.4 Simple commands (`<cmd> <arg1>...<argn>`) 50
6.5 Operators 51
6.5.1 Sequence (`;`, `&`) 51
6.5.2 Logical operators (`&&`, `||`, `!`) 52
6.5.3 String matching (`~`) 52
6.5.4 Pipe (`|`) 52
6.5.5 Redirections (`>`, `<`) 52
6.6 Control flow statements (`if`, `if not`, `while`, `switch`, `for`) 53
6.7 Grouping (`{...}`) 54
6.8 Functions (`fn <f> ...`) 54
6.9 Variables (`<x> = ...`, `$x`) 54
6.10 Lists (`(...)`) 55

7 Opcode Generation and Interpretation 56
7.1 Overview 56
7.1.1 `compile()` 56
7.1.2 `outcode_seq()` skeleton 57
7.2 Simple commands 58
7.2.1 Opcode generation 58
7.2.2 Stack management 59
7.2.3 `O.Simple()` 60

7.2.4 Builtin dispatch . . . 61
7.2.5 Fork . . . 61
7.2.6 Exec . . . 62
7.2.7 Error management . . . 63
7.2.8 `$status` management . . . 64
7.2.9 Wait . . . 64
7.2.10 `$path` management . . . 65
7.3 Operators . . . 66
7.3.1 Basic sequence . . . 66
7.3.2 Logical operators . . . 66
7.3.3 String matching . . . 68
7.4 Redirection . . . 68
7.4.1 Opcode generation . . . 68
7.4.2 `O.Write()` . . . 69
7.4.3 `O.Read()` . . . 70
7.4.4 `O.Append()` . . . 70
7.4.5 Closing redirection opened files . . . 70
7.5 Pipe . . . 71
7.5.1 Opcode generation . . . 71
7.5.2 `O.Pipe()` . . . 72
7.5.3 `O.Exit()` . . . 72
7.5.4 `O.PipeWait()` . . . 73
7.6 Asynchronous execution . . . 74
7.7 Control flow statements . . . 74
7.7.1 `if` . . . 74
7.7.2 `if not` . . . 74
7.7.3 `while` . . . 75
7.7.4 `for` . . . 75
7.7.5 `switch` . . . 75
7.7.6 Blocks: `'{...}'` . . . 77
7.8 Functions . . . 77
7.8.1 Function definitions (`fn <foo> ... `) . . . 77
7.8.2 Function uses (`<foo>(...)`) . . . 77
7.9 Variables . . . 77
7.9.1 Variable definitions (`<x>=...`) . . . 77
7.9.2 Variable uses (`$<x>`) . . . 79
7.9.3 Special variables . . . 79

8 Builtins **81**
8.1 Overview . . . 81
8.2 `cd` . . . 82
8.3 `show` . . . 83
8.4 `exit` . . . 83
8.5 `'.'` . . . 83
8.6 `eval` . . . 84
8.7 `flag` . . . 84
8.8 `finit` . . . 85
8.9 `wait` . . . 85

9 Environment 86
- 9.1 `Var.init()` 86
- 9.2 `Var.update_env()` 86

10 Signals 87

11 Initialization 88
- 11.1 Actual bootstrapping code 88
- 11.2 Intitialization script and `rc -m /path/to/rcmain` 89
- 11.3 Actual environment 89

12 Globbing 90
- 12.1 Lexing globbing characters 90
- 12.2 Expanding globbing characters 90
- 12.3 `glob()` 90
- 12.4 `match()` 90

13 Debugging for the `rc` User 91
- 13.1 Printing commands: `rc -x` 91
- 13.2 Printing subprocesses status: `rc -s` 91
- 13.3 Logging: `rc -v` 91
- 13.4 Failing fast: `rc -e` 92
- 13.5 Strict mode: `rc -strict` 93

14 Advanced Features 94
- 14.1 Advanced flags 94
  - 14.1.1 Reading commands from a string: `rc -c` 94
  - 14.1.2 Restrict `$path` candidates: `rc -p` 94
- 14.2 Advanced constructs 94
  - 14.2.1 Subshell: `@ <cmd>` 95
  - 14.2.2 Count and indexing of variables: `$#<foo>`, `$<foo>(...)` 95
  - 14.2.3 Command output as a file: `'<{<cmd>}'` 96
  - 14.2.4 Command substitution: `` `{<cmd>} `` 96
  - 14.2.5 Read-write redirections: `<> <file>` 96
  - 14.2.6 General redirections: `>[2] <file>` 96
  - 14.2.7 Advanced dup: `>[<fd0>=<fd1>]`, `<>[<fd0>=<fd1>]`, `<[<fd0>=<fd1>]` 97
  - 14.2.8 Advanced pipes: `|[<fd>]`, `|[<fd0>=<fd1>]` 97
  - 14.2.9 Here documents: `<< <HERE>` 97
  - 14.2.10 Stringification of variables: `$"<foo>` 97
  - 14.2.11 String concatenation 97
- 14.3 Advanced builtins 98
  - 14.3.1 `exec` 98
  - 14.3.2 `whatis` 98
  - 14.3.3 `rfork` 98
  - 14.3.4 `shift` 98

15 Conclusion 99

A Debugging for the rc Developer 100
A.1 Exception backtraces . . . 100
A.2 Dumpers . . . 100
A.2.1 Dumping tokens . . . 100
A.2.2 Dumping the AST . . . 101
A.2.3 Dumping the opcodes . . . 101
A.3 Opcode generator trace: `rc -r` . . . 101
A.4 CLI actions . . . 102

B Examples of rc scripts 103
B.1 `/rc/lib/rcmain` . . . 103
B.2 `/home/pad/lib/profile` . . . 104

C Extra Code 105
C.1 `Ast.ml` . . . 105
C.2 `Builtin.mli` . . . 105
C.3 `Builtin.ml` . . . 105
C.4 `CLI.mli` . . . 106
C.5 `CLI.ml` . . . 106
C.6 `Compile.mli` . . . 107
C.7 `Compile.ml` . . . 107
C.8 `Env.mli` . . . 108
C.9 `Env.ml` . . . 108
C.10 `Error.mli` . . . 108
C.11 `Error.ml` . . . 109
C.12 `Flags.ml` . . . 109
C.13 `Fn.mli` . . . 109
C.14 `Fn.ml` . . . 109
C.15 `Glob.mli` . . . 109
C.16 `Glob.ml` . . . 110
C.17 `Globals.ml` . . . 110
C.18 `Heredoc.mli` . . . 110
C.19 `Heredoc.ml` . . . 110
C.20 `Interpreter.mli` . . . 110
C.21 `Interpreter.ml` . . . 110
C.22 `Lexer.mli` . . . 111
C.23 `Lexer.mll` . . . 111
C.24 `Main.ml` . . . 111
C.25 `Op_process.ml` . . . 111
C.26 `Op_repl.ml` . . . 111
C.27 `Opcode.ml` . . . 112
C.28 `PATH.mli` . . . 112
C.29 `PATH.ml` . . . 112
C.30 `Parser.mly` . . . 112
C.31 `Parse.mli` . . . 112
C.32 `Parse.ml` . . . 113
C.33 `Pattern.mli` . . . 113
C.34 `Pattern.ml` . . . 113
C.35 `Process.mli` . . . 113
C.36 `Process.ml` . . . 114

C.37 Prompt.mli 114
C.38 Prompt.ml 114
C.39 Runtime.mli 114
C.40 Runtime.ml 115
C.41 Status.mli 116
C.42 Status.ml 117
C.43 Var.mli 117
C.44 Var.ml 117

Glossary **118**

Index **119**

References **123**

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 [Pad16b]).

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

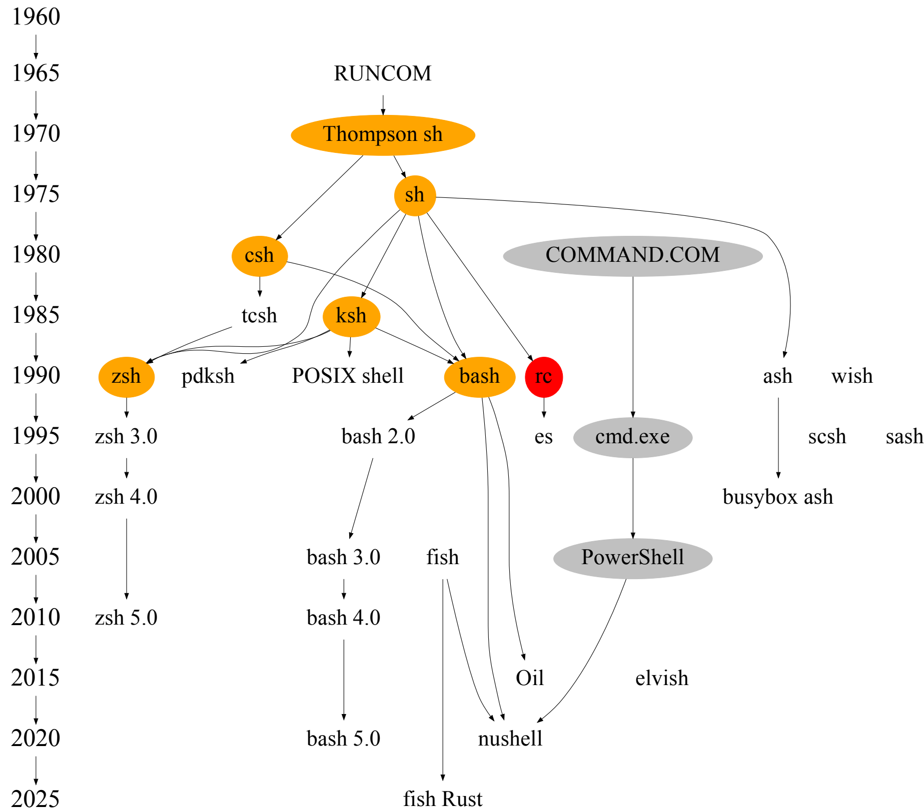

Figure 1.1: Shells timeline

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).

[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.

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). 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
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 7.9.3 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 8 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 6, 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 [Pad16a]).

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

Most of this document is made of source code from Plan 9, so those parts are copyright by Plan 9 Foundation. The prose is mine and is licensed under the All Rights Reserved License.

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 Essential shell features

The following sections will explain the essential features of a shell. 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.

2.1.1 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 13⟩≡
```
#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 12.

- *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.

2.1.2 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`.

2.1.3 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.

2.1.4 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.

---

[3] You can also enter commands on multiple lines, which requires to *escape* the newline as explained in Section 5.4.

[4] `rc` allows to redirect not just the standard input and output, as explained in Section 14.2.6.

2.1.5 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.

2.2 `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 14.1.1 for the code handling `rc -c`).

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

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

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

2.3 boot.rc

In this section, I will present an example of an rc script showing a few features of rc's scripting language. The goal here is to illustrate the general features of a shell you have seen in Section 2.1 with the concrete syntax of rc's scripting language. This will also help you understand the grammar of rc I will present in Chapter 6. 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* 16⟩≡

```
#!/boot/rc -m /boot/rcmain

/boot/echo booooooooting...

path=(/bin /boot)

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

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

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

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

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

2.3.1 #!

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.

2.3.2 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 B.1 and Chapter 11 for details).

2.3.3 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.

2.3.4 `$path`

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

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

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

See Section 7.2.10 for the implementation.

2.3.5 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.

2.3.6 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 5.6).

2.3.7 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 9 for the implementation.

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

2.3.8 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 8 for the full list of builtins.

2.3.9 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.

2.4 Code organization

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

2.5 Software architecture

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

2.5.1 The command-line user interface

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

2.5.2 `rc`'s components

At its core, `rc` is a small compiler paired with a bytecode interpreter. The compiler transforms shell commands through three stages: characters are grouped into tokens (lexing), tokens are assembled into an abstract syntax tree (parsing), and the AST is compiled into a flat array of opcodes. The interpreter then executes these opcodes

```
filesystem           +---------------+ +-------------------+
 interface           |               | |                   |  A
|-- bin              |     Shell     | |command (e.g., ls) |  P
|   |-- cp  +--------+-> stdin       | |      stdin        |  P
|   |-- ls  |      +-+---stdout      | |      stdout       |  S
|   |-- rc  |      | +---------------+ +-------------------+
|   `-- ... |      |------------syscall interface------------
|-- boot    |      | +-------------------------------------+
|-- dev     |      | |               Kernel                |
|   |--  cons <----+ |                                     |  K
|   |         < - - +|  device drivers-+                   |  E
|   |--  draw        |  |              |                   |  R
|   |--  mouse      +|- + >console     |                   |  N
|   `--  ...         |  |              |                   |  E
|-- env          + -|- + -display      |                   |  L
|-- sys              |  |  keyboard- - |                   |
`-- ...          |   |  |  mouse - - -|+ - - - - - - - +   |
                     |  |              |                   |
                 |   |  +-------------++             |     |
                     +-------------------------------------+
                 v                 |              |       D
            + - - - - - - - -                             E
                           |         v              v     V
            |   Monitor      + - - - - - -      + - - - - - -   I
                           |   keyboard |       mouse   |   C
            + - - - - - - - -  + - - - - - -      + - - - - - -   E
                       Terminal                            S
                      (console)
```

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

Function	Ch.	File	Entities	LOC
abstract syntax tree	3	`Ast.ml`	`line`[27a] `cmd`[98] `Ast.value`[105a]	110
opcodes	3	`Opcode.ml`	`codevecX` `Opcode.t`[96f] `operation`[27e]	109
runtime structures	3	`Runtime.ml`	`varX` `globals`[33b] `thread`[29h] `runqX` `cur()`[30d]	233
variable management	3	`Var.ml`	`gvlook()X` `vlook()X` `setvar()X` `init()X`	46
function management	3	`Fn.ml`	`flook()`[30a]	10
entry point	4	`Main.ml`		18
main functions	4	`CLI.ml`	`main()`[106b] `interpret_bootstrap()`[88]	298
read eval print loop	4	`Op_REPL.ml`	`op_REPL()`[38g]	61
lexer	5	`Lexer.mll`	`Lexer.token()`[41] `incr_lineno()`[48a]	170
prompt management	5	`Prompt.ml`	`doprompt`[42f] `prompt`[42i] `pprompt()`[43a]	39
globals	5	`Globals.ml`	`skipnl`[44g] `ifnot`[74d] `errstr`[63d]	33
parser	6	`Parse.ml`	`parse_line()X` `Parse.error()`[48b]	96
grammar	6	`Parser.mly`	`Parser.tokens`[24] `Parser.line()`[50a]	311
opcode generation	7	`Compile.ml`	`compile()`[56a] `outcode_seq()`[58a]	341
opcode interpretation	7	`Interpret.ml`	`interpret_operation()X`	357
simple command interpretation	7	`Op_process.ml`	`op_Simple()`[61f] `forkexec()`[62a]	120
process status	7	`Status.ml`	`setstatus()X` `getstatus()X`	39
process management	7	`Process.ml`	`return()`[57c] `waitfor()`[65a] `exit()`[73b]	81
error management	7	`Error.ml`	`Error.error()`[63c]	18
`$path` management	7	`PATH.ml`	`find_in_path()`[66a]	27
pattern matching	7	`Pattern.ml`	`match_str()`[113e]	12
shell builtins	8	`Builtin.ml`	`dispatch()X` `dochdir()X`	157
shell environment	9	`Env.ml`	`read_environment()X`	2
signal management	10	`Trap.ml`		
initialization (bootstrap)	11	`CLI.ml`	`bootstrap()`[88] `dotcmdsX`	
wildcard matching	12	`Glob.ml`		2
debugging flags	13	`Flags.ml`	`xflag`[91a] `sflag`[91d] `eflag`[92c]	58
here documents	14	`Heredoc.ml`		2
Total				3200

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

using a stack machine model. This architecture is the same as the C version, but the OCaml implementation replaces many hand-written components with tool-generated ones: `ocamllex` generates the lexer from regular expressions, and `ocamlyacc` generates the parser from a grammar specification.

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 `Lexer.token()`[41]). Then, `rc` parses those tokens (with `Parse.parse_line()`[47a]) and builds an abstract syntax tree (AST) of the program (see `Ast.line`[27a] and `Ast.cmd`[98]). Finally, `rc` transforms this tree in a series of bytecodes or opcodes (see `Opcode.opcode`[27d] and `Opcode.operation`[27e]). This is similar to what a compiler such as `5c` does (see Compiler book [Pad16a]), except an opcode here is not an instruction from a concrete machine but from a *virtual machine*.

After this compilation, `rc` goes through the series of opcodes and interprets them. `rc` keeps track of what is

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

Figure 2.2: Data flow diagram of `rc`.

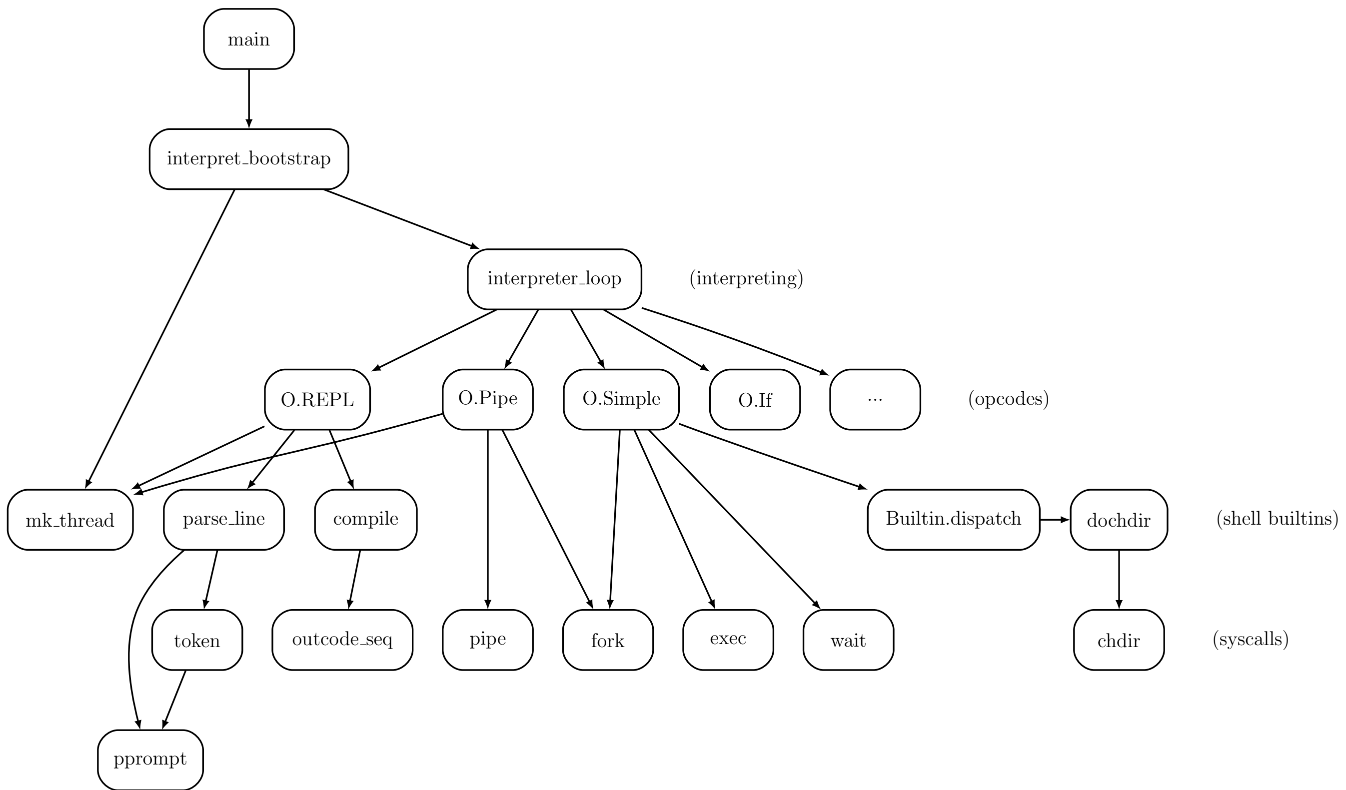

Figure 2.3: Control flow diagram of `rc`.

currently executing in a `thread`[29h] data structure and in a queue `runqX`. This data structure is called a "thread" because `rc` must sometimes manage multiple threads of execution. Indeed, some of the opcodes can create new processes running concurrently (e.g., `O.Pipe()`[72a], the opcode handling pipes).

In fact, `rc` does not start by compiling, but by interpreting. Indeed, `rc` starts first by interpreting some special opcodes, called the *bootstrap*, that contains an opcode (`Opcode.operation.REPL`[95c]) that when interpreted triggers the compiler. I will now explain briefly the control flow of `rc`, starting from the top of Figure 2.3. After some basic initializations, `main()`[106b] calls `interpret_bootstrap()`[88] which first calls a function `bootstrap()`[88] returning the initial set of opcodes to execute. Internally, `interpret_bootstrap()` just modifies the global `runqX` to point to a newly created `thread`, and sets the field `Runtime.thread.code`[29h] to the content of `bootstrap`. `interpret_bootstrap()` then goes in a loop that interprets the opcodes in `runq.code`.

The most important opcode in `bootstrap()` is `Opcode.operation.REPL`, which reads and evaluates a command (hence its name) and starts a new `thread`. `O.REPL()`[38f] first calls the parser `parse_line()X`, which calls the lexer `Lexer.token()` to get the next token. By default, `parse_line()X` will read characters from a `lexbuf` data structure attached to the current thread (in `Runtime.thread.lexbufX`). This `lexbuf` is set initially to be the standard input in `interpret_bootstrap()` but this can be changed if `O.REPL()` is asked at some point to parse a script instead. Once you finished to enter a command with a newline, `parse_line()X` will return and `O.REPL()` will call `compile()`[56a] with the AST it built during parsing as an argument. `compile()` then returns the opcodes deriving from the tree. Then, `O.REPL()` calls `Runtime.mk_thread()`[71c] to start a new thread with the returned opcodes as a parameter. `O.REPL()` then modifies again the global `runqX`, and when `O.REPL()` returns, the main bytecode interpreter loop will process a new series of opcodes stored in `runq.code`.

Note that `O.REPL()` modifies `runqX` by adding new threads but keep the old threads in the queue still. Moreover, `compile()` adds the opcode `Opcode.operation.Return`[27e] at the end of the series of opcodes deriving

from your command. Thus, after the interpreter loop finished interpreting the opcodes of your command, it will process the Return opcode in `O.Return()`[56c], which will modify `runqX` topoint to the old thread, the one containing the bootstrap opcodes. Then, after `O.Return()` returns, the main bytecode interpreter loop will process again the opcodes from the bootstrap, which will read another command through `O.REPL()`.

In `rc`, the opcodes are represented by regular constructors (e.g., `REPL`, `Pipe`, `If`). Thus, the bytecode interpreter is mainly an opcode 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 opcode is `Opcode.operation.Simple`[96c], which is interpreted by the function `O.Simple()`[60c], which `rc` uses to run a "simple" command. `O.Simple()` 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). `O.Simple()` is also responsible for managing the multiple shell builtins. Indeed, if the name of the "simple" command is a builtin (e.g., `cd`), then `O.Simple()` calls `Builtin.dispatch()`[81b] which dispatches the appropriate function (e.g., `dochdir()X`) instead of forking a new process.

2.5.3 Trace of a simple command: `ls /`

You can see the opcodes 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.3 for the full explanation and for the code handling `rc -r`), but you can recognize a few of the opcodes I mentioned before: `O.REPL()`[38f], `O.Simple()`[60c], and `O.Return()`[56c]. I will explain `O.Mark()`[59e] and `O.Word()`[59d] later in this document.

2.6 Book structure

You now have enough background to understand the source code of `rc`. The rest of the book is organized as follows. I will start by describing the core data structures of `rc` in Chapter 3. Then, I will use a top-down approach, starting with Chapter 4 with the description of `main()`[106b], the initialization of `rc`, the bytecode interpreter loop, and the function `O.REPL()`[38f]. The following chapters will describe the main components of the compiler pipeline: Chapter 5 will present the lexer, Chapter 6 the parser, and finally Chapter 7 the opcode generator and opcode interpreter for the main features of `rc`. In Chapter 8, I will present the code of the different shell builtins (e.g., `dochdir()X` for `cd`). Chapter 9 contains the code to manage the environment of the processes launched from the shell, and Chapter 10 the code to manage the signals sent to the processes (also known as *notes* under Plan 9). Chapter 11 presents the actual code initializing `rc`. Indeed, Chapter 4 presents

only a simplified initialization to not introduce too much complexity early-on. The codebase is organized into 35 small OCaml modules totaling about 3,200 lines of code—roughly half the size of the C version (6,700 LOC). OCaml's module system encourages this fine-grained decomposition: each module (e.g., `Ast.ml`, `Opcode.ml`, `Runtime.ml`) has a clear responsibility and a small interface. In the C version, global variables and function pointers are scattered across fewer, larger files.

Then, I will present advanced features of `rc` that I did not present before to simplify the explanations, for instance, wildcard matching (e.g., `ls *.c`) in Chapter 12, logging and other debugging helps 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 opcode generator, and the opcode 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 Finally, Appendix B 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`, following the data flow from Figure 2.2. The first three sections cover the compiler side: tokens, the abstract syntax tree, and opcodes. The remaining sections cover the interpreter side: runtime values (words), variables, functions, and the thread/run queue that drives execution.

3.1 Tokens

The first step of the compilation pipeline is to transform a stream of characters into a stream of tokens. The lexer `Lexer.token()`[41] (described in Chapter 5) is responsible for this transformation.

The token kinds are declared below using the `ocamlyacc` syntax `%token`. They are grouped into categories: words, operators, variables, punctuation, and keywords. Unlike the C version where single-character operators like `';'` or `'='` are returned as raw ASCII values, here every token has an explicit constructor (`TSemicolon`, `TEq`, etc.), which makes the grammar easier to read and the type checker can verify exhaustiveness.

⟨*Parser tokens* 24⟩≡ (48c)

```
/*(*-----------------------------------------*)*/
/*(*2 Word *)*/
/*(*-----------------------------------------*)*/
%token<string * bool(* quoted *)> TWord

/*(*-----------------------------------------*)*/
/*(*2 Operators *)*/
/*(*-----------------------------------------*)*/
%token TPipe
%token<Ast.redirection_kind> TRedir
%token TAndAnd TOrOr TBang
%token TTwiddle
%token TSemicolon
%token TAnd
⟨Parser tokens, operators other cases 95b⟩

/*(*-----------------------------------------*)*/
/*(*2 Variables *)*/
/*(*-----------------------------------------*)*/
%token TEq
%token TDollar
```

⟨*Parser tokens, variables other cases* 95f⟩

```
/*(*-----------------------------------------*)*/
/*(*2 Punctuation *)*/
/*(*-----------------------------------------*)*/
%token TOPar TCPar
%token TOBrace TCBrace
```
⟨*Parser tokens, punctuation other cases* 96d⟩

```
/*(*-----------------------------------------*)*/
/*(*2 Keywords *)*/
/*(*-----------------------------------------*)*/
%token TIf TNot TWhile TSwitch TFor TIn TFn

/*(*-----------------------------------------*)*/
/*(*2 Misc *)*/
/*(*-----------------------------------------*)*/
%token TNewline

%token EOF
```

Most of these tokens are standard, but `TNewline` is the interesting one. In traditional programming languages, newlines are whitespace and ignored by the lexer. In a shell, a newline is *meaningful*: it terminates a command and triggers compilation and execution of whatever the user typed. This is what makes the shell interactive—each line is a complete program that is compiled and run immediately.

3.2 Abstract syntax tree (AST)

The second step of the compilation pipeline transforms the stream of tokens into an abstract syntax tree (AST). The parser `Parse.parse_line()`[47a] (described in Chapter 6) 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 types below.

⟨*type* `Ast.line` 25a⟩≡ (105a)
```
(* None when reads EOF *)
type line = cmd_sequence option
```

⟨*type* `Ast.cmd_sequence` 25b⟩≡ (105a)
```
  and cmd_sequence = cmd list
```

⟨*type* `Ast.cmd` 25c⟩≡ (105a)
```
and cmd =
  | EmptyCommand

  (* Base *)
  | Simple of value * values
  | Pipe of cmd * cmd (* less: lfd, rfd option *)
  | Async of cmd

  (* Redirections *)
  | Redir of cmd * redirection
```
⟨`Ast.cmd` *other redirection cases* 97d⟩
```
  (* expressions *)
  | And of cmd * cmd
  | Or of cmd * cmd
  | Not of cmd
  (* can also run the program 'test' for other comparisons ('[' in bash) *)
```

```
    | Match of value * values

    (* stmts *)
    | If of cmd_sequence * cmd
    (* Note that you can not put a 'cmd option' in If instead of IfNot below.
     * rc has to process 'if(...) cmd\n' now! It can not wait for an else.
     *)
    | IfNot of cmd
    | While of cmd_sequence * cmd
    ⟨Ast.cmd other statement cases 53f⟩

    (* definitions *)
    (* can do x=a; but also $x=b !
     * less: could have AssignGlobal and AssignLocal of ... cmd
     *)
    | Assign of value * value * cmd (* can be EmptyCommand *)
    | Fn of value * cmd_sequence
    ⟨Ast.cmd other definition cases 94b⟩
```

Uses `Ast.cmd` 98, `Ast.redirection_kind`, and `Ast.value` 105a.

This algebraic type makes the full grammar of `rc` visible at a glance. In the C version, the AST is a single `Tree` struct with a `type` field and up to three generic `child` pointers—you must mentally decode which child means what for each node type. Here, each constructor carries exactly the data it needs: `If` takes a condition (a `cmd_sequence`) and a body (a `cmd`), `Pipe` takes two commands, and so on. Note the `If`/`IfNot` split: because `rc` is interactive and must execute commands as soon as a newline is entered, it cannot wait to see if an `else` follows. So `if(test) cmd` and `if not cmd` are separate constructs, processed on separate lines.

⟨*type* `Ast.values` 26a⟩≡ (105a)

```
  (* separated by spaces *)
  and values = value list
```

Uses `Ast.cmd` 98.

In `rc`, there is only one type of value: a list of strings. There are no integers, no booleans, no floats. Even a single word like `hello` is really a one-element list. The `Word` constructor carries a `quoted` flag that records whether the string came from a quoted context (`'\ldots'`), which matters later for globbing: quoted strings are not subject to wildcard expansion.

⟨*type* `Ast.value` 26b⟩≡ (105a)

```
  (* rc does not use types; there is no integer, no boolean, no float.
   * The only value in rc is the list of strings. Even a single
   * string is really a list with one element.
   *)
  type value =
    (* The string can contain the * ? [ special characters.
     * So, even a single word can expand to a list of strings.
     * less: they should be preceded by \001
     * less: W of word_elt list and word_elt = Star | Question | ...Str of string
     *)
    | Word of string * bool (* quoted *)
    | Dollar of value
    | List of values
    ⟨Ast.value other cases 95h⟩
```

Uses `Ast.value` 105a.

⟨*type* `Ast.redirection` 26c⟩≡ (105a)

```
    and redirection = redirection_kind * value (* the filename *)
```

⟨*type* Ast.redirection_kind 27a⟩≡ (105a)
```
      and redirection_kind =
        | RWrite (* > *)
        | RRead  (* < *)
        | RAppend (* > > *)
        ⟨Ast.redirection_kind other cases 97c⟩
```

3.3 Opcodes

The third step of the compilation pipeline transforms the AST into a series of opcodes. The function `compile()`[56a] (described in Chapter 7) is responsible for this transformation.

⟨*signature* Compile.compile 27b⟩≡ (107a)
```
  val compile : Ast.cmd_sequence -> Opcode.codevec
```

⟨*type* Opcode.codevec 27c⟩≡ (112a)
```
  type codevec = t array
```

⟨*type* Opcode.opcode 27d⟩≡ (112a)
```
  type t =
    | F of operation
    (* The int can be a program counter, a file descriptor depending on ctx *)
    | I of int
    (* command name and arguments from the user *)
    | S of string
```

Why compile to bytecodes rather than interpret the AST directly? The flat array representation makes the bytecode interpreter simpler: it is just a loop with a program counter, incrementing through the array. Forward jumps (for `if`, `&&`, `||`) are easy to implement by storing a target index. Backpatching—filling in a jump target after the target code has been emitted—is trivial with array indices but awkward with tree pointers. The `codevec` is essentially a tagged union array: each element is either an opcode (`F`), an integer operand (`I`) for jump targets or file descriptors, or a string operand (`S`) for command names and arguments.

⟨*type* Opcode.operation 27e⟩≡ (112a)
```
  (* Operations generated by the compiler.
   *
   * Semantic of comments below:
   *  - Arguments on stack (...)
   *  - Arguments in line [...]
   *  - Code in line with jump around {...}
   *
   * alt: type operation = ((unit -> unit) * string)
   *)
  type operation =

    (* Stack (argv) *)
    | Mark
    | Word (* [string] *)
    | Popm (* (value) *)
    ⟨Opcode.operation other stack cases 96b⟩

    (* Variable *)
    | Assign (* (name)(val) *)
    | Dollar (* (name) *)
    ⟨Opcode.operation other variable cases 96c⟩

    (* Process! *)
    | Simple (* (args) *)
```

```
  | Async  (* {... Xreturn} *)

  (* Control flow *)
  | If
  | IfNot
  (* While are compiled in jumps *)
  | Jump  (* [addr] *)
  | Exit
  ⟨Opcode.operation other control cases 56b⟩

  (* Boolean return status  *)
  | Wastrue
  | Not
  | False (* {...} *)
  | True  (* {...} *)
  | Match (* (pat, str) *)

  (* Redirections *)
  | Read (* (file)[fd] *)
  | Write (* (file)[fd] *)
  | Append (* (file)[fd] *)
  ⟨Opcode.operation other redirection cases 37d⟩

  (* Pipes *)
  | Pipe (* [i j]{... Xreturn}{... Xreturn} *)
  ⟨Opcode.operation other pipe cases 37e⟩

  (* Functions *)
  | Fn (* (name){... Xreturn } *)

  ⟨Opcode.operation other cases 90a⟩

  | REPL (* *)
```

3.4 Words

⟨*type* Runtime.value 28a⟩≡ (115 114d)

```
(* In rc the basic (and only) value is the list of strings.
 * A single word is considered as a list with one element.
 *)
type value = string list
```

Note that `Runtime.value` is distinct from `Ast.value`. The AST type represents *unevaluated* expressions (a `Dollar` referencing a variable, a `List` of sub-expressions), whereas the runtime type is the *evaluated* result: just a flat `string list`. In OCaml, this distinction is enforced by the type system—you cannot accidentally pass an unevaluated AST value where a runtime value is expected.

3.5 Variables

⟨*type* Runtime.varname 28b⟩≡ (115 114d)

```
(* can be anything: "foo", but also "*", "1", "2", etc *)
type varname = string
```

⟨*type* Runtime.var 28c⟩≡ (115 114d)

```
type var = {
  (* can be None when lookup for a value that was never set before,
```

```
   * which is different from Some [] (when do A=()), which is also
   * different from Some [""] (when do A='').
   *)
  mutable v: value option;
  (* less: opti: changed: bool *)
}
```

The `option` type captures a three-way distinction that the C version must encode with NULL pointers and empty strings: `None` means the variable was never set, `Some []` means it was explicitly set to the empty list (e.g., `A=()`), and `Some [""]` means it was set to a single empty string (e.g., `A=''`). These three states have different semantics in `rc`.

⟨*signature* `Runtime.globals` 29a⟩≡ (114d)
```
  val globals : (varname, var) Hashtbl.t
```

⟨*global* `Runtime.globals` 29b⟩≡ (115)
```
  let globals: (varname, var) Hashtbl.t =
    Hashtbl.create 101
```

⟨*signature* `Var.gvlook` 29c⟩≡ (117b)
```
  (* This will only look for globals (in Runtime.globals) *)
  val gvlook : Runtime.varname -> Runtime.var
```

⟨*function* `Var.gvlook` 29d⟩≡ (117c)
```
  let gvlook (name : R.varname) : R.var =
    try
      Hashtbl.find R.globals name
    with Not_found ->
      let var = { R.v = None } in
      Hashtbl.add R.globals name var;
      var
```

Uses `Runtime.globals` 33b and `Runtime.var.v` 28c.

⟨*signature* `Var.vinit` 29e⟩≡ (117b)
```
  (* Populate Runtime.globals from the current environment *)
  val init : < Cap.env ; .. > -> unit
```

3.6 Functions

⟨*signature* `Runtime.fns` 29f⟩≡ (114d)
```
  val fns : (string, fn) Hashtbl.t
```

⟨*global* `Runtime.fns` 29g⟩≡ (115)
```
  let fns: (string, fn) Hashtbl.t =
    Hashtbl.create 101
```

⟨*type* `Runtime.fn` 29h⟩≡ (115 114d)
```
  type fn = {
    code: Opcode.codevec;
    pc: int;
    (* less: fnchanged *)
  }
```

⟨*signature* `Fn.flook` 29i⟩≡ (109c)
```
  val flook : string -> Runtime.fn option
```

⟨*function* Fn.flook 30a⟩≡ (109d)

```
let flook s =
  try
    Some (Hashtbl.find R.fns s)
  with Not_found -> None
```

Uses Runtime.fns.

3.7 Threads and run queue

⟨*type* Runtime.thread 30b⟩≡ (115 114d)

```
type thread = {
  code: Opcode.codevec;
  pc: int ref;

  mutable argv: string list;
  locals: (varname, var) Hashtbl.t;

  ⟨Runtime.thread other fields 32e⟩
}
```

Uses Runtime.redir and Runtime.waitstatus 32f.

The `argv` field is the operand stack for the bytecode interpreter. Bytecodes like `O.Word`[59d] push strings onto it, and `O.Simple`[60c] consumes the accumulated list as the command's arguments. The name "argv" is apt: the strings accumulated here are ultimately what gets passed to `execv()` as the argument vector of the child process.

Despite the name, a "thread" here is not an OS thread. It is an execution frame inside the interpreter: a code vector, a program counter, a local variable scope, and a stack of arguments. The `pc` field is an `int ref`—a mutable reference to an integer—because bytecodes like `O.True` and `O.Jump` need to modify the program counter directly (for conditional and unconditional jumps). In the C version, `pc` is a plain integer mutated through a pointer. The `runqX` is a list of threads acting as a call stack: when `O.REPL`[38f] compiles a new command, it pushes a new thread; when the command finishes (`O.Return`[56c]), the thread is popped and the interpreter resumes the previous one (ultimately the bootstrap thread, which loops back to `O.REPL`).

⟨*signature* Runtime.runq 30c⟩≡ (114d)

```
val runq : thread list ref
```

⟨*constant* Runtime.runq 30d⟩≡ (115)

```
(* less: could have also  'let cur = { code = boostrap; pc = 1; chan = stdin }'
 * in addition to runq. Then there will be no need for cur ().
 *)
let runq = ref []
```

⟨*signature* Runtime.cur 30e⟩≡ (114d)

```
val cur : unit -> thread
```

⟨*function* Runtime.cur 30f⟩≡ (115)

```
let cur () =
  match !runq with
  | [] -> failwith "empty runq"
  | x::_xs -> x
```

Uses Runtime.cur() 30d, Runtime.thread.argv, and Runtime.thread.argv_stack 38c.

⟨*signature* Runtime.mk_thread 30g⟩≡ (114d)

```
val mk_thread :
  Opcode.codevec -> int -> (varname, var) Hashtbl.t -> thread
```

⟨*function* `Runtime.mk_thread` 31a⟩≡ (115)

```
(* This function was called start(), but it does not really start right
 * away the new thread. So better to call it mk_thread.
 * It starts with pc <> 0 when handle async, traps, pipes, etc.
 *)
let mk_thread code pc locals =
  let t = {
    code = code;
    pc = ref pc;
    argv = [];
    locals = locals;

    ⟨Runtime.mk_thread() set other fields 32g⟩
  } in
  t
```

Uses `Runtime.thread.argv_stack` 38c, `Runtime.thread.file` 36b, `Runtime.thread.iflag`, `Runtime.thread.lexbuf`, and `Runtime.thread.line` 36b.

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

```
runq = [ thread ]

thread:
  code = bootstrap [| Mark; Word; "*"; Assign; ... |]
  pc   = ref 0
  argv = ["myscript.rc"; "foo"; "bar"]
  locals = {}
  waitstatus = NothingToWaitfor
```

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

3.7.1 The stack

The `argv` field is the operand stack for the bytecode interpreter. Bytecodes like `O.Word`[59d] push strings onto it, and `O.Simple`[60c] consumes the accumulated list as the command's arguments. In the C version, this is a two-level linked list (`List` of `Word` lists) requiring `Xmark` to push a new sub-list. Here it is a flat `string list`, which is simpler but means the mark/pop operations must be handled differently.

⟨*signature* `Runtime.push_word` 31b⟩≡ (114d)

```
val push_word : string -> unit
```

⟨*function* `Runtime.push_word` 31c⟩≡ (115)

```
let push_word s =
  let t = cur () in
  t.argv <- s::t.argv
```

⟨*signature* `Runtime.pop_word` 31d⟩≡ (114d)

```
val pop_word : unit -> unit
```

⟨*function* `Runtime.pop_word` 31e⟩≡ (115)

```
let pop_word () =
  let t = cur () in
  match t.argv with
  | [] -> failwith "pop_word but no word!"
  | _x::xs ->
      t.argv <- xs
```

Uses `Runtime.cur()` 30d and `Runtime.thread.redirections` 30b.

3.7.2 Local variables

Each thread has its own hashtable of local variables. These are created by constructs like `for` that bind a variable for the duration of a block. When looking up a variable with `vlook()X`, `rc` first searches the current thread's locals before falling back to the global variable table.

⟨*signature* `Var.vlook` 32a⟩≡ (117b)
```
(* This will look first in the locals (in Runtime.cur().locals) and then
 * in globals (in Runtime.globals)
 *)
val vlook : Runtime.varname -> Runtime.var
```

⟨*function* `Var.vlook` 32b⟩≡ (117c)
```
let vlook name =
  if !Runtime.runq <> []
  then
    let t = Runtime.cur () in
    try
      Hashtbl.find t.R.locals name
    with Not_found ->
      gvlook name
  else gvlook name
```
Uses `Runtime.cur()` 30d, `Runtime.runq`, `Runtime.thread.locals` 115, and `Var.gvlook()`.

⟨*signature* `Var.setvar` 32c⟩≡ (117b)
```
(* Override the content of a (local or global) variable *)
val setvar : Runtime.varname -> Runtime.value -> unit
```

⟨*function* `Var.setvar` 32d⟩≡ (117c)
```
let setvar (name : Runtime.varname) (v : Runtime.value) : unit =
  let var = vlook name in
  var.R.v <- Some v
```
Uses `Runtime.var.v` 28c and `Var.vlook()`.

3.7.3 Redirections

⟨`Runtime.thread` *other fields* 32e⟩≡ (30b) 33a ▷
```
(* things to do before exec'ing the simple command *)
mutable redirections: (redir list) list;
```

The redirections field is a list of lists—essentially a stack of redirection scopes. Each time a redirection opcode (`O.Write`, `O.Read`) executes, it pushes a `FromTo` entry into the current (innermost) scope. When `O.Popredir`[69a] executes, it pops and closes the most recent redirection. When a new thread is created (in `mk_thread`[71c]), a fresh empty scope is pushed on top of the parent's redirections, so child commands can add redirections without affecting the parent's scope.

⟨*type* `Runtime.redir` 32f⟩≡ (115 114d)
```
  and redir =
    (* the file descriptor From becomes To, e.g., /tmp/foo becomes stdout,
     * which means your process output will now go in /tmp/foo.
     *)
    | FromTo of Unix.file_descr (* from *) * Unix.file_descr (* to *)
    | Close of Unix.file_descr
```

⟨`Runtime.mk_thread()` *set other fields* 32g⟩≡ (31a) 33c ▷
```
redirections =
  (match !runq with
  | t::_ts -> []::t.redirections
  | [] -> []::[]
  );
```

3.7.4 Wait status

The last major field in the thread structure tracks the relationship between a thread and an external process. When `O.Pipe`[72a] forks a child, it records the child's PID in `WaitFor` so that `O.PipewaitX` knows which process to wait for. When the child terminates, the status is stored as `ChildStatus`. The algebraic type `waitstatus` makes the three states explicit: `NothingToWaitfor`, `WaitFor of int`, and `ChildStatus of string`. In the C version, this is encoded as a plain `int pid` field where `-1` means "nothing to wait for" and a separate `char status[]` buffer.

⟨`Runtime.thread` *other fields* 33a⟩+≡ (30b) ◁32e 36b▷

```
  (* things to wait for after a thread forked a process *)
  mutable waitstatus: waitstatus;
```

⟨*type* `Runtime.waitstatus` 33b⟩≡ (115 114d)

```
    and waitstatus =
      | NothingToWaitfor
      (* process pid to wait for in Xpipewait (set from Xpipe) *)
      | WaitFor of int
      (* exit status from child process returned from a wait() *)
      | ChildStatus of string
```

⟨`Runtime.mk_thread()` *set other fields* 33c⟩+≡ (31a) ◁32g 36c▷

```
  waitstatus = NothingToWaitfor;
```

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()`[106b], the entry point of `rc`.

4.1 Overview

The `CLI.main`[106b] function performs four tasks: it processes the command-line arguments (using OCaml's `Arg` module), it initializes the shell (environment variables, traps), 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 11.

The `caps` type is an OCaml capability pattern: it declares exactly which OS operations the shell needs (fork, exec, wait, chdir, exit, open). Any function that wants to, say, fork a process must receive a value of type `< Cap.forkew; .. >`—there is no way to call `fork` without one. This makes the security boundary explicit in the type signatures, and makes it easy to audit which parts of the code perform which system calls. In the C version, any function can call any system call since they are all globally available.

⟨*type* `CLI.caps` 34a⟩≡ (106)
```
(* Need:
 *  - fork/exec/wait: obviously as we are a shell
 *  - chdir: for the builtin 'cd'
 *  - env: to ??
 *  - exit: as many commands can abruptely exit 'rc' itself or children
 *    created by 'rc'
 *  - open_in: for '.' that can source and eval scripts
 *
 * alt: could remove Cap.exit and use Exit.ExitCode exn in Process.ml instead
*)
type caps = < Cap.forkew; Cap.chdir; Cap.env; Cap.exit; Cap.open_in >
```

⟨*signature* `CLI.main` 34b⟩≡ (106a)
```
(* entry point (can also raise Exit.ExitCode) *)
val main: <caps; Cap.stdout; Cap.stderr; ..> ->
  string array -> Exit.t
```

⟨*toplevel* `Main._1` 34c⟩≡ (111b)
```
let _ =
  Cap.main (fun (caps : Cap.all_caps) ->
     let argv = CapSys.argv caps in
     Exit.exit caps
        (Exit.catch (fun () ->
            CLI.main caps argv))
    )
```
Uses `CLI.main()` 106b, `Cap.main()`, `CapSys.argv()`, `Exit.catch()`, and `Exit.exit()`.

⟨*function* CLI.main 35a⟩≡ (106b)

```
  let main (caps : <caps; Cap.stdout; Cap.stderr; .. >) (argv : string array) :
      Exit.t =
    let args = ref [] in
    ⟨CLI.main() debugging initializations 91g⟩

    let options = [
      ⟨CLI.main() options elements 36a⟩
    ] |> Arg.align
    in
    (* This may raise ExitCode *)
    Arg_.parse_argv caps argv options (fun t -> args := t::!args) usage;
    ⟨CLI.main() logging initializations 92b⟩
    ⟨CLI.main() CLI action processing 102c⟩
    Var.init caps;
    (* todo: trap_init () *)
    ⟨CLI.main() other initializations 35f⟩
    try
      interpret_bootstrap (caps :> < caps >) (List.rev !args);
      Exit.OK
    with exn ->
      ⟨CLI.main() when exn thrown in interpret() 100c⟩
```

Uses Flags.debugger, Flags.login 36d, Logs.info(), and Logs.set_level().

⟨CLI.main() *when* Failure exn *thrown in* interpret() 35b⟩≡ (100c)

```
  | Failure s ->
      (* useful to indicate that error comes from rc, not subprocess *)
      Logs.err (fun m -> m  "rc: %s" s);
      Exit.Code 1
```

⟨*signature* CLI.interpret_bootstrap 35c⟩≡ (106a)

```
  val interpret_bootstrap : < caps > ->
    string list -> unit
```

4.2 Command-line arguments processing

⟨*constant* CLI.usage 35d⟩≡ (106b)

```
  let usage =
    "usage: rc [-SsriIlxepv] [-c arg] [-m command] [file [arg ...]]"
```

4.2.1 Interactive mode: `rc -i`

Interactive mode determines whether `rc` displays a prompt and handles errors differently (e.g., not exiting on a parse error). The logic is subtle: `rc` is interactive by default when no script file is given and standard input is a terminal. The `-I` flag forces non-interactive mode (useful when piping commands into `rc` from another program).

⟨*constant* Flags.interactive 35e⟩≡ (109b)

```
  (* -i (on by default when detects that stdin is /dev/cons *)
  let interactive = ref false
```

⟨CLI.main() *other initializations* 35f⟩≡ (35a) 84c▷

```
  (* todo:
   * if argc=1 and Isatty then Flags.interactive := true
   *)
```

Uses Common.=~().

⟨CLI.main() options *elements* 36a⟩≡ (35a) 36e▷
```
  "-i", Arg.Set Flags.interactive,
  " interactive mode (display prompt)";
  "-I", Arg.Clear Flags.interactive,
  " non-interactive mode (no prompt)";
```
Uses `Flags.rcmain` 89a.

⟨Runtime.thread *other fields* 36b⟩+≡ (30b) ◁33a 38c▷
```
  (* to display a prompt or not *)
  mutable iflag: bool;
```

⟨Runtime.mk_thread() *set other fields* 36c⟩+≡ (31a) ◁33c 38d▷
```
  iflag = false;
```

The interactive flag is stored per-thread rather than as a global because the dot builtin (`. file`) creates a new thread that reads from a file—it should not display a prompt even if the top-level shell is interactive.

4.2.2 Login mode: `rc -l`

When invoked with the `-l` (login shell) flag, `rc` will source the user's profile (`$home/lib/profile`). In the C version, `rc` also detects login mode when `argv[0]` starts with a dash—a UNIX convention used by the kernel. The OCaml version relies solely on the `-l` flag via `Arg`.

⟨*constant* Flags.login 36d⟩≡ (109b)
```
  (* -l (on by default if argv0 starts with a -) *)
  let login = ref false
```

⟨CLI.main() options *elements* 36e⟩+≡ (35a) ◁36a 89b▷
```
  "-l", Arg.Set Flags.login,
  " login mode (execute ~/lib/profile)";
```

4.3 Bytecode interpreter loop

`interpret_bootstrap`[88] creates the initial thread from the bootstrap bytecodes, pushes it onto `runqX`, populates it with command-line arguments, and enters the infinite interpreter loop. In the C version, these steps are split across several sections of `main()`[106b]—`start()X`, argument pushing, and the interpreter loop are separate. Here they are gathered in one function, which is cleaner.

⟨*function* CLI.interpret_bootstrap 36f⟩≡ (106b)
```
  let interpret_bootstrap (caps : < caps >) (args : string list) : unit =
    let t = R.mk_thread (bootstrap ()) 0 (Hashtbl.create 11) in
    R.runq := t::!R.runq;

    ⟨CLI.interpret_bootstrap() other initializations for t 38a⟩

    while true do
      (* bugfix: need to fetch the current thread each time,
       * as the interpreted code may have modified runq.
       *)
      let t = Runtime.cur () in
      let pc = t.R.pc in
      ⟨CLI.interpret() if rflag 101h⟩
      incr pc;
      (match t.R.code.(!pc - 1) with
      (* opcode dispatch ! *)
      | O.F operation ->  Interpreter.interpret_operation caps operation
      | O.S s -> failwith (spf "was expecting a F, not a S: %s" s)
```

```
    | O.I i -> failwith (spf "was expecting a F, not a I: %d" i)
    );
    (* todo: handle trap *)
  done
[@@profiling]
```

Uses `CLI.interpret_bootstrap()` 88, `Flags.rflag` 101f, `Logs.app()`, `Profiling.profile_code()`, `Runtime.push_word()` 60f, `Runtime.thread.lexbuf`, and `Runtime.thread.pc` 29h.

The interpreter loop is deceptively simple: it fetches the current thread from `runqX`, pre-increments `pc`, then dispatches the opcode at `pc-1`. The pre-increment matters: when an opcode like `O.Word`[59d] reads an inline operand, it finds it at `!pc` (the slot just past the opcode) and increments `pc` again to skip over it. This convention means opcodes never need to worry about advancing past their own slot—that is already done. Note that `runqX` is re-fetched every iteration because the previously interpreted opcode may have modified it (e.g., `O.REPL`[38f] pushes a new thread, `O.Return`[56c] pops one).

⟨*constant* `CLI._bootstrap_simple` 37a⟩≡ (106b)

```
let _bootstrap_simple : O.codevec =
  [| O.F O.REPL |]
```

Uses `Opcode.operation.REPL` 95c and `Opcode.t.F`.

The simplest possible bootstrap is a single-element array containing just `O.REPL`. This is enough to start the read-eval loop: the interpreter executes `REPL`, which reads a command, compiles it, pushes a new thread, and decrements `pc` so that when the new thread finishes, the interpreter returns here and executes `REPL` again. The real `bootstrap()` (in Chapter 11) is more elaborate: it sets `\$*` from the command-line arguments and sources the `rcmain` initialization script.

⟨*signature* `Interpreter.interpret_operation` 37b⟩≡ (110f)

```
(* Interpret one operation. Called from a loop in main(). *)
val interpret_operation :
  < Cap.fork ; Cap.exec ; Cap.wait; Cap.chdir ; Cap.exit ; Cap.open_in; .. > ->
  Opcode.operation ->
  unit
```

⟨*function* `Interpreter.interpret_operation` 37c⟩≡ (110g)

```
let interpret_operation (caps: < Cap.fork; Cap.exec; Cap.chdir; Cap.exit; .. >) op : unit =
  match op with
  ⟨Interpreter.interpret_operation() match operation cases 38f⟩
  | (O.Concatenate|O.Stringify    |O.Index|
     O.Unlocal|
     O.Fn|
     O.For|
     O.Read|O.Append |O.ReadWrite|
     O.Close|O.Dup|O.PipeFd|
     O.Subshell|O.Backquote|O.Async
    ) ->
    failwith ("TODO: " ^ Opcode.show (O.F op))
```

Uses `Opcode.operation.Append` 27e, `Opcode.operation.Async` 96c, `Opcode.operation.Backquote` 94c, `Opcode.operation.Close` 27e, `Opcode.operation.Dup` 27e, `Opcode.operation.Fn` 37e, `Opcode.operation.For` 27e, `Opcode.operation.PipeFd` 37d, `Opcode.operation.Read` 27e, `Opcode.operation.ReadWrite` 27e, `Opcode.operation.Subshell` 92e, `Opcode.operation.Unlocal` 27e, and `Opcode.t.F`.

⟨`Opcode.operation` *other redirection cases* 37d⟩≡ (27e)

```
| ReadWrite (* (file)[fd] *)
| Close  (* [fd] *)
| Dup    (* [fd0 fd1] *)
| Popredir (* *)
```

⟨`Opcode.operation` *other pipe cases* 37e⟩≡ (27e) 73c ▷

```
| PipeFd (* [type]{... Xreturn} *)
```

The command-line arguments are pushed onto the thread's argument stack. They are reversed first because `push_word`[60f] prepends to the list, so reversing ensures the first argument ends up at the top. The bootstrap code will later assign this list to `\$*`. In the C version, `pushlist()X` must be called first to create an empty word list on the stack, and the loop goes backwards. Here, `List.rev` followed by `List.iter push_word` is more idiomatic.

⟨`CLI.interpret_bootstrap()` *other initializations for* `t` 38a⟩≡ (36f) 38b▷
```
(* less: set argv0 *)
args |> List.rev |> List.iter Runtime.push_word;
```

Each thread has its own `lexbufX` (input source), because the dot builtin (`. file`) creates a new thread that reads from a different file. The bootstrap thread reads from `stdin`. In the C version, threads store a raw file descriptor (`cmdfd`) and a filename (`cmdfile`). OCaml's `Lexing.lexbuf` bundles both the input buffer and position tracking into one value.

⟨`CLI.interpret_bootstrap()` *other initializations for* `t` 38b⟩+≡ (36f) ◁38a 38e▷
```
t.R.lexbuf <- Lexing.from_channel stdin;
```

⟨`Runtime.thread` *other fields* 38c⟩+≡ (30b) ◁36b 47c▷
```
(* connected on stdin by default (changed when do '. file') *)
mutable lexbuf: Lexing.lexbuf;
```

⟨`Runtime.mk_thread()` *set other fields* 38d⟩+≡ (31a) ◁36c 47d▷
```
lexbuf = Lexing.from_function (fun _ _ -> failwith "unconnected lexbuf");
```

⟨`CLI.interpret_bootstrap()` *other initializations for* `t` 38e⟩+≡ (36f) ◁38b
```
t.R.iflag <- !Flags.interactive;
```

4.4 Reading commands: `O.REPL()`

`O.REPL`[38f] (called `Xrdcmds` in the C version) is the bytecode that implements the read-eval loop. It calls `Parse.parse_lineX` to read and parse one command, then `Compile.compile`[56a] to generate bytecodes, and pushes a new thread to execute them.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 38f⟩≡ (37c) 56c▷
```
(* *)
| O.REPL -> Op_repl.op_REPL caps ()
```
Uses `Op_repl.op_REPL()` 38g and `Opcode.operation.REPL` 95c.

⟨*function* `Op_repl.op_REPL` 38g⟩≡ (111d)
```
(* was called Xrdcmds *)
let op_REPL (caps : < Cap.exit; ..>) () =
  let t = R.cur () in

  ⟨Op_repl.op_REPL() if sflag 91f⟩
  ⟨Op_repl.op_REPL() set prompt if iflag 43b⟩
  (* less: call Noerror before yyparse *)

  let lexbuf = t.R.lexbuf in
  try
    let cmdseq_opt = Parse.parse_line lexbuf in
    match cmdseq_opt with
    | Some seq ->
        (* should contain an op_return *)
        let codevec = Compile.compile seq in

        let newt = R.mk_thread codevec 0 t.R.locals in
        R.runq := newt::!(R.runq);
```

```
        decr t.R.pc;
        (* when codevec does a op_return(), then interpreter loop
         * in main should call us back since the pc was decremented above
         *)
    ⟨Op_repl.op_REPL() match cmdset_opt other cases 57a⟩

  with Failure s ->
    ⟨Op_repl.op_REPL() when Failure s thrown 39b⟩
```

Uses Compile.compile() 56a, Parse.parse_line(), Runtime.cur() 30d, Runtime.mk_thread() 71c, Runtime.runq, Runtime.thread.lexbuf, Runtime.thread.locals 115, and Runtime.thread.pc 29h.

The `decr t.R.pc` is the key trick that turns the bytecode interpreter into a REPL loop. After `O.REPL` pushes a new thread for the compiled command, it decrements the bootstrap thread's `pc` so it still points at the `REPL` opcode. When the command thread finishes (via `O.Return`[56c]), the interpreter pops it and resumes the bootstrap thread—which re-executes `O.REPL`, reading the next command. This gives us an infinite read-eval loop without any explicit "while" in the shell logic itself.

⟨*signature* Parse.parse_line 39a⟩≡ (112d)

```
val parse_line : Lexing.lexbuf -> Ast.line
```

Error recovery depends on whether the shell is interactive. In interactive mode (`iflag`), a parse error just prints a message and decrements `pc` to loop back for the next command—the user gets another prompt. In non-interactive mode (running a script), the error propagates as an exception, which ultimately causes the shell to exit with a failure status.

⟨Op_repl.op_REPL() *when* Failure s *thrown* 39b⟩≡ (38g)

```
(* todo: check signals  *)
(* less: was doing Xreturn originally *)
if t.R.iflag
then begin
  Logs.err (fun m -> m "%s" s);
  (* go back for next command *)
  decr t.R.pc;
end
else failwith s
```

Uses Logs.err(), Runtime.thread.iflag, and Runtime.thread.pc 29h.

Chapter 5

Lexing

The C version of `rc` has a hand-written lexer (`yylex()X`), with a separate input layer (`getnext()X`, `nextc()X`, `advance()X`) handling character-by-character reading, lookahead, and escaped newlines. In the OCaml version, all of this is replaced by `ocamllex`: the lexer is declared as a set of regular expression rules in `Lexer.mll`, and `ocamllex` generates the lexer automatically. However, one piece of complexity remains in a wrapper function `lexfuncX`: the `skipnl` mechanism and the prompt display, which sit between the raw lexer and the parser.

5.1 lexfunc() skeleton

⟨`Parse.parse_line()` *nested function* `lexfunc` 40a⟩≡ (47a)

```
let lexfunc lexbuf =
  ⟨Parse.lexfunc() possibly print the prompt 42g⟩
  let tok = ref (Lexer.token lexbuf) in
  ⟨Parse.lexfunc() adjustment for skipnl depending on tok read 44j⟩
  ⟨Parse.lexfunc() adjust curtok with tok 47b⟩
  (* todo:
     - handle lastdol
     - handle SUB
     - handle free caret insertion
  *)
  !tok
  ⟨Parse.lexfunc() possibly dump tokens 101a⟩
in
```

⟨*signature* `Lexer.token` 40b⟩≡ (111a)

```
val token: Lexing.lexbuf -> Parser.token
```

⟨*exception* `Lexer.Lexical_error` 40c⟩≡ (111a 40e)

```
exception Lexical_error of string
```

⟨*function* `Lexer.error` 40d⟩≡ (40e)

```
let error s =
  raise (Lexical_error s)
```

5.2 token() skeleton

⟨`Lexer.mll` 40e⟩≡

```
{
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Parser
```

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Limitations compared to rc:
 *  - no unicode support
*)
⟨exception Lexer.Lexical_error 40c⟩
⟨function Lexer.error 40d⟩
⟨function Lexer.incr_lineno 48a⟩
}
(*****************************************************************************)
(* Regexps aliases *)
(*****************************************************************************)
⟨lexer regexp aliases 42a⟩

(*****************************************************************************)
(* Main rule *)
(*****************************************************************************)
⟨rule Lexer.token 41⟩

(*****************************************************************************)
(* Quote rule *)
(*****************************************************************************)
⟨rule Lexer.quote 45e⟩
```

The lexer is written using `ocamllex`, OCaml's lexer generator. Instead of hand-writing a character-by-character state machine (as in the C version's `yylex()X`), the programmer declares regular expression rules and the tool generates the lexer. Each rule consists of a pattern (e.g., `'\textbackslash n'`, `"&&"`) and an action returning a token. The `lexbuf` argument is a mutable buffer managed by `ocamllex` that tracks the current position in the input. The `Lexing.lexeme lexbuf` call extracts the matched string.

⟨*rule* `Lexer.token` 41⟩≡ (40e)

```
rule token = parse

  (* ----------------------------------------------------------------------- *)
  (* Spacing/comments *)
  (* ----------------------------------------------------------------------- *)
  ⟨Lexer.token() space cases 42b⟩
  ⟨Lexer.token() comment cases 42c⟩

  (* ----------------------------------------------------------------------- *)
  (* Symbols *)
  (* ----------------------------------------------------------------------- *)
  ⟨Lexer.token() symbol cases 43e⟩

  (* ----------------------------------------------------------------------- *)
  (* Variables *)
  (* ----------------------------------------------------------------------- *)
  ⟨Lexer.token() variable cases 44c⟩

  (* ----------------------------------------------------------------------- *)
  (* Quoted word *)
  (* ----------------------------------------------------------------------- *)
  ⟨Lexer.token() quoted word cases 45d⟩

  (* ----------------------------------------------------------------------- *)
  (* Keywords and unquoted words *)
  (* ----------------------------------------------------------------------- *)
  ⟨Lexer.token() keywords and unquoted words cases 46⟩
```

```
(* ------------------------------------------------------------------------ *)
| eof { EOF }
| _ (*as c*)   { error (spf "unrecognized character: '%s'" (Lexing.lexeme lexbuf)) }
```

⟨*lexer regexp aliases* 42a⟩≡ (40e) 45f ▷
```
(* less: not used yet, special regexp used with lastdol *)
let idchr = ['a'-'z''A'-'Z''0'-'9''_''*']
```

5.3 Spaces and comments (#)

⟨Lexer.token() *space cases* 42b⟩≡ (41) 42d ▷
```
| [' ''\t']+ { token lexbuf }
```

⟨Lexer.token() *comment cases* 42c⟩≡ (41)
```
| '#' [^'\n']* { token lexbuf }
```

5.4 Newlines and prompts

⟨Lexer.token() *space cases* 42d⟩+≡ (41) ◁ 42b 43c ▷
```
| '\n'       { incr_lineno(); Prompt.doprompt := true; TNewline }
```

⟨*signature* Prompt.doprompt 42e⟩≡ (114b)
```
val doprompt : bool ref
```

⟨*constant* Prompt.doprompt 42f⟩≡ (114c)
```
(* Set to true so the prompt is displayed before the first character is read.
 *
 * Do we need that global? Can we not just call pprompt() explicitely?
 * No because when we get a newline, we know we should display the prompt,
 * but not before finishing executing the command. So the display
 * prompt should be done before the next round of input.
 * less: actually we could do it in the caller of parse_line, in the REPL.
 *)
let doprompt = ref true
```

⟨Parse.lexfunc() *possibly print the prompt* 42g⟩≡ (40a)
```
(* less: could do that in caller? would remove need for doprompt *)
if !Prompt.doprompt
then Prompt.pprompt ();
```
Uses Prompt.doprompt 42f and Prompt.pprompt() 43a.

⟨*signature* Prompt.prompt 42h⟩≡ (114b)
```
val prompt : string ref
```

⟨*constant* Prompt.prompt 42i⟩≡ (114c)
```
let prompt = ref "% "
```

In rc, the \$prompt variable is a two-element list: the first element is the normal prompt (e.g., ;), and the second is displayed for continuation lines (after a backslash-newline). After printing the first prompt, pprompt[43a] switches to the continuation prompt, so the user sees a different prompt while typing multi-line commands.

⟨*signature* Prompt.pprompt 42j⟩≡ (114b)
```
(* !will reset doprompt! *)
val pprompt : unit -> unit
```

⟨*function* `Prompt.pprompt` 43a⟩≡ (114c)

```
  let pprompt () =
    let t = R.cur () in
    if t.R.iflag then begin
      prerr_string !prompt;
      flush stderr;
      ⟨Prompt.pprompt() adjust prompt for next time 43d⟩
    end;
    (* alt: incr t.R.line; this is done in the lexer instead *)
    doprompt := false
```

Uses `Prompt.doprompt` 42f, `Prompt.prompt` 42i, `Runtime.cur()` 30d, and `Runtime.thread.iflag`.

⟨`Op_repl.op_REPL()` *set* `prompt` *if* `iflag` 43b⟩≡ (38g)

```
  (* set promptstr *)
  if t.R.iflag then begin
    let promptv = (Var.vlook "prompt").R.v in
    Prompt.prompt :=
      (match promptv with
      | Some (x::_xs) -> x
      (* stricter? display error message if prompt set but no element?*)
      | Some [] | None -> "% "
      );
  end;
```

Uses `Prompt.prompt` 42i, `Runtime.thread.iflag`, `Runtime.var.v` 28c, and `Var.vlook()`.

A backslash before a newline tells the shell that the command continues on the next line. The lexer consumes both characters silently (returning no token) and displays the continuation prompt so the user knows more input is expected. In the C version, this is handled deep inside `getnext()X`, which returns a space instead of a newline. With `ocamllex`, it is a simple two-character pattern.

⟨`Lexer.token()` *space cases* 43c⟩+≡ (41) ◁42d

```
  | '\\''\n'  {
      incr_lineno ();
      Prompt.pprompt ();
      token lexbuf
  }
```

⟨`Prompt.pprompt()` *adjust* `prompt` *for next time* 43d⟩≡ (43a)

```
  (* set promptstr for the next pprompt() *)
  let promptv = (Var.vlook "prompt").R.v in
  prompt :=
    (match promptv with
    | Some [_x;y] -> y
     (* stricter? display error message if prompt set no 2 elements?*)
    | Some _ | None -> "\t"
    );
```

Uses `Prompt.prompt` 42i, `Runtime.var.v` 28c, and `Var.vlook()`.

5.5 Operators (;, &, |, <, >, $, &&, ||, ...)

In the C version, operators are handled as cases in a large `switch` statement, and multi-character operators like `&&` require a manual lookahead call (`nextis()X`). With `ocamllex`, each operator is simply a pattern—multi-character operators like `&&` and `||` are matched directly by the longest-match rule. Note that after binary operators like `|`, `&&`, and `||`, the lexer sets `skipnl := true` so that newlines on the next line are treated as whitespace (allowing multi-line pipelines).

⟨`Lexer.token()` *symbol cases* 43e⟩≡ (41) 44a▷

```
  | ';'  { TSemicolon }
```

⟨Lexer.token() *symbol cases* 44a⟩+≡ (41) ◁43e 44b▷

```
| "&"  { TAnd }
```

⟨Lexer.token() *symbol cases* 44b⟩+≡ (41) ◁44a 44d▷

```
| ">"  { TRedir Ast.RWrite }
| "<"  { TRedir Ast.RRead }
| ">>" { TRedir Ast.RAppend }
```

⟨Lexer.token() *variable cases* 44c⟩≡ (41) 96a▷

```
| "$"   { TDollar }
```

⟨Lexer.token() *symbol cases* 44d⟩+≡ (41) ◁44b 44e▷

```
| '('  { TOPar }   | ')' { TCPar }
| '{'  { TOBrace } | '}' { TCBrace }
```

⟨Lexer.token() *symbol cases* 44e⟩+≡ (41) ◁44d 44f▷

```
| '='  { TEq }
```

⟨Lexer.token() *symbol cases* 44f⟩+≡ (41) ◁44e 45a▷

```
| "|"  { Globals.skipnl := true; TPipe }
```

⟨*constant* Globals.skipnl 44g⟩≡ (110c)

```
(* This global is used by the lexer and parser to consider certain
 * newlines as regular spaces. Because rc is an interactive
 * interpreter, newline has a special meaning: it terminates a command.
 * However, sometimes we just want to add a newline because the command
 * is too long. Escaping the newline is one way to do it.
 * But when we start to parse 'cmd1 &&' we know that we are expecting
 * more stuff. So after the && and other binary operators  we consider
 * the newline not as a command terminator, but as a space.
 * Also we indent the prompt to let the user know he can still enter
 * more characters.
 *)
let skipnl = ref false
```

⟨Parse.parse_line() *locals and inits* 44h⟩≡ (47a) 44i▷

```
Globals.skipnl := false;
```

⟨Parse.parse_line() *locals and inits* 44i⟩+≡ (47a) ◁44h

```
let got_skipnl_last_round = ref false in
```

⟨Parse.lexfunc() *adjustment for skipnl depending on* tok *read* 44j⟩≡ (40a)

```
if !got_skipnl_last_round then begin
  if !tok = Parser.TNewline
  then begin
    let rec loop () =
      Prompt.pprompt ();
      tok := Lexer.token lexbuf;
      if !tok = Parser.TNewline
      then loop ()
    in
    loop ()
  end;
  got_skipnl_last_round := false;
end;
if !Globals.skipnl
then got_skipnl_last_round := true;
Globals.skipnl := false;
```

Uses Globals.skipnl 44g, Lexer.token(), Parser.token.TNewline, and Prompt.pprompt() 43a.

The `skipnl` mechanism is subtle because it operates with a one-token delay. When the lexer returns `TPipe`, it sets `skipnl := true`. But the *current* token is the pipe, not the newline—we want to skip the *next* newline. So `lexfunc` records the flag in `got_skipnl_last_round` and only acts on the *following* call. On that next call, if the token is a newline, `lexfunc` consumes it silently (and any consecutive newlines), displaying the continuation prompt each time. This is what allows users to write multi-line pipelines like `ls |<newline>grep foo`—the newline after `|` is treated as whitespace rather than as a command terminator.

⟨`Lexer.token()` *symbol cases* 45a⟩+≡ (41) ◁44f 45b▷

```
  | "&&" { Globals.skipnl := true; TAndAnd }
  | "||" { Globals.skipnl := true; TOrOr }
```

⟨`Lexer.token()` *symbol cases* 45b⟩+≡ (41) ◁45a 45c▷

```
  | "!" { TBang }
```

⟨`Lexer.token()` *symbol cases* 45c⟩+≡ (41) ◁45b 95d▷

```
  | "~" { TTwiddle }
```

5.6 Quoted strings ('...')

⟨`Lexer.token()` *quoted word cases* 45d⟩≡ (41)

```
  | "'" { let s = quote lexbuf in TWord (s, true) }
```

Unlike `sh` and `bash`, `rc` has only single quotes—no double quotes. This eliminates the confusing distinction between single-quoted (literal) and double-quoted (interpolated) strings. To include a literal single quote inside a quoted string, you double it: `'it''s'` produces `it's`. The `quote` rule below is a recursive lexer rule: it accumulates characters until it hits a closing quote, handling the `''` escape and multi-line strings (with continuation prompts) along the way.

⟨*rule* `Lexer.quote` 45e⟩≡ (40e)

```
  and quote = parse
    | "'" { "" }
    | "''" { "'" ^ quote lexbuf }
    | "\n" {
        incr_lineno ();
        Prompt.pprompt ();
        "\n" ^ quote lexbuf
      }
    | [^'\'' '\n']+ { let s = Lexing.lexeme lexbuf in s ^ quote lexbuf }
    (* stricter: generate error *)
    | eof { error "unterminated quote" }
```

5.7 Keywords and words (if, for, while, switch, fn, ...)

⟨*lexer regexp aliases* 45f⟩+≡ (40e) ◁42a

```
  (* original: !strchr("\n \t#;&|^$=`'{}()<>", c) && c!=EOF;
   * note that wordchr allows '~', '!', '@', '"', ','
   *)
  let wordchr = [^'\n' ' ' '\t' '#'
                  ';' '&' '|' '^' '$' '='
                  '`' '\''
                 '{''}' '('')' '<''>'
                  ]
```

Keywords are recognized *after* lexing, not during: the lexer first matches a generic "word" (any sequence of `wordchr`), then pattern-matches the resulting string against the keyword list. This is simpler than the C version, which must check keywords character by character during scanning. The `false` in `TWord (s, false)` means "not quoted"—this word came from unquoted input and is therefore subject to globbing expansion.

⟨`Lexer.token()` *keywords and unquoted words cases* 46⟩≡ (41)

```
  | wordchr+ {
    match Lexing.lexeme lexbuf with
    | "if"    -> TIf
    | "while" -> TWhile
    | "for"   -> TFor
    | "in"    -> TIn
    | "not"   -> TNot
    | "switch" -> TSwitch
    | "fn"     -> TFn
    | s -> TWord (s, false)
  }
```

Chapter 6

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.

6.1 parse_line() skeleton

The parser is generated by `ocamlyacc` from a grammar specification in `Parser.mly`. This replaces the hand-written recursive descent parser in the C version with a declarative grammar: each rule directly states what constructs are valid. The trade-off is that error recovery is harder with a generated parser. The `parse_lineX` function wraps the generated parser, providing the lexer callback and handling parse errors. It returns `Some cmd_sequence` for a valid command line, or `None` on end-of-file.

```
⟨function Parse.parse_line 47a⟩≡                                  (113a)
  let parse_line lexbuf =
    let curtok = ref (Parser.EOF, "EOF") in
    ⟨Parse.parse_line() locals and inits 44h⟩
    ⟨Parse.parse_line() nested function lexfunc 40a⟩
    try
      Parser.rc lexfunc lexbuf
      ⟨Parse.parse_line() possibly dump AST 101c⟩
    with
      | Parsing.Parse_error ->
          error "syntax error" !curtok
      | Lexer.Lexical_error s ->
          error (spf "lexical error, %s" s) !curtok
```

Uses `Common.spf()`, `Parse.error()`, `Parser.rc()`, and `Parser.token.EOF`.

```
⟨Parse.lexfunc() adjust curtok with tok 47b⟩≡                      (40a)
  let s = Lexing.lexeme lexbuf in
  curtok := (!tok, s);
```

6.2 Error location reporting

```
⟨Runtime.thread other fields 47c⟩+≡                     (30b) ◁38c 60a▷
  (* for error reporting (None when reading from stdin) *)
  (* less: file has to be mutable? could be a param of start? like chan? *)
  mutable file: Fpath.t option;
  line: int ref;
```

```
⟨Runtime.mk_thread() set other fields 47d⟩+≡            (31a) ◁38d 60b▷
  file = None;
  line = ref 1;
```

⟨*function* Lexer.incr_lineno 48a⟩≡ (40e)
```
(* we could do that in pprompt() too *)
let incr_lineno () =
  let t = Runtime.cur () in
  incr t.Runtime.line
```

⟨*function* Parse.error 48b⟩≡ (113a)
```
let error s (curtok, curtokstr) =
  let t = R.cur () in
  let locstr =
    match t.R.file, t.R.iflag with
    | Some f, false -> spf "%s:%d: " !!f !(t.R.line)
    | Some f, true -> spf "%s: " !!f
    | None, false -> spf "%d: " !(t.R.line)
    | None, true -> ""
  in
  let tokstr =
    match curtok with
    | Parser.TNewline -> ""
    | _ -> spf "token %s: " (curtokstr)
  in
  let str = spf "rc: %s%s%s" locstr tokstr s in

  (* todo: reset globals like lastdol, lastword *)
  (* less: error recovery, skip until next newline *)
  Status.setstatus s;

  (* less: nerror++; *)
  failwith str
```
Uses `Common.spf()`, `Parser.token.TNewline`, `Runtime.cur()` 30d, `Runtime.thread.file` 36b, `Runtime.thread.iflag`, `Runtime.thread.line` 36b, and `Status.setstatus()`.

6.3 Grammar overview

⟨Parser.mly 48c⟩≡
```
%{
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Ast

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function Parser.mk_seq 51f⟩
%}

/*(*************************************************************************)*/
/*(*1 Tokens *)*/
/*(*************************************************************************)*/
⟨Parser tokens 24⟩

/*(*************************************************************************)*/
/*(*1 Priorities *)*/
/*(*************************************************************************)*/
```

```
⟨Parser token priorities 49a⟩

/*(*************************************************************************)*/
/*(*1 Rules type declaration *)*/
/*(*************************************************************************)*/
⟨Parser entry points types 49b⟩

%%

⟨grammar 49c⟩
```

⟨*Parser token priorities* 49a⟩≡ (48c)

```
/*(* from low to high *)*/
%left TIf TWhile TFor TSwitch TCPar TNot
%left TAndAnd TOrOr
%left TBang TSubshell
%left TPipe
%left TCaret
/*(* $$A -> ($ ($ A)) not (($ $) A) *)*/
%right TDollar TCount TStringify
%left TSub
```

⟨*Parser entry points types* 49b⟩≡ (48c)

```
%type <Ast.line> rc
⟨Parser other rule types 50b⟩
%start rc
```

⟨*grammar* 49c⟩≡ (48c)

```
/*(*************************************************************************)*/
/*(*1 line *)*/
/*(*************************************************************************)*/
rc:
  | line TNewline { Some $1 }
  | EOF           { None }

⟨rule Parser.line 50a⟩
⟨other line related rules 51e⟩

/*(*************************************************************************)*/
/*(*1 Command *)*/
/*(*************************************************************************)*/

⟨rule Parser.cmd 50c⟩
⟨rule Parser.simple 50e⟩
⟨other command related rules 53d⟩

/*(*************************************************************************)*/
/*(*1 Word *)*/
/*(*************************************************************************)*/

⟨rule Parser.comword 51a⟩
⟨rule Parser.word 51b⟩
⟨other word related rules 50g⟩

/*(*************************************************************************)*/
/*(*1 Redirection *)*/
/*(*************************************************************************)*/

⟨rule Parser.redir 52i⟩
⟨other redirection related rules 54f⟩
```

```
/*(**************************************************************************)*/
/*(*1 Skipnl hacks *)*/
/*(**************************************************************************)*/
```

⟨*skipnl rules* 53c⟩

The top-level grammar rule `rc` is what makes `rc` interactive: it parses exactly one `line` terminated by `TNewline`, then returns. This means the parser processes one command at a time and gives control back to the interpreter. The `EOF` case returns `None`, signaling the end of input (e.g., the user typed Ctrl-D).

6.4 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. 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.

⟨*rule* `Parser.line` 50a⟩≡ (49c)

```
/*(* =~ stmt *)*/
line:
  | cmd { [$1] }
  ⟨Parser.line other cases 51d⟩
```

⟨*Parser other rule types* 50b⟩≡ (49b) 50d ▷

```
%type <Ast.cmd> cmd
```

⟨*rule* `Parser.cmd` 50c⟩≡ (49c)

```
/*(* =~ expr *)*/
cmd:
  | /*empty*/ { EmptyCommand }
  | simple    {
      let (cmd, args, redirs) = $1 in
      let args = List.rev args in
      let base = Simple (cmd, args) in
      ⟨Parser.cmd adjust base with redis 52j⟩
    }
  ⟨Parser.cmd other cases 52a⟩
```

⟨*Parser other rule types* 50d⟩+≡ (49b) ◁ 50b 50f ▷

```
%type <value * value list * (redirection_kind * value, redirection_kind * int * int) either list> simple
```

⟨*rule* `Parser.simple` 50e⟩≡ (49c)

```
/*(* =~ primary expr *)*/
simple:
  | first       { $1, [], [] }
  | simple word { let (cmd, args, redirs) = $1 in (cmd, $2::args, redirs) }
  ⟨Parser.simple other cases 52h⟩
```

The `simple` rule accumulates arguments in reverse order (each new `word` is prepended with `$2::args`), then the `cmd` rule reverses the list back. This is a standard pattern in `yacc` grammars: left-recursive rules are efficient for LALR parsers but naturally produce reversed lists.

⟨*Parser other rule types* 50f⟩+≡ (49b) ◁ 50d

```
%type <Ast.value> first
```

⟨*other word related rules* 50g⟩≡ (49c) 51c ▷

```
first:
  | comword           { $1 }
  ⟨Parser.first other cases 97l⟩
```

⟨*rule* `Parser.comword` 51a⟩≡ (49c)

```
  comword:
    | TWord                         { Word (fst $1, snd $1) }
    ⟨Parser.comword other cases 55b⟩
```

⟨*rule* `Parser.word` 51b⟩≡ (49c)

```
  word:
    | comword           { $1 }
    | keyword           { Word ($1, false) }
    ⟨Parser.word other cases 97k⟩
```

The three-level word hierarchy (`comword`, `word`, `keyword`) reflects an important design decision in `rc`. A `comword` is what can appear as a *command name* (the first word)—keywords are excluded, so `if` cannot accidentally be treated as a program to execute. A `word` includes keywords, so `if` *can* appear as an argument: `ls if` lists a file named "if". This makes `rc` more flexible than languages where keywords are universally reserved.

⟨*other word related rules* 51c⟩+≡ (49c) ◁50g 52e▷

```
  keyword:
    | TFor { "for" }  | TIn { "in" }
    | TIf { "if" }  | TNot { "not" }
    | TWhile { "while" }
    | TSwitch { "switch" }
    | TFn { "fn" }
    ⟨Parser.keyword other cases 52b⟩
```

6.5 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).

6.5.1 Sequence (;, &)

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`.

⟨`Parser.line` *other cases* 51d⟩≡ (50a)

```
  | cmdsa line { $1 $2  }
```

⟨*other line related rules* 51e⟩≡ (49c)

```
  cmdsa:
    | cmd TSemicolon { (fun x -> mk_Seq ($1, x)) }
    | cmd TAnd       { (fun x -> mk_Seq (Async $1, x)) }
```

The `mk_Seq` helper optimizes away empty commands in sequences. Since every newline produces an `EmptyCommand` in the grammar, a simple `\textbackslash n ls\textbackslash n` block would generate two unnecessary empty nodes. This optimization is important in practice because empty commands are pervasive in interactive use.

⟨*function* `Parser.mk_seq` 51f⟩≡ (48c)

```
  (* This is a useful optimisation as you can get lots of EmptyCommand,
   * for instance, in {\nls\n} you will get 2 EmptyCommand (for each \n)
  *)
  let mk_Seq (a, b) =
    match a, b with
    | EmptyCommand, _ -> b
    | _, [EmptyCommand] -> [a]
    | _ -> a::b
```

6.5.2 Logical operators (&&, ||, !)

⟨Parser.cmd *other cases* 52a⟩≡ (50c) 52c ▷
```
  | cmd TAndAnd cmd { And ($1, $3) }
  | cmd TOrOr cmd   { Or  ($1, $3) }
  | TBang cmd       { Not $2 }
```

⟨Parser.keyword *other cases* 52b⟩≡ (51c) 52d ▷
```
  | TBang { "!" }
```

6.5.3 String matching (~)

The ~ (twiddle) operator is the fundamental comparison operator in `rc`. Since all values are strings (or lists of strings), there are no arithmetic comparisons; instead, ~ performs pattern matching: `~ $x foo*` tests whether `\$x` matches the pattern `foo*`. It can also be used 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 `[ ]`).

⟨Parser.cmd *other cases* 52c⟩+≡ (50c) ◁ 52a 52g ▷
```
  | TTwiddle word words { Match ($2, $3) }
```

⟨Parser.keyword *other cases* 52d⟩+≡ (51c) ◁ 52b 95e ▷
```
  | TTwiddle { "~" }
```

⟨*other word related rules* 52e⟩+≡ (49c) ◁ 51c 52f ▷
```
 words: words_rev { List.rev $1 }
```

⟨*other word related rules* 52f⟩+≡ (49c) ◁ 52e
```
 words_rev:
   | /*empty*/  { [] }
   | words_rev word { $2::$1 }
```

6.5.4 Pipe (|)

⟨Parser.cmd *other cases* 52g⟩+≡ (50c) ◁ 52c 53a ▷
```
  | cmd TPipe cmd  { Pipe ($1, $3) }
```

6.5.5 Redirections (>, <)

Redirections are at the `simple` level in the grammar, which means they bind tighter than pipes and logical operators. For example, `ls | grep foo > out` is parsed as `ls | (grep foo > out)`, not `(ls | grep foo) > out`—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 foo} > out`.

⟨Parser.simple *other cases* 52h⟩≡ (50e)
```
  | simple redir { let (cmd, args, redirs) = $1 in (cmd, args, $2::redirs) }
```

⟨*rule* Parser.redir 52i⟩≡ (49c)
```
 redir:
   | TRedir word { Left ($1, $2) }
   ⟨Parser.redir other cases 97b⟩
```

⟨Parser.cmd *adjust* base *with* redis 52j⟩≡ (50c)
```
 let redirs = List.rev redirs in
 redirs |> List.fold_left (fun acc e ->
   match e with
   | Left (kind, word)    -> Redir (acc, (kind, word))
   | Right (kind, fd0, fd1) -> Dup (acc, kind, fd0, fd1)
 ) base
```

For example, `grep pattern < input > output` produces this AST:

```
Redir (RWrite, "output")
 |
Redir (RRead, "input")
 |
Simple ("grep", ["pattern"])
```

The `fold_left` wraps each redirection around the command, outermost last. This means the bytecode generator encounters the redirections first and sets them up before executing the command—the same tree shape that the C version achieves via a post-hoc `simplemung()X` transformation, but here it falls out naturally from the fold.

6.6 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 execute it immediately. It cannot look ahead to see if the next line starts with `else`. The Bourne shell solves this with `if...then...fi`: the explicit `fi` tells the parser when the `if` block ends. `rc` takes a simpler approach: `if not` is a completely separate command that checks the exit status left by the previous `if`. No lookahead, no extra keywords, and each line is self-contained. The `for` loop has two forms: `for(x in a b c)` iterates over an explicit list, and `for(x)` iterates over `\$*` (the script arguments).

⟨`Parser.cmd` *other cases* 53a⟩+≡ (50c) ◁52g 53b▷

```
  | TIf paren_skipnl cmd { If ($2, $3) }
  | TIf tnot_skipnl  cmd { IfNot $3 }
```

⟨`Parser.cmd` *other cases* 53b⟩+≡ (50c) ◁53a 53e▷

```
  | TWhile paren_skipnl cmd   { While ($2, $3) }
```

⟨*skipnl rules* 53c⟩≡ (49c) 53i▷

```
 paren_skipnl: paren { Globals.skipnl := true; $1 }
 tnot_skipnl: TNot   { Globals.skipnl := true; }
```

⟨*other command related rules* 53d⟩≡ (49c) 54c▷

```
 paren: TOPar body TCPar { $2 }
```

⟨`Parser.cmd` *other cases* 53e⟩+≡ (50c) ◁53b 53g▷

```
  | TSwitch word_skipnl brace { Switch ($2, $3) }
```

⟨`Ast.cmd` *other statement cases* 53f⟩≡ (25c) 53h▷

```
  | Switch of value * cmd_sequence
```

⟨`Parser.cmd` *other cases* 53g⟩+≡ (50c) ◁53e 54a▷

```
 /*(* I added the %prec TFor. *)*/
  | TFor TOPar word TIn words tcpar_skipnl cmd %prec TFor { ForIn ($3, $5, $7) }
  | TFor TOPar word tcpar_skipnl cmd           %prec TFor { For ($3, $5) }
```

⟨`Ast.cmd` *other statement cases* 53h⟩+≡ (25c) ◁53f 54b▷

```
  | ForIn of value * values * cmd
  (* less: could desugar as ForIn value $* *)
  | For of value * cmd
```

Uses `Ast.cmd` 98 and `Ast.value` 105a.

⟨*skipnl rules* 53i⟩+≡ (49c) ◁53c

```
 word_skipnl: word   { Globals.skipnl := true; $1 }
 tcpar_skipnl: TCPar { Globals.skipnl := true; }
```

6.7 Grouping ({...})

```
⟨Parser.cmd other cases 54a⟩+≡                                  (50c) ◁53g 54h▷
  | brace epilog   {
      let cmd = (Compound $1) in
      ⟨Parser.cmd in brace case, adjust cmd with epilog 54g⟩
  }
```

```
⟨Ast.cmd other statement cases 54b⟩+≡                                (25c) ◁53h
  | Compound of cmd_sequence
```

Uses Ast.value 105a.

```
⟨other command related rules 54c⟩+≡                              (49c) ◁53d 54d▷
  brace: TOBrace body TCBrace { $2 }
```

```
⟨other command related rules 54d⟩+≡                              (49c) ◁54c 54e▷
  body:
    | cmd         { [$1] }
    | cmdsan body { $1 $2 }
```

```
⟨other command related rules 54e⟩+≡                              (49c) ◁54d 55a▷
  cmdsan:
    | cmdsa        { $1 }
    | cmd TNewline { (fun x -> mk_Seq ($1, x)) }
```

```
⟨other redirection related rules 54f⟩≡                                     (49c)
  epilog:
    | /*empty*/    { [] }
    | redir epilog { $1::$2 }
```

```
⟨Parser.cmd in brace case, adjust cmd with epilog 54g⟩≡                    (54a)
  $2 |> List.fold_left (fun acc e ->
    match e with
    | Left (kind, word) -> Redir (acc, (kind, word))
    | Right (kind, fd0, fd1) -> Dup (acc, kind, fd0, fd1)
  )  cmd
```

6.8 Functions (fn <f> ...)

Function definition in `rc` uses `fn name {body}`. In the C version, `fn name` without a body *deletes* the function, and `words` (not just a single word) is accepted so you can define the same handler for multiple signals at once (e.g., `fn sigint sighup {...}`).

```
⟨Parser.cmd other cases 54h⟩+≡                                  (50c) ◁54a 54i▷
  /*(* stricter: allow only word, not words *)*/
  | TFn word brace { Fn ($2, $3) }
```

6.9 Variables (<x> = ..., $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 (just a newline), the assignment is global.

```
⟨Parser.cmd other cases 54i⟩+≡                                  (50c) ◁54h 94d▷
  | assign cmd %prec TBang { $1 $2 }
```

⟨*other command related rules* 55a⟩+≡ (49c) ◁54e

```
  assign: first TEq word { (fun x -> Assign ($1, $3, x))  }
```

⟨`Parser.comword` *other cases* 55b⟩≡ (51a) 55c▷

```
  | TDollar word                    { Dollar $2 }
```

6.10 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.

⟨`Parser.comword` *other cases* 55c⟩+≡ (51a) ◁55b 96h▷

```
  | TOPar words TCPar               { List $2 }
```

Chapter 7

Opcode Generation and Interpretation

7.1 Overview

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.

7.1.1 compile()

The `compile`[56a] function encapsulates the code buffer and emission functions in local closures. The `emit` closure appends an opcode to the growing array (doubling it when full), and the `set` closure patches a previously emitted slot—used for backpatching forward jump targets. The `idx` ref tracks the current write position. Unlike the C version which uses a global `codebuf`, the OCaml version keeps the code buffer local to `compile`, returning only the trimmed result array. The `emit` and `set` closures capture the buffer by reference, avoiding the need for globals.

```
⟨function Compile.compile 56a⟩≡                                          (107b)
  let compile (seq : Ast.cmd_sequence) : Opcode.codevec =

    (* a growing array *)
    let codebuf = ref [| |] in
    let len_codebuf = ref 0 in
    (* pointer in codebuf *)
    let idx = ref 0 in

    ⟨Compile.compile() nested function emit 57d⟩
    ⟨Compile.compile() nested function set 57f⟩

    outcode_seq seq !Flags.eflag (emit, set, idx);
    emit (O.F O.Return);
    (* less: O.F O.End *)
    (* less: heredoc, readhere() *)

    (* return the trimmed array *)
    Array.sub !codebuf 0 !idx
    ⟨Compile.compile() possibly dump returned opcodes 101e⟩
```

Uses `Compile.outcode_seq()` 58a, `Flags.eflag` 92c, `Opcode.operation.Return` 27e, and `Opcode.t.F`.

```
⟨Opcode.operation other control cases 56b⟩≡                        (27e) 75b▷
  | Return
```

```
⟨Interpreter.interpret_operation() match operation cases 56c⟩+≡    (37c) ◁38f 59d▷
  | O.Return -> Process.return caps ()
```

⟨Op_repl.op_REPL() *match* cmdset_opt *other cases* 57a⟩≡ (38g)
```
  | None -> Process.return caps ()
```
Uses Process.return() 57c.

⟨*signature* Process.return 57b⟩≡ (113g)
```
  val return : < Cap.exit ; .. > -> unit -> unit
```

⟨*function* Process.return 57c⟩≡ (114a)
```
  (* Was called Xreturn but called not only from the opcode interpreter.
   * It is an helper function really.
   *)
  let return (caps : < Cap.exit; .. >) () =
    ⟨Process.return() initializations 70f⟩
    match !R.runq with
    | [] -> failwith "empty runq"
    (* last thread in runq, we exit then *)
    | [_x] -> exit caps (Status.getstatus ())
    | _x::xs ->
        R.runq := xs
```
Uses Process.exit() 73b, Runtime.runq, and Status.getstatus().

Process.return is the interpreter's equivalent of "return from a function": it pops the current thread from runqX, resuming the previous one. If only one thread remains (the bootstrap thread), it means we have reached the outermost level and the shell exits. Before popping, turf_redir[70e] closes any redirections that the finishing thread had opened, restoring the original file descriptor state—a bracket pattern (open before, close after).

⟨Compile.compile() *nested function* emit 57d⟩≡ (56a)
```
  let emit x =
    ⟨Compile.compile() nested function emit possibly grow codebuf 57e⟩
    !codebuf.(!idx) <- x;
    incr idx
  in
```

⟨Compile.compile() *nested function* emit *possibly grow* codebuf 57e⟩≡ (57d)
```
  (* grow the array if needed *)
  if !idx = !len_codebuf then begin
    len_codebuf := !len_codebuf + 100;
    codebuf := Array.append !codebuf (Array.make 100 (O.I 0));
  end;
```
Uses Opcode.t.I 27e.

⟨Compile.compile() *nested function* set 57f⟩≡ (56a)
```
  let set idx2 x =
    ⟨Compile.compile() nested function set array bound checking 57g⟩
    !codebuf.(idx2) <- x;
  in
```

⟨Compile.compile() *nested function* set *array bound checking* 57g⟩≡ (57f)
```
  if idx2 < 0 || idx2 >= !len_codebuf
  then failwith (spf "Bad address %d in set()" idx2);
```
Uses Common.spf().

7.1.2 outcode_seq() skeleton

The code generator is structured as three mutually recursive functions that mirror the AST: xseq handles command sequences, xcmd handles individual commands, and xword handles values. The mutual recursion is necessary because commands can contain values (e.g., the arguments of Simple) and values can contain

commands (e.g., `CommandOutput`, a.k.a. backquote). Each function walks its part of the AST and emits opcodes into the shared code buffer via the `emit` closure.

⟨*function* `Compile.outcode_seq` 58a⟩≡ (107b)
```
  let outcode_seq (seq : Ast.cmd_sequence) eflag (emit,set,idx) : unit =

    let rec xseq (seq : Ast.cmd_sequence) eflag : unit =
     ⟨Compile.outcode_seq in nested xseq() 66c⟩

    and xcmd (cmd : Ast.cmd) eflag : unit =
      match cmd with
      ⟨Compile.outcode_seq in nested xcmd() match cmd cases 58b⟩
      | (A.Async _|
         A.Dup (_, _, _, _)|
         A.While (_, _)|
         A.ForIn (_, _, _)|
         A.For (_, _)
         )
         -> failwith ("TODO compile: " ^ Ast.show_cmd cmd)

   (* Do we need to pass eflag here too?
    * Even though types are mutually recursive because of Backquote, the
    * compilation of backquote does not use eflag!
    *)
    and xword (w : Ast.value) : unit =
      match w with
      ⟨Compile.outcode_seq in nested xword() match w cases 59a⟩
      | (A.CommandOutput _|
         A.Index (_, _)|
         A.Concat (_, _)|
         A.Stringify _
        )
         -> failwith ("TODO compile: " ^ Ast.show_value w)

    and xwords (ws : Ast.value list) : unit =
      ⟨Compile.outcode_seq in nested xwords() 59b⟩
    in
    xseq seq eflag
```
Uses `Ast.cmd.Async` 26a, `Ast.cmd.Dup` 25c, `Ast.cmd.For` 25c, `Ast.cmd.ForIn` 25c, `Ast.cmd.While` 25c, `Ast.value.CommandOutput` 26b, `Ast.value.Concat`, `Ast.value.Index` 26b, and `Ast.value.Stringify` 26b.

7.2 Simple commands

7.2.1 Opcode generation

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 58b⟩≡ (58a) 66d ▷
```
  | A.Simple (w, ws) ->
      emit (O.F O.Mark);
      xwords ws;
      xword w;
      emit (O.F O.Simple);
      ⟨Compile.outcode_seq in A.Simple case after emit O.Simple 92f⟩
```
Uses `Ast.cmd.Simple`, `Opcode.operation.Mark` 27e, `Opcode.operation.Simple` 96c, and `Opcode.t.F`.

The opcode sequence for a simple command like `ls /etc` is: `Mark`, `Word "/etc"`, `Word "ls"`, `Simple`. `Mark` pushes a new scope on the argument stack (via `push_listX`), each `Word` pushes a string onto it, and `Simple` consumes the accumulated list as the command's argv. Note that arguments are emitted in reverse order:

xwords reverses the list before emitting, and the command name w is emitted last (after the arguments). Since each Word *prepends* to the stack, the final order is correct: the command name ends up first.

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

```
AST:                         codevec:

Simple                       Mark
 ("echo",                    Word "world"
  ["hello"; "world"])         Word "hello"
                             Word "echo"
                             Simple
```

The OCaml AST is simpler than the C version: no intermediate ARGLIST tree—just a constructor with a flat argument list. The reverse-order emission comes from xwords calling List.rev before iterating.

⟨Compile.outcode_seq *in nested* xword() *match* w *cases* 59a⟩≡ (58a) 59c ▷
```
  | A.Word (s, _quoted) ->
      emit (O.F O.Word);
      emit (O.S s);
```
Uses Ast.value.Word 26b, Opcode.operation.Word 27e, Opcode.t.F, and Opcode.t.S 27e.

7.2.2 Stack management

⟨Compile.outcode_seq *in nested* xwords() 59b⟩≡ (58a)
```
  ws |> List.rev |> List.iter (fun w -> xword w);
```

⟨Compile.outcode_seq *in nested* xword() *match* w *cases* 59c⟩+≡ (58a) ◁ 59a 79b ▷
```
  | A.List ws ->
      xwords ws
```
Uses Ast.value.List 26b.

⟨Interpreter.interpret_operation() *match* operation *cases* 59d⟩+≡ (37c) ◁ 56c 59e ▷
```
  (* [string] *)
  | O.Word ->
      let t = R.cur () in
      let pc = t.R.pc in
      let x = t.R.code.(!pc) in
      incr pc;
      (match x with
      | O.S s -> R.push_word s
      (* stricter: but should never happen *)
      | op -> failwith (spf "was expecting a S, not %s" (Opcode.show op))
      )
```
Uses Common.spf(), Opcode.operation.Word 27e, Opcode.t.S 27e, Runtime.cur() 30d, Runtime.push_word() 60f, Runtime.thread.code 29h, and Runtime.thread.pc 29h.

⟨Interpreter.interpret_operation() *match* operation *cases* 59e⟩+≡ (37c) ◁ 59d 60c ▷
```
  | O.Mark -> R.push_list ()
```

⟨*signature* Runtime.push_list 59f⟩≡ (114d)
```
  val push_list : unit -> unit
```

⟨*function* Runtime.push_list 59g⟩≡ (115)
```
  let push_list () =
    let t = cur () in
    t.argv_stack <- t.argv :: t.argv_stack;
    t.argv <- []
```
Uses Runtime.thread.argv.

⟨Runtime.thread *other fields* 60a⟩+≡ (30b) ◁47c
```
(* Used for switch but also assignments. *)
mutable argv_stack: (string list) list;
```

⟨Runtime.mk_thread() *set other fields* 60b⟩+≡ (31a) ◁47d
```
argv_stack = [];
```

The `argv_stack` is the "rest of the stack" below the current working area. When `O.Mark`[59e] calls `push_listX`, the current `argv` is saved onto `argv_stack` and a fresh empty list becomes the new `argv`. When `O.Simple`[60c] or `O.Assign`[78a] calls `pop_list`[115], the process reverses: the current `argv` is consumed and the previous one is restored from `argv_stack`. This two-level structure lets the interpreter handle nested expressions like `echo \$x` where the variable expansion and the command arguments each need their own scope.

7.2.3 O.Simple()

`O.Simple`[60c] is the central bytecode: it takes the word list accumulated by `O.Mark`[59e] and `O.Word`[59d], and executes it as a command. It checks the first word (`argv0`) against three cases in order: builtins (like `cd`, `exit`, `.`), then user-defined functions (TODO in this version), and finally external programs via `forkexec`[62a].

⟨Interpreter.interpret_operation() *match* operation *cases* 60c⟩+≡ (37c) ◁59e 67a▷
```
(* (args) *)
| O.Simple -> Op_process.op_Simple caps ()
```
Uses `Op_process.op_Simple()` 61f and `Opcode.operation.Simple` 96c.

⟨*function* Op_process.op_Simple 60d⟩≡ (111c)
```
let op_Simple (caps : < Cap.fork; Cap.exec; Cap.wait; Cap.chdir; Cap.exit; ..>) () =
  let t = R.cur () in
  let argv = t.R.argv in
  ⟨Op_process.op_Simple() possibly dump command 91c⟩
  match argv with
  ⟨Op_process.op_Simple() match argv empty case 63a⟩
  | argv0::args ->
      match argv0 with
      ⟨Op_process.op_Simple() match argv0 builtin cases 61b⟩
      | _ ->
        ⟨Op_process.op_Simple() when default case, before the fork 61a⟩
        (try
          let pid = forkexec caps () in
          R.pop_list ();
          ⟨Op_process.op_Simple() when default case, after the fork 64g⟩
        with
          | Failure s ->
              E.error caps ("try again: " ^ s)
          | Unix.Unix_error (err, s1, s2) ->
              E.error caps (Process.s_of_unix_error err s1 s2)
)
```
Uses `Error.error()` 63c, `Process.s_of_unix_error()` 63f, `Process.waitfor()` 65a, `Process.waitfor_result.WaitforInterrupted` 64i, and `Runtime.pop_list()` 115.

⟨*signature* Runtime.pop_list 60e⟩≡ (114d)
```
val pop_list : unit -> unit
```

⟨*function* Runtime.pop_list 60f⟩≡ (115)
```
let pop_list () =
  let t = cur () in
  match t.argv_stack with
  (* At the very beginning we do *=(argv) in bootstrap.
   * In that case, Assign will generate two pop_list, but for
```

```
     * the second one argv_stack becomes empty. The only thing
     * we must do is to empty argv then.
     *)
    | [] ->
       t.argv <- []
    | x::xs ->
        t.argv_stack <- xs;
        t.argv <- x
```
Uses Runtime.cur() 30d and Runtime.thread.argv.

⟨Op_process.op_Simple() *when default case, before the fork* 61a⟩≡ (60d)
```
  (* if exitnext opti *)
  flush stderr;
  (* less: Updenv *)
```

7.2.4 Builtin dispatch

⟨Op_process.op_Simple() *match* argv0 *builtin cases* 61b⟩≡ (60d) 61e▷
```
  | s when Builtin.is_builtin s -> Builtin.dispatch caps argv0
```
Uses Builtin.dispatch().

⟨*signature* Builtin.is_builtin 61c⟩≡ (105b)
```
  val is_builtin : string -> bool
```

⟨*signature* Builtin.dispatch 61d⟩≡ (105b)
```
  (* execute the builtin *)
  val dispatch : < Cap.chdir ; Cap.exit ; Cap.open_in; .. > -> string -> unit
```

⟨Op_process.op_Simple() *match* argv0 *builtin cases* 61e⟩+≡ (60d) ◁61b
```
  (* todo: if argv0 is a function *)
  | "builtin" ->
      (match args with
      | [] ->
          Logs.err (fun m -> m "builtin: empty argument list");
          Status.setstatus "empty arg list";
          R.pop_list ()
      | argv0::_args ->
          R.pop_word ();
          Builtin.dispatch caps argv0
      )
```
Uses Op_process.forkexec() 62a and Runtime.pop_list() 115.

7.2.5 Fork

This is the essence of a shell: the fork/exec/wait trinity. `forkexec`[62a] forks a new process; in the child, it calls `exec`[62b] which applies any pending redirections (`doredir`[70e]) and then calls `execv()` to replace the child process with the target program. The parent receives the child's `pid` and later calls `waitfor`[65a] to block until the child terminates. A minimal shell could be just these 50 lines. Everything else in `rc`—pipes, redirections, variables, control flow—is layered on top of this fundamental pattern.

⟨*function* Op_process.forkexec 61f⟩≡ (111c)
```
  let forkexec (caps : < Cap.fork; Cap.exec; Cap.exit; .. >) () : int =
    let pid = CapUnix.fork caps () in
    (* child *)
    if pid = 0
    then begin
      (* less: clearwaitpids *)
```

```
      (* less: could simplify and remove this word if exec was not a builtin *)
      R.push_word "exec";
      exec caps ();
      (* should not be reached, unless prog could not be executed *)
      Process.exit caps ("can't exec: " ^ !Globals.errstr);
      0
    end
    else
      (* parent *)
      (* less: addwaitpid *)
      pid
```

Uses Flags.xflag 91a, Logs.app(), Runtime.cur() 30d, and Runtime.thread.argv.

7.2.6 Exec

⟨*function* Op_process.exec 62a⟩≡ (111c)

```
  let exec (caps : < Cap.exec; Cap.exit; .. >) () : unit =
    R.pop_word (); (* "exec" *)

    let t = R.cur () in
    let argv = t.R.argv in
    match argv with
    | [] -> E.error caps "empty argument list"
    | prog::_xs ->
        ⟨Op_process.exec() before execute 71a⟩
        execute caps argv (PATH.var_PATH_for_cmd_opt prog);
        (* should not be reached, unless prog could not be executed *)
        R.pop_list ()
```

Uses CapUnix.fork(), Op_process.execute(), PATH.search_path_for_cmd() 112c, Runtime.pop_list() 115, and Runtime.push_word() 60f.

⟨*function* Op_process.execute 62b⟩≡ (111c)

```
  let execute (caps : <Cap.exec; ..>) (args : string list) (var_PATH : Fpath.t list option) =

    let argv = Array.of_list args in
    let arg0 = argv.(0) in
    let errstr = ref "" in

    (* less: Updenv () *)
    let final_path : Fpath.t =
      match var_PATH with
      | None -> Fpath.v arg0
      | Some xs ->
          try
            let dir = PATH.find_dir_with_cmd_in_PATH arg0 xs in
            dir / arg0
          with Not_found ->
              Logs.err (fun m -> m "could not find %s in $path" arg0);
              Fpath.v arg0
    in
    (try
        CapUnix.execv caps !!final_path argv |> ignore
      with Unix.Unix_error (err, s1, s2) ->
       errstr := Process.s_of_unix_error err s1 s2;
       Globals.errstr := s2
    );
    (* reached only when could not find a path *)
    Logs.err (fun m -> m "%s: %s" argv.(0) !errstr)
```

Uses CapUnix.execv(), Error.error() 63c, Globals.errstr 63d, Logs.err(), Process.s_of_unix_error() 63f, Runtime.cur() 30d, Runtime.pop_word() 60f, and Runtime.thread.argv.

7.2.7 Error management

⟨Op_process.op_Simple() *match* argv *empty case* 63a⟩≡ (60d)
```
(* How can you get an empty list as Simple has at least one word?
 * If you do A=()\n and then $A\n then Simple has a word, but after
 * expansion the list becomes empty.
 * stricter: I give extra explanations
 *)
| [] -> E.error caps "empty argument list (after variable expansion)"
```
Uses Logs.err().

⟨*signature* Error.error 63b⟩≡ (108c)
```
(* Will behave like a 'raise'. Will Return until you reach
 * the interactive thread and set the status to the error argument.
 * Mostly used by the builtins.
 *)
val error : < Cap.exit ; .. > -> string -> unit
```

Error.error[63c] is the interpreter's exception mechanism. When a runtime error occurs (e.g., a command not found, a bad redirection), it logs the error, sets the status to "error", and then *loops* calling Process.return[57c] to pop threads from the run queue until it reaches one with iflag set (the interactive REPL thread). This effectively unwinds the interpreter's call stack, much like throwing an exception that is caught at the interactive prompt level. Interestingly, this does not use OCaml's native exception mechanism. The unwinding is done manually by popping threads, matching the C version's approach.

⟨*function* Error.error 63c⟩≡ (109a)
```
(* less: error1 similar to error but without %r *)
let error (caps: < Cap.exit; ..>) (s : string) =
  (* less: use argv0 *)
  (* less: use %r *)
  Logs.err (fun m -> m "rc: %s" s);

  Status.setstatus "error";

  while (R.cur ()).R.iflag do
    (* goes up the call stack, like when we have an exception *)
    Process.return caps ();
  done
```
Uses Logs.err(), Process.return() 57c, Runtime.cur() 30d, Runtime.thread.iflag, and Status.setstatus().

⟨*constant* Globals.errstr 63d⟩≡ (110c)
```
(* to mimic Plan 9 errstr() *)
let errstr = ref ""
```

⟨*signature* Process.s_of_unix_error 63e⟩≡ (113g)
```
val s_of_unix_error : Unix.error -> string -> string -> string
```

⟨*function* Process.s_of_unix_error 63f⟩≡ (114a)
```
let s_of_unix_error err _s1 _s2 =
  spf "%s" (Unix.error_message err)
```
Uses Common.spf().

7.2.8 $status management

The exit status of the last command is stored in the `\$status` variable as a string (not an integer like in the Bourne shell). An empty string or "0" means success; anything else means failure. For pipelines like `ls | wc`, `rc` combines the statuses into "0|0". `truestatusX` considers a status "true" if it contains only `0` and `|`—meaning every stage succeeded. This preserves per-stage failure information, unlike `bash` where `\$?` only reports the last command.

⟨*signature* Status.setstatus 64a⟩≡ (116)
```
  val setstatus : string -> unit
```

⟨*function* Status.setstatus 64b⟩≡ (117a)
```
  let setstatus s =
    Var.setvar "status" [s]
```
Uses Var.setvar().

⟨*signature* Status.getstatus 64c⟩≡ (116)
```
  val getstatus : unit -> string
```

⟨*function* Status.getstatus 64d⟩≡ (117a)
```
  let getstatus () =
    let v = (Var.vlook "status").R.v in
    match v with
    | None -> ""
    | Some [x] -> x
    (* stricter: should never happen *)
    | Some _ -> failwith "getstatus: $status is a list with more than one element"
```
Uses Runtime.var.v 28c and Var.vlook().

⟨*signature* Status.truestatus 64e⟩≡ (116)
```
  val truestatus : unit -> bool
```

⟨*function* Status.truestatus 64f⟩≡ (117a)
```
  let truestatus () : bool =
    let s = getstatus () in
    s = ""
    || s =~ "0+\\(|0+\\)*"
```
Uses Common.=~() and Status.getstatus().

7.2.9 Wait

⟨Op_process.op_Simple() *when default case, after the fork* 64g⟩≡ (60d)
```
  (* do again even if was interrupted *)
  while Process.waitfor caps pid = Process.WaitforInterrupted do
    ()
  done
```

⟨*signature* Process.waitfor 64h⟩≡ (113g)
```
  val waitfor : < Cap.wait; .. > -> int (* pid *) -> waitfor_result
```

⟨*type* Process.waitfor_result 64i⟩≡ (114a 113g)
```
  type waitfor_result =
    | WaitforInterrupted
    | WaitforFound
    | WaitforNotfound
```

`waitfor`[65a] loops calling `Unix.wait()`, which returns the next child that exits—but that might not be the one we are looking for. When a different child exits, its status is recorded in the corresponding thread's `waitstatus` field (found by scanning `runqX` for a thread with a matching `WaitFor pid`). The loop continues until our target `pid` is found. This design handles the case where multiple children run concurrently (e.g., both sides of a pipe): `wait()` returns them in whatever order they finish, and `waitfor` distributes the results to the right threads.

⟨*function* `Process.waitfor` 65a⟩≡ (114a)
```
let waitfor (caps : < Cap.wait; .. >) (pid : int) : waitfor_result =
  (* less: check for havewaitpid *)

  try
    let rec loop () =
      let (pid2, status) = CapUnix.wait caps () in
      let status_str =
        match status with
        | Unix.WEXITED i -> spf "%d" i
        | Unix.WSIGNALED i -> spf "signaled %d" i
        | Unix.WSTOPPED i -> spf "stopped %d" i
      in
      if pid = pid2
      then begin
        Status.setstatus status_str;
        WaitforFound
      end else begin
        !R.runq |> List.iter (fun t ->
          match t.R.waitstatus with
          | R.WaitFor pid ->
              if pid = pid2
              then t.R.waitstatus <- R.ChildStatus status_str;
          | R.ChildStatus _ | R.NothingToWaitfor -> ()
        );
        loop ()
      end
    in
    loop ()
  with Unix.Unix_error (err, s1, s2) ->
    Globals.errstr := s_of_unix_error err s1 s2;
    if err = Unix.EINTR
    then WaitforInterrupted
    else WaitforNotfound
```
Uses `CapUnix.wait()`, `Common.spf()`, `Globals.errstr` 63d, `Process.s_of_unix_error()` 63f, `Process.waitfor_result.WaitforFound` 64i, `Process.waitfor_result.WaitforInterrupted` 64i, `Process.waitfor_result.WaitforNotfound` 64i, `Runtime.runq`, `Runtime.thread.waitstatus` 30b, `Runtime.waitstatus.ChildStatus`, `Runtime.waitstatus.NothingToWaitfor` 32f, `Runtime.waitstatus.WaitFor` 32f, and `Status.setstatus()`.

7.2.10 $path management

When executing an external command, `rc` must find it on the filesystem. If the command name contains a slash (`/bin/ls`, `./myprog`, `../tool`) or starts with `#` (a Plan 9 device path), it is used as-is. Otherwise, `rc` searches each directory in `\$path` until it finds an executable with that name.

⟨*signature* `PATH.find_in_path` 65b⟩≡ (112b)
```
val find_dir_with_cmd_in_PATH :
  string (* cmd *) -> Fpath.t list (* $path *) -> Fpath.t
```

⟨*signature* `PATH.search_path_for_cmd` 65c⟩≡ (112b)
```
(* Returns None when there is no need to use $path because the cmd is
 * already using an absolute, relative, or special path.
```

```
  *)
 val var_PATH_for_cmd_opt : string (* cmd *) ->
   Fpath.t list (* content of $path *) option
```

⟨*function* PATH.search_path_for_cmd 66a⟩≡ (112c)

```
 let var_PATH_for_cmd_opt (s : string) : Fpath.t list option =
   (* no need for PATH resolution when commands use absolute, relative, or
    * special paths.
    *)
   if s =~ "^/" ||
      (* Plan 9 device paths *)
      s =~ "^#" ||
      s =~ "\\./" ||
      s =~ "\\.\\./"
   then None
   else
     let v = (Var.vlook "path").R.v in
     (match v with
     | None -> Some []
     | Some xs -> Some (Fpath_.of_strings xs)
     )
```

Uses Common.=~(), Runtime.var.v 28c, and Var.vlook().

⟨*function* PATH.find_in_path 66b⟩≡ (112c)

```
 let find_dir_with_cmd_in_PATH (cmd : string) (paths : Fpath.t list) : Fpath.t =
   paths |> List.find (fun dir ->
     let res = Sys.file_exists !!(dir / cmd) in
     if res
     then Logs.info (fun m -> m "found %s in %s" cmd !!dir);
     res
   )
```

7.3 Operators

7.3.1 Basic sequence

⟨Compile.outcode_seq *in nested* xseq() 66c⟩≡ (58a)

```
 (* less set iflast, for if not syntax error checking *)
 seq |> List.iter (fun x -> xcmd x eflag)
```

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 66d⟩+≡ (58a) ◁58b 66e▷

```
 | A.EmptyCommand -> ()
```

Uses Ast.cmd.EmptyCommand 98.

7.3.2 Logical operators

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 66e⟩+≡ (58a) ◁66d 67c▷

```
 | A.And (cmd1, cmd2) ->
     xcmd cmd1 false;
     emit (O.F O.True);
     let p = !idx in
     xcmd cmd2 eflag;

     set p (O.I !idx)
```

Uses Ast.cmd.And 25c, Opcode.operation.True, Opcode.t.F, and Opcode.t.I 27e.

Logical operators use a forward-jump-with-backpatching pattern. For `cmd1 \&\& cmd2`, the compiler emits: opcodes for `cmd1`, then `O.True`, then a placeholder integer, then opcodes for `cmd2`. After `cmd2`'s opcodes are emitted, the compiler backpatches the placeholder with the current index (the address just past `cmd2`). At runtime, `O.True` checks the exit status: if true (status is empty or zero), it increments `pc` to execute `cmd2`; if false, it jumps over `cmd2` to the backpatched address. `O.False` (for `||`) does the opposite.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 67a⟩+≡ (37c) ◁60c 67d▷

```
  | O.True ->
      let t = R.cur () in
      let pc = t.R.pc in
      if Status.truestatus ()
      then incr pc
      else pc := int_at_address t (!pc)
```

Uses `Interpreter.int_at_address()`, `Runtime.cur()` 30d, `Runtime.thread.pc` 29h, and `Status.truestatus()`.

⟨*function* `Interpreter.int_at_address` 67b⟩≡ (110g)

```
  let int_at_address t pc =
    match t.R.code.(pc) with
    | O.I i -> i
    (* stricter: generate error, but should never happen *)
    | op -> failwith (spf "was expecting I, not %s at %d"
                        (Opcode.show op) pc)
```

Uses `Common.spf()`, `Opcode.t.I` 27e, and `Runtime.thread.code` 29h.

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 67c⟩+≡ (58a) ◁66e 67e▷

```
  | A.Or (cmd1, cmd2) ->
      xcmd cmd1 false;
      emit (O.F O.False);
      let p = !idx in
      xcmd cmd2 eflag;

      set p (O.I !idx)
```

Uses `Ast.cmd.Or` 25c, `Opcode.operation.False` 76b, `Opcode.t.F`, and `Opcode.t.I` 27e.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 67d⟩+≡ (37c) ◁67a 67f▷

```
  | O.False ->
      let t = R.cur () in
      let pc = t.R.pc in
      if Status.truestatus ()
      then pc := int_at_address t (!pc)
      else incr pc
```

Uses `Interpreter.int_at_address()`, `Runtime.cur()` 30d, `Runtime.thread.pc` 29h, and `Status.truestatus()`.

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 67e⟩+≡ (58a) ◁67c 68a▷

```
  | A.Not cmd ->
      xcmd cmd eflag;
      emit (O.F O.Not);
```

Uses `Ast.cmd.Not` 25c, `Opcode.operation.Not`, and `Opcode.t.F`.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 67f⟩+≡ (37c) ◁67d 68b▷

```
  | O.Not ->
      Status.setstatus (if Status.truestatus() then "false" else "");
```

Uses `Status.setstatus()` and `Status.truestatus()`.

7.3.3 String matching

The ~ operator needs two `O.Mark`[59e] bytecodes because both the subject and the patterns can be multi-word (after variable expansion). The first Mark/words group collects the patterns, the second Mark/word collects the subject. `O.Match`[68b] pops both, concatenates the subject words into a single string, and tests each pattern with `Pattern.match_str`[113e].

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 68a⟩+≡ (58a) ◁67e 68c▷

```
  | A.Match (w, ws) ->
      emit (O.F O.Mark);
      xwords ws;
      emit (O.F O.Mark);
      xword w;
      emit (O.F O.Match);
      ⟨Compile.outcode_seq in A.Match case after emit O.Match 92h⟩
```

Uses `Ast.cmd.Match` 25c, `Opcode.operation.Mark` 27e, `Opcode.operation.Match` 27e, and `Opcode.t.F`.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 68b⟩+≡ (37c) ◁67f 69a▷

```
  (* (pat, str) *)
  | O.Match ->
      let t = R.cur () in
      let argv = t.R.argv in
      let subject = String.concat " " argv in
      Status.setstatus "no match";
      R.pop_list ();
      let argv = t.R.argv in
      argv |> List.exists (fun w ->
        if Pattern.match_str subject w
        then begin
          Status.setstatus "";
          true
        end else false
      ) |> ignore;
      R.pop_list ();
```

Uses `Opcode.operation.Match` 27e, `Pattern.match_str()` 113e, `Runtime.cur()` 30d, `Runtime.pop_list()` 115, `Runtime.thread.argv`, and `Status.setstatus()`.

7.4 Redirection

7.4.1 Opcode generation

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 68c⟩+≡ (58a) ◁68a 71d▷

```
  | A.Redir (cmd, (redir_kind, word)) ->
      (* resolve the filename *)
      emit (O.F O.Mark);
      xword word;
      emit (O.F O.Glob);

      (match redir_kind with
      | A.RWrite ->
          emit (O.F O.Write);
          emit (O.I 1);
      (* less: and A.RHere *)
      | A.RRead ->
          emit (O.F O.Read);
          emit (O.I 0);
      | A.RAppend ->
          emit (O.F O.Append);
```

```
        emit (O.I 1);
    | _ -> failwith ("TODO compile: " ^ Ast.show_cmd cmd)
    );

    (* perform the command *)
    xcmd cmd eflag;
    emit (O.F O.Popredir);
```

Uses Ast.cmd.Redir, Ast.redirection_kind.RAppend, Ast.redirection_kind.RRead 25b, Ast.redirection_kind.RWrite, Opcode.operation.Append 27e, Opcode.operation.Glob 27e, Opcode.operation.Mark 27e, Opcode.operation.Popredir 27e, Opcode.operation.Read 27e, Opcode.operation.Write 27e, Opcode.t.F, and Opcode.t.I 27e.

The redirection compilation follows a bracket pattern: first resolve the filename (Mark, emit the word, Glob to expand wildcards), then emit the redirection opcode (`Write`, `Read`, or `Append`) with an inline file descriptor number, then emit the command that runs under this redirection, and finally emit `Popredir` to close the opened file descriptor. The inline integer after the redirection opcode specifies which standard file descriptor to redirect: 1 for stdout (with `>`), 0 for stdin (with `<`).

⟨Interpreter.interpret_operation() *match* operation *cases* 69a⟩+≡ (37c) ◁68b 69b▷

```
  | O.Popredir ->
      R.pop_redir ()
```

Uses Runtime.pop_redir() 31e.

7.4.2 O.Write()

`O.Write`[69b] pops the filename from `argv`, opens the file for writing (creating it if necessary), then pushes a redirection record pairing the opened file descriptor with the target (stdout by default). The actual `dup2()` call happens later, in `doredir`[70e], when the child process is about to `exec`. `O.Popredir`[69a] later closes the opened file descriptor to prevent leaks in the parent.

⟨Interpreter.interpret_operation() *match* operation *cases* 69b⟩+≡ (37c) ◁69a 72a▷

```
  (* (file)[fd] *)
  | O.Write ->
      let t = R.cur () in
      let argv = t.R.argv in
      let pc = t.R.pc in
      (match argv with
      | [file] ->
          (try
            let fd_from =
              Unix.openfile file [Unix.O_CREAT;Unix.O_WRONLY] 0o666 in
            (* should be stdout *)
            let fd_to =
              let i = int_at_address t !pc in
              file_descr_of_int i
            in
            R.push_redir (R.FromTo (fd_from, fd_to));
            incr pc;
            R.pop_list();
          with Unix.Unix_error (_err, _s1, _s2) ->
            prerr_string (spf "%s: " file);
            E.error caps "can't open"
          )
      | []      -> E.error caps "> requires file"
      | _x::_y::_xs -> E.error caps "> requires singleton"
      )
```

Uses Common.spf(), Error.error() 63c, Interpreter.file_descr_of_int() 70a, Interpreter.int_at_address(), Opcode.operation.Write 27e, Runtime.cur() 30d, Runtime.pop_list() 115, Runtime.push_redir() 31c, Runtime.redir.FromTo 30b, Runtime.thread.argv, and Runtime.thread.pc 29h.

⟨*function* Interpreter.file_descr_of_int 70a⟩≡ (110g)

```
(* could be in runtime.ml *)
let file_descr_of_int i =
  match i with
  | 0 -> Unix.stdin
  | 1 -> Unix.stdout
  | 2 -> Unix.stderr
  (* todo: how do that? if do >[1=4] ?? *)
  | n -> failwith (spf "file_descr_of_int: unsupported int %d" n)
```

Uses Common.spf().

⟨*signature* Runtime.push_redir 70b⟩≡ (114d)

```
val push_redir : redir -> unit
```

⟨*function* Runtime.push_redir 70c⟩≡ (115)

```
let push_redir (x : redir) : unit =
  let t = cur () in
  match t.redirections with
  | [] -> failwith "push_redir: no starting redir"
  | xs::xxs -> t.redirections <- (x::xs)::xxs
```

Uses Runtime.redir.Close.

7.4.3 O.Read()

7.4.4 O.Append()

7.4.5 Closing redirection opened files

⟨*signature* Runtime.pop_redir 70d⟩≡ (114d)

```
val pop_redir : unit -> unit
```

⟨*function* Runtime.pop_redir 70e⟩≡ (115)

```
let pop_redir () =
  let t= cur () in
  match t.redirections with
  | [] -> failwith "pop_redir: no starting redir"
  | []::_xxs -> failwith "popredir null!"
  | (x::xs)::xxs ->
      t.redirections <- xs::xxs;
      (match x with
      | FromTo (fd_from, _fd_to) ->
          Unix.close fd_from
      | Close _ ->
          ()
      )
```

Uses Runtime.cur() 30d, Runtime.pop_redir() 31e, Runtime.redir.FromTo 30b, and Runtime.thread.redirections 30b.

⟨Process.return() *initializations* 70f⟩≡ (57c)

```
R.turf_redir ();
```

Uses Runtime.turf_redir() 70e.

⟨*signature* Runtime.turf_redir 70g⟩≡ (114d)

```
val turf_redir : unit -> unit
```

⟨*function* Runtime.turf_redir 70h⟩≡ (115)

```
let turf_redir () =
  let t = cur () in
  while List.hd t.redirections <> [] do
    pop_redir ()
  done
```

⟨Op_process.exec() *before* execute 71a⟩≡ (62a)
```
R.doredir t.R.redirections;
```

⟨*signature* Runtime.doredir 71b⟩≡ (114d)
```
val doredir : redir list list -> unit
```

doredir[70e] is called in the child process just before exec. It flattens all redirection scopes into a single list, reverses it (so the outermost redirections are applied first), then performs the classic Unix “fd dance”: dup2(from, to) makes the target file descriptor (to, e.g., stdout) point to the opened file (from), then close(from) releases the now-redundant original descriptor.

⟨*function* Runtime.doredir 71c⟩≡ (115)
```
let doredir xxs =
  xxs |> List.flatten |> List.rev |> List.iter (fun redir ->
    match redir with
    | FromTo (xfrom, xto) ->
       Unix.dup2 xfrom xto;
       Unix.close xfrom
    | Close from ->
       Unix.close from
  )
```

Uses Runtime.thread.argv, Runtime.thread.code 29h, Runtime.thread.locals 115, Runtime.thread.pc 29h, and Runtime.thread.redirections 30b.

7.5 Pipe

7.5.1 Opcode generation

Pipe compilation is the most complex opcode sequence. The emitted layout is: Pipe, fd numbers (1 for stdout, 0 for stdin), two jump-target placeholders, the left command’s opcodes (terminated by Exit—the child process exits after running), the right command’s opcodes (terminated by Return—it runs in a new thread in the parent process), and finally PipeWait for the parent to collect the child’s exit status. The two placeholders are backpatched: one points to the start of the right command, the other to PipeWait.

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 71d⟩+≡ (58a) ◁68c 74a▷
```
| A.Pipe (cmd1, cmd2) ->
    emit (O.F O.Pipe);
    emit (O.I 1); (* left fd *)
    emit (O.I 0); (* right fd *)

    let p = !idx in
    emit (O.I 0);
    let q = !idx in
    emit (O.I 0);

    (* will be executed in a forked child, hence Exit *)
    xcmd cmd1 eflag;
    emit (O.F O.Exit);

    (* will be executed in a children thread, hence Return *)
    set p (O.I !idx);
    xcmd cmd2 eflag;
    emit (O.F O.Return);

    (* will be executed by parent once the children thread finished *)
    set q (O.I !idx);
    emit (O.F O.PipeWait);
```

Uses Ast.cmd.Pipe 26a, Opcode.operation.Exit 27e, Opcode.operation.Pipe 37d, Opcode.operation.PipeWait 27e, Opcode.operation.Return 27e, Opcode.t.F, and Opcode.t.I 27e.

7.5.2 O.Pipe()

When `O.Pipe` executes, it creates a Unix pipe, then forks. The *child* process runs the left side of the pipe: it redirects stdout to the write end of the pipe and executes `cmd1` (which ends with `O.Exit`). The *parent* creates a new thread for the right side: it redirects stdin to the read end and executes `cmd2` (which ends with `O.Return`). After the right-side thread finishes (its `Return` pops it), the parent reaches `O.PipeWait`, which calls `waitfor` to collect the child's exit status. The combined status is the two statuses joined by "—".

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 72a⟩+≡ (37c) ◁69b 72b▷

```
  (* [i j]{... Xreturn}{... Xreturn} *)
  | O.Pipe ->
      let t = R.cur () in
      let pc = t.R.pc in
      (* left file descriptor, should be stdout *)
      let lfd =
        let i = int_at_address t !pc in
        file_descr_of_int i
      in
      incr pc;
      (* right file descriptor, should be stdin *)
      let rfd =
        let i = int_at_address t !pc in
        file_descr_of_int i
      in
      incr pc;

      let (pipe_read, pipe_write) = Unix.pipe () in
      let forkid = CapUnix.fork caps () in

      (* child *)
      if forkid = 0 then begin
        (* less: clearwaitpids () *)
        (* pc + 2 to jump over the jump addresses *)
        let newt = R.mk_thread t.R.code (!pc + 2) t.R.locals in
        R.runq := [newt];
        Unix.close pipe_read;
        R.push_redir (R.FromTo (pipe_write, lfd));
      (* parent *)
      end else begin
        (* less: addwaitpid () *)
        let newt =
          R.mk_thread t.R.code (int_at_address t (!pc+0)) t.R.locals in
        R.runq := newt::!R.runq;
        Unix.close pipe_write;
        R.push_redir (R.FromTo (pipe_read, rfd));

       (* once newt finished, jump to Xpipewait *)
        pc := int_at_address t (!pc+1);
        t.R.waitstatus <- R.WaitFor forkid;
      end
```

Uses `CapUnix.fork()`, `Interpreter.file_descr_of_int()` 70a, `Interpreter.int_at_address()`, `Opcode.operation.Pipe` 37d, `Runtime.cur()` 30d, `Runtime.mk_thread()` 71c, `Runtime.push_redir()` 31c, `Runtime.redir.FromTo` 30b, `Runtime.runq`, `Runtime.thread.code` 29h, `Runtime.thread.locals` 115, `Runtime.thread.pc` 29h, `Runtime.thread.waitstatus` 30b, and `Runtime.waitstatus.WaitFor` 32f.

7.5.3 O.Exit()

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 72b⟩+≡ (37c) ◁72a 73d▷

```
  | O.Exit ->
      (* todo: trapreq *)
      Process.exit caps (Status.getstatus())
```
Uses `Process.exit()` 73b and `Status.getstatus()`.

⟨*signature* `Process.exit` 73a⟩≡ (113g)
```
  val exit : < Cap.exit ; .. > -> string -> unit
```

⟨*function* `Process.exit` 73b⟩≡ (114a)
```
  let exit (caps: < Cap.exit; ..>) s =
    (* todo: Updenv *)
    Status.setstatus s;
    (* todo: how communicate error to parent process under Unix?
     * alt: raise Exit.ExitCode which removes the need for Cap.exit
     *)
    CapStdlib.exit caps (if Status.truestatus () then 0 else 1)
```
Uses `CapStdlib.exit()`, `Status.setstatus()`, and `Status.truestatus()`.

7.5.4 `O.PipeWait()`

⟨`Opcode.operation` *other pipe cases* 73c⟩+≡ (27e) ◁37e
```
  | PipeWait (* argument passed through Thread.pid *)
```

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 73d⟩+≡ (37c) ◁72b 74b▷
```
  (* argument passed through Thread.pid *)
  | O.PipeWait ->
      let t = R.cur () in
      (match t.R.waitstatus with
      (* stricter: *)
      | R.NothingToWaitfor ->
         failwith "Impossible: NothingToWaitfor for PipeWait"
      (* a previous waitfor() already got it *)
      | R.ChildStatus status ->
         Status.setstatus (Status.concstatus status (Status.getstatus()));
      | R.WaitFor pid ->
         let status = Status.getstatus () in
         (* will internally call setstatus() when it found the right child *)
         Process.waitfor caps pid |> ignore;
         t.R.waitstatus <- R.NothingToWaitfor;
         Status.setstatus (Status.concstatus (Status.getstatus()) status);
      )
```
Uses `Opcode.operation.PipeWait` 27e, `Process.waitfor()` 65a, `Runtime.cur()` 30d, `Runtime.thread.waitstatus` 30b, `Runtime.waitstatus.ChildStatus`, `Runtime.waitstatus.NothingToWaitfor` 32f, `Runtime.waitstatus.WaitFor` 32f, `Status.concstatus()`, `Status.getstatus()`, and `Status.setstatus()`.

⟨*signature* `Status.concstatus` 73e⟩≡ (116)
```
  val concstatus : string -> string -> string
```

⟨*function* `Status.concstatus` 73f⟩≡ (117a)
```
  let concstatus s1 s2 =
    s1 ^ "|" ^ s2
```

7.6 Asynchronous execution

7.7 Control flow statements

7.7.1 if

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 74a⟩+≡ (58a) ◁71d 75a▷

```
  | A.If (cmds, cmd) ->
      xseq cmds false;
      emit (O.F O.If);
      let p = !idx in
      emit (O.I 0);
      xcmd cmd eflag;
      emit (O.F O.Wastrue);
      set p (O.I !idx);
```

Uses Ast.cmd.If, Opcode.operation.If 27e, Opcode.operation.Wastrue 75b, Opcode.t.F, and Opcode.t.I 27e.

⟨Interpreter.interpret_operation() *match* operation *cases* 74b⟩+≡ (37c) ◁73d 74c▷

```
  | O.If ->
      let t = R.cur () in
      let pc = t.R.pc in

      Globals.ifnot := true;
      if Status.truestatus()
      then incr pc
      else pc := int_at_address t (!pc);
```

Uses Globals.ifnot 74d, Interpreter.int_at_address(), Runtime.cur() 30d, Runtime.thread.pc 29h, and Status.truestatus().

7.7.2 if not

The if/if not protocol uses a global ifnot flag as a communication channel between separate command lines. When O.If executes, it sets ifnot := true. If the condition was true and the then-branch executes, O.Wastrue resets ifnot := false. On the *next line*, if the user types if not cmd, O.IfNot[74c] checks the flag: if still true (meaning the previous if took the false branch), it executes cmd; otherwise it skips. This is a clever workaround for interactive parsing: since rc must execute each line immediately, it cannot look ahead to see if an "else" follows.

⟨Interpreter.interpret_operation() *match* operation *cases* 74c⟩+≡ (37c) ◁74b 74e▷

```
  | O.IfNot ->
      let t = R.cur () in
      let pc = t.R.pc in
      if !Globals.ifnot
      then incr pc
      else pc := int_at_address t (!pc);
```

Uses Globals.ifnot 74d, Interpreter.int_at_address(), Runtime.cur() 30d, and Runtime.thread.pc 29h.

⟨*constant* Globals.ifnot 74d⟩≡ (110c)

```
  (* Set to true at the beginning of an if and back to false
   * if rc executes the then branch. Kept to true otherwise
   * so next IfNot will run.
   *)
  let ifnot = ref false
```

⟨Interpreter.interpret_operation() *match* operation *cases* 74e⟩+≡ (37c) ◁74c 76a▷

```
  | O.Wastrue ->
      Globals.ifnot := false
```

Uses Globals.ifnot 74d.

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 75a⟩+≡ (58a) ◁74a 75c▷

```
  | A.IfNot cmd ->
      emit (O.F O.IfNot);
      let p = !idx in
      emit (O.I 0);
      xcmd cmd eflag;
      set p (O.I !idx);
```

Uses Ast.cmd.IfNot 25c, Opcode.operation.IfNot 27e, Opcode.t.F, and Opcode.t.I 27e.

7.7.3 while

7.7.4 for

⟨Opcode.operation *other control cases* 75b⟩+≡ (27e) ◁56b 76b▷

```
  | For   (* (var, list){... Xreturn} *)
```

7.7.5 switch

The switch compilation uses a chain of forward jumps and backpatching. For each case, the compiler emits Mark, the case patterns, O.Case, and a placeholder for the next case's address. If the case matches at runtime, execution falls through to the case body; otherwise O.Case jumps to the next case. Each case body ends with a Jump to the common exit point (leave), and Popm at the end removes the switch value from the stack. The out variable holds the address of the exit jump target, shared by all cases. This is the same forward-jump backpatching technique used for logical operators, just applied multiple times in a chain.

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 75c⟩+≡ (58a) ◁75a 77a▷

```
  | A.Switch (w, cmds) ->

      (match cmds with
      | (A.Simple (A.Word ("case", false), _))::_ -> ()
      | _ -> failwith "case missing in switch"
      );

      emit (O.F O.Mark);
      xword w;
      emit (O.F O.Jump);

      let nextcase = !idx in
      emit (O.I 0);
      let out = !idx in
      emit (O.F O.Jump);
      let leave = !idx in
      emit (O.I 0);

      set nextcase (O.I !idx);

      let aux cmds =
        match cmds with
        | [] -> ()
        | (A.Simple (A.Word ("case", false), ws))::cmds ->
            emit (O.F O.Mark);
            xwords ws;
            emit (O.F O.Case);
            let nextcase = !idx in
            emit (O.I 0);

            let cmds_for_this_case, _other_cases = split_when_case cmds in
```

```
          cmds_for_this_case |> List.iter (fun cmd ->
            xcmd cmd eflag
          );
          emit (O.F O.Jump);
          emit (O.I out);
          set nextcase (O.I !idx);
      | _ -> failwith "case missing in switch"
    in
    aux cmds;
    set leave (O.I !idx);
    (* can not call pop_list(), here, otherwise circular deps *)
    emit (O.F O.Popm);
```

Uses Ast.cmd.Simple, Ast.cmd.Switch 25c, Ast.value.Word 26b, Compile.split_when_case() 76e, Opcode.operation.Case 27e, Opcode.operation.Jump 27e, Opcode.operation.Mark 27e, Opcode.operation.Popm 27e, Opcode.t.F, and Opcode.t.I 27e.

⟨Interpreter.interpret_operation() *match* operation *cases* 76a⟩+≡ (37c) ◁74e 76c▷

```
  (* [addr] *)
  | O.Jump ->
      let t = R.cur () in
      let pc = t.R.pc in
      pc := int_at_address t (!pc);
```

Uses Interpreter.int_at_address(), Opcode.operation.Jump 27e, Runtime.cur() 30d, and Runtime.thread.pc 29h.

⟨Opcode.operation *other control cases* 76b⟩+≡ (27e) ◁75b

```
  | Case  (* (pat, value){...} *)
```

⟨Interpreter.interpret_operation() *match* operation *cases* 76c⟩+≡ (37c) ◁76a 76d▷

```
  (* (pat, value){...} *)
  | O.Case ->
      let t = R.cur () in
      let pc = t.R.pc in
      let s = List.hd t.R.argv_stack |> String.concat " " in
      let argv = t.R.argv in
      let match_found = argv |> List.exists (fun w -> Pattern.match_str s w) in
      (if match_found
      then incr pc
      else pc := int_at_address t (!pc)
      );
      R.pop_list ();
```

Uses Interpreter.int_at_address(), Opcode.operation.Case 27e, Pattern.match_str() 113e, Runtime.cur() 30d, Runtime.pop_list() 115, Runtime.thread.argv, Runtime.thread.argv_stack 38c, and Runtime.thread.pc 29h.

⟨Interpreter.interpret_operation() *match* operation *cases* 76d⟩+≡ (37c) ◁76c 78a▷

```
  (* (value) *)
  | O.Popm ->
      R.pop_list ()
```

Uses Opcode.operation.Popm 27e and Runtime.pop_list() 115.

⟨*function* Compile.split_when_case 76e⟩≡ (107b)

```
  let split_when_case cmds =
    cmds |> List_.span (function
      | (A.Simple (A.Word ("case", false), _)) -> false
      | _ -> true
    )
```

Uses Ast.cmd.Simple, Ast.value.Word 26b, and Common2.span().

7.7.6 Blocks: '{...}'

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 77a⟩+≡ (58a) ◁75c 77b▷
```
  | A.Compound seq -> xseq seq eflag
```
Uses Ast.cmd.Compound 25c.

7.8 Functions

7.8.1 Function definitions (fn <foo> ...)

When defining a function with `fn foo {body}`, the compiler emits the function name, then `O.FnX`, a forward-jump placeholder (to skip over the body during normal execution), the body bytecodes, `Unlocal` (to remove the local `\$*` binding), and `Return`. At runtime, `O.FnX` stores the current code vector and the body's start address in a separate `fns` hashtable. Unlike the C version which stores functions in the same `Var` structure as variables (using `fn` and `pc` fields), the OCaml version uses a separate `Runtime.fns`[29g] hashtable, cleanly separating the two namespaces.

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 77b⟩+≡ (58a) ◁77a 77c▷
```
  | A.Fn (w, cmds) ->
      emit (O.F O.Mark);
      xword w;
      emit (O.F O.Fn);
      let p = !idx in
      (* less: emit str of fn *)
      emit (O.S "Fn String Todo?");
      xseq cmds eflag;
      emit (O.F O.Unlocal);
      emit (O.F O.Return);
      set p (O.I !idx);
```
Uses Ast.cmd.Fn 53h, Opcode.operation.Fn 37e, Opcode.operation.Mark 27e, Opcode.operation.Return 27e, Opcode.operation.Unlocal 27e, Opcode.t.F, Opcode.t.I 27e, and Opcode.t.S 27e.

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

7.9 Variables

7.9.1 Variable definitions (<x>=...)

Variable assignment has two forms that compile differently. A bare assignment (`A=b`) uses `O.Assign` to set the variable globally. An assignment followed by a command (`A=b cmd`) uses `O.Local` to create a temporary binding in the thread's local scope, executes the command, then `O.Unlocal` removes the binding. This is `rc`'s equivalent of environment variables in `sh`'s `VAR=val cmd` syntax. The `split_at_non_assign` helper handles chained assignments like `A=1 B=2 cmd`, collecting all the assignments before the command.

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 77c⟩+≡ (58a) ◁77b 94e▷
```
  | A.Assign (val1, val2, cmd) ->
      let all_assigns, cmd =
        split_at_non_assign (A.Assign (val1, val2, cmd)) in
      (match cmd with
      (* A=b; *)
      | A.EmptyCommand ->
          all_assigns |> List.iter (fun (val1, val2) ->
            emit (O.F O.Mark);
            xword val2;
            emit (O.F O.Mark);
```

```
          xword val1;
          emit (O.F O.Assign);
        )

      (* A=b cmd; *)
      | _ ->
          all_assigns |> List.iter (fun (val1, val2) ->
            emit (O.F O.Mark);
            xword val2;
            emit (O.F O.Mark);
            xword val1;
            emit (O.F O.Local);
          );
          xcmd cmd eflag;
          all_assigns |> List.iter (fun (_, _) ->
            emit (O.F O.Unlocal);
          )
      )
```

Uses Ast.cmd.Assign 53h, Ast.cmd.EmptyCommand 98, Compile.split_at_non_assign() 79a, Opcode.operation.Assign 96b, Opcode.operation.Local 27e, Opcode.operation.Mark 27e, Opcode.operation.Unlocal 27e, and Opcode.t.F.

⟨Interpreter.interpret_operation() *match* operation *cases* 78a⟩+≡ (37c) ◁76d 78b▷

```
  (* (name) (val) *)
  | O.Assign ->
      let t = R.cur () in
      let argv = t.R.argv in
      (match argv with
      | [varname] ->
          (* no call to globlist for varname as it can be "*" for $* *)
          (* less: deglob varname *)
          let v = Var.vlook varname in
          R.pop_list ();

          (* less: globlist for the arguments *)
          let argv = t.R.argv in
          v.R.v <- Some argv;
          R.pop_list ();

      | _ -> E.error caps "variable name not singleton!"
      )
```

Uses Error.error() 63c, Opcode.operation.Assign 96b, Runtime.cur() 30d, Runtime.pop_list() 115, Runtime.thread.argv, Runtime.var.v 28c, and Var.vlook().

⟨Interpreter.interpret_operation() *match* operation *cases* 78b⟩+≡ (37c) ◁78a 79c▷

```
  (* (name) (val) *)
  | O.Local ->
      let t = R.cur () in
      let argv = t.R.argv in
      (match argv with
      | [varname] ->
        (* less: deglob varname *)
        R.pop_list ();
        (* less: globlist *)
        let argv = t.R.argv in
        Hashtbl.add t.R.locals varname { R.v = Some argv };
        R.pop_list ();
      | _ -> E.error caps "variable name not singleton!"
      )
```

Uses Error.error() 63c, Opcode.operation.Local 27e, Runtime.cur() 30d, Runtime.pop_list() 115, Runtime.thread.argv, Runtime.thread.locals 115, and Runtime.var.v 28c.

⟨*function* `Compile.split_at_non_assign` 79a⟩≡ (107b)

```
  let rec split_at_non_assign = function
    | A.Assign (val1, val2, cmd) ->
       let (a,b) = split_at_non_assign cmd in
       (val1, val2)::a, b
    | b -> [], b
```

Uses `Ast.cmd.Assign` 53h and `Compile.split_at_non_assign()` 79a.

7.9.2 Variable uses ($<x>)

`O.Dollar`[79c] handles variable expansion at runtime. It pops the variable name from **argv**, looks it up in the variable table, and prepends the variable's value (a word list) to the current **argv**. If the variable name is a number (e.g., \$1, \$2), it indexes into \$* instead of looking up a named variable.

⟨`Compile.outcode_seq` *in nested* `xword()` *match* `w` *cases* 79b⟩+≡ (58a) ◁59c 95g▷

```
 | A.Dollar w ->
     emit (O.F O.Mark);
     xword w;
     emit (O.F O.Dollar);
```

Uses `Ast.value.Dollar` 26b, `Opcode.operation.Dollar` 96b, `Opcode.operation.Mark` 27e, and `Opcode.t.F`.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 79c⟩+≡ (37c) ◁78b 90b▷

```
 (* (name) *)
 | O.Dollar ->
     let t = R.cur () in
     let argv = t.R.argv in
     (match argv with
     | [varname] ->
        (* less: deglob varname *)
        (try
           let value = vlook_varname_or_index varname in
           R.pop_list ();
           let argv = t.R.argv in
           let newargv =
             (match value with None -> [] | Some xs -> xs) @ argv in
           t.R.argv <- newargv
         with Failure s -> E.error caps s
        )
     | _ -> E.error caps "variable name not singleton!"
     )
```

Uses `Error.error()` 63c, `Interpreter.vlook_varname_or_index()`, `Opcode.operation.Dollar` 96b, `Runtime.cur()` 30d, `Runtime.pop_list()` 115, and `Runtime.thread.argv`.

7.9.3 Special variables

⟨*function* `Interpreter.vlook_varname_or_index` 79d⟩≡ (110g)

```
  let vlook_varname_or_index varname =
    if varname =~ "^[0-9]+$"
    then
      let i = int_of_string varname in
      let v = (Var.vlook "*").R.v in
      (match v with
      (* stricter: array out of bound checking *)
      | None -> failwith "undefined $*"
      | Some xs ->
          (* list indexes in rc starts at 1, not 0 *)
          if i >= 1 && i <= List.length xs
```

```
          then Some ([List.nth xs (i-1)])
          else failwith (spf "out of bound, $%d too big for $*" i)
      )
    else (Var.vlook varname).R.v
```

Uses `Common.=~()`, `Common.spf()`, `Runtime.var.v` 28c, and `Var.vlook()`.

Chapter 8

Builtins

8.1 Overview

Builtins are commands that the shell executes directly in its own process, without forking. They *must* be builtins because they need to modify the shell's own state: `cd` changes the shell's working directory (a child process changing its directory would have no effect on the parent), `.` (dot/source) loads and evaluates a script in the current shell, and `exit` terminates the shell itself. The dispatch is a simple list membership check followed by a match on the command name—no registration table or hash map, just straightforward pattern matching.

⟨*function* `Builtin.is_builtin` 81a⟩≡ (105c)

```
let is_builtin s =
  List.mem s [
    "cd";
    ".";
    "eval";
    "exit";
    "flag";
    "finit";
    (* new in ocaml *)
    "show";
  ]
```

⟨*function* `Builtin.dispatch` 81b⟩≡ (105c)

```
let dispatch (caps : < Cap.chdir; Cap.exit; Cap.open_in; ..>) s =
  match s with
  ⟨Builtin.dispatch() match s cases 82a⟩
  | "show" ->
      let t = R.cur () in
      let argv = t.R.argv in
      (match argv with
      | [_show;"vars"] ->
          Logs.app (fun m -> m "--- GLOBALS ---");
          R.globals |> Hashtbl_.to_list |> Assoc.sort_by_key_lowfirst |>
          List.iter (fun (k, v) ->
              Logs.app (fun m -> m "%s=%s" k (Runtime.string_of_var v))
          );
          Logs.app (fun m -> m "--- LOCALS ---");
          t.R.locals |> Hashtbl.iter (fun k v ->
              Logs.app (fun m -> m "%s=%s" k (Runtime.string_of_var v))
          );
          Logs.app (fun m -> m "--- FUNCTIONS ---");
          R.fns |> Hashtbl.iter (fun k fn ->
              Logs.app (fun m -> m "%s=%s" k (Runtime.show_fn fn))
          );
```

```
            ()
        | _ -> E.error caps ("Usage: show (vars|thread)")
        );


    | _ -> failwith (spf "unsupported builtin %s" s)
```
Uses Common.spf().

8.2 cd

cd with no argument changes to \$home. With one argument, it attempts to change to that directory. The status is optimistically set to "can't cd" before trying, and reset to empty on success—a pattern that avoids a separate error flag.

⟨Builtin.dispatch() *match* s *cases* 82a⟩≡ (81b) 83a▷

```
  | "cd" ->
      let t = R.cur () in
      let argv = t.R.argv in
      (* default value in case something goes wrong below *)
      Status.setstatus "can't cd";

      (* less: cdpath vlook *)
      (match argv with
      | [_cd] ->
          let v = (Var.vlook "home").R.v in
          (match v with
          | Some (dir::_) ->
            if dochdir caps dir
            then Status.setstatus ""
            (* less: %r *)
            else Logs.err (fun m -> m "Can't cd %s" dir)
          | _ -> Logs.err (fun m -> m "Can't cd -- $home empty")
          )
      | [_cd;dir] ->
          (* less: cdpath iteration *)
          if dochdir caps dir
          then Status.setstatus ""
          else Logs.err (fun m -> m "Can't cd %s" dir)
      | _ ->
          Logs.err (fun m -> m "Usage: cd [directory]");
      );
      R.pop_list()
```
Uses Builtin.dochdir(), Logs.err(), Runtime.pop_list() 115, Runtime.thread.argv, Runtime.var.v 28c, Status.setstatus(), and Var.vlook().

⟨*function* Builtin.dochdir 82b⟩≡ (105c)

```
  let dochdir (caps : < Cap.chdir; .. >) s =
    try
      Logs.info (fun m -> m "about to chdir to %s" s);
      CapUnix.chdir caps s;
      true
    with Unix.Unix_error _ ->
      false
```
Uses CapUnix.chdir() and Logs.info().

8.3 show

8.4 exit

8.5 '.'

The . (dot) builtin sources a script: it reads and executes commands from a file in the current shell process (without forking). This is how rc loads initialization files like rcmain and the user's profile. It creates a new thread whose lexbuf reads from the sourced file, pushes \$0 and \$* as local variables, then enters the REPL loop to evaluate the file's commands.

⟨Builtin.dispatch() *match* s *cases* 83a⟩+≡ (81b) ◁82a 84d▷

```
  | "." ->
      let t = R.cur () in
      R.pop_word (); (* "." *)

      if !ndots > 0
      then Globals.eflagok := true;
      incr ndots;

      let iflag =
        match t.R.argv with
        | "-i"::_xs -> R.pop_word (); true
        | _ -> false
      in
      (match t.R.argv with
      | [] -> E.error caps "Usage: . [-i] file [arg ...]"
      | zero::args ->
          R.pop_word ();
          (* less: searchpath, also for dot? seems wrong *)
          (try
            let file = zero in
            Logs.info (fun m -> m "evaluating %s" file);
            let chan = CapStdlib.open_in caps file in
            let newt = R.mk_thread dotcmds 0 (Hashtbl.create 10) in
            R.runq := [newt];
            R.push_redir (R.Close (Unix.descr_of_in_channel chan));
            newt.R.file <- Some (Fpath.v file);
            newt.R.lexbuf <- Lexing.from_channel chan;
            newt.R.iflag <- iflag;
            (* push for $* *)
            R.push_list ();
            newt.R.argv <- args;
            (* push for $0 *)
            R.push_list ();
            newt.R.argv <- [zero];

          with Failure _ ->
            prerr_string (spf "%s: " zero);
            E.error caps ".: can't open"
          )
      )
```

Uses Builtin.dotcmds, Builtin.ndots, CapStdlib.open_in(), Common.spf(), Error.error() 63c, Globals.eflagok 110b, Logs.info(), Runtime.mk_thread() 71c, Runtime.pop_word() 60f, Runtime.push_list(), Runtime.push_redir() 31c, Runtime.redir.Close, Runtime.runq, Runtime.thread.argv, Runtime.thread.file 36b, Runtime.thread.iflag, and Runtime.thread.lexbuf.

⟨*constant* Builtin.ndots 83b⟩≡ (105c)

```
let ndots = ref 0
```

The `dotcmds` array is a hand-compiled bytecode sequence equivalent to the shell commands: `local \$0 = <zero>; local \$* = <args>; REPL`. It first creates local bindings for `\$0` (the script name) and `\$*` (the script arguments), then enters the REPL loop to read and evaluate commands from the sourced file. When the sourced file reaches EOF, `O.REPL`[38f] calls `Process.return`[57c], which pops the dotcmds thread. The `Unlocal` opcodes then restore the previous `\$0` and `\$*` values.

⟨*constant* `Builtin.dotcmds` 84a⟩≡ (105c)
```
  (* for the builtin '.' (called 'source' in bash) *)
  let dotcmds =
    [|
      O.F O.Mark;
        O.F O.Word;
        O.S "0";
      O.F O.Local;

      O.F O.Mark;
        O.F O.Word;
        O.S "*";
      O.F O.Local;

      O.F O.REPL;

      O.F O.Unlocal;
      O.F O.Unlocal;
      O.F O.Return;
    |]
```
Uses `Opcode.operation.Local` 27e, `Opcode.operation.Mark` 27e, `Opcode.operation.REPL` 95c, `Opcode.operation.Return` 27e, `Opcode.operation.Unlocal` 27e, `Opcode.operation.Word` 27e, `Opcode.t.F`, and `Opcode.t.S` 27e.

8.6 eval

8.7 flag

⟨*global* `Flags.hflags` 84b⟩≡ (109b)
```
  let hflags: char Hashtbl_.set = Hashtbl.create 10
```

⟨`CLI.main()` *other initializations* 84c⟩+≡ (35a) ◁35f
```
  (* for 'flags' builtin (see builtin.ml) *)
  argv |> Array.iter (fun s ->
    if s =~ "^-\\([a-zA-Z]\\)"
    then begin
      let letter = Regexp.matched1 s in
      let char = String.get letter 0 in
      Hashtbl.add Flags.hflags char true
    end
  );
```
Uses `CLI.interpret_bootstrap()` 88, `Exit.t.OK`, and `Flags.hflags`.

⟨`Builtin.dispatch()` *match* `s` *cases* 84d⟩+≡ (81b) ◁83a 85▷
```
  | "flag" ->
      let t = R.cur () in
      let argv = t.R.argv in
      (match argv with
      | [_flag;letter] ->
          (* stricter: *)
```

```
        if String.length letter <> 1
        then E.error caps "flag argument must be a single letter"
        else begin
          let char = String.get letter 0 in
          let is_set = Hashtbl.mem Flags.hflags char in
          Status.setstatus (if is_set then "" else "flag not set");
        end

    | [_flag;_letter;_set] ->
        failwith "TODO: flag letter +- not handled yet"

    | _ -> E.error caps ("Usage: flag [letter] [+-]")
    );
    R.pop_list()
```

Uses `Error.error()` 63c, `Flags.hflags`, `Runtime.pop_list()` 115, `Runtime.thread.argv`, and `Status.setstatus()`.

8.8 finit

⟨`Builtin.dispatch()` *match* `s` *cases* 85⟩+≡ (81b) ◁84d

```
  | "finit" ->
     (* less: Xrdfn *)
     R.pop_list ()
```

8.9 wait

Chapter 9

Environment

9.1 Var.init()

Var.initX reads the process environment (environ) and populates the shell's global variable table. The special case for PATH converts the Unix colon-separated format (/usr/bin:/bin) into rc's list-valued \$path variable ((/usr/bin /bin)). This is a bridge between Unix conventions (uppercase PATH, colon-separated) and Plan 9 conventions (lowercase path, list-valued).

⟨*function* Var.vinit 86⟩≡ (117c)

```
let init (caps : < Cap.env; .. >) =
  Logs.info (fun m -> m "load globals from the environment");
  let xs = Env.read_environment caps in
  xs |> List.iter (fun (k, vs) ->
    match k, vs with
    | "PATH", [x] ->
       Logs.info (fun m -> m "adjust $PATH to $path");
       let vs = Regexp.split "[:]" x in
       let k = "path" in
       setvar k vs
    | _ -> setvar k vs
  );
```

Uses Logs.err().

9.2 Var.update_env()

Chapter 10

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.

Chapter 11

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.

11.1 Actual bootstrapping code

The real bootstrap is a hand-compiled bytecode array equivalent to the shell command: `\$*=(argv); . /usr/lib/rcmain \$*`. It first assigns the command-line arguments (pushed onto the stack by `interpret_bootstrap`) to `\$*`, then executes the dot builtin to source `rcmain`. The `rcmain` script (shown in Chapter B) handles initialization details like setting up the prompt, sourcing the user's profile, and ultimately running `. /dev/stdin` to start reading commands from the terminal. This "shell bootstraps itself in its own language" design is elegant: the core interpreter only needs to know how to execute bytecodes, and the initialization policy lives in an editable script.

⟨*function* `CLI.bootstrap` 88⟩≡ (106b)

```
(* This is more complex than just [| O.REPL |] in _bootstrap_simple.
 * The opcodes below correspond to this shell code:
 *
 *       *=(argv); . /usr/lib/rcmain $*
 *
 * /usr/lib/rcmain itself contains code roughly equivalent to '. /dev/stdin'
 * that will trigger reading commands from the standard input.
 *
 * Note that boostrap is now a function because it uses a flag that can be
 * modified after startup.
 *)
let bootstrap () : O.codevec =
  [|
     O.F O.Mark;
       O.F O.Word;
       O.S "*";
     O.F O.Assign; (* will pop_list twice *)

     O.F O.Mark;
       O.F O.Mark;
         O.F O.Word;
         O.S "*";
       O.F O.Dollar; (* will pop_list once *)
       O.F O.Word;
       O.S !Flags.rcmain; (* can be changed -m *)
       O.F O.Word;
       O.S ".";
```

```
        O.F O.Simple; (* will pop_list once *)

        O.F O.Exit;
    |]
```

Uses `CLI.bootstrap()` 88, `Flags.rcmain` 89a, `Opcode.operation.Assign` 96b, `Opcode.operation.Dollar` 96b, `Opcode.operation.Exit` 27e, `Opcode.operation.Mark` 27e, `Opcode.operation.Simple` 96c, `Opcode.operation.Word` 27e, `Opcode.t.F`, `Opcode.t.S` 27e, `Runtime.mk_thread()` 71c, and `Runtime.runq`.

11.2 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 profile. The `-m` flag allows overriding this path, which is useful during system bootstrap or testing.

⟨*constant* `Flags.rcmain` 89a⟩≡ (109b)

```
(* can be changed with -m *)
let rcmain = ref "/rc/lib/rcmain"
```

⟨`CLI.main()` options *elements* 89b⟩+≡ (35a) ◁36e 91b▷

```
"-m", Arg.Set_string Flags.rcmain,
" <file> read commands to initialize rc from file, not /rc/lib/rcmain";
```

11.3 Actual environment

Chapter 12

Globbing

Globbing (also called wildcard expansion or pathname expansion) is one of the key features of a shell: when you write `ls *.c`, the shell expands `*.c` into the list of matching filenames *before* passing them to `ls`. This is a deliberate design choice: the expansion is done by the shell, not by individual programs, which factorizes the functionality.

⟨`Opcode.operation` *other cases* 90a⟩≡ (27e) 92e ▷

```
  (* Globbing *)
  | Glob (* (value?) *)
```

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 90b⟩+≡ (37c) ◁ 79c 92g ▷

```
  (* (value?) *)
  | O.Glob ->
      Logs.err (fun m -> m "TODO: interpret Glob");
      ()
```

Uses `Logs.err()` and `Opcode.operation.Glob` 27e.

12.1 Lexing globbing characters

12.2 Expanding globbing characters

12.3 `glob()`

12.4 `match()`

Chapter 13

Debugging for the rc User

13.1 Printing commands: rc -x

⟨*constant* Flags.xflag 91a⟩≡ (109b)
```
(* -x, to print simple commands before executing them *)
let xflag = ref false
```

⟨CLI.main() options *elements* 91b⟩+≡ (35a) ◁89b 91e▷
```
"-x", Arg.Set Flags.xflag,
" print each simple command before executing it";
```
Uses Logs.level.Info.

⟨Op_process.op_Simple() *possibly dump command* 91c⟩≡ (60d)
```
(* less: globlist () *)
if !Flags.xflag
then Logs.app (fun m -> m "|%s|" (String.concat " " argv));
```
Uses Error.error() 63c.

13.2 Printing subprocesses status: rc -s

⟨*constant* Flags.sflag 91d⟩≡ (109b)
```
(* -s, to print status when error in command just ran *)
let sflag = ref false
```

⟨CLI.main() options *elements* 91e⟩+≡ (35a) ◁91b 92a▷
```
"-s", Arg.Set Flags.sflag,
" print exit status after any command where the status is non-null";
```
Uses Logs.level.Info.

⟨Op_repl.op_REPL() *if* sflag 91f⟩≡ (38g)
```
(* todo: flush error and reset error count *)
if !Flags.sflag && not (Status.truestatus())
then Logs.app (fun m -> m "status=%s" (Status.getstatus ()));
```
Uses Flags.sflag 91d, Logs.app(), Status.getstatus(), and Status.truestatus().

13.3 Logging: rc -v

⟨CLI.main() *debugging initializations* 91g⟩≡ (35a) 102a▷
```
let level = ref (Some Logs.Warning) in
```
Uses Flags.interactive 35e.

⟨CLI.main() options *elements* 92a⟩+≡ (35a) ◁91e 92d▷
```
(* pad: I added that *)
(* alt: reuse Logs_.cli_flags *)
"-v", Arg.Unit (fun () -> level := Some Logs.Info),
 " verbose mode";
"-verbose", Arg.Unit (fun () -> level := Some Logs.Info),
" verbose mode";
"-quiet", Arg.Unit (fun () -> level := None),
" ";
"-debug", Arg.Unit (fun () -> level := Some Logs.Debug),
" trace the main functions";
```
Uses Flags.eflag 92c, Flags.strict_mode, and Logs.level.Debug.

⟨CLI.main() *logging initializations* 92b⟩≡ (35a)
```
Logs_.setup !level ();
Logs.info (fun m -> m "ran as %s from %s" argv.(0) (Sys.getcwd ()));
```
Uses CLI.do_action() 102d.

13.4 Failing fast: rc -e

⟨*constant* Flags.eflag 92c⟩≡ (109b)
```
(* -e, for strict error checking. Abort the script when an error happens.*)
let eflag = ref false
```

The `-e` flag ("exit on error") is implemented at the opcode level, not as a runtime check. The compiler emits an `O.Eflag` opcode after each `Simple` and `Match` command when `eflag` is true. At runtime, `O.Eflag` checks the status and exits the shell if it is non-null. This per-command granularity is important: `eflag` is *not* checked after control flow operators like `if` or `&&`, only after commands that actually run programs. The `eflag` parameter is threaded through the compilation functions rather than using the global `Flags.eflag` directly. This allows the compiler to suppress error checking in contexts where it would be incorrect (e.g., inside an `if` condition, which is expected to produce non-zero statuses).

⟨CLI.main() options *elements* 92d⟩+≡ (35a) ◁92a 93b▷
```
"-e", Arg.Set Flags.eflag,
" exit if $status is non-null after a simple command";
```
Uses Flags.debugger.

⟨Opcode.operation *other cases* 92e⟩+≡ (27e) ◁90a 94c▷
```
(* Error management *)
| Eflag
```

⟨Compile.outcode_seq *in* A.Simple *case after emit* O.Simple 92f⟩≡ (58b)
```
if eflag
then emit (O.F O.Eflag);
```
Uses Opcode.operation.Eflag 27e and Opcode.t.F.

⟨Interpreter.interpret_operation() *match* operation *cases* 92g⟩+≡ (37c) ◁90b 95a▷
```
| O.Eflag ->
    if !Globals.eflagok && not (Status.truestatus())
    then Process.exit caps (Status.getstatus())
```
Uses Globals.eflagok 110b, Process.exit() 73b, Status.getstatus(), and Status.truestatus().

⟨Compile.outcode_seq *in* A.Match *case after emit* O.Match 92h⟩≡ (68a)
```
if eflag
then emit (O.F O.Eflag);
```
Uses Opcode.operation.Eflag 27e and Opcode.t.F.

13.5 Strict mode: `rc -strict`

⟨*constant* `Flags.strict_mode` 93a⟩≡ (109b)

```
let strict_mode = ref false
```

⟨`CLI.main() options` *elements* 93b⟩+≡ (35a) ◁92d 94a▷

```
(* pad: I added that *)
"-strict", Arg.Set Flags.strict_mode,
" strict mode";
```

Uses `Flags.dump_tokens`.

Chapter 14

Advanced Features

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, general redirections, subshells—are what make a shell truly useful for scripting.

14.1 Advanced flags

14.1.1 Reading commands from a string: `rc -c`

14.1.2 Restrict `$path` candidates: `rc -p`

⟨`CLI.main() options` *elements* 94a⟩+≡ (35a) ◁93b 100b▷
```
  "-p", Arg.Unit (fun () -> ()),
  " restrict $path to just (/bin /usr/bin)";
```
Uses `Flags.dump_opcodes`.

14.2 Advanced constructs

⟨`Ast.cmd` *other definition cases* 94b⟩≡ (25c)
```
  | DelFn of value
```

⟨`Opcode.operation` *other cases* 94c⟩+≡ (27e) ◁92e 95c▷
```
  | DelFn (* (name) *)
  (* less: RdFn *)
```

⟨`Parser.cmd` *other cases* 94d⟩+≡ (50c) ◁54i
```
  | TFn word       { DelFn $2 }
```

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 94e⟩+≡ (58a) ◁77c
```
  | A.DelFn w ->
      emit (O.F O.Mark);
      xword w;
      emit (O.F O.DelFn);
```
Uses `Ast.cmd.DelFn` 54b, `Opcode.operation.DelFn` 90a, `Opcode.operation.Mark` 27e, and `Opcode.t.F`.

⟨Interpreter.interpret_operation() *match* operation *cases* 95a⟩+≡ (37c) ◁92g 95i▷

```
  (* (name) *)
  | O.DelFn ->
      let t = R.cur () in
      let argv = t.R.argv in
      argv |> List.iter (fun s ->
        let x = Fn.flook s in
        match x with
        | Some _ -> Hashtbl.remove R.fns s
        | None ->
            (* stricter: *)
            if !Flags.strict_mode
            then E.error caps (spf "deleting undefined function %s" s)
      );
```

Uses Common.spf(), Error.error() 63c, Flags.strict_mode, Fn.flook() 30a, Opcode.operation.DelFn 90a, Runtime.cur() 30d, Runtime.fns, and Runtime.thread.argv.

14.2.1 Subshell: @ <cmd>

⟨*Parser tokens, operators other cases* 95b⟩≡ (24) 97a▷

```
  %token TSubshell
```

⟨Opcode.operation *other cases* 95c⟩+≡ (27e) ◁94c 96f▷

```
  (* ?? *)
  | Subshell (* {... Xexit} *)
```

⟨Lexer.token() *symbol cases* 95d⟩+≡ (41) ◁45c 96g▷

```
  | "@" { TSubshell }
```

⟨Parser.keyword *other cases* 95e⟩+≡ (51c) ◁52d

```
  | TSubshell { "@" }
```

14.2.2 Count and indexing of variables: $#<foo>, $<foo>(...)

⟨*Parser tokens, variables other cases* 95f⟩≡ (24) 97e▷

```
  %token TCount
```

⟨Compile.outcode_seq *in nested* xword() *match* w *cases* 95g⟩+≡ (58a) ◁79b

```
  | A.Count w ->
      emit (O.F O.Mark);
      xword w;
      emit (O.F O.Count);
```

Uses Ast.value.Count 26b, Opcode.operation.Count 27e, Opcode.operation.Mark 27e, and Opcode.t.F.

⟨Ast.value *other cases* 95h⟩≡ (26b) 96e▷

```
  | Count of value (* $#foo *)
  | Index of value * values (* $foo(...) *)
```

⟨Interpreter.interpret_operation() *match* operation *cases* 95i⟩+≡ (37c) ◁95a

```
  (* (name) *)
  | O.Count ->
      let t = R.cur () in
      let argv = t.R.argv in
      (match argv with
      | [varname] ->
          (* less: deglob *)
          let value = vlook_varname_or_index varname in
          let num =
```

```
          match value with
          | None -> 0
          | Some xs -> List.length xs
        in
        R.pop_list ();
        R.push_word (spf "%d" num)
    | _ -> E.error caps "variable name not singleton!"
    )
```

Uses `Common.spf()`, `Error.error()` 63c, `Interpreter.vlook_varname_or_index()`, `Opcode.operation.Count` 27e, `Runtime.cur()` 30d, `Runtime.pop_list()` 115, `Runtime.push_word()` 60f, and `Runtime.thread.argv`.

⟨`Lexer.token()` *variable cases* 96a⟩+≡ (41) ◁44c 97f▷

```
  | "$#"  { TCount }
```

⟨`Opcode.operation` *other stack cases* 96b⟩≡ (27e)

```
  | Count (* (name) *)
  | Concatenate (* (left)(right) *)
  | Stringify (* (name) *)
```

⟨`Opcode.operation` *other variable cases* 96c⟩≡ (27e)

```
  | Index  (* ??? *)
  | Local (* (name)(val) *)
  | Unlocal (* *)
```

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

⟨*Parser tokens, punctuation other cases* 96d⟩≡ (24) 97i▷

```
  %token TBackquote
```

⟨`Ast.value` *other cases* 96e⟩+≡ (26b) ◁95h 97g▷

```
  (* this causes the value and cmd types to be mutually recursive. Uses $IFS *)
  | CommandOutput of cmd_sequence
```

Uses `Ast.value` 105a.

⟨`Opcode.operation` *other cases* 96f⟩+≡ (27e) ◁95c

```
  | Backquote (* {... Xreturn} *)
```

⟨`Lexer.token()` *symbol cases* 96g⟩+≡ (41) ◁95d 96i▷

```
  | "‘" { TBackquote }
```

⟨`Parser.comword` *other cases* 96h⟩+≡ (51a) ◁55c 97h▷

```
  | TBackquote brace               { CommandOutput $2 }
```

14.2.4 Command substitution: ‘{<cmd>}

14.2.5 Read-write redirections: <> <file>

14.2.6 General redirections: >[2] <file>

⟨`Lexer.token()` *symbol cases* 96i⟩+≡ (41) ◁96g 97m▷

```
  | ">[" (['0'-'9']+ (*as fd0*)) "=" (['0'-'9']+ (*as fd1*)) "]"
         {  let fd0 = failwith "TODO: fd0" in
     let fd1 = failwith "TODO: fd1" in
     TDup (Ast.RWrite, int_of_string fd0, int_of_string fd1)
  }
  | ">>[" (['0'-'9']+ (*as fd0*)) "=" (['0'-'9']+ (*as fd1*)) "]"
         {
             let fd0 = failwith "TODO: fd0" in
```

```
    let fd1 = failwith "TODO: fd1" in
    TDup (Ast.RAppend, int_of_string fd0, int_of_string fd1)
}
(* less: advanced pipe and redirection *)
```

14.2.7 Advanced dup: `>[<fd0>=<fd1> ]`, `<>[<fd0>=<fd1>]` , `<[<fd0>=<fd1>]`

⟨*Parser tokens, operators other cases* 97a⟩+≡ (24) ◁95b
```
%token<Ast.redirection_kind * int * int> TDup
```

⟨`Parser.redir` *other cases* 97b⟩≡ (52i)
```
| TDup        { Right $1 }
```

⟨`Ast.redirection_kind` *other cases* 97c⟩≡ (27a)
```
(* less: RHere *) (* < < *)
| RDup of int * int (* >[x=y] *)
```

⟨`Ast.cmd` *other redirection cases* 97d⟩≡ (25c)
```
| Dup of cmd * redirection_kind * int * int (* >[1=2] *)
```

14.2.8 Advanced pipes: `|[<fd>]` , `|[<fd0>=<fd1>]`

14.2.9 Here documents: `<< <HERE>`

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

⟨*Parser tokens, variables other cases* 97e⟩+≡ (24) ◁95f
```
%token TStringify
```

⟨`Lexer.token()` *variable cases* 97f⟩+≡ (41) ◁96a
```
| "$\"" { TStringify }
```

⟨`Ast.value` *other cases* 97g⟩+≡ (26b) ◁96e 98▷
```
| Stringify of value (* $"foo " *)
```

⟨`Parser.comword` *other cases* 97h⟩+≡ (51a) ◁96h
```
| TCount word                   { Count $2 }
| TDollar word TSub words TCPar { Index ($2, $4) }
| TStringify word               { Stringify $2 }
```

⟨*Parser tokens, punctuation other cases* 97i⟩+≡ (24) ◁96d 97j▷
```
%token TSub
```

14.2.11 String concatenation

⟨*Parser tokens, punctuation other cases* 97j⟩+≡ (24) ◁97i
```
%token TCaret
```

⟨`Parser.word` *other cases* 97k⟩≡ (51b)
```
| word TCaret word { Concat ($1, $3) }
```

⟨`Parser.first` *other cases* 97l⟩≡ (50g)
```
| first TCaret word { Concat ($1, $3) }
```

⟨`Lexer.token()` *symbol cases* 97m⟩+≡ (41) ◁96i
```
| "^" { TCaret }
```

⟨Ast.value *other cases* 98⟩+≡ (26b) ◁97g

```
  (* ^ distributes over lists *)
  | Concat of value * value
```

14.3 Advanced builtins

14.3.1 exec

14.3.2 whatis

14.3.3 rfork

14.3.4 shift

Chapter 15

Conclusion

You now know how the Plan 9 shell `rc` works, and more generally how many shells work. Despite its modest size, `rc` implements a complete programming language: a lexer, a 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—`fork()`, `exec()`, `wait()`, `pipe()`, `open()`, `close()`, `dup2()`—to implement pipes, redirections, 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.

This OCaml port demonstrates that the core ideas of `rc` translate cleanly to a higher-level language. Algebraic types replace the generic `Tree` struct, pattern matching replaces the `switch` on type tags, and closures replace global code buffers. The garbage collector eliminates reference counting on code vectors and manual memory management on word lists. Yet the fundamental architecture—compile to bytecodes, interpret with a run queue of threads—remains the same.

Appendix A

Debugging for the rc Developer

A.1 Exception backtraces

⟨*constant* Flags.debugger 100a⟩≡ (109b)
```
let debugger = ref false
```

⟨CLI.main() options *elements* 100b⟩+≡ (35a) ◁94a 100d▷
```
"-debugger", Arg.Set Flags.debugger,
" ";
```
Uses Flags.rflag 101f.

⟨CLI.main() *when* exn *thrown in* interpret() 100c⟩≡ (35a)
```
if !Flags.debugger
then raise exn
else
  (match exn with
  ⟨CLI.main() when Failure exn thrown in interpret() 35b⟩
  (* alt: could catch Exit.ExitCode here but this is done in Main.ml *)
  | _ -> raise exn
  )
```
Uses Exit.t.Code and Logs.err().

A.2 Dumpers

⟨CLI.main() options *elements* 100d⟩+≡ (35a) ◁100b 101g▷
```
(* pad: I added that *)
"-dump_tokens", Arg.Set Flags.dump_tokens,
" dump the tokens as they are generated";
"-dump_ast", Arg.Set Flags.dump_ast,
" dump the parsed AST";
"-dump_opcodes", Arg.Set Flags.dump_opcodes,
" dump the generated opcodes ";
```

A.2.1 Dumping tokens

⟨*constant* Flags.dump_tokens 100e⟩≡ (109b)
```
let dump_tokens = ref false
```

⟨Parse.lexfunc() *possibly dump tokens* 101a⟩≡ (40a)
```
  |> (fun tok ->
      if !Flags.dump_tokens
      then Logs.app (fun m -> m "%s" (Dumper.dump (tok,s)));
      tok)
```
Uses Dumper.dump(), Flags.dump_tokens, and Logs.app().

A.2.2 Dumping the AST

⟨*constant* Flags.dump_ast 101b⟩≡ (109b)
```
  let dump_ast = ref false
```

⟨Parse.parse_line() *possibly dump AST* 101c⟩≡ (47a)
```
  |> (fun ast -> if !Flags.dump_ast then Logs.app (fun m -> m "%s" (Ast.show_line ast)); ast)
```

A.2.3 Dumping the opcodes

⟨*constant* Flags.dump_opcodes 101d⟩≡ (109b)
```
  let dump_opcodes = ref false
```

⟨Compile.compile() *possibly dump returned opcodes* 101e⟩≡ (56a)
```
  |> (fun x -> if !Flags.dump_opcodes then Logs.app (fun m -> m "%s" (Opcode.show_codevec x)); x)
```
Uses Flags.dump_opcodes and Logs.app().

The -r flag is invaluable for understanding how the interpreter works. At each step of the interpreter loop, it prints the process ID, the program counter, the current opcode, and the contents of the argument stack. This lets you trace exactly how a command is compiled and executed, seeing the Mark/Word/Simple stack machine protocol in action.

A.3 Opcode generator trace: rc -r

⟨*constant* Flags.rflag 101f⟩≡ (109b)
```
  (* -r, similar to dump_opcodes, but at each step *)
  let rflag = ref false
```

⟨CLI.main() options *elements* 101g⟩+≡ (35a) ◁100d 102b▷
```
  "-r", Arg.Set Flags.rflag,
  " print internal form of commands (opcodes)";
```
Uses CLI.usage 35d.

⟨CLI.interpret() *if* rflag 101h⟩≡ (36f)
```
  (* less: cycle =~ codevec pointer *)
  if !Flags.rflag
  then Logs.app (fun m -> m "pid %d %d %s %s"
              (Unix.getpid ())
              !pc
              (Opcode.show t.R.code.(!pc))
              ( (t.R.argv::t.R.argv_stack) |> List.map (fun xs ->
                  spf "(%s)" (String.concat " " xs)
                ) |> String.concat " "));
```
Uses Common.spf(), Interpreter.interpret_operation(), Opcode.t.F, Opcode.t.S 27e, Runtime.thread.argv, Runtime.thread.argv_stack 38c, and Runtime.thread.code 29h.

A.4 CLI actions

⟨CLI.main() *debugging initializations* 102a⟩+≡ (35a) ◁91g
```
  let action = ref "" in
```
Uses Flags.interactive 35e.

⟨CLI.main() options *elements* 102b⟩+≡ (35a) ◁101g
```
  (* pad: I added that *)
  "-test_parser", Arg.Unit (fun () -> action := "-test_parser"), " ";
```
Uses UConsole.print().

⟨CLI.main() *CLI action processing* 102c⟩≡ (35a)
```
  (* to test and debug components of mk *)
  if !action <> "" then begin
    do_action caps !action (List.rev !args);
    raise (Exit.ExitCode 0)
  end;
```
Uses Var.vinit().

⟨*function* CLI.do_action 102d⟩≡ (106b)
```
  let do_action caps s xs =
    match s with
    | "-test_parser" ->
        xs |> List.iter (fun file ->
          let file = Fpath.v file in
          Logs.info (fun m -> m "processing %s" !!file);
          file |> FS.with_open_in caps (fun (chan : Chan.i) ->
            let lexbuf = Lexing.from_channel chan.Chan.ic in

            (* for error reporting I need a runq *)
            let t = Runtime.mk_thread [||] 0 (Hashtbl.create 0) in
            R.runq := t::!R.runq;
            t.R.file <- Some file;

            let rec loop () =
              let line = Parse.parse_line lexbuf in
              match line with
              | Some seq ->
                Logs.app (fun m -> m "%s" (Ast.show_cmd_sequence seq));
                loop ();
              | None -> ()
            in
            loop ()
         )
        )

    | _ -> failwith ("action not supported: " ^ s)
    (* old: do that in caller now, so more explicit
    runq := t::!runq
    *)
```
Uses Chan.i.ic, FS.with_open_in(), Fpath_.Operators.!!(), Logs.app(), Logs.info(), Parse.parse_line(), Runtime.mk_thread() 71c, Runtime.runq, and Runtime.thread.file 36b.

Appendix B

Examples of rc scripts

B.1 /rc/lib/rcmain

This is the initialization script sourced by the bootstrap bytecodes (see Chapter 11). 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 on Unix version
if(~ $#home 0) home=$HOME
if(~ $#home 0) home=/
if(~ $#ifs 0) ifs='
'

switch($#prompt){
case 0
 prompt=('; ' ' ')
case 1
 prompt=($prompt ' ')
}
if(~ $rcname ?.out ?.rc */?.rc */?.out) prompt=('broken! ' ' ')
if(flag p) path=(/bin /usr/bin)
if not{
 finit
 # should be taken care of by rc now, but leave just in case
}
fn sigexit
if(! ~ $#cflag 0){
 if(flag l && test -r $home/lib/profile) . $home/lib/profile
 status=''
 eval $cflag
 exit $status
}
if(flag i){
 if(flag l && test -r $home/lib/profile) . $home/lib/profile
 status=''
 if(! ~ $#* 0) . $*
 . -i '/dev/stdin'
```

```
 exit $status
}
if(flag l && test -r $home/lib/profile) . $home/lib/profile
if(~ $#* 0){
 . /dev/stdin
 exit $status
}
status=''
. $*
exit $status
```

B.2 /home/pad/lib/profile

Appendix C

Extra Code

C.1 Ast.ml

```
⟨shell/Ast.ml 105a⟩≡
  (* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)

  (* for boostrap-mk.sh and ocaml-light to work without deriving *)
  let show_cmd _ = "NO DERIVING"
  [@@warning "-32"]
  let show_value _ = "NO DERIVING"
  [@@warning "-32"]
  let show_cmd_sequence _ = "NO DERIVING"
  [@@warning "-32"]
  let show_line _ = "NO DERIVING"
  [@@warning "-32"]

  ⟨type Ast.value 26b⟩
  ⟨type Ast.values 26a⟩

  ⟨type Ast.cmd 25c⟩
  ⟨type Ast.cmd_sequence 25b⟩

  (* todo: RDup does not have a filename? so push value inside RWrite of value*)
  ⟨type Ast.redirection 26c⟩
  ⟨type Ast.redirection_kind 27a⟩
  [@@deriving show]

  ⟨type Ast.line 25a⟩
  [@@deriving show]
```

C.2 Builtin.mli

```
⟨Builtin.mli 105b⟩≡
  ⟨signature Builtin.is_builtin 61c⟩

  ⟨signature Builtin.dispatch 61d⟩
```

C.3 Builtin.ml

```
⟨shell/Builtin.ml 105c⟩≡
  (* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
```

```
open Common

module R = Runtime
module E = Error
module O = Opcode

⟨function Builtin.is_builtin 81a⟩
⟨constant Builtin.ndots 83b⟩
⟨function Builtin.dochdir 82b⟩
⟨constant Builtin.dotcmds 84a⟩
⟨function Builtin.dispatch 81b⟩
```

C.4 CLI.mli

⟨CLI.mli 106a⟩≡

```
⟨type CLI.caps 34a⟩

⟨signature CLI.main 34b⟩

(* internals *)
⟨signature CLI.interpret_bootstrap 35c⟩
```

C.5 CLI.ml

⟨shell/CLI.ml 106b⟩≡

```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators
open Regexp.Operators

module R = Runtime
module O = Opcode

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An OCaml port of rc, the Plan 9 shell.
 *
 * Main limitations compared to rc:
 *  - no unicode support
 *  - not all of the fancy redirections and fancy pipes
 *  - no storing of functions in the environment
 *    (used by rcmain. But can do the same by using '. rcmain')
 *
 * Improvements (IMHO):
 *  - a strict mode where we report when deleting undefined function
 *  - Logs with errors and warnings and debug
 *  - more?
 *
 * todo:
 *  - read environment variables and export variables
 *  - globbing
 *  - Isatty rc -i detection
 *  - add ~ shortcut for HOME (from csh?)
 *  - rc -c
```

```
 *)

(****************************************************************************)
(* Types and constants *)
(****************************************************************************)

⟨type CLI.caps 34a⟩

(* -d and -p are dead according to man page so I removed them *)
⟨constant CLI.usage 35d⟩

(****************************************************************************)
(* Testing *)
(****************************************************************************)

⟨function CLI.do_action 102d⟩

(****************************************************************************)
(* Main algorithm *)
(****************************************************************************)

⟨constant CLI._bootstrap_simple 37a⟩
⟨function CLI.bootstrap 88⟩

⟨function CLI.interpret_bootstrap 36f⟩

(****************************************************************************)
(* Entry point *)
(****************************************************************************)

⟨function CLI.main 35a⟩
```

Uses Logs.level.Warning.

C.6 Compile.mli

```
⟨Compile.mli 107a⟩≡
  ⟨signature Compile.compile 27b⟩
```

C.7 Compile.ml

```
⟨shell/Compile.ml 107b⟩≡
  open Common

  module A = Ast
  module O = Opcode

  (****************************************************************************)
  (* Prelude *)
  (****************************************************************************)
  (* AST to opcodes.
  *)

  (****************************************************************************)
  (* Helpers *)
  (****************************************************************************)
```

```
⟨function Compile.split_at_non_assign 79a⟩
⟨function Compile.split_when_case 76e⟩

(*****************************************************************************)
(* Compilation algorithm *)
(*****************************************************************************)

⟨function Compile.outcode_seq 58a⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function Compile.compile 56a⟩
```

C.8 Env.mli

```
⟨Env.mli 108a⟩≡
  type t = (string * string list) list

  val read_environment : < Cap.env ; .. > -> t
```

C.9 Env.ml

```
⟨shell/Env.ml 108b⟩≡
  open Common
  open Regexp.Operators

  type t = (string * string list) list

  (* copy paste of mk/Shellenv.ml
   * alt: move to lib_core/commons/Proc.ml to factorize?
   * This is Unix specific; In plan9 one needs to read /env/.
  *)
  let read_environment (caps : < Cap.env; ..>) =
    CapUnix.environment caps () |> Array.to_list |> List.map (fun s ->
      if s =~ "\\([^=]+\\)=\\(.*\\)"
      then
        let (var, str) = Regexp.matched2 s in
        var, Regexp.split "[ \t]+" str
      else failwith (spf "wrong format for environment variable: %s" s)
    )
```

C.10 Error.mli

```
⟨Error.mli 108c⟩≡
  ⟨signature Error.error 63b⟩
```

C.11 Error.ml

```
⟨shell/Error.ml 109a⟩≡
  module R = Runtime

  ⟨function Error.error 63c⟩
```

C.12 Flags.ml

```
⟨shell/Flags.ml 109b⟩≡
  open Common

  ⟨constant Flags.interactive 35e⟩
  ⟨constant Flags.login 36d⟩

  ⟨constant Flags.eflag 92c⟩
  ⟨constant Flags.rflag 101f⟩
  ⟨constant Flags.sflag 91d⟩
  ⟨constant Flags.xflag 91a⟩

  (* less: let cflag = ref "" *)

  ⟨global Flags.hflags 84b⟩

  ⟨constant Flags.rcmain 89a⟩

  (* pad: I added this one *)
  ⟨constant Flags.strict_mode 93a⟩

  ⟨constant Flags.dump_tokens 100e⟩
  ⟨constant Flags.dump_ast 101b⟩
  ⟨constant Flags.dump_opcodes 101d⟩

  ⟨constant Flags.debugger 100a⟩
```

C.13 Fn.mli

```
⟨Fn.mli 109c⟩≡
  ⟨signature Fn.flook 29i⟩
```

C.14 Fn.ml

```
⟨shell/Fn.ml 109d⟩≡
  module R = Runtime

  ⟨function Fn.flook 30a⟩
```

C.15 Glob.mli

```
⟨Glob.mli 109e⟩≡
```

C.16 Glob.ml

⟨shell/Glob.ml 110a⟩≡

C.17 Globals.ml

⟨*constant* Globals.eflagok 110b⟩≡ (110c)
```
(* this is set after the first . of rcmain *)
let eflagok = ref false
```

⟨shell/Globals.ml 110c⟩≡

⟨*constant* Globals.skipnl 44g⟩
⟨*constant* Globals.errstr 63d⟩

⟨*constant* Globals.ifnot 74d⟩
⟨*constant* Globals.eflagok 110b⟩

C.18 Heredoc.mli

⟨Heredoc.mli 110d⟩≡

C.19 Heredoc.ml

⟨shell/Heredoc.ml 110e⟩≡

C.20 Interpreter.mli

⟨Interpreter.mli 110f⟩≡
⟨*signature* Interpreter.interpret_operation 37b⟩

C.21 Interpreter.ml

⟨shell/Interpreter.ml 110g⟩≡
```
open Common
open Regexp.Operators

module O = Opcode
module R = Runtime
module E = Error

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* Interpreter.file_descr_of_int 70a⟩
⟨*function* Interpreter.int_at_address 67b⟩

```
⟨function Interpreter.vlook_varname_or_index 79d⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function Interpreter.interpret_operation 37c⟩
```

C.22 Lexer.mli

```
⟨Lexer.mli 111a⟩≡
  ⟨signature Lexer.token 40b⟩

  ⟨exception Lexer.Lexical_error 40c⟩
```

C.23 Lexer.mll

C.24 Main.ml

```
⟨shell/Main.ml 111b⟩≡
  (* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
  open Xix_shell

  (*****************************************************************************)
  (* Entry point *)
  (*****************************************************************************)

  ⟨toplevel Main._1 34c⟩
```

C.25 Op_process.ml

```
⟨shell/Op_process.ml 111c⟩≡
  open Fpath_.Operators

  module R = Runtime
  module E = Error

  ⟨function Op_process.execute 62b⟩

  ⟨function Op_process.exec 62a⟩

  ⟨function Op_process.forkexec 61f⟩

  ⟨function Op_process.op_Simple 60d⟩
```

Uses `Runtime.doredir()` 70e and `Runtime.thread.redirections` 30b.

C.26 Op_repl.ml

```
⟨shell/Op_repl.ml 111d⟩≡

  module R = Runtime
```

⟨*function* Op_repl.op_REPL 38g⟩

C.27 Opcode.ml

⟨shell/Opcode.ml 112a⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)

(* for boostrap-mk.sh and ocaml-light to work without deriving *)
let show _ = "NO DERIVING"
[@@warning "-32"]
let show_codevec _ = "NO DERIVING"
[@@warning "-32"]

⟨type Opcode.operation 27e⟩
[@@deriving show]
⟨type Opcode.opcode 27d⟩
[@@deriving show]
⟨type Opcode.codevec 27c⟩
[@@deriving show]
```

C.28 PATH.mli

⟨PATH.mli 112b⟩≡
```
⟨signature PATH.find_in_path 65b⟩

⟨signature PATH.search_path_for_cmd 65c⟩
```

C.29 PATH.ml

⟨shell/PATH.ml 112c⟩≡
```
open Common
open Fpath_.Operators
open Regexp.Operators

module R = Runtime

⟨function PATH.search_path_for_cmd 66a⟩
⟨function PATH.find_in_path 66b⟩
```

C.30 Parser.mly

C.31 Parse.mli

⟨Parse.mli 112d⟩≡
```
⟨signature Parse.parse_line 39a⟩
```

C.32 Parse.ml

⟨shell/Parse.ml 113a⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators

module R = Runtime

⟨function Parse.error 48b⟩

⟨function Parse.parse_line 47a⟩
```

C.33 Pattern.mli

⟨signature Pattern.match_str 113b⟩≡ (113c)
```
val match_str : string -> pattern -> bool
```

⟨Pattern.mli 113c⟩≡
```
(* todo: Glob char *)
⟨type Pattern.pattern 113d⟩

⟨signature Pattern.match_str 113b⟩
```

C.34 Pattern.ml

⟨type Pattern.pattern 113d⟩≡ (113)
```
type pattern = string
```

⟨function Pattern.match_str 113e⟩≡ (113f)
```
(* todo: handle [ * ?  *)
let match_str s1 s2 =
  s1 = s2
```

⟨shell/Pattern.ml 113f⟩≡
```

⟨type Pattern.pattern 113d⟩

⟨function Pattern.match_str 113e⟩
```

C.35 Process.mli

⟨Process.mli 113g⟩≡
```
⟨type Process.waitfor_result 64i⟩

⟨signature Process.return 57b⟩
⟨signature Process.exit 73a⟩
⟨signature Process.waitfor 64h⟩
⟨signature Process.s_of_unix_error 63e⟩
```

C.36 Process.ml

```
⟨shell/Process.ml 114a⟩≡
  open Common

  module R = Runtime

  ⟨function Process.s_of_unix_error 63f⟩

  ⟨function Process.exit 73b⟩

  ⟨function Process.return 57c⟩


  ⟨type Process.waitfor_result 64i⟩

  ⟨function Process.waitfor 65a⟩
```

C.37 Prompt.mli

```
⟨Prompt.mli 114b⟩≡
  ⟨signature Prompt.doprompt 42e⟩
  ⟨signature Prompt.prompt 42h⟩

  ⟨signature Prompt.pprompt 42j⟩
```

C.38 Prompt.ml

```
⟨shell/Prompt.ml 114c⟩≡
  module R = Runtime

  ⟨constant Prompt.doprompt 42f⟩
  ⟨constant Prompt.prompt 42i⟩

  ⟨function Prompt.pprompt 43a⟩
```

C.39 Runtime.mli

```
⟨Runtime.mli 114d⟩≡

  ⟨type Runtime.varname 28b⟩
  ⟨type Runtime.value 28a⟩
  ⟨type Runtime.var 28c⟩

  val string_of_var : var -> string

  ⟨type Runtime.fn 29h⟩
  val show_fn: fn -> string

  ⟨type Runtime.thread 30b⟩
  ⟨type Runtime.redir 32f⟩

  ⟨type Runtime.waitstatus 33b⟩
```

```
(* globals *)

⟨signature Runtime.globals 29a⟩
⟨signature Runtime.fns 29f⟩
⟨signature Runtime.runq 30c⟩

(* API *)

⟨signature Runtime.cur 30e⟩
⟨signature Runtime.push_list 59f⟩
⟨signature Runtime.pop_list 60e⟩
⟨signature Runtime.push_word 31b⟩
⟨signature Runtime.pop_word 31d⟩
⟨signature Runtime.push_redir 70b⟩
⟨signature Runtime.pop_redir 70d⟩
⟨signature Runtime.turf_redir 70g⟩
⟨signature Runtime.doredir 71b⟩

⟨signature Runtime.mk_thread 30g⟩
```

C.40 Runtime.ml

```
⟨shell/Runtime.ml 115⟩≡
  (* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Globals used by rc during interpretation.
   *)

  (*****************************************************************************)
  (* Types and globals *)
  (*****************************************************************************)

  ⟨type Runtime.varname 28b⟩
  ⟨type Runtime.value 28a⟩
  ⟨type Runtime.var 28c⟩

  (* for boostrap-mk.sh and ocaml-light to work without deriving *)
  let show_fn _ = "NO DERIVING"
  [@@warning "-32"]

  ⟨type Runtime.fn 29h⟩
  [@@deriving show]

  ⟨type Runtime.thread 30b⟩

  ⟨type Runtime.redir 32f⟩

  ⟨type Runtime.waitstatus 33b⟩

  ⟨global Runtime.globals 29b⟩

  ⟨global Runtime.fns 29g⟩

  (* less: argv0 *)
```

```
⟨constant Runtime.runq 30d⟩

(*****************************************************************************)
(* cur *)
(*****************************************************************************)

⟨function Runtime.cur 30f⟩

(*****************************************************************************)
(* Stack (argv) API *)
(*****************************************************************************)

⟨function Runtime.push_list 59g⟩
⟨function Runtime.pop_list 60f⟩

⟨function Runtime.push_word 31c⟩
⟨function Runtime.pop_word 31e⟩

(*****************************************************************************)
(* Redirection *)
(*****************************************************************************)

⟨function Runtime.push_redir 70c⟩
⟨function Runtime.pop_redir 70e⟩

⟨function Runtime.turf_redir 70h⟩

⟨function Runtime.doredir 71c⟩

(*****************************************************************************)
(* Constructor *)
(*****************************************************************************)

⟨function Runtime.mk_thread 31a⟩

(*****************************************************************************)
(* Misc *)
(*****************************************************************************)
let string_of_var (var : var) : string =
  match var.v with
  | None -> ""
  | Some [] -> ""
  | Some [x] -> x
  | Some xs -> "(" ^ String.concat " " xs ^ ")"
```

Uses `Runtime.cur()` 30d, `Runtime.redir.Close`, `Runtime.thread.argv`, `Runtime.thread.argv_stack` 38c, `Runtime.thread.redirections` 30b, `Runtime.thread.waitstatus` 30b, `Runtime.var`, and `Runtime.waitstatus.NothingToWaitfor` 32f.

C.41 Status.mli

```
⟨Status.mli 116⟩≡
  (* helpers to manipulate the "status" special variable (see Var.ml) *)

  ⟨signature Status.setstatus 64a⟩
  ⟨signature Status.getstatus 64c⟩
  ⟨signature Status.concstatus 73e⟩
  ⟨signature Status.truestatus 64e⟩
```

C.42 Status.ml

⟨shell/Status.ml 117a⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Regexp.Operators

module R = Runtime

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Small helpers to manipulate the "status" special variable (see Var.ml) *)

(*****************************************************************************)
(* API *)
(*****************************************************************************)

⟨function Status.setstatus 64b⟩
⟨function Status.getstatus 64d⟩
⟨function Status.concstatus 73f⟩
⟨function Status.truestatus 64f⟩
```

C.43 Var.mli

⟨Var.mli 117b⟩≡
```
(* Helpers to manipulate rc shell variables *)

⟨signature Var.gvlook 29c⟩
⟨signature Var.vlook 32a⟩
⟨signature Var.setvar 32c⟩
⟨signature Var.vinit 29e⟩
```

C.44 Var.ml

⟨shell/Var.ml 117c⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common

module R = Runtime

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Helpers to manipulate rc shell variables *)

(*****************************************************************************)
(* API *)
(*****************************************************************************)

⟨function Var.gvlook 29d⟩
⟨function Var.vlook 32b⟩

⟨function Var.setvar 32d⟩

⟨function Var.vinit 86⟩
```

Glossary

LOC = Lines Of Code
CLI = Command-Line Interface

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.

Ast.cmd: 25c, 26a, 53h, 98
Ast.cmd.And: 25c, 66e
Ast.cmd.Assign: 53h, 77c, 79a
Ast.cmd.Async: 26a, 58a
Ast.cmd.Compound: 25c, 77a
Ast.cmd.DelFn: 54b, 94e
Ast.cmd.Dup: 25c, 58a
Ast.cmd.EmptyCommand: 66d, 77c, 98
Ast.cmd.Fn: 53h, 77b
Ast.cmd.For: 25c, 58a
Ast.cmd.ForIn: 25c, 58a
Ast.cmd.If: 74a
Ast.cmd.IfNot: 25c, 75a
Ast.cmd.Match: 25c, 68a
Ast.cmd.Not: 25c, 67e
Ast.cmd.Or: 25c, 67c
Ast.cmd.Pipe: 26a, 71d
Ast.cmd.Redir: 68c
Ast.cmd.Simple: 58b, 75c, 76e
Ast.cmd.Switch: 25c, 75c
Ast.cmd.While: 25c, 58a
Ast.cmd_sequence: 25c
Ast.line: 27a
Ast.redirection_kind.RAppend: 68c
Ast.redirection_kind.RRead: 25b, 68c
Ast.redirection_kind.RWrite: 68c
Ast.redirection_kind: 25c
Ast.value: 25c, 26b, 53h, 54b, 96e, 105a
Ast.value.CommandOutput: 26b, 58a
Ast.value.Concat: 58a
Ast.value.Count: 26b, 95g
Ast.value.Dollar: 26b, 79b
Ast.value.Index: 26b, 58a
Ast.value.List: 26b, 59c
Ast.value.Stringify: 26b, 58a
Ast.value.Word: 26b, 59a, 75c, 76e
Ast.values: 96e
Builtin.dispatch(): 61b

Builtin.dochdir(): 82a
Builtin.dotcmds: 83a
Builtin.is_builtin(): 81a
Builtin.ndots: 83a
Cap.main(): 34c
CapStdlib.exit(): 73b
CapStdlib.open_in(): 83a
CapSys.argv(): 34c
CapUnix.chdir(): 82b
CapUnix.execv(): 62b
CapUnix.fork(): 62a, 72a
CapUnix.wait(): 65a
Chan.i.ic: 102d
CLI.bootstrap(): 88, 88
CLI.caps: 34a
CLI.do_action(): 92b, 102d
CLI.interpret_bootstrap(): 36f, 84c, 88
CLI.main(): 34c, 106b
CLI.usage: 35d, 101g
CLI._bootstrap_simple: 37a
Common.= (): 35f, 64f, 66a, 79d
Common.spf(): 47a, 48b, 57g, 59d, 63f, 65a, 67b, 69b, 70a, 79d, 81b, 83a, 95a, 95i, 101h
Common2.span(): 76e
Compile.compile(): 38g, 56a
Compile.outcode_seq(): 56a, 58a
Compile.split_at_non_assign(): 77c, 79a, 79a
Compile.split_when_case(): 75c, 76e
Dumper.dump(): 101a
Error.error(): 60d, 62b, 63c, 69b, 78a, 78b, 79c, 83a, 84d, 91c, 95a, 95i
Exit.catch(): 34c
Exit.exit(): 34c
Exit.t.Code: 100c
Exit.t.OK: 84c
Flags.debugger: 35a, 92d
Flags.dump_opcodes: 94a, 101e
Flags.dump_tokens: 93b, 101a
Flags.eflag: 56a, 92a, 92c
Flags.hflags: 84c, 84d
Flags.interactive: 35e, 91g, 102a
Flags.login: 35a, 36d
Flags.rcmain: 36a, 88, 89a
Flags.rflag: 36f, 100b, 101f
Flags.sflag: 91d, 91f
Flags.strict_mode: 92a, 95a
Flags.xflag: 61f, 91a
Fn.flook(): 30a, 95a
Fpath_.Operators.!!(): 102d
FS.with_open_in(): 102d
Globals.eflagok: 83a, 92g, 110b

Globals.errstr: 62b, 63d, 65a
Globals.ifnot: 74b, 74c, 74d, 74e
Globals.skipnl: 44g, 44j
Interpreter.file_descr_of_int(): 69b, 70a, 72a
Interpreter.interpret_operation(): 101h
Interpreter.int_at_address(): 67a, 67d, 69b, 72a, 74b, 74c, 76a, 76c
Interpreter.vlook_varname_or_index(): 79c, 95i
Lexer.token(): 44j
Logs.app(): 36f, 61f, 91f, 101a, 101e, 102d
Logs.err(): 39b, 62b, 63a, 63c, 82a, 86, 90b, 100c
Logs.info(): 35a, 82b, 83a, 102d
Logs.level.Debug: 92a
Logs.level.Info: 91b, 91e
Logs.level.Warning: 106b
Logs.set_level(): 35a
Opcode.operation: 27e
Opcode.operation.Append: 27e, 37c, 68c
Opcode.operation.Assign: 77c, 78a, 88, 96b
Opcode.operation.Async: 37c, 96c
Opcode.operation.Backquote: 37c, 94c
Opcode.operation.Case: 27e, 75c, 76c
Opcode.operation.Close: 27e, 37c
Opcode.operation.Concatenate: 27e
Opcode.operation.Count: 27e, 95g, 95i
Opcode.operation.DelFn: 90a, 94e, 95a
Opcode.operation.Dollar: 79b, 79c, 88, 96b
Opcode.operation.Dup: 27e, 37c
Opcode.operation.Eflag: 27e, 92f, 92h
Opcode.operation.Exit: 27e, 71d, 88
Opcode.operation.False: 67c, 76b
Opcode.operation.Fn: 37c, 37e, 77b
Opcode.operation.For: 27e, 37c
Opcode.operation.Glob: 27e, 68c, 90b
Opcode.operation.If: 27e, 74a
Opcode.operation.IfNot: 27e, 75a
Opcode.operation.Jump: 27e, 75c, 76a
Opcode.operation.Local: 27e, 77c, 78b, 84a
Opcode.operation.Mark: 27e, 58b, 68a, 68c, 75c, 77b, 77c, 79b, 84a, 88, 94e, 95g
Opcode.operation.Match: 27e, 68a, 68b
Opcode.operation.Not: 67e
Opcode.operation.Pipe: 37d, 71d, 72a
Opcode.operation.PipeFd: 37c, 37d
Opcode.operation.PipeWait: 27e, 71d, 73d
Opcode.operation.Popm: 27e, 75c, 76d
Opcode.operation.Popredir: 27e, 68c
Opcode.operation.Read: 27e, 37c, 68c
Opcode.operation.ReadWrite: 27e, 37c
Opcode.operation.REPL: 37a, 38f, 84a, 95c
Opcode.operation.Return: 27e, 56a, 71d, 77b, 84a

Opcode.operation.Simple: 58b, 60c, 88, 96c
Opcode.operation.Stringify: 27e
Opcode.operation.Subshell: 37c, 92e
Opcode.operation.True: 66e
Opcode.operation.Unlocal: 27e, 37c, 77b, 77c, 84a
Opcode.operation.Wastrue: 74a, 75b
Opcode.operation.Word: 27e, 59a, 59d, 84a, 88
Opcode.operation.Write: 27e, 68c, 69b
Opcode.t: 96f
Opcode.t.F: 37a, 37c, 56a, 58b, 59a, 66e, 67c, 67e, 68a, 68c, 71d, 74a, 75a, 75c, 77b, 77c, 79b, 84a, 88, 92f, 92h, 94e, 95g, 101h
Opcode.t.I: 27e, 57e, 66e, 67b, 67c, 68c, 71d, 74a, 75a, 75c, 77b
Opcode.t.S: 27e, 59a, 59d, 77b, 84a, 88, 101h
Op_process.exec(): 62b
Op_process.execute(): 62a
Op_process.forkexec(): 61e, 62a
Op_process.op_Simple(): 60c, 61f
Op_repl.op_REPL(): 38f, 38g
Parse.error(): 47a
Parse.parse_line(): 38g, 102d
Parser.rc(): 47a
Parser.token.EOF: 47a
Parser.token.TNewline: 44j, 48b
PATH.find_in_path(): 66a
PATH.search_path_for_cmd(): 62a, 112c
Pattern.match_str(): 68b, 76c, 113e
Pattern.pattern: 113d
Process.exit(): 57c, 72b, 73b, 92g
Process.return(): 57a, 57c, 63c
Process.s_of_unix_error(): 60d, 62b, 63f, 65a
Process.waitfor(): 60d, 65a, 73d
Process.waitfor_result.WaitforFound: 64i, 65a
Process.waitfor_result.WaitforInterrupted: 60d, 64i, 65a
Process.waitfor_result.WaitforNotfound: 64i, 65a
Process.waitfor_result: 64i
Profiling.profile_code(): 36f
Prompt.doprompt: 42f, 42g, 43a
Prompt.pprompt(): 42g, 43a, 44j
Prompt.prompt: 42i, 43a, 43b, 43d
Runtime.cur(): 30d, 30f, 31e, 32b, 38g, 43a, 48b, 59d, 60f, 61f, 62b, 63c, 67a, 67d, 68b, 69b, 70e, 72a, 73d, 74b, 74c, 76a, 76c, 78a, 78b, 79c, 95a, 95i, 115
Runtime.doredir(): 70e, 111c
Runtime.fn.code: 115
Runtime.fn.pc: 115
Runtime.fns: 30a, 95a
Runtime.globals: 29d, 33b
Runtime.mk_thread(): 38g, 71c, 72a, 83a, 88, 102d
Runtime.pop_list(): 60d, 61e, 62a, 68b, 69b, 76c, 76d, 78a, 78b, 79c, 82a, 84d, 95i, 115
Runtime.pop_redir(): 31e, 69a, 70e

Runtime.pop_word(): 60f, 62b, 83a
Runtime.push_list(): 83a
Runtime.push_redir(): 31c, 69b, 72a, 83a
Runtime.push_word(): 36f, 59d, 60f, 62a, 95i
Runtime.redir: 30b
Runtime.redir.Close: 70c, 83a, 115
Runtime.redir.FromTo: 30b, 69b, 70e, 72a
Runtime.runq: 32b, 38g, 57c, 65a, 72a, 83a, 88, 102d
Runtime.thread: 29h
Runtime.thread.argv: 30f, 59g, 60f, 61f, 62b, 68b, 69b, 71c, 76c, 78a, 78b, 79c, 82a, 83a, 84d, 95a, 95i, 101h, 115
Runtime.thread.argv_stack: 30f, 31a, 38c, 76c, 101h, 115
Runtime.thread.code: 29h, 59d, 67b, 71c, 72a, 101h
Runtime.thread.file: 31a, 36b, 48b, 83a, 102d
Runtime.thread.iflag: 31a, 39b, 43a, 43b, 48b, 63c, 83a
Runtime.thread.lexbuf: 31a, 36f, 38g, 83a
Runtime.thread.line: 31a, 36b, 48b
Runtime.thread.locals: 32b, 38g, 71c, 72a, 78b, 115
Runtime.thread.pc: 29h, 36f, 38g, 39b, 59d, 67a, 67d, 69b, 71c, 72a, 74b, 74c, 76a, 76c
Runtime.thread.redirections: 30b, 31e, 70e, 71c, 111c, 115
Runtime.thread.waitstatus: 30b, 65a, 72a, 73d, 115
Runtime.turf_redir(): 70e, 70f
Runtime.value: 28a
Runtime.var: 115
Runtime.var.v: 28c, 29d, 32d, 43b, 43d, 64d, 66a, 78a, 78b, 79d, 82a
Runtime.waitstatus: 30b, 32f
Runtime.waitstatus.ChildStatus: 65a, 73d
Runtime.waitstatus.NothingToWaitfor: 32f, 65a, 73d, 115
Runtime.waitstatus.WaitFor: 32f, 65a, 72a, 73d
Status.concstatus(): 73d
Status.getstatus(): 57c, 64f, 72b, 73d, 91f, 92g
Status.setstatus(): 48b, 63c, 65a, 67f, 68b, 73b, 73d, 82a, 84d
Status.truestatus(): 67a, 67d, 67f, 73b, 74b, 91f, 92g
UConsole.print(): 102b
Var.gvlook(): 32b
Var.setvar(): 64b
Var.vinit(): 102c
Var.vlook(): 32d, 43b, 43d, 64d, 66a, 78a, 79d, 82a

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) 11

[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, 11

[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, 11

[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) 11

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 11

[Mic08] Randal K. Michael. *Master UNIX Shell Scripting*. Wiley, 2008. cited page(s) 11

[New95] Cameron Newham. *Learning the bash Shell*. O'Reilly, 1995. cited page(s) 9, 11

[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) 11, 12, 15, 16, 18

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 12, 20

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Windowing System rio*. 2016. cited page(s) 9

[Pou00] Louis Pouzin. The origin of the shell, 2000. http://multicians.org/shell.html. cited page(s) 13, 14

[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