# 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 10, 2026

# Contents

# Chapter 1

# Introduction

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

## 1.1 Motivations

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

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

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 type `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 implemented redirections and pipes? 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?
- How does `C-c`, which interrupts a process, work? Which process receives the signal after an interruption? The shell or the command ran from the shell?

[1] For example, [BKM86] contrasts writing a program to count words in Pascal and in a 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 succintely 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 small size of Ken Thompson's shell reflects the UNIX philosophy of keeping the shell minimal: `if` and `goto` were separate programs (not builtins), and wildcard expansion was handled by an external `glob` program. This made the shell itself trivial at the cost of some awkwardness—for example, you could not write a loop without creating a script file for `goto` to jump into.

- 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.

  Part of the difficulty comes from the source code itself: Bourne wrote extensive C macros to make the code look like Algol (`IF`, `THEN`, `BEGIN`, `END`, etc.), which allegedly inspired the International Obfuscated C Code Contest. More fundamentally, the Bourne shell has no formal grammar—Tom Duff once remarked that "nobody knows the grammar of `sh`"—so the parser is full of special cases and context-dependent disambiguation that make it hard to reason about.

[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.

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

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

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

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

Figure 1.1 presents a timeline of major UNIX shells (and a few non-UNIX shells). I think `rc` represents the best compromise for this book: it implements the essential features of a shell while still having a small and understandable codebase (6700 LOC).

## 1.4 Getting started

To play with `rc`, you will first need to install the Plan 9 fork used in Principia Softwarica (see `http://www.principia-softwarica.org/wiki/Install`). 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='$ '
```

[8] https://www.gnu.org/software/bash/
[9] http://www.gnu.org
[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.

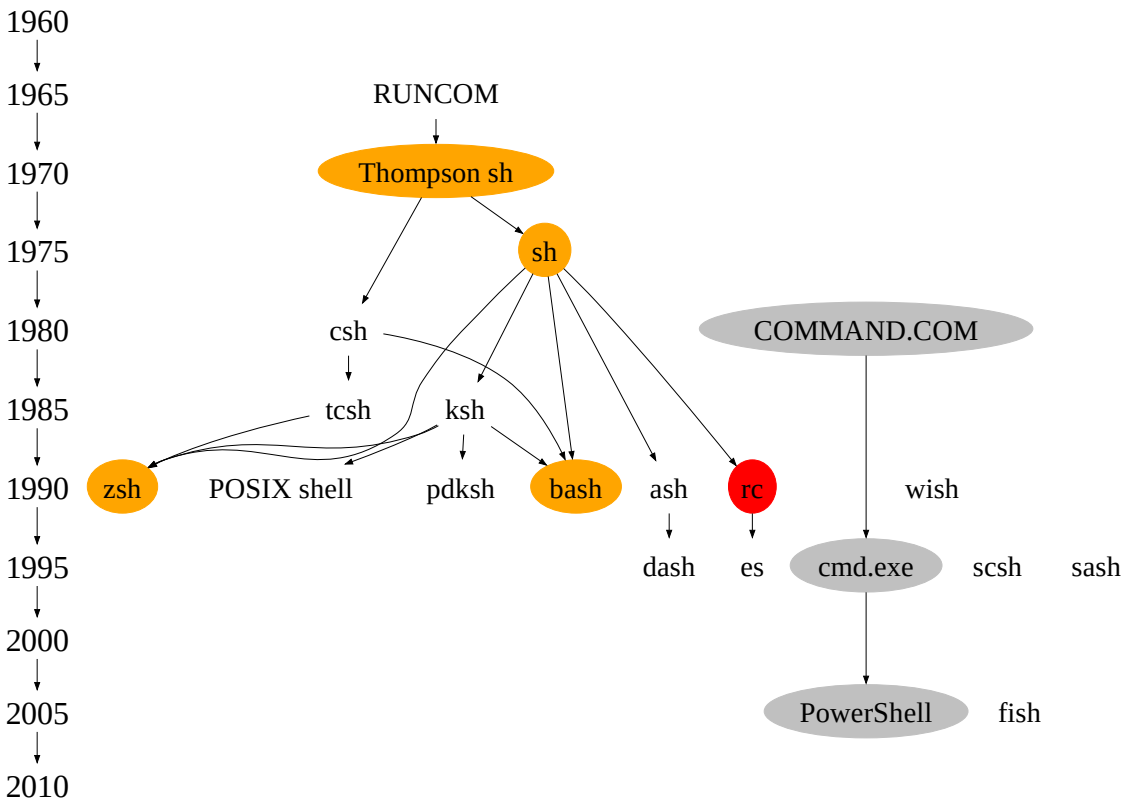


Figure 1.1: Shells timeline

```
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



## 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 script 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[] = {"/"};
  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 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

Line 15 of `boot.rc` 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.

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

Line 33 of `boot.rc` assigns `objtype=arm`. This is not just a local shell variable: in Plan 9, all shell variables are automatically exported to the environment. 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.

### 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). 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

### 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`.

```
%\t stdin stdout regular program
%plan9:
%\t under \plan kernel run rc (see \book{Kernel} and Section{boot.rc}).
%unix:
%\t Under \unix run init, then login, then shell in terminal. Even virtual
%\t  terminal in Linux,
%\t under windowing system like rio also virtual terminals connected
%\t stdin and stdout of process (see \book{Windows}).
% Figure X presents archi for plan9.

% shell = stuff around a kernel in a coquillage.
% history: multics.
% A shell is a bastard. See es paper quote in abstract
%  that reference unix programmnig environment book

% separate prog from kernel, can change shell!
%history: originated in Multics (one of the 2 things Thompson liked about
% Multics, the other being the hierarchical filesystem)
```

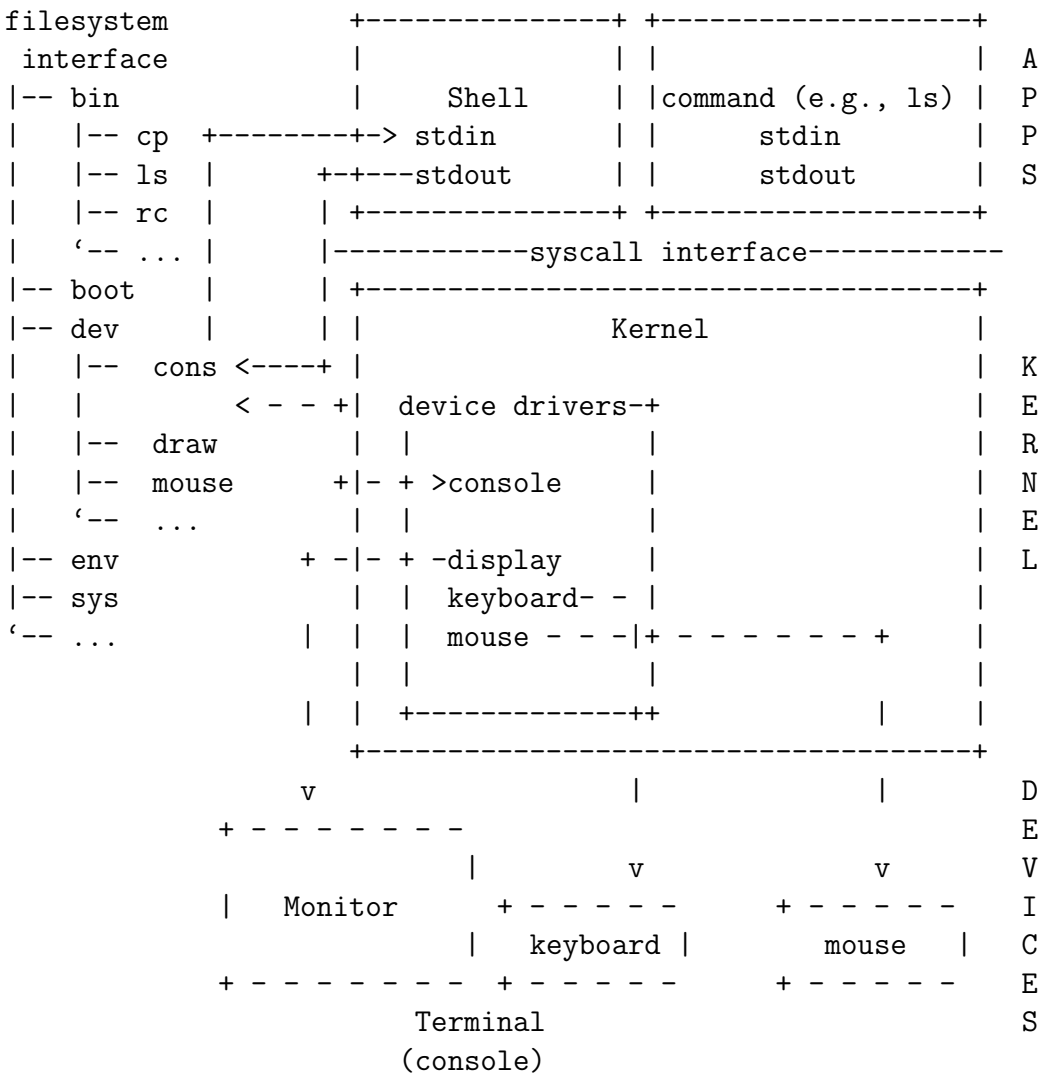


Figure 2.1: Components of the command-line user interface under Plan 9.

| Function | Ch. | File | Entities | LOC |
|---|---|---|---|---|
| abstract syntax tree | 3 | Ast.ml | line[26c] cmd[87m] Ast.value[94a] | 110 |
| opcodes | 3 | Opcode.ml | codevecX Opcode.t[86f] operation[26g] | 109 |
| runtime structures | 3 | Runtime.ml | varX globals[31c] thread[28i] runqX cur()[29c] | 233 |
| variable management | 3 | Var.ml | gvlook()X vlook()X setvar()X init()X | 46 |
| function management | 3 | Fn.ml | flook()[28k] | 10 |
| entry point | 4 | Main.ml | | 18 |
| main functions | 4 | CLI.ml | main()[95b] interpret_bootstrap()[79a] | 298 |
| read eval print loop | 4 | Op_REPL.ml | op_REPL()[35i] | 61 |
| lexer | 5 | Lexer.mll | Lexer.token()[38a] incr_lineno()[43e] | 170 |
| prompt management | 5 | Prompt.ml | doprompt[39d] prompt[39g] pprompt()[39i] | 39 |
| globals | 5 | Globals.ml | skipnl[40j] ifnot[67a] errstr[56d] | 33 |
| parser | 6 | Parse.ml | parse_line()X Parse.error()[44a] | 96 |
| grammar | 6 | Parser.mly | Parser.tokens[24] Parser.line()[46a] | 311 |
| opcode generation | 7 | Compile.ml | compile()[50a] outcode_seq()[51f] | 341 |
| opcode interpretation | 7 | Interpret.ml | interpret_operation()X | 357 |
| simple command interpretation | 7 | Op_process.ml | op_Simple()[55a] forkexec()[55b] | 120 |
| process status | 7 | Status.ml | setstatus()X getstatus()X | 39 |
| process management | 7 | Process.ml | return()[51a] waitfor()[58a] exit()[65c] | 81 |
| error management | 7 | Error.ml | Error.error()[56c] | 18 |
| $path management | 7 | PATH.ml | find_in_path()[58d] | 27 |
| pattern matching | 7 | Pattern.ml | match_str()[102e] | 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 | | |
| wildcard matching | 12 | Glob.ml | | 2 |
| Debugging flags | 13 | Flags.ml | xflag[82a] sflag[82d] eflag[83c] | 58 |
| here documents | 14 | Heredoc.ml | | 2 |
| Total | | | | 3200 |

Table 2.1: Chapters and associated rc source files.

```
% for a long time actually there was no (external) shell? Shell was
% part of kernel? With Unix it changed and shell can be a regular user program.

% for interactive use, and for programming. Both uses,
% and different features for those 2 things.

% regular program
% backspace handled by kerne
% line editing not shell
% \footnote not libreadline raw mode, completion
```

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

Figure 2.2: Data flow diagram of `rc`.

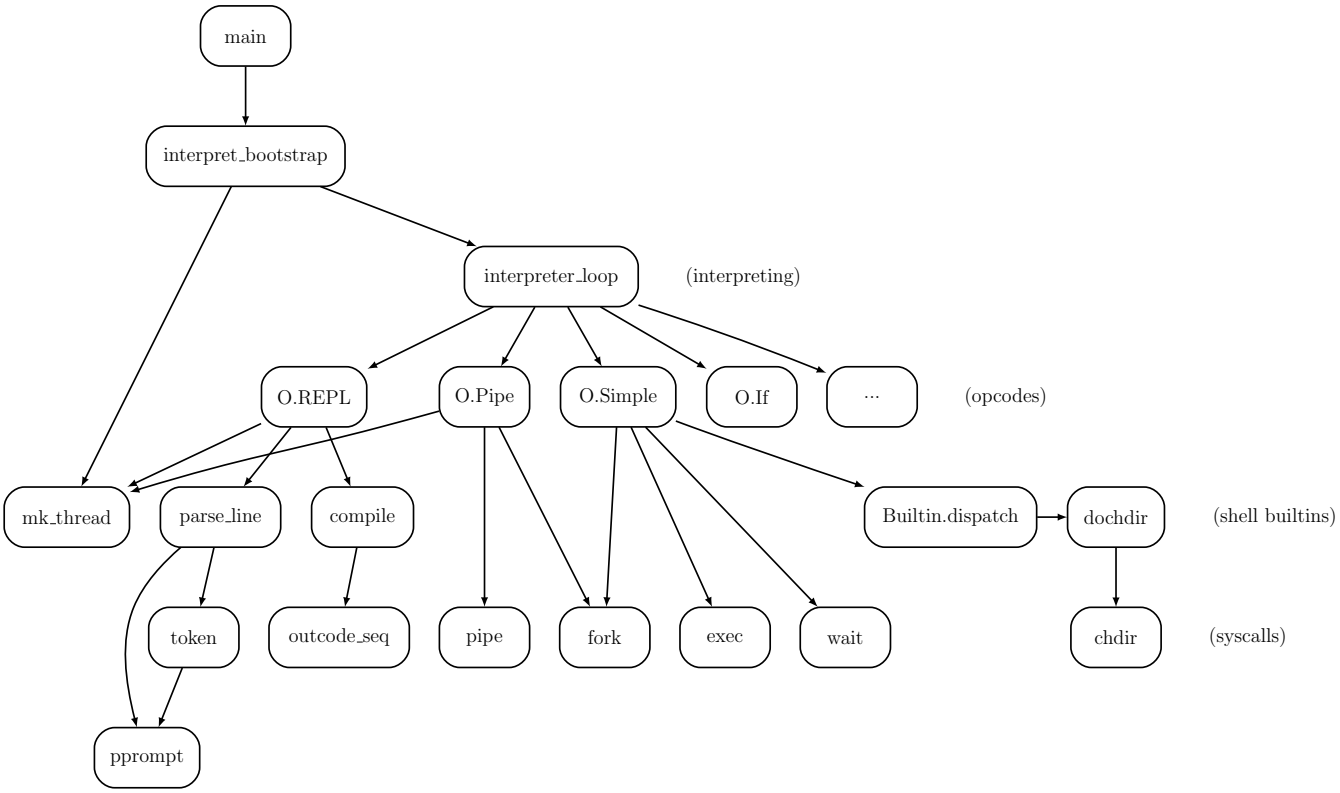


Figure 2.3: Control flow diagram of `rc`.

### 2.5.2 `rc`'s components

Figure 2.2 describes the main data flow of `rc`, whereas Figure 2.3 describes the main control flow of `rc`. At its core, `rc` is both a compiler and a bytecode interpreter. As shown by Figure 2.2, given a series of characters (coming either from what you typed in the terminal or from the content of a script), `rc` first groups those characters in tokens (with `Lexer.token()`[38a]). Then, `rc` parses those tokens (with `Parse.parse_line()`[43a]) and builds an abstract syntax tree (AST) of the program (see `Ast.line`[26c] and `Ast.cmd`[87m]). Finally, `rc` transforms this tree in a series of bytecodes or opcodes (see `Opcode.opcode`[26f] and `Opcode.operation`[26g]). 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 currently executing in a `thread`[28i] 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()`[64b], 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`[85b]) 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()`[95b] calls `interpret_bootstrap()`[79a] which first calls a function `bootstrap()`[79a] 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`[28i] 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 com-

mand (hence its name) and starts a new `thread`. `O.REPL()`[35h] 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()`[50a] 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()`[63h] 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`[26g] 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()`[50c], 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`[86c], which is interpreted by the function `O.Simple()`[53g], 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()`[72b] 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()`[35h], `O.Simple()`[53g], and `O.Return()`[50c]. I will explain `O.Mark()`[53b] and `O.Word()`[53a] later in this document.

```
% It is interesting to understand the full trace of a simple command
% through the multiple layers of the software stack.
% Here is again the trace of the command {\tt{}ls /<newline>}
% entered in a terminal:

%\t full trace ls\n
\begin{enumerate}
\item After {\tt{}rc} displayed its prompt,
%\t 1 shell read(), sysread, connected to /dev/cons, consread
%\t blocked (see \book{kernel})

%\t item l

%\t item s space /

%\t item newline
\end{enumerate}
```

## 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()`[95b], the initialization of `rc`, the bytecode interpreter loop, and the function `O.REPL()`[35h]. 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. 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*

## 3.1 Tokens

⟨*Parser tokens* 24⟩≡ (44b)

```
/*(*-----------------------------------------*)*/
/*(*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 85a⟩

/*(*-----------------------------------------*)*/
/*(*2 Variables *)*/
/*(*-----------------------------------------*)*/
%token TEq
%token TDollar
⟨Parser tokens, variables other cases 85e⟩

/*(*-----------------------------------------*)*/
/*(*2 Punctuation *)*/
/*(*-----------------------------------------*)*/
%token TOPar TCPar
%token TOBrace TCBrace
⟨Parser tokens, punctuation other cases 86d⟩

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

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

%token EOF
```

## 3.2 Abstract syntax tree (AST)

⟨*type* `Ast.line` 25a⟩≡ (94a)
```
(* None when reads EOF *)
type line = cmd_sequence option
```

⟨*type* `Ast.cmd_sequence` 25b⟩≡ (94a)
```
and cmd_sequence = cmd list
```

⟨*type* `Ast.cmd` 25c⟩≡ (94a)
```
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 87c⟩

  (* 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 48h⟩

  (* 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 84b⟩
```
Uses `Ast.cmd` 87m, `Ast.redirection_kind`, and `Ast.value` 94a.

⟨*type* `Ast.values` 25d⟩≡ (94a)
```
(* separated by spaces *)
and values = value list
```
Uses `Ast.cmd` 87m.

⟨*type* `Ast.value` 26a⟩≡ (94a)
```
(* 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 85g⟩
```
Uses `Ast.value` 94a.

⟨*type* `Ast.redirection` 26b⟩≡ (94a)
```
    and redirection = redirection_kind * value (* the filename *)
```

⟨*type* `Ast.redirection_kind` 26c⟩≡ (94a)
```
      and redirection_kind =
        | RWrite (* > *)
        | RRead  (* < *)
        | RAppend (* > > *)
        ⟨Ast.redirection_kind other cases 87b⟩
```

## 3.3 Opcodes

⟨*signature* `Compile.compile` 26d⟩≡ (96a)
```
val compile : Ast.cmd_sequence -> Opcode.codevec
```

⟨*type* `Opcode.codevec` 26e⟩≡ (101a)
```
type codevec = t array
```

⟨*type* `Opcode.opcode` 26f⟩≡ (101a)
```
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
```

⟨*type* `Opcode.operation` 26g⟩≡ (101a)
```
(* 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 86b⟩

(* Variable *)
| Assign (* (name)(val) *)
| Dollar (* (name) *)
⟨Opcode.operation other variable cases 86c⟩

(* Process! *)
| Simple (* (args) *)
| Async  (* {... Xreturn} *)

(* Control flow *)
| If
| IfNot
(* While are compiled in jumps *)
| Jump  (* [addr] *)
| Exit
⟨Opcode.operation other control cases 50b⟩

(* 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 35a⟩

(* Pipes *)
| Pipe (* [i j]{... Xreturn}{... Xreturn} *)
⟨Opcode.operation other pipe cases 35b⟩

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

⟨Opcode.operation other cases 81a⟩

| REPL (* *)
```

## 3.4 Words

⟨*type* Runtime.value 27a⟩≡ (104 103d)

```
(* 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
```

## 3.5 Variables

⟨*type* Runtime.varname 27b⟩≡ (104 103d)

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

⟨*type* Runtime.var 28a⟩≡ (104 103d)
```
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 *)
}
```

⟨*signature* Runtime.globals 28b⟩≡ (103d)
```
val globals : (varname, var) Hashtbl.t
```

⟨*global* Runtime.globals 28c⟩≡ (104)
```
let globals: (varname, var) Hashtbl.t =
  Hashtbl.create 101
```

⟨*signature* Var.gvlook 28d⟩≡ (106b)
```
(* This will only look for globals (in Runtime.globals) *)
val gvlook : Runtime.varname -> Runtime.var
```

⟨*function* Var.gvlook 28e⟩≡ (106c)
```
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 31c and Runtime.var.v 28a.

⟨*signature* Var.vinit 28f⟩≡ (106b)
```
(* Populate Runtime.globals from the current environment *)
val init : < Cap.env ; .. > -> unit
```

## 3.6 Functions

⟨*signature* Runtime.fns 28g⟩≡ (103d)
```
val fns : (string, fn) Hashtbl.t
```

⟨*global* Runtime.fns 28h⟩≡ (104)
```
let fns: (string, fn) Hashtbl.t =
  Hashtbl.create 101
```

⟨*type* Runtime.fn 28i⟩≡ (104 103d)
```
type fn = {
  code: Opcode.codevec;
  pc: int;
  (* less: fnchanged *)
}
```

⟨*signature* Fn.flook 28j⟩≡ (98c)
```
val flook : string -> Runtime.fn option
```

⟨*function* Fn.flook 28k⟩≡ (98d)
```
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 29a⟩≡ (104 103d)
```
type thread = {
  code: Opcode.codevec;
  pc: int ref;

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

  ⟨Runtime.thread other fields 30h⟩
}
```
Uses Runtime.redir and Runtime.waitstatus 30i.

⟨*signature* Runtime.runq 29b⟩≡ (103d)
```
val runq : thread list ref
```

⟨*constant* Runtime.runq 29c⟩≡ (104)
```
(* 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 29d⟩≡ (103d)
```
val cur : unit -> thread
```

⟨*function* Runtime.cur 29e⟩≡ (104)
```
let cur () =
  match !runq with
  | [] -> failwith "empty runq"
  | x::_xs -> x
```
Uses Runtime.cur() 29c, Runtime.thread.argv, and Runtime.thread.argv_stack 35e.

⟨*signature* Runtime.mk_thread 29f⟩≡ (103d)
```
val mk_thread :
  Opcode.codevec -> int -> (varname, var) Hashtbl.t -> thread
```

⟨*function* Runtime.mk_thread 29g⟩≡ (104)
```
(* 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 31a⟩
  } in
  t
```
Uses Runtime.thread.argv_stack 35e, Runtime.thread.file 33g, Runtime.thread.iflag, Runtime.thread.lexbuf, and Runtime.thread.line 33g.

### 3.7.1 The stack

⟨*signature* Runtime.push_word 29h⟩≡ (103d)
```
val push_word : string -> unit
```

⟨*function* `Runtime.push_word` 30a⟩≡ (104)
```
let push_word s =
  let t = cur () in
  t.argv <- s::t.argv
```

⟨*signature* `Runtime.pop_word` 30b⟩≡ (103d)
```
val pop_word : unit -> unit
```

⟨*function* `Runtime.pop_word` 30c⟩≡ (104)
```
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()` 29c and `Runtime.thread.redirections` 29a.

### 3.7.2 Local variables

⟨*signature* `Var.vlook` 30d⟩≡ (106b)
```
(* 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` 30e⟩≡ (106c)
```
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()` 29c, `Runtime.runq`, `Runtime.thread.locals` 104, and `Var.gvlook()`.

⟨*signature* `Var.setvar` 30f⟩≡ (106b)
```
(* Override the content of a (local or global) variable *)
val setvar : Runtime.varname -> Runtime.value -> unit
```

⟨*function* `Var.setvar` 30g⟩≡ (106c)
```
let setvar (name : Runtime.varname) (v : Runtime.value) : unit =
  let var = vlook name in
  var.R.v <- Some v
```
Uses `Runtime.var.v` 28a and `Var.vlook()`.

### 3.7.3 Redirections

⟨`Runtime.thread` *other fields* 30h⟩≡ (29a) 31b▷
```
(* things to do before exec'ing the simple command *)
mutable redirections: (redir list) list;
```

⟨*type* `Runtime.redir` 30i⟩≡ (104 103d)
```
  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* 31a⟩≡ (29g) 31d ▷
```
redirections =
  (match !runq with
  | t::_ts -> []::t.redirections
  | [] -> []::[]
  );
```

### 3.7.4 Wait status

⟨Runtime.thread *other fields* 31b⟩+≡ (29a) ◁30h 33g ▷
```
(* things to wait for after a thread forked a process *)
mutable waitstatus: waitstatus;
```

⟨*type* Runtime.waitstatus 31c⟩≡ (104 103d)
```
  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* 31d⟩+≡ (29g) ◁31a 33h ▷
```
waitstatus = NothingToWaitfor;
```

# Chapter 4

# `main()`

## 4.1 Overview

⟨*type* `CLI.caps` 32a⟩≡ (95)
```
(* 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` 32b⟩≡ (95a)
```
(* entry point (can also raise Exit.ExitCode) *)
val main: <caps; Cap.stdout; Cap.stderr; ..> ->
  string array -> Exit.t
```

⟨*toplevel* `Main._1` 32c⟩≡ (100b)
```
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()` 95b, `Cap.main()`, `CapSys.argv()`, `Exit.catch()`, and `Exit.exit()`.

⟨*function* `CLI.main` 32d⟩≡ (95b)
```
let main (caps : <caps; Cap.stdout; Cap.stderr; .. >) (argv : string array) :
    Exit.t =
  let args = ref [] in
  ⟨CLI.main() debugging initializations 82g⟩

  let options = [
    ⟨CLI.main() options elements 33f⟩
  ] |> Arg.align
  in
  (* This may raise ExitCode *)
  Arg_.parse_argv caps argv options (fun t -> args := t::!args) usage;
  ⟨CLI.main() logging initializations 83b⟩
  ⟨CLI.main() CLI action processing 91b⟩
```

```
    Var.init caps;
    (* todo: trap_init () *)
    ⟨CLI.main() other initializations 33e⟩
    try
      interpret_bootstrap (caps :> < caps >) (List.rev !args);
      Exit.OK
    with exn ->
      ⟨CLI.main() when exn thrown in interpret() 89c⟩
```

Uses Flags.debugger, Flags.login 33i, Logs.info(), and Logs.set_level().

⟨CLI.main() *when* Failure exn *thrown in* interpret() 33a⟩≡ (89c)
```
  | 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 33b⟩≡ (95a)
```
  val interpret_bootstrap : < caps > ->
    string list -> unit
```

## 4.2 Command-line arguments processing

⟨*constant* CLI.usage 33c⟩≡ (95b)
```
  let usage =
    "usage: rc [-SsriIlxepv] [-c arg] [-m command] [file [arg ...]]"
```

### 4.2.1 Interactive mode: `rc -i`

⟨*constant* Flags.interactive 33d⟩≡ (98b)
```
  (* -i (on by default when detects that stdin is /dev/cons *)
  let interactive = ref false
```

⟨CLI.main() *other initializations* 33e⟩≡ (32d) 75b▷
```
  (* todo:
   * if argc=1 and Isatty then Flags.interactive := true
   *)
```

Uses Common.=~().

⟨CLI.main() options *elements* 33f⟩≡ (32d) 34a▷
```
  "-i", Arg.Set Flags.interactive,
  " interactive mode (display prompt)";
  "-I", Arg.Clear Flags.interactive,
  " non-interactive mode (no prompt)";
```

Uses Flags.rcmain 79b.

⟨Runtime.thread *other fields* 33g⟩+≡ (29a) ◁31b 35e▷
```
  (* to display a prompt or not *)
  mutable iflag: bool;
```

⟨Runtime.mk_thread() *set other fields* 33h⟩+≡ (29g) ◁31d 35f▷
```
  iflag = false;
```

### 4.2.2 Login mode: `rc -l`

⟨*constant* Flags.login 33i⟩≡ (98b)
```
  (* -l (on by default if argv0 starts with a -) *)
  let login = ref false
```

⟨CLI.main() options *elements* 34a⟩+≡ (32d) ◁33f 80▷
```
"-l", Arg.Set Flags.login,
" login mode (execute ~/lib/profile)";
```

## 4.3 Bytecode interpreter loop

⟨*function* CLI.interpret_bootstrap 34b⟩≡ (95b)
```
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 35c⟩

  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 90h⟩
    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() 79a, Flags.rflag 90f, Logs.app(), Profiling.profile_code(), Runtime.push_word() 54b, Runtime.thread.lexbuf, and Runtime.thread.pc 28i.

⟨*constant* CLI._bootstrap_simple 34c⟩≡ (95b)
```
let _bootstrap_simple : O.codevec =
  [| O.F O.REPL |]
```
Uses Opcode.operation.REPL 85b and Opcode.t.F.

⟨*signature* Interpreter.interpret_operation 34d⟩≡ (99f)
```
(* 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 34e⟩≡ (99g)
```
let interpret_operation (caps: < Cap.fork; Cap.exec; Cap.chdir; Cap.exit; .. >) op : unit =
  match op with
  ⟨Interpreter.interpret_operation() match operation cases 35h⟩
  | (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 26g, Opcode.operation.Async 86c, Opcode.operation.Backquote 84c, Opcode.operation.Close 26g, Opcode.operation.Dup 26g, Opcode.operation.Fn 35b, Opcode.operation.For 26g, Opcode.operation.PipeFd 35a, Opcode.operation.Read 26g, Opcode.operation.ReadWrite 26g, Opcode.operation.Subshell 83e, Opcode.operation.Unlocal 26g, and Opcode.t.F.

⟨Opcode.operation *other redirection cases* 35a⟩≡ (26g)
```
| ReadWrite (* (file)[fd] *)
| Close  (* [fd] *)
| Dup    (* [fd0 fd1] *)
| Popredir (* *)
```

⟨Opcode.operation *other pipe cases* 35b⟩≡ (26g) 65d▷
```
| PipeFd (* [type]{... Xreturn} *)
```

⟨CLI.interpret_bootstrap() *other initializations for* t 35c⟩≡ (34b) 35d▷
```
(* less: set argv0 *)
args |> List.rev |> List.iter Runtime.push_word;
```

⟨CLI.interpret_bootstrap() *other initializations for* t 35d⟩+≡ (34b) ◁35c 35g▷
```
t.R.lexbuf <- Lexing.from_channel stdin;
```

⟨Runtime.thread *other fields* 35e⟩+≡ (29a) ◁33g 43c▷
```
(* connected on stdin by default (changed when do '. file') *)
mutable lexbuf: Lexing.lexbuf;
```

⟨Runtime.mk_thread() *set other fields* 35f⟩+≡ (29g) ◁33h 43d▷
```
lexbuf = Lexing.from_function (fun _ _ -> failwith "unconnected lexbuf");
```

⟨CLI.interpret_bootstrap() *other initializations for* t 35g⟩+≡ (34b) ◁35d
```
t.R.iflag <- !Flags.interactive;
```

## 4.4 Reading commands: O.REPL()

⟨Interpreter.interpret_operation() *match* operation *cases* 35h⟩≡ (34e) 50c▷
```
(* *)
| O.REPL -> Op_repl.op_REPL caps ()
```
Uses Op_repl.op_REPL() 35i and Opcode.operation.REPL 85b.

⟨*function* Op_repl.op_REPL 35i⟩≡ (100d)
```
(* was called Xrdcmds *)
let op_REPL (caps : < Cap.exit; ..>) () =
  let t = R.cur () in

  ⟨Op_repl.op_REPL() if sflag 82f⟩
  ⟨Op_repl.op_REPL() set prompt if iflag 39j⟩
  (* 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 50d⟩

    with Failure s ->
      ⟨Op_repl.op_REPL() when Failure s thrown 36b⟩
```

Uses Compile.compile() 50a, Parse.parse_line(), Runtime.cur() 29c, Runtime.mk_thread() 63h, Runtime.runq, Runtime.thread.lexbuf, Runtime.thread.locals 104, and Runtime.thread.pc 28i.

⟨*signature* Parse.parse_line 36a⟩≡ (101d)
```
val parse_line : Lexing.lexbuf -> Ast.line
```

⟨Op_repl.op_REPL() *when* Failure s *thrown* 36b⟩≡ (35i)
```
(* 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 28i.

# Chapter 5

# Lexing

## 5.1 `lexfunc()` skeleton

⟨`Parser.parse_line()` *nested function* `lexfunc` 37a⟩≡ (43a)
```
let lexfunc lexbuf =
  ⟨Parser.lexfunc() possibly print the prompt 39e⟩
  let tok = ref (Lexer.token lexbuf) in
  ⟨Parser.lexfunc() adjustment for skipnl depending on tok read 41c⟩
  ⟨Parser.lexfunc() adjust curtok with tok 43b⟩
  (* todo:
     - handle lastdol
     - handle SUB
     - handle free caret insertion
  *)
  !tok
  ⟨Parser.lexfunc() possibly dump tokens 90a⟩
in
```

⟨*signature* `Lexer.token` 37b⟩≡ (100a)
```
val token: Lexing.lexbuf -> Parser.token
```

⟨*exception* `Lexer.Lexical_error` 37c⟩≡ (100a 37e)
```
exception Lexical_error of string
```

⟨*function* `Lexer.error` 37d⟩≡ (37e)
```
let error s =
  raise (Lexical_error s)
```

## 5.2 `token()` skeleton

⟨`Lexer.mll` 37e⟩≡
```
{
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Parser

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Limitations compared to rc:
 *  - no unicode support
*)
⟨exception Lexer.Lexical_error 37c⟩
```

```
⟨function Lexer.error 37d⟩
⟨function Lexer.incr_lineno 43e⟩
}
(*****************************************************************************)
(* Regexps aliases *)
(*****************************************************************************)
⟨lexer regexp aliases 38b⟩

(*****************************************************************************)
(* Main rule *)
(*****************************************************************************)
⟨rule Lexer.token 38a⟩

(*****************************************************************************)
(* Quote rule *)
(*****************************************************************************)
⟨rule Lexer.quote 41h⟩
```

⟨*rule* Lexer.token 38a⟩≡ (37e)

```
rule token = parse

  (* ---------------------------------------------------------------------- *)
  (* Spacing/comments *)
  (* ---------------------------------------------------------------------- *)
  ⟨Lexer.token() space cases 38c⟩
  ⟨Lexer.token() comment cases 39a⟩

  (* ---------------------------------------------------------------------- *)
  (* Symbols *)
  (* ---------------------------------------------------------------------- *)
  ⟨Lexer.token() symbol cases 40c⟩

  (* ---------------------------------------------------------------------- *)
  (* Variables *)
  (* ---------------------------------------------------------------------- *)
  ⟨Lexer.token() variable cases 40f⟩

  (* ---------------------------------------------------------------------- *)
  (* Quoted word *)
  (* ---------------------------------------------------------------------- *)
  ⟨Lexer.token() quoted word cases 41g⟩

  (* ---------------------------------------------------------------------- *)
  (* Keywords and unquoted words *)
  (* ---------------------------------------------------------------------- *)
  ⟨Lexer.token() keywords and unquoted words cases 42b⟩

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

⟨*lexer regexp aliases* 38b⟩≡ (37e) 42a▷

```
(* 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* 38c⟩≡ (38a) 39b▷

```
| [' ''\t']+ { token lexbuf }
```

⟨Lexer.token() *comment cases* 39a⟩≡ (38a)
```
| '#' [^'\n']* { token lexbuf }
```

## 5.4 Newlines and prompts

⟨Lexer.token() *space cases* 39b⟩+≡ (38a) ◁38c 40a▷
```
| '\n'       { incr_lineno(); Prompt.doprompt := true; TNewline }
```

⟨*signature* Prompt.doprompt 39c⟩≡ (103b)
```
val doprompt : bool ref
```

⟨*constant* Prompt.doprompt 39d⟩≡ (103c)
```
(* 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* 39e⟩≡ (37a)
```
(* less: could do that in caller? would remove need for doprompt *)
if !Prompt.doprompt
then Prompt.pprompt ();
```
Uses Prompt.doprompt 39d and Prompt.pprompt() 39i.

⟨*signature* Prompt.prompt 39f⟩≡ (103b)
```
val prompt : string ref
```

⟨*constant* Prompt.prompt 39g⟩≡ (103c)
```
let prompt = ref "% "
```

⟨*signature* Prompt.pprompt 39h⟩≡ (103b)
```
(* !will reset doprompt! *)
val pprompt : unit -> unit
```

⟨*function* Prompt.pprompt 39i⟩≡ (103c)
```
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 40b⟩
  end;
  (* alt: incr t.R.line; this is done in the lexer instead *)
  doprompt := false
```
Uses Prompt.doprompt 39d, Prompt.prompt 39g, Runtime.cur() 29c, and Runtime.thread.iflag.

⟨Op_repl.op_REPL() *set* prompt *if* iflag 39j⟩≡ (35i)
```
(* 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 39g, Runtime.thread.iflag, Runtime.var.v 28a, and Var.vlook().

⟨Lexer.token() *space cases* 40a⟩+≡ (38a) ◁39b
```
  | '\\''\n'  {
      incr_lineno ();
      Prompt.pprompt ();
      token lexbuf
  }
```

⟨Prompt.pprompt() *adjust* prompt *for next time* 40b⟩≡ (39i)
```
  (* 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 39g, Runtime.var.v 28a, and Var.vlook().

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

⟨Lexer.token() *symbol cases* 40c⟩≡ (38a) 40d▷
```
  | ';'  { TSemicolon }
```

⟨Lexer.token() *symbol cases* 40d⟩+≡ (38a) ◁40c 40e▷
```
  | "&"  { TAnd }
```

⟨Lexer.token() *symbol cases* 40e⟩+≡ (38a) ◁40d 40g▷
```
  | ">"  { TRedir Ast.RWrite }
  | "<"  { TRedir Ast.RRead }
  | ">>" { TRedir Ast.RAppend }
```

⟨Lexer.token() *variable cases* 40f⟩≡ (38a) 86a▷
```
  | "$"   { TDollar }
```

⟨Lexer.token() *symbol cases* 40g⟩+≡ (38a) ◁40e 40h▷
```
  | '('  { TOPar }   | ')' { TCPar }
  | '{'  { TOBrace } | '}' { TCBrace }
```

⟨Lexer.token() *symbol cases* 40h⟩+≡ (38a) ◁40g 40i▷
```
  | '='  { TEq }
```

⟨Lexer.token() *symbol cases* 40i⟩+≡ (38a) ◁40h 41d▷
```
  | "|"  { Globals.skipnl := true; TPipe }
```

⟨*constant* Globals.skipnl 40j⟩≡ (99c)
```
  (* 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* 41a⟩≡ (43a) 41b▷
```
Globals.skipnl := false;
```

⟨Parse.parse_line() *locals and inits* 41b⟩+≡ (43a) ◁41a
```
let got_skipnl_last_round = ref false in
```

⟨Parse.lexfunc() *adjustment for skipnl depending on* tok *read* 41c⟩≡ (37a)
```
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 40j, Lexer.token(), Parser.token.TNewline, and Prompt.pprompt() 39i.

⟨Lexer.token() *symbol cases* 41d⟩+≡ (38a) ◁40i 41e▷
```
| "&&" { Globals.skipnl := true; TAndAnd }
| "||" { Globals.skipnl := true; TOrOr }
```

⟨Lexer.token() *symbol cases* 41e⟩+≡ (38a) ◁41d 41f▷
```
| "!" { TBang }
```

⟨Lexer.token() *symbol cases* 41f⟩+≡ (38a) ◁41e 85c▷
```
| "~" { TTwiddle }
```

## 5.6 Quoted strings ('...')

⟨Lexer.token() *quoted word cases* 41g⟩≡ (38a)
```
| "'" { let s = quote lexbuf in TWord (s, true) }
```

⟨*rule* Lexer.quote 41h⟩≡ (37e)
```
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* 42a⟩+≡ (37e) ◁38b

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

⟨`Lexer.token()` *keywords and unquoted words cases* 42b⟩≡ (38a)

```
| 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

## 6.1 `parse_line()` skeleton

⟨*function* `Parse.parse_line` 43a⟩≡ (102a)
```
let parse_line lexbuf =
  let curtok = ref (Parser.EOF, "EOF") in
  ⟨Parse.parse_line() locals and inits 41a⟩
  ⟨Parse.parse_line() nested function lexfunc 37a⟩
  try
    Parser.rc lexfunc lexbuf
    ⟨Parse.parse_line() possibly dump AST 90c⟩
  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` 43b⟩≡ (37a)
```
let s = Lexing.lexeme lexbuf in
curtok := (!tok, s);
```

## 6.2 Error location reporting

⟨`Runtime.thread` *other fields* 43c⟩+≡ (29a) ◁35e 53e▷
```
(* 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* 43d⟩+≡ (29g) ◁35f 53f▷
```
file = None;
line = ref 1;
```

⟨*function* `Lexer.incr_lineno` 43e⟩≡ (37e)
```
(* we could do that in pprompt() too *)
let incr_lineno () =
  let t = Runtime.cur () in
  incr t.Runtime.line
```

⟨*function* Parse.error 44a⟩≡ (102a)
```
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() 29c, Runtime.thread.file 33g, Runtime.thread.iflag, Runtime.thread.line 33g, and Status.setstatus().

## 6.3 Grammar overview

⟨Parser.mly 44b⟩≡
```
%{
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Ast

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

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function Parser.mk_seq 47c⟩
%}

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

/*(*************************************************************************)*/
/*(*1 Priorities *)*/
/*(*************************************************************************)*/
⟨Parser token priorities 45a⟩

/*(*************************************************************************)*/
/*(*1 Rules type declaration *)*/
/*(*************************************************************************)*/
⟨Parser entry points types 45b⟩
```

```
%%

⟨grammar 45c⟩
```

⟨Parser token priorities 45a⟩≡ (44b)
```
/*(* 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 45b⟩≡ (44b)
```
%type <Ast.line> rc
⟨Parser other rule types 46b⟩
%start rc
```

⟨grammar 45c⟩≡ (44b)
```
/*(*************************************************************************)*/
/*(*1 line *)*/
/*(*************************************************************************)*/
rc:
  | line TNewline { Some $1 }
  | EOF           { None }

⟨rule Parser.line 46a⟩
⟨other line related rules 47b⟩

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

⟨rule Parser.cmd 46c⟩
⟨rule Parser.simple 46e⟩
⟨other command related rules 48f⟩

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

⟨rule Parser.comword 46h⟩
⟨rule Parser.word 46i⟩
⟨other word related rules 46g⟩

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

⟨rule Parser.redir 48a⟩
⟨other redirection related rules 49e⟩

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

⟨skipnl rules 48e⟩
```

## 6.4 Simple commands (<cmd> <arg1>...<argn>)

⟨*rule* Parser.line 46a⟩≡ (45c)
```
/*(* =~ stmt *)*/
line:
  | cmd { [$1] }
  ⟨Parser.line other cases 47a⟩
```

⟨*Parser other rule types* 46b⟩≡ (45b) 46d▷
```
%type <Ast.cmd> cmd
```

⟨*rule* Parser.cmd 46c⟩≡ (45c)
```
/*(* =~ 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 48b⟩
    }
  ⟨Parser.cmd other cases 47d⟩
```

⟨*Parser other rule types* 46d⟩+≡ (45b) ◁46b 46f▷
```
%type <value * value list * (redirection_kind * value, redirection_kind * int * int) either list> simple
```

⟨*rule* Parser.simple 46e⟩≡ (45c)
```
/*(* =~ primary expr *)*/
simple:
  | first        { $1, [], [] }
  | simple word  { let (cmd, args, redirs) = $1 in (cmd, $2::args, redirs) }
  ⟨Parser.simple other cases 47k⟩
```

⟨*Parser other rule types* 46f⟩+≡ (45b) ◁46d
```
%type <Ast.value> first
```

⟨*other word related rules* 46g⟩≡ (45c) 46j▷
```
first:
  | comword           { $1 }
  ⟨Parser.first other cases 87k⟩
```

⟨*rule* Parser.comword 46h⟩≡ (45c)
```
comword:
  | TWord                        { Word (fst $1, snd $1) }
  ⟨Parser.comword other cases 49j⟩
```

⟨*rule* Parser.word 46i⟩≡ (45c)
```
word:
  | comword          { $1 }
  | keyword          { Word ($1, false) }
  ⟨Parser.word other cases 87j⟩
```

⟨*other word related rules* 46j⟩+≡ (45c) ◁46g 47h▷
```
keyword:
  | TFor { "for" }  | TIn { "in" }
  | TIf { "if" }  | TNot { "not" }
  | TWhile { "while" }
  | TSwitch { "switch" }
  | TFn { "fn" }
  ⟨Parser.keyword other cases 47e⟩
```

## 6.5 Operators

### 6.5.1 Sequence (;, &)

⟨Parser.line *other cases* 47a⟩≡ (46a)
```
| cmdsa line { $1 $2  }
```

⟨*other line related rules* 47b⟩≡ (45c)
```
cmdsa:
  | cmd TSemicolon { (fun x -> mk_Seq ($1, x)) }
  | cmd TAnd       { (fun x -> mk_Seq (Async $1, x)) }
```

⟨*function* Parser.mk_seq 47c⟩≡ (44b)
```
(* 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* 47d⟩≡ (46c) 47f▷
```
| cmd TAndAnd cmd { And ($1, $3) }
| cmd TOrOr cmd   { Or  ($1, $3) }
| TBang cmd       { Not $2 }
```

⟨Parser.keyword *other cases* 47e⟩≡ (46j) 47g▷
```
| TBang { "!" }
```

### 6.5.3 String matching (~)

⟨Parser.cmd *other cases* 47f⟩+≡ (46c) ◁47d 47j▷
```
| TTwiddle word words { Match ($2, $3) }
```

⟨Parser.keyword *other cases* 47g⟩+≡ (46j) ◁47e 85d▷
```
| TTwiddle { "~" }
```

⟨*other word related rules* 47h⟩+≡ (45c) ◁46j 47i▷
```
words: words_rev { List.rev $1 }
```

⟨*other word related rules* 47i⟩+≡ (45c) ◁47h
```
words_rev:
  | /*empty*/  { [] }
  | words_rev word { $2::$1 }
```

### 6.5.4 Pipe (|)

⟨Parser.cmd *other cases* 47j⟩+≡ (46c) ◁47f 48c▷
```
| cmd TPipe cmd  { Pipe ($1, $3) }
```

### 6.5.5 Redirections (>, <)

⟨Parser.simple *other cases* 47k⟩≡ (46e)
```
| simple redir { let (cmd, args, redirs) = $1 in (cmd, args, $2::redirs) }
```

⟨*rule* `Parser.redir` 48a⟩≡ (45c)
```
redir:
  | TRedir word { Left ($1, $2) }
  ⟨Parser.redir other cases 87a⟩
```

⟨`Parser.cmd` *adjust* `base` *with* `redis` 48b⟩≡ (46c)
```
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
```

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

⟨`Parser.cmd` *other cases* 48c⟩+≡ (46c) ◁47j 48d▷
```
| TIf paren_skipnl cmd { If ($2, $3) }
| TIf tnot_skipnl  cmd { IfNot $3 }
```

⟨`Parser.cmd` *other cases* 48d⟩+≡ (46c) ◁48c 48g▷
```
| TWhile paren_skipnl cmd   { While ($2, $3) }
```

⟨*skipnl rules* 48e⟩≡ (45c) 48k▷
```
paren_skipnl: paren { Globals.skipnl := true; $1 }
tnot_skipnl: TNot   { Globals.skipnl := true; }
```

⟨*other command related rules* 48f⟩≡ (45c) 49b▷
```
paren: TOPar body TCPar { $2 }
```

⟨`Parser.cmd` *other cases* 48g⟩+≡ (46c) ◁48d 48i▷
```
| TSwitch word_skipnl brace { Switch ($2, $3) }
```

⟨`Ast.cmd` *other statement cases* 48h⟩≡ (25c) 48j▷
```
| Switch of value * cmd_sequence
```

⟨`Parser.cmd` *other cases* 48i⟩+≡ (46c) ◁48g 48l▷
```
/*(* 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* 48j⟩+≡ (25c) ◁48h 49a▷
```
| ForIn of value * values * cmd
(* less: could desugar as ForIn value $* *)
| For of value * cmd
```
Uses `Ast.cmd` 87m and `Ast.value` 94a.

⟨*skipnl rules* 48k⟩+≡ (45c) ◁48e
```
word_skipnl: word   { Globals.skipnl := true; $1 }
tcpar_skipnl: TCPar { Globals.skipnl := true; }
```

## 6.7 Grouping ({ . . . })

⟨`Parser.cmd` *other cases* 48l⟩+≡ (46c) ◁48i 49g▷
```
| brace epilog   {
    let cmd = (Compound $1) in
    ⟨Parser.cmd in brace case, adjust cmd with epilog 49f⟩
}
```

⟨Ast.cmd *other statement cases* 49a⟩+≡ (25c) ◁48j
```
| Compound of cmd_sequence
```
Uses Ast.value 94a.

⟨*other command related rules* 49b⟩+≡ (45c) ◁48f 49c▷
```
brace: TOBrace body TCBrace { $2 }
```

⟨*other command related rules* 49c⟩+≡ (45c) ◁49b 49d▷
```
body:
  | cmd          { [$1] }
  | cmdsan body { $1 $2 }
```

⟨*other command related rules* 49d⟩+≡ (45c) ◁49c 49i▷
```
cmdsan:
  | cmdsa        { $1 }
  | cmd TNewline { (fun x -> mk_Seq ($1, x)) }
```

⟨*other redirection related rules* 49e⟩≡ (45c)
```
epilog:
  | /*empty*/    { [] }
  | redir epilog { $1::$2 }
```

⟨Parser.cmd *in* brace *case, adjust* cmd *with* epilog 49f⟩≡ (48l)
```
$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> ... )

⟨Parser.cmd *other cases* 49g⟩+≡ (46c) ◁48l 49h▷
```
/*(* stricter: allow only word, not words *)*/
| TFn word brace { Fn ($2, $3) }
```

## 6.9 Variables (<x> = ..., $x)

⟨Parser.cmd *other cases* 49h⟩+≡ (46c) ◁49g 84d▷
```
| assign cmd %prec TBang { $1 $2 }
```

⟨*other command related rules* 49i⟩+≡ (45c) ◁49d
```
assign: first TEq word { (fun x -> Assign ($1, $3, x))  }
```

⟨Parser.comword *other cases* 49j⟩≡ (46h) 49k▷
```
| TDollar word                { Dollar $2 }
```

## 6.10 Lists ((...))

⟨Parser.comword *other cases* 49k⟩+≡ (46h) ◁49j 86h▷
```
| TOPar words TCPar           { List $2 }
```

# Chapter 7

# Opcode Generation and Interpretation

## 7.1 Overview

### 7.1.1 compile()

⟨*function* Compile.compile 50a⟩≡ (96b)
```
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 51b⟩
  ⟨Compile.compile() nested function set 51d⟩

  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 90e⟩
```
Uses Compile.outcode_seq() 51f, Flags.eflag 83c, Opcode.operation.Return 26g, and Opcode.t.F.

⟨Opcode.operation *other control cases* 50b⟩≡ (26g) 67d ▷
```
| Return
```

⟨Interpreter.interpret_operation() *match* operation *cases* 50c⟩+≡ (34e) ◁35h 53a ▷
```
| O.Return -> Process.return caps ()
```

⟨Op_repl.op_REPL() *match* cmdset_opt *other cases* 50d⟩≡ (35i)
```
| None -> Process.return caps ()
```
Uses Process.return() 51a.

⟨*signature* Process.return 50e⟩≡ (102g)
```
val return : < Cap.exit ; .. > -> unit -> unit
```

⟨*function* `Process.return` 51a⟩≡ (103a)
```
(* 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 63c⟩
  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()` 65c, `Runtime.runq`, and `Status.getstatus()`.

⟨`Compile.compile()` *nested function* `emit` 51b⟩≡ (50a)
```
let emit x =
  ⟨Compile.compile() nested function emit possibly grow codebuf 51c⟩
  !codebuf.(!idx) <- x;
  incr idx
in
```

⟨`Compile.compile()` *nested function* `emit` *possibly grow* `codebuf` 51c⟩≡ (51b)
```
(* 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` 26g.

⟨`Compile.compile()` *nested function* `set` 51d⟩≡ (50a)
```
let set idx2 x =
  ⟨Compile.compile() nested function set array bound checking 51e⟩
  !codebuf.(idx2) <- x;
in
```

⟨`Compile.compile()` *nested function* `set` *array bound checking* 51e⟩≡ (51d)
```
if idx2 < 0 || idx2 >= !len_codebuf
then failwith (spf "Bad address %d in set()" idx2);
```
Uses `Common.spf()`.

### 7.1.2 `outcode_seq()` skeleton

⟨*function* `Compile.outcode_seq` 51f⟩≡ (96b)
```
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() 59b⟩

  and xcmd (cmd : Ast.cmd) eflag : unit =
    match cmd with
    ⟨Compile.outcode_seq in nested xcmd() match cmd cases 52a⟩
    | (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 52b⟩
     | (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() 52c⟩
   in
   xseq seq eflag
```

Uses Ast.cmd.Async 25d, Ast.cmd.Dup 25c, Ast.cmd.For 25c, Ast.cmd.ForIn 25c, Ast.cmd.While 25c, Ast.value.CommandOutput 26a, Ast.value.Concat, Ast.value.Index 26a, and Ast.value.Stringify 26a.

# 7.2 Simple commands

## 7.2.1 Opcode generation

⟨Compile.outcode_seq in nested xcmd() match cmd cases 52a⟩≡ (51f) 59c▷

```
  | 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 83f⟩
```

Uses Ast.cmd.Simple, Opcode.operation.Mark 26g, Opcode.operation.Simple 86c, and Opcode.t.F.

⟨Compile.outcode_seq in nested xword() match w cases 52b⟩≡ (51f) 52d▷

```
  | A.Word (s, _quoted) ->
      emit (O.F O.Word);
      emit (O.S s);
```

Uses Ast.value.Word 26a, Opcode.operation.Word 26g, Opcode.t.F, and Opcode.t.S 26g.

## 7.2.2 Stack management

⟨Compile.outcode_seq in nested xwords() 52c⟩≡ (51f)

```
  ws |> List.rev |> List.iter (fun w -> xword w);
```

⟨Compile.outcode_seq in nested xword() match w cases 52d⟩+≡ (51f) ◁52b 71a▷

```
  | A.List ws ->
      xwords ws
```

Uses Ast.value.List 26a.

⟨Interpreter.interpret_operation() *match* operation *cases* 53a⟩+≡ (34e) ◁50c 53b▷
```
(* [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 26g, Opcode.t.S 26g, Runtime.cur() 29c, Runtime.push_word() 54b, Runtime.thread.code 28i, and Runtime.thread.pc 28i.

⟨Interpreter.interpret_operation() *match* operation *cases* 53b⟩+≡ (34e) ◁53a 53g▷
```
| O.Mark -> R.push_list ()
```

⟨*signature* Runtime.push_list 53c⟩≡ (103d)
```
val push_list : unit -> unit
```

⟨*function* Runtime.push_list 53d⟩≡ (104)
```
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* 53e⟩+≡ (29a) ◁43c
```
(* Used for switch but also assignments. *)
mutable argv_stack: (string list) list;
```

⟨Runtime.mk_thread() *set other fields* 53f⟩+≡ (29g) ◁43d
```
argv_stack = [];
```

### 7.2.3 O.Simple()

⟨Interpreter.interpret_operation() *match* operation *cases* 53g⟩+≡ (34e) ◁53b 59e▷
```
(* (args) *)
| O.Simple -> Op_process.op_Simple caps ()
```
Uses Op_process.op_Simple() 55a and Opcode.operation.Simple 86c.

⟨*function* Op_process.op_Simple 53h⟩≡ (100c)
```
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 82c⟩
  match argv with
  ⟨Op_process.op_Simple() match argv empty case 56a⟩
  | argv0::args ->
      match argv0 with
      ⟨Op_process.op_Simple() match argv0 builtin cases 54d⟩
      | _ ->
        ⟨Op_process.op_Simple() when default case, before the fork 54c⟩
        (try
          let pid = forkexec caps () in
          R.pop_list ();
          ⟨Op_process.op_Simple() when default case, after the fork 57g⟩
```

```
        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() 56c, Process.s_of_unix_error() 56f, Process.waitfor() 58a, Process.waitfor_result.WaitforInterrupted 57i, and Runtime.pop_list() 104.

⟨*signature* Runtime.pop_list 54a⟩≡ (103d)
```
  val pop_list : unit -> unit
```

⟨*function* Runtime.pop_list 54b⟩≡ (104)
```
  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() 29c and Runtime.thread.argv.

⟨Op_process.op_Simple() *when default case, before the fork* 54c⟩≡ (53h)
```
  (* if exitnext opti *)
  flush stderr;
  (* less: Updenv *)
```

### 7.2.4 Builtin dispatch

⟨Op_process.op_Simple() *match* argv0 *builtin cases* 54d⟩≡ (53h) 54g▷
```
  | s when Builtin.is_builtin s -> Builtin.dispatch caps argv0
```

Uses Builtin.dispatch().

⟨*signature* Builtin.is_builtin 54e⟩≡ (94b)
```
  val is_builtin : string -> bool
```

⟨*signature* Builtin.dispatch 54f⟩≡ (94b)
```
  (* execute the builtin *)
  val dispatch : < Cap.chdir ; Cap.exit ; Cap.open_in; .. > -> string -> unit
```

⟨Op_process.op_Simple() *match* argv0 *builtin cases* 54g⟩+≡ (53h) ◁54d
```
  (* 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() 55b and Runtime.pop_list() 104.

### 7.2.5 Fork

⟨*function* Op_process.forkexec 55a⟩≡ (100c)

```
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 82a, Logs.app(), Runtime.cur() 29c, and Runtime.thread.argv.

### 7.2.6 Exec

⟨*function* Op_process.exec 55b⟩≡ (100c)

```
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 63f⟩
      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() 101c, Runtime.pop_list() 104, and Runtime.push_word() 54b.

⟨*function* Op_process.execute 55c⟩≡ (100c)

```
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() 56c, Globals.errstr 56d, Logs.err(), Process.s_of_unix_error() 56f, Runtime.cur() 29c, Runtime.pop_word() 54b, and Runtime.thread.argv.

### 7.2.7 Error management

⟨Op_process.op_Simple() *match* argv *empty case* 56a⟩≡ (53h)

```
  (* 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 56b⟩≡ (97c)

```
  (* 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
```

⟨*function* Error.error 56c⟩≡ (98a)

```
  (* 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() 51a, Runtime.cur() 29c, Runtime.thread.iflag, and Status.setstatus().

⟨*constant* Globals.errstr 56d⟩≡ (99c)

```
  (* to mimic Plan 9 errstr() *)
  let errstr = ref ""
```

⟨*signature* Process.s_of_unix_error 56e⟩≡ (102g)

```
  val s_of_unix_error : Unix.error -> string -> string -> string
```

⟨*function* Process.s_of_unix_error 56f⟩≡ (103a)

```
  let s_of_unix_error err _s1 _s2 =
    spf "%s" (Unix.error_message err)
```

Uses Common.spf().

### 7.2.8 $status management

⟨*signature* Status.setstatus 57a⟩≡ (105)
```
val setstatus : string -> unit
```

⟨*function* Status.setstatus 57b⟩≡ (106a)
```
let setstatus s =
  Var.setvar "status" [s]
```
Uses Var.setvar().

⟨*signature* Status.getstatus 57c⟩≡ (105)
```
val getstatus : unit -> string
```

⟨*function* Status.getstatus 57d⟩≡ (106a)
```
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 28a and Var.vlook().

⟨*signature* Status.truestatus 57e⟩≡ (105)
```
val truestatus : unit -> bool
```

⟨*function* Status.truestatus 57f⟩≡ (106a)
```
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* 57g⟩≡ (53h)
```
(* do again even if was interrupted *)
while Process.waitfor caps pid = Process.WaitforInterrupted do
  ()
done
```

⟨*signature* Process.waitfor 57h⟩≡ (102g)
```
val waitfor : < Cap.wait; .. > -> int (* pid *) -> waitfor_result
```

⟨*type* Process.waitfor_result 57i⟩≡ (103a 102g)
```
type waitfor_result =
  | WaitforInterrupted
  | WaitforFound
  | WaitforNotfound
```

⟨*function* Process.waitfor 58a⟩≡ (103a)

```
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 56d, Process.s_of_unix_error() 56f, Process.waitfor_result.WaitforFound 57i, Process.waitfor_result.WaitforInterrupted 57i, Process.waitfor_result.WaitforNotfound 57i, Runtime.runq, Runtime.thread.waitstatus 29a, Runtime.waitstatus.ChildStatus, Runtime.waitstatus.NothingToWaitfor 30i, Runtime.waitstatus.WaitFor 30i, and Status.setstatus().

### 7.2.10 $path management

⟨*signature* PATH.find_in_path 58b⟩≡ (101b)

```
val find_dir_with_cmd_in_PATH :
  string (* cmd *) -> Fpath.t list (* $path *) -> Fpath.t
```

⟨*signature* PATH.search_path_for_cmd 58c⟩≡ (101b)

```
(* 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 58d⟩≡ (101c)

```
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` 28a, and `Var.vlook()`.

⟨*function* `PATH.find_in_path` 59a⟩≡ (101c)

```
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()` 59b⟩≡ (51f)

```
(* 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* 59c⟩+≡ (51f) ◁52a 59d▷

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

Uses `Ast.cmd.EmptyCommand` 87m.

## 7.3.2 Logical operators

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 59d⟩+≡ (51f) ◁59c 60b▷

```
| 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` 26g.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 59e⟩+≡ (34e) ◁53g 60c▷

```
| 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()` 29c, `Runtime.thread.pc` 28i, and `Status.truestatus()`.

⟨*function* `Interpreter.int_at_address` 60a⟩≡ (99g)
```
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` 26g, and `Runtime.thread.code` 28i.

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 60b⟩+≡ (51f) ◁59d 60d▷
```
| 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` 68b, `Opcode.t.F`, and `Opcode.t.I` 26g.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 60c⟩+≡ (34e) ◁59e 60e▷
```
| 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()` 29c, `Runtime.thread.pc` 28i, and `Status.truestatus()`.

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 60d⟩+≡ (51f) ◁60b 60f▷
```
| 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* 60e⟩+≡ (34e) ◁60c 61a▷
```
| O.Not ->
    Status.setstatus (if Status.truestatus() then "false" else "");
```
Uses `Status.setstatus()` and `Status.truestatus()`.

### 7.3.3 String matching

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 60f⟩+≡ (51f) ◁60d 61b▷
```
| 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 83h⟩
```
Uses `Ast.cmd.Match` 25c, `Opcode.operation.Mark` 26g, `Opcode.operation.Match` 26g, and `Opcode.t.F`.

⟨Interpreter.interpret_operation() *match* operation *cases* 61a⟩+≡ (34e) ◁60e 61c▷
```
(* (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 26g, Pattern.match_str() 102e, Runtime.cur() 29c, Runtime.pop_list() 104, Runtime.thread.argv, and Status.setstatus().

# 7.4 Redirection

## 7.4.1 Opcode generation

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 61b⟩+≡ (51f) ◁60f 64a▷
```
| 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 26g, Opcode.operation.Glob 26g, Opcode.operation.Mark 26g, Opcode.operation.Popredir 26g, Opcode.operation.Read 26g, Opcode.operation.Write 26g, Opcode.t.F, and Opcode.t.I 26g.

⟨Interpreter.interpret_operation() *match* operation *cases* 61c⟩+≡ (34e) ◁61a 62a▷
```
| O.Popredir ->
    R.pop_redir ()
```
Uses Runtime.pop_redir() 30c.

### 7.4.2 O.Write()

⟨Interpreter.interpret_operation() *match* operation *cases* 62a⟩+≡ (34e) ◁61c 64b▷
```
(* (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() 56c, Interpreter.file_descr_of_int() 62b, Interpreter.int_at_address(), Opcode.operation.Write 26g, Runtime.cur() 29c, Runtime.pop_list() 104, Runtime.push_redir() 30a, Runtime.redir.FromTo 29a, Runtime.thread.argv, and Runtime.thread.pc 28i.

⟨*function* Interpreter.file_descr_of_int 62b⟩≡ (99g)
```
(* 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 62c⟩≡ (103d)
```
val push_redir : redir -> unit
```

⟨*function* Runtime.push_redir 62d⟩≡ (104)
```
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 63a⟩≡ (103d)
```
val pop_redir : unit -> unit
```

⟨*function* Runtime.pop_redir 63b⟩≡ (104)
```
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() 29c, Runtime.pop_redir() 30c, Runtime.redir.FromTo 29a, and Runtime.thread.redirections 29a.

⟨Process.return() *initializations* 63c⟩≡ (51a)
```
R.turf_redir ();
```
Uses Runtime.turf_redir() 63b.

⟨*signature* Runtime.turf_redir 63d⟩≡ (103d)
```
val turf_redir : unit -> unit
```

⟨*function* Runtime.turf_redir 63e⟩≡ (104)
```
let turf_redir () =
  let t = cur () in
  while List.hd t.redirections <> [] do
    pop_redir ()
  done
```

⟨Op_process.exec() *before* execute 63f⟩≡ (55b)
```
R.doredir t.R.redirections;
```

⟨*signature* Runtime.doredir 63g⟩≡ (103d)
```
val doredir : redir list list -> unit
```

⟨*function* Runtime.doredir 63h⟩≡ (104)
```
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 28i, Runtime.thread.locals 104, Runtime.thread.pc 28i, and Runtime.thread.redirections 29a.

## 7.5 Pipe

### 7.5.1 Opcode generation

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 64a⟩+≡ (51f) ◁61b 66c▷

```
| 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 25d, Opcode.operation.Exit 26g, Opcode.operation.Pipe 35a, Opcode.operation.PipeWait 26g, Opcode.operation.Return 26g, Opcode.t.F, and Opcode.t.I 26g.

### 7.5.2 O.Pipe()

⟨Interpreter.interpret_operation() *match* operation *cases* 64b⟩+≡ (34e) ◁62a 65a▷

```
(* [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() 62b, Interpreter.int_at_address(), Opcode.operation.Pipe 35a, Runtime.cur() 29c, Runtime.mk_thread() 63h, Runtime.push_redir() 30a, Runtime.redir.FromTo 29a, Runtime.runq, Runtime.thread.code 28i, Runtime.thread.locals 104, Runtime.thread.pc 28i, Runtime.thread.waitstatus 29a, and Runtime.waitstatus.WaitFor 30i.

### 7.5.3 O.Exit()

⟨Interpreter.interpret_operation() *match* operation *cases* 65a⟩+≡ (34e) ◁64b 65e▷

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

Uses Process.exit() 65c and Status.getstatus().

⟨*signature* Process.exit 65b⟩≡ (102g)

```
  val exit : < Cap.exit ; .. > -> string -> unit
```

⟨*function* Process.exit 65c⟩≡ (103a)

```
  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* 65d⟩+≡ (26g) ◁35b

```
  | PipeWait (* argument passed through Thread.pid *)
```

⟨Interpreter.interpret_operation() *match* operation *cases* 65e⟩+≡ (34e) ◁65a 66d▷

```
  (* 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 26g, Process.waitfor() 58a, Runtime.cur() 29c, Runtime.thread.waitstatus 29a, Runtime.waitstatus.ChildStatus, Runtime.waitstatus.NothingToWaitfor 30i, Runtime.waitstatus.WaitFor 30i, Status.concstatus(), Status.getstatus(), and Status.setstatus().

⟨*signature* Status.concstatus 66a⟩≡ (105)

```
val concstatus : string -> string -> string
```

⟨*function* Status.concstatus 66b⟩≡ (106a)

```
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* 66c⟩+≡ (51f) ◁64a 67c▷

```
| 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 26g, Opcode.operation.Wastrue 67d, Opcode.t.F, and Opcode.t.I 26g.

⟨Interpreter.interpret_operation() *match* operation *cases* 66d⟩+≡ (34e) ◁65e 66e▷

```
| 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 67a, Interpreter.int_at_address(), Runtime.cur() 29c, Runtime.thread.pc 28i, and Status.truestatus().

### 7.7.2 if not

⟨Interpreter.interpret_operation() *match* operation *cases* 66e⟩+≡ (34e) ◁66d 67b▷

```
| 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 67a, Interpreter.int_at_address(), Runtime.cur() 29c, and Runtime.thread.pc 28i.

⟨*constant* Globals.ifnot 67a⟩≡ (99c)

```
  (* 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* 67b⟩+≡ (34e) ◁66e 68a▷

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

Uses Globals.ifnot 67a.

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 67c⟩+≡ (51f) ◁66c 67e▷

```
  | 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 26g, Opcode.t.F, and Opcode.t.I 26g.

### 7.7.3 while

### 7.7.4 for

⟨Opcode.operation *other control cases* 67d⟩+≡ (26g) ◁50b 68b▷

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

### 7.7.5 switch

⟨Compile.outcode_seq *in nested* xcmd() *match* cmd *cases* 67e⟩+≡ (51f) ◁67c 69b▷

```
  | 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 26a, Compile.split_when_case() 69a, Opcode.operation.Case 26g, Opcode.operation.Jump 26g, Opcode.operation.Mark 26g, Opcode.operation.Popm 26g, Opcode.t.F, and Opcode.t.I 26g.

⟨Interpreter.interpret_operation() *match* operation *cases* 68a⟩+≡ (34e) ◁67b 68c▷

```
  (* [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 26g, Runtime.cur() 29c, and Runtime.thread.pc 28i.

⟨Opcode.operation *other control cases* 68b⟩+≡ (26g) ◁67d

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

⟨Interpreter.interpret_operation() *match* operation *cases* 68c⟩+≡ (34e) ◁68a 68d▷

```
  (* (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 26g, Pattern.match_str() 102e, Runtime.cur() 29c, Runtime.pop_list() 104, Runtime.thread.argv, Runtime.thread.argv_stack 35e, and Runtime.thread.pc 28i.

⟨Interpreter.interpret_operation() *match* operation *cases* 68d⟩+≡ (34e) ◁68c 70a▷

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

Uses Opcode.operation.Popm 26g and Runtime.pop_list() 104.

⟨*function* `Compile.split_when_case` 69a⟩≡ (96b)
```
let split_when_case cmds =
  cmds |> List_.span (function
    | (A.Simple (A.Word ("case", false), _)) -> false
    | _ -> true
  )
```
Uses `Ast.cmd.Simple`, `Ast.value.Word` 26a, and `Common2.span()`.

### 7.7.6 Blocks: '{...}'

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 69b⟩+≡ (51f) ◁67e 69c▷
```
| A.Compound seq -> xseq seq eflag
```
Uses `Ast.cmd.Compound` 25c.

## 7.8 Functions

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

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 69c⟩+≡ (51f) ◁69b 69d▷
```
| 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` 48j, `Opcode.operation.Fn` 35b, `Opcode.operation.Mark` 26g, `Opcode.operation.Return` 26g, `Opcode.operation.Unlocal` 26g, `Opcode.t.F`, `Opcode.t.I` 26g, and `Opcode.t.S` 26g.

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

## 7.9 Variables

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

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 69d⟩+≡ (51f) ◁69c 84e▷
```
| 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 48j, Ast.cmd.EmptyCommand 87m, Compile.split_at_non_assign() 70c, Opcode.operation.Assign 86b, Opcode.operation.Local 26g, Opcode.operation.Mark 26g, Opcode.operation.Unlocal 26g, and Opcode.t.F.

⟨Interpreter.interpret_operation() *match* operation *cases* 70a⟩+≡ (34e) ◁68d 70b▷

```
  (* (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() 56c, Opcode.operation.Assign 86b, Runtime.cur() 29c, Runtime.pop_list() 104, Runtime.thread.argv, Runtime.var.v 28a, and Var.vlook().

⟨Interpreter.interpret_operation() *match* operation *cases* 70b⟩+≡ (34e) ◁70a 71b▷

```
  (* (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() 56c, Opcode.operation.Local 26g, Runtime.cur() 29c, Runtime.pop_list() 104, Runtime.thread.argv, Runtime.thread.locals 104, and Runtime.var.v 28a.

⟨*function* Compile.split_at_non_assign 70c⟩≡ (96b)

```
  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 48j and Compile.split_at_non_assign() 70c.

### 7.9.2 Variable uses ($<x>)

⟨Compile.outcode_seq *in nested* xword() *match* w *cases* 71a⟩+≡ (51f) ◁52d 85f▷

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

Uses Ast.value.Dollar 26a, Opcode.operation.Dollar 86b, Opcode.operation.Mark 26g, and Opcode.t.F.

⟨Interpreter.interpret_operation() *match* operation *cases* 71b⟩+≡ (34e) ◁70b 81b▷

```
  (* (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() 56c, Interpreter.vlook_varname_or_index(), Opcode.operation.Dollar 86b, Runtime.cur() 29c, Runtime.pop_list() 104, and Runtime.thread.argv.

### 7.9.3 Special variables

⟨*function* Interpreter.vlook_varname_or_index 71c⟩≡ (99g)

```
  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 28a, and Var.vlook().

# Chapter 8

# Builtins

## 8.1 Overview

⟨*function* `Builtin.is_builtin` 72a⟩≡ (94c)
```
let is_builtin s =
  List.mem s [
    "cd";
    ".";
    "eval";
    "exit";
    "flag";
    "finit";
    (* new in ocaml *)
    "show";
  ]
```

⟨*function* `Builtin.dispatch` 72b⟩≡ (94c)
```
let dispatch (caps : < Cap.chdir; Cap.exit; Cap.open_in; ..>) s =
  match s with
  ⟨Builtin.dispatch() match s cases 73a⟩
  | "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

⟨Builtin.dispatch() *match* s *cases* 73a⟩≡ (72b) 73c ▷
```
  | "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() 104, Runtime.thread.argv, Runtime.var.v 28a, Status.setstatus(), and Var.vlook().

⟨*function* Builtin.dochdir 73b⟩≡ (94c)
```
  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 '.'

⟨Builtin.dispatch() *match* s *cases* 73c⟩+≡ (72b) ◁73a 75c ▷
```
  | "." ->
      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() 56c, Globals.eflagok 99b, Logs.info(), Runtime.mk_thread() 63h, Runtime.pop_word() 54b, Runtime.push_list(), Runtime.push_redir() 30a, Runtime.redir.Close, Runtime.runq, Runtime.thread.argv, Runtime.thread.file 33g, Runtime.thread.iflag, and Runtime.thread.lexbuf.

⟨*constant* Builtin.ndots 74a⟩≡ (94c)

```
let ndots = ref 0
```

⟨*constant* Builtin.dotcmds 74b⟩≡ (94c)

```
(* 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 26g, Opcode.operation.Mark 26g, Opcode.operation.REPL 85b, Opcode.operation.Return 26g, Opcode.operation.Unlocal 26g, Opcode.operation.Word 26g, Opcode.t.F, and Opcode.t.S 26g.

## 8.6 eval

## 8.7 flag

⟨*global* Flags.hflags 75a⟩≡ (98b)

```
let hflags: char Hashtbl_.set = Hashtbl.create 10
```

⟨CLI.main() *other initializations* 75b⟩+≡ (32d) ◁33e

```
(* 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() 79a, Exit.t.OK, and Flags.hflags.

⟨Builtin.dispatch() *match* s *cases* 75c⟩+≡ (72b) ◁73c 75d▷

```
| "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() 56c, Flags.hflags, Runtime.pop_list() 104, Runtime.thread.argv, and Status.setstatus().

## 8.8 finit

⟨Builtin.dispatch() *match* s *cases* 75d⟩+≡ (72b) ◁75c

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

## 8.9 wait

# Chapter 9

# Environment

## 9.1 Var.init()

⟨*function* Var.vinit 77⟩≡ (106c)

```
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

# Chapter 11

# Initialization

## 11.1 Actual bootstrapping code

⟨*function* `CLI.bootstrap` 79a⟩≡ (95b)

```
(* 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()` 79a, `Flags.rcmain` 79b, `Opcode.operation.Assign` 86b, `Opcode.operation.Dollar` 86b, `Opcode.operation.Exit` 26g, `Opcode.operation.Mark` 26g, `Opcode.operation.Simple` 86c, `Opcode.operation.Word` 26g, `Opcode.t.F`, `Opcode.t.S` 26g, `Runtime.mk_thread()` 63h, and `Runtime.runq`.

## 11.2 Intitialization script and `rc -m /path/to/rcmain`

⟨*constant* `Flags.rcmain` 79b⟩≡ (98b)

```
(* can be changed with -m *)
```

```
  let rcmain = ref "/rc/lib/rcmain"
```

⟨CLI.main() options *elements* 80⟩+≡ (32d) ◁34a 82b▷

```
  "-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

⟨`Opcode.operation` *other cases* 81a⟩≡ (26g) 83e▷
```
  (* Globbing *)
  | Glob (* (value?) *)
```

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 81b⟩+≡ (34e) ◁71b 83g▷
```
  (* (value?) *)
  | O.Glob ->
      Logs.err (fun m -> m "TODO: interpret Glob");
      ()
```
Uses `Logs.err()` and `Opcode.operation.Glob` 26g.

## 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` 82a⟩≡ (98b)
```
(* -x, to print simple commands before executing them *)
let xflag = ref false
```

⟨`CLI.main()` `options` *elements* 82b⟩+≡ (32d) ◁80 82e▷
```
"-x", Arg.Set Flags.xflag,
" print each simple command before executing it";
```
Uses `Logs.level.Info`.

⟨`Op_process.op_Simple()` *possibly dump command* 82c⟩≡ (53h)
```
(* less: globlist () *)
if !Flags.xflag
then Logs.app (fun m -> m "|%s|" (String.concat " " argv));
```
Uses `Error.error()` 56c.

## 13.2 Printing subprocesses status: `rc -s`

⟨*constant* `Flags.sflag` 82d⟩≡ (98b)
```
(* -s, to print status when error in command just ran *)
let sflag = ref false
```

⟨`CLI.main()` `options` *elements* 82e⟩+≡ (32d) ◁82b 83a▷
```
"-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` 82f⟩≡ (35i)
```
(* 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` 82d, `Logs.app()`, `Status.getstatus()`, and `Status.truestatus()`.

## 13.3 Logging: `rc -v`

⟨`CLI.main()` *debugging initializations* 82g⟩≡ (32d) 90i▷
```
let level = ref (Some Logs.Warning) in
```
Uses `Flags.interactive` 33d.

⟨CLI.main() options *elements* 83a⟩+≡ (32d) ◁82e 83d▷
```
  (* 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 83c, Flags.strict_mode, and Logs.level.Debug.

⟨CLI.main() *logging initializations* 83b⟩≡ (32d)
```
  Logs_.setup !level ();
  Logs.info (fun m -> m "ran as %s from %s" argv.(0) (Sys.getcwd ()));
```
Uses CLI.do_action() 91c.

## 13.4 Failing fast: `rc -e`

⟨*constant* Flags.eflag 83c⟩≡ (98b)
```
  (* -e, for strict error checking. Abort the script when an error happens.*)
  let eflag = ref false
```

⟨CLI.main() options *elements* 83d⟩+≡ (32d) ◁83a 83j▷
```
  "-e", Arg.Set Flags.eflag,
  " exit if $status is non-null after a simple command";
```
Uses Flags.debugger.

⟨Opcode.operation *other cases* 83e⟩+≡ (26g) ◁81a 84c▷
```
  (* Error management *)
  | Eflag
```

⟨Compile.outcode_seq *in* A.Simple *case after emit* O.Simple 83f⟩≡ (52a)
```
  if eflag
  then emit (O.F O.Eflag);
```
Uses Opcode.operation.Eflag 26g and Opcode.t.F.

⟨Interpreter.interpret_operation() *match* operation *cases* 83g⟩+≡ (34e) ◁81b 84f▷
```
  | O.Eflag ->
      if !Globals.eflagok && not (Status.truestatus())
      then Process.exit caps (Status.getstatus())
```
Uses Globals.eflagok 99b, Process.exit() 65c, Status.getstatus(), and Status.truestatus().

⟨Compile.outcode_seq *in* A.Match *case after emit* O.Match 83h⟩≡ (60f)
```
  if eflag
  then emit (O.F O.Eflag);
```
Uses Opcode.operation.Eflag 26g and Opcode.t.F.

## 13.5 Strict mode: `rc -strict`

⟨*constant* Flags.strict_mode 83i⟩≡ (98b)
```
  let strict_mode = ref false
```

⟨CLI.main() options *elements* 83j⟩+≡ (32d) ◁83d 84a▷
```
  (* pad: I added that *)
  "-strict", Arg.Set Flags.strict_mode,
  " strict mode";
```
Uses Flags.dump_tokens.

# Chapter 14

# Advanced Features

## 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* 84a⟩+≡ (32d) ◁83j 89b▷
```
"-p", Arg.Unit (fun () -> ()),
" restrict $path to just (/bin /usr/bin)";
```
Uses `Flags.dump_opcodes`.

## 14.2 Advanced constructs

⟨`Ast.cmd` *other definition cases* 84b⟩≡ (25c)
```
| DelFn of value
```

⟨`Opcode.operation` *other cases* 84c⟩+≡ (26g) ◁83e 85b▷
```
| DelFn (* (name) *)
(* less: RdFn *)
```

⟨`Parser.cmd` *other cases* 84d⟩+≡ (46c) ◁49h
```
| TFn word       { DelFn $2 }
```

⟨`Compile.outcode_seq` *in nested* `xcmd()` *match* `cmd` *cases* 84e⟩+≡ (51f) ◁69d
```
| A.DelFn w ->
    emit (O.F O.Mark);
    xword w;
    emit (O.F O.DelFn);
```
Uses `Ast.cmd.DelFn` 49a, `Opcode.operation.DelFn` 81a, `Opcode.operation.Mark` 26g, and `Opcode.t.F`.

⟨`Interpreter.interpret_operation()` *match* `operation` *cases* 84f⟩+≡ (34e) ◁83g 85h▷
```
(* (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() 56c, Flags.strict_mode, Fn.flook() 28k, Opcode.operation.DelFn 81a, Runtime.cur() 29c, Runtime.fns, and Runtime.thread.argv.

### 14.2.1 Subshell: @ <cmd>

⟨*Parser tokens, operators other cases* 85a⟩≡ (24) 86j▷
```
%token TSubshell
```

⟨Opcode.operation *other cases* 85b⟩+≡ (26g) ◁84c 86f▷
```
(* ?? *)
| Subshell (* {... Xexit} *)
```

⟨Lexer.token() *symbol cases* 85c⟩+≡ (38a) ◁41f 86g▷
```
| "@" { TSubshell }
```

⟨Parser.keyword *other cases* 85d⟩+≡ (46j) ◁47g
```
| TSubshell { "@" }
```

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

⟨*Parser tokens, variables other cases* 85e⟩≡ (24) 87d▷
```
%token TCount
```

⟨Compile.outcode_seq *in nested* xword() *match* w *cases* 85f⟩+≡ (51f) ◁71a
```
| A.Count w ->
    emit (O.F O.Mark);
    xword w;
    emit (O.F O.Count);
```

Uses Ast.value.Count 26a, Opcode.operation.Count 26g, Opcode.operation.Mark 26g, and Opcode.t.F.

⟨Ast.value *other cases* 85g⟩≡ (26a) 86e▷
```
| Count of value (* $#foo *)
| Index of value * values (* $foo(...) *)
```

⟨Interpreter.interpret_operation() *match* operation *cases* 85h⟩+≡ (34e) ◁84f
```
(* (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() 56c, Interpreter.vlook_varname_or_index(), Opcode.operation.Count 26g, Runtime.cur() 29c, Runtime.pop_list() 104, Runtime.push_word() 54b, and Runtime.thread.argv.

⟨Lexer.token() *variable cases* 86a⟩+≡ (38a) ◁40f 87e▷
```
  | "$#"  { TCount }
```

⟨Opcode.operation *other stack cases* 86b⟩≡ (26g)
```
  | Count (* (name) *)
  | Concatenate (* (left)(right) *)
  | Stringify (* (name) *)
```

⟨Opcode.operation *other variable cases* 86c⟩≡ (26g)
```
  | Index  (* ??? *)
  | Local (* (name)(val) *)
  | Unlocal (* *)
```

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

⟨*Parser tokens, punctuation other cases* 86d⟩≡ (24) 87h▷
```
  %token TBackquote
```

⟨Ast.value *other cases* 86e⟩+≡ (26a) ◁85g 87f▷
```
  (* this causes the value and cmd types to be mutually recursive. Uses $IFS *)
  | CommandOutput of cmd_sequence
```
Uses Ast.value 94a.

⟨Opcode.operation *other cases* 86f⟩+≡ (26g) ◁85b
```
  | Backquote (* {... Xreturn} *)
```

⟨Lexer.token() *symbol cases* 86g⟩+≡ (38a) ◁85c 86i▷
```
  | "`" { TBackquote }
```

⟨Parser.comword *other cases* 86h⟩+≡ (46h) ◁49k 87g▷
```
  | 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* 86i⟩+≡ (38a) ◁86g 87l▷
```
  | ">[" (['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* 86j⟩+≡ (24) ◁85a
```
  %token<Ast.redirection_kind * int * int> TDup
```

⟨Parser.redir *other cases* 87a⟩≡ (48a)
```
| TDup        { Right $1 }
```

⟨Ast.redirection_kind *other cases* 87b⟩≡ (26c)
```
(* less: RHere *) (* < < *)
| RDup of int * int (* >[x=y] *)
```

⟨Ast.cmd *other redirection cases* 87c⟩≡ (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* 87d⟩+≡ (24) ◁85e
```
%token TStringify
```

⟨Lexer.token() *variable cases* 87e⟩+≡ (38a) ◁86a
```
| "$\"" { TStringify }
```

⟨Ast.value *other cases* 87f⟩+≡ (26a) ◁86e 87m▷
```
| Stringify of value (* $"foo " *)
```

⟨Parser.comword *other cases* 87g⟩+≡ (46h) ◁86h
```
| TCount word                    { Count $2 }
| TDollar word TSub words TCPar { Index ($2, $4) }
| TStringify word               { Stringify $2 }
```

⟨*Parser tokens, punctuation other cases* 87h⟩+≡ (24) ◁86d 87i▷
```
%token TSub
```

### 14.2.11 String concatenation

⟨*Parser tokens, punctuation other cases* 87i⟩+≡ (24) ◁87h
```
%token TCaret
```

⟨Parser.word *other cases* 87j⟩≡ (46i)
```
| word TCaret word { Concat ($1, $3) }
```

⟨Parser.first *other cases* 87k⟩≡ (46g)
```
| first TCaret word { Concat ($1, $3) }
```

⟨Lexer.token() *symbol cases* 87l⟩+≡ (38a) ◁86i
```
| "^" { TCaret }
```

⟨Ast.value *other cases* 87m⟩+≡ (26a) ◁87f
```
(* ^ 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

# Appendix A

# Debugging for the `mk` Developer

## A.1 Exception backtraces

⟨*constant* `Flags.debugger` 89a⟩≡ (98b)
```
let debugger = ref false
```

⟨`CLI.main()` `options` *elements* 89b⟩+≡ (32d) ◁84a 89d▷
```
"-debugger", Arg.Set Flags.debugger,
" ";
```
Uses `Flags.rflag` 90f.

⟨`CLI.main()` *when* `exn` *thrown in* `interpret()` 89c⟩≡ (32d)
```
if !Flags.debugger
then raise exn
else
  (match exn with
  ⟨CLI.main() when Failure exn thrown in interpret() 33a⟩
  (* 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* 89d⟩+≡ (32d) ◁89b 90g▷
```
(* 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` 89e⟩≡ (98b)
```
let dump_tokens = ref false
```

⟨Parse.lexfunc() *possibly dump tokens* 90a⟩≡ (37a)
```
|> (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 90b⟩≡ (98b)
```
let dump_ast = ref false
```

⟨Parse.parse_line() *possibly dump AST* 90c⟩≡ (43a)
```
|> (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 90d⟩≡ (98b)
```
let dump_opcodes = ref false
```

⟨Compile.compile() *possibly dump returned opcodes* 90e⟩≡ (50a)
```
|> (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().

## A.3 Opcode generator trace: `rc -r`

⟨*constant* Flags.rflag 90f⟩≡ (98b)
```
(* -r, similar to dump_opcodes, but at each step *)
let rflag = ref false
```

⟨CLI.main() options *elements* 90g⟩+≡ (32d) ◁89d 91a▷
```
"-r", Arg.Set Flags.rflag,
" print internal form of commands (opcodes)";
```
Uses CLI.usage 33c.

⟨CLI.interpret() *if* rflag 90h⟩≡ (34b)
```
(* 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 26g, Runtime.thread.argv, Runtime.thread.argv_stack 35e, and Runtime.thread.code 28i.

## A.4 CLI actions

⟨CLI.main() *debugging initializations* 90i⟩+≡ (32d) ◁82g
```
let action = ref "" in
```
Uses Flags.interactive 33d.

⟨CLI.main() options *elements* 91a⟩+≡ (32d) ◁90g
```
  (* pad: I added that *)
  "-test_parser", Arg.Unit (fun () -> action := "-test_parser"), " ";
```
Uses UConsole.print().

⟨CLI.main() *CLI action processing* 91b⟩≡ (32d)
```
  (* 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 91c⟩≡ (95b)
```
  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() 63h, Runtime.runq, and Runtime.thread.file 33g.

# Appendix B

# Examples of rc scripts

## B.1 /rc/lib/rcmain

```
# 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 94a⟩≡
```
(* 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 26a⟩
⟨type Ast.values 25d⟩

⟨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 26b⟩
⟨type Ast.redirection_kind 26c⟩
[@@deriving show]

⟨type Ast.line 25a⟩
[@@deriving show]
```

## C.2 Builtin.mli

⟨Builtin.mli 94b⟩≡
```
⟨signature Builtin.is_builtin 54e⟩

⟨signature Builtin.dispatch 54f⟩
```

## C.3 Builtin.ml

⟨shell/Builtin.ml 94c⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
```

```
open Common

module R = Runtime
module E = Error
module O = Opcode

⟨function Builtin.is_builtin 72a⟩
⟨constant Builtin.ndots 74a⟩
⟨function Builtin.dochdir 73b⟩
⟨constant Builtin.dotcmds 74b⟩
⟨function Builtin.dispatch 72b⟩
```

## C.4 CLI.mli

⟨CLI.mli 95a⟩≡

```
⟨type CLI.caps 32a⟩

⟨signature CLI.main 32b⟩

(* internals *)
⟨signature CLI.interpret_bootstrap 33b⟩
```

## C.5 CLI.ml

⟨shell/CLI.ml 95b⟩≡

```
(* 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 32a⟩

(* -d and -p are dead according to man page so I removed them *)
⟨constant CLI.usage 33c⟩

(*****************************************************************************)
(* Testing *)
(*****************************************************************************)

⟨function CLI.do_action 91c⟩

(*****************************************************************************)
(* Main algorithm *)
(*****************************************************************************)

⟨constant CLI._bootstrap_simple 34c⟩
⟨function CLI.bootstrap 79a⟩

⟨function CLI.interpret_bootstrap 34b⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function CLI.main 32d⟩
```

Uses Logs.level.Warning.

## C.6 Compile.mli

```
⟨Compile.mli 96a⟩≡
  ⟨signature Compile.compile 26d⟩
```

## C.7 Compile.ml

```
⟨shell/Compile.ml 96b⟩≡
  open Common

  module A = Ast
  module O = Opcode

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* AST to opcodes.
  *)

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)
```

```
⟨function Compile.split_at_non_assign 70c⟩
⟨function Compile.split_when_case 69a⟩

(*****************************************************************************)
(* Compilation algorithm *)
(*****************************************************************************)

⟨function Compile.outcode_seq 51f⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function Compile.compile 50a⟩
```

## C.8 Env.mli

```
⟨Env.mli 97a⟩≡
  type t = (string * string list) list

  val read_environment : < Cap.env ; .. > -> t
```

## C.9 Env.ml

```
⟨shell/Env.ml 97b⟩≡
  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 97c⟩≡
  ⟨signature Error.error 56b⟩
```

## C.11 Error.ml

⟨shell/Error.ml 98a⟩≡
```
module R = Runtime

⟨function Error.error 56c⟩
```

## C.12 Flags.ml

⟨shell/Flags.ml 98b⟩≡
```
open Common

⟨constant Flags.interactive 33d⟩
⟨constant Flags.login 33i⟩

⟨constant Flags.eflag 83c⟩
⟨constant Flags.rflag 90f⟩
⟨constant Flags.sflag 82d⟩
⟨constant Flags.xflag 82a⟩

(* less: let cflag = ref "" *)

⟨global Flags.hflags 75a⟩

⟨constant Flags.rcmain 79b⟩

(* pad: I added this one *)
⟨constant Flags.strict_mode 83i⟩

⟨constant Flags.dump_tokens 89e⟩
⟨constant Flags.dump_ast 90b⟩
⟨constant Flags.dump_opcodes 90d⟩

⟨constant Flags.debugger 89a⟩
```

## C.13 Fn.mli

⟨Fn.mli 98c⟩≡
```
⟨signature Fn.flook 28j⟩
```

## C.14 Fn.ml

⟨shell/Fn.ml 98d⟩≡
```
module R = Runtime

⟨function Fn.flook 28k⟩
```

## C.15 Glob.mli

⟨Glob.mli 98e⟩≡

## C.16 Glob.ml

⟨shell/Glob.ml 99a⟩≡

## C.17 Globals.ml

⟨*constant* Globals.eflagok 99b⟩≡ (99c)
```
(* this is set after the first . of rcmain *)
let eflagok = ref false
```

⟨shell/Globals.ml 99c⟩≡

⟨*constant* Globals.skipnl 40j⟩
⟨*constant* Globals.errstr 56d⟩

⟨*constant* Globals.ifnot 67a⟩
⟨*constant* Globals.eflagok 99b⟩

## C.18 Heredoc.mli

⟨Heredoc.mli 99d⟩≡

## C.19 Heredoc.ml

⟨shell/Heredoc.ml 99e⟩≡

## C.20 Interpreter.mli

⟨Interpreter.mli 99f⟩≡
⟨*signature* Interpreter.interpret_operation 34d⟩

## C.21 Interpreter.ml

⟨shell/Interpreter.ml 99g⟩≡
```
open Common
open Regexp.Operators

module O = Opcode
module R = Runtime
module E = Error

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

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* Interpreter.file_descr_of_int 62b⟩
⟨*function* Interpreter.int_at_address 60a⟩

⟨*function* `Interpreter.vlook_varname_or_index` 71c⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```

⟨*function* `Interpreter.interpret_operation` 34e⟩

## C.22 Lexer.mli

⟨`Lexer.mli` 100a⟩≡
⟨*signature* `Lexer.token` 37b⟩

⟨*exception* `Lexer.Lexical_error` 37c⟩

## C.23 Lexer.mll

## C.24 Main.ml

⟨`shell/Main.ml` 100b⟩≡

```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Xix_shell

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```

⟨*toplevel* `Main._1` 32c⟩

## C.25 Op_process.ml

⟨`shell/Op_process.ml` 100c⟩≡

```
open Fpath_.Operators

module R = Runtime
module E = Error
```

⟨*function* `Op_process.execute` 55c⟩

⟨*function* `Op_process.exec` 55b⟩

⟨*function* `Op_process.forkexec` 55a⟩

⟨*function* `Op_process.op_Simple` 53h⟩

Uses `Runtime.doredir()` 63b and `Runtime.thread.redirections` 29a.

## C.26 Op_repl.ml

⟨`shell/Op_repl.ml` 100d⟩≡

```
module R = Runtime
```

⟨*function* Op_repl.op_REPL 35i⟩

## C.27 Opcode.ml

⟨shell/Opcode.ml 101a⟩≡
```
(* 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 26g⟩
```
[@@deriving show]
```
⟨*type* Opcode.opcode 26f⟩
```
[@@deriving show]
```
⟨*type* Opcode.codevec 26e⟩
```
[@@deriving show]
```

## C.28 PATH.mli

⟨PATH.mli 101b⟩≡
⟨*signature* PATH.find_in_path 58b⟩

⟨*signature* PATH.search_path_for_cmd 58c⟩

## C.29 PATH.ml

⟨shell/PATH.ml 101c⟩≡
```
open Common
open Fpath_.Operators
open Regexp.Operators

module R = Runtime
```

⟨*function* PATH.search_path_for_cmd 58d⟩
⟨*function* PATH.find_in_path 59a⟩

## C.30 Parser.mly

## C.31 Parse.mli

⟨Parse.mli 101d⟩≡
⟨*signature* Parse.parse_line 36a⟩

## C.32 Parse.ml

⟨shell/Parse.ml 102a⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators

module R = Runtime

⟨function Parse.error 44a⟩

⟨function Parse.parse_line 43a⟩
```

## C.33 Pattern.mli

⟨signature Pattern.match_str 102b⟩≡ (102c)
```
val match_str : string -> pattern -> bool
```

⟨Pattern.mli 102c⟩≡
```
(* todo: Glob char *)
⟨type Pattern.pattern 102d⟩

⟨signature Pattern.match_str 102b⟩
```

## C.34 Pattern.ml

⟨type Pattern.pattern 102d⟩≡ (102)
```
type pattern = string
```

⟨function Pattern.match_str 102e⟩≡ (102f)
```
(* todo: handle [ * ?  *)
let match_str s1 s2 =
  s1 = s2
```

⟨shell/Pattern.ml 102f⟩≡
```
⟨type Pattern.pattern 102d⟩

⟨function Pattern.match_str 102e⟩
```

## C.35 Process.mli

⟨Process.mli 102g⟩≡
```
⟨type Process.waitfor_result 57i⟩

⟨signature Process.return 50e⟩
⟨signature Process.exit 65b⟩
⟨signature Process.waitfor 57h⟩
⟨signature Process.s_of_unix_error 56e⟩
```

## C.36 Process.ml

⟨shell/Process.ml 103a⟩≡
```
open Common

module R = Runtime

```
⟨*function* Process.s_of_unix_error 56f⟩

⟨*function* Process.exit 65c⟩

⟨*function* Process.return 51a⟩

⟨*type* Process.waitfor_result 57i⟩

⟨*function* Process.waitfor 58a⟩

## C.37 Prompt.mli

⟨Prompt.mli 103b⟩≡
⟨*signature* Prompt.doprompt 39c⟩
⟨*signature* Prompt.prompt 39f⟩

⟨*signature* Prompt.pprompt 39h⟩

## C.38 Prompt.ml

⟨shell/Prompt.ml 103c⟩≡
```
module R = Runtime

```
⟨*constant* Prompt.doprompt 39d⟩
⟨*constant* Prompt.prompt 39g⟩

⟨*function* Prompt.pprompt 39i⟩

## C.39 Runtime.mli

⟨Runtime.mli 103d⟩≡

⟨*type* Runtime.varname 27b⟩
⟨*type* Runtime.value 27a⟩
⟨*type* Runtime.var 28a⟩

```
val string_of_var : var -> string

```
⟨*type* Runtime.fn 28i⟩
```
val show_fn: fn -> string
```

⟨*type* Runtime.thread 29a⟩
⟨*type* Runtime.redir 30i⟩

⟨*type* Runtime.waitstatus 31c⟩

```
(* globals *)

⟨signature Runtime.globals 28b⟩
⟨signature Runtime.fns 28g⟩
⟨signature Runtime.runq 29b⟩

(* API *)

⟨signature Runtime.cur 29d⟩
⟨signature Runtime.push_list 53c⟩
⟨signature Runtime.pop_list 54a⟩
⟨signature Runtime.push_word 29h⟩
⟨signature Runtime.pop_word 30b⟩
⟨signature Runtime.push_redir 62c⟩
⟨signature Runtime.pop_redir 63a⟩
⟨signature Runtime.turf_redir 63d⟩
⟨signature Runtime.doredir 63g⟩

⟨signature Runtime.mk_thread 29f⟩
```

## C.40 Runtime.ml

```
⟨shell/Runtime.ml 104⟩≡
  (* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Globals used by rc during interpretation.
   *)

  (*****************************************************************************)
  (* Types and globals *)
  (*****************************************************************************)

  ⟨type Runtime.varname 27b⟩
  ⟨type Runtime.value 27a⟩
  ⟨type Runtime.var 28a⟩

  (* for boostrap-mk.sh and ocaml-light to work without deriving *)
  let show_fn _ = "NO DERIVING"
  [@@warning "-32"]

  ⟨type Runtime.fn 28i⟩
  [@@deriving show]

  ⟨type Runtime.thread 29a⟩

  ⟨type Runtime.redir 30i⟩

  ⟨type Runtime.waitstatus 31c⟩

  ⟨global Runtime.globals 28c⟩

  ⟨global Runtime.fns 28h⟩

  (* less: argv0 *)
```

⟨*constant* Runtime.runq 29c⟩

```
(*****************************************************************************)
(* cur *)
(*****************************************************************************)
```

⟨*function* Runtime.cur 29e⟩

```
(*****************************************************************************)
(* Stack (argv) API *)
(*****************************************************************************)
```

⟨*function* Runtime.push_list 53d⟩
⟨*function* Runtime.pop_list 54b⟩

⟨*function* Runtime.push_word 30a⟩
⟨*function* Runtime.pop_word 30c⟩

```
(*****************************************************************************)
(* Redirection *)
(*****************************************************************************)
```

⟨*function* Runtime.push_redir 62d⟩
⟨*function* Runtime.pop_redir 63b⟩

⟨*function* Runtime.turf_redir 63e⟩

⟨*function* Runtime.doredir 63h⟩

```
(*****************************************************************************)
(* Constructor *)
(*****************************************************************************)
```

⟨*function* Runtime.mk_thread 29g⟩

```
(*****************************************************************************)
(* 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() 29c, Runtime.redir.Close, Runtime.thread.argv, Runtime.thread.argv_stack 35e, Runtime.thread.redirections 29a, Runtime.thread.waitstatus 29a, Runtime.var, and Runtime.waitstatus.NothingToWaitfor 30i.

## C.41 Status.mli

⟨Status.mli 105⟩≡

```
(* helpers to manipulate the "status" special variable (see Var.ml) *)
```

⟨*signature* Status.setstatus 57a⟩
⟨*signature* Status.getstatus 57c⟩
⟨*signature* Status.concstatus 66a⟩
⟨*signature* Status.truestatus 57e⟩

## C.42 Status.ml

⟨shell/Status.ml 106a⟩≡
```
(* 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 57b⟩
⟨*function* Status.getstatus 57d⟩
⟨*function* Status.concstatus 66b⟩
⟨*function* Status.truestatus 57f⟩

## C.43 Var.mli

⟨Var.mli 106b⟩≡
```
(* Helpers to manipulate rc shell variables *)
```

⟨*signature* Var.gvlook 28d⟩
⟨*signature* Var.vlook 30d⟩
⟨*signature* Var.setvar 30f⟩
⟨*signature* Var.vinit 28f⟩

## C.44 Var.ml

⟨shell/Var.ml 106c⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common

module R = Runtime

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Helpers to manipulate rc shell variables *)

(*****************************************************************************)
(* API *)
(*****************************************************************************)
```

⟨*function* Var.gvlook 28e⟩
⟨*function* Var.vlook 30e⟩

⟨*function* Var.setvar 30g⟩

⟨*function* Var.vinit 77⟩

# 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.

# 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, 12

[Bou15] Stephen R. Bourne. Early days of unix and design of sh. In *BSDCan - The BSD Conference*, 2015. `https://www.bsdcan.org/2015/schedule/attachments/306_srbBSDCan2015.pdf`, also in `shells/docs/early-days-unix-sh-bourne.pdf`. cited page(s) 12

[Duf90] Tom Duff. Rc – a shell for plan 9 and unix. In *Unix Research System: Papers (Vol 2) 10th ed.* Saunders College, 1990. cited page(s) 9

[Duf00] Tom Duff. Rc – the plan 9 shell. Technical report, Bell Labs, 2000. Also available at `shells/docs/rc.pdf`. cited page(s) 12

[Joh79] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual Vol 2b*, 1979. Also available at `generators/docs/yacc.pdf`. cited page(s) 9

[Joy86] William Joy. An introduction to the c shell. In *UNIX Programmer's Supplementary Documents 1*, 1986. Available at `https://docs.freebsd.org/44doc/usd/04.csh/paper.pdf`. cited page(s) 10

[Kid05] Oliver Kiddle. *From Bash to Z Shell*. Apress, 2005. cited page(s) 10, 12

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 12

[Kor94] David G. Korn. ksh - an extensible high level language. In *Proceedings of the USENIX 1994 Very High Level Languages Symposium*, 1994. cited page(s) 10

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984. cited page(s) 12

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 11

[Mic08] Randal K. Michael. *Master UNIX Shell Scripting*. Wiley, 2008. cited page(s) 12

[New95] Cameron Newham. *Learning the bash Shell*. O'Reilly, 1995. cited page(s) 10, 12

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 12

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 10, 12, 15, 16, 18

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 12, 21

[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, 10

[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