Principia Softwarica
Fundamental Literate System Programs
version 0.5

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Ken Thompson, Rob Pike, Dave Presotto, Phil Winterbottom,
Tom Duff, Andrew Hume, Russ Cox,
Xavier Leroy, Fabrice Le Fessant, and Francois Rouaix.

April 28, 2026

Contents

1 Introduction **6**
- 1.1 Motivations . . . 6
- 1.2 The ideal teaching operating system . . . 8
- 1.3 The Plan 9 operating system . . . 8
- 1.4 The C language and ARM architecture . . . 10
- 1.5 Literate programming . . . 10
- 1.6 Getting started . . . 11
- 1.7 Requirements . . . 11
- 1.8 Copyright . . . 12
- 1.9 Acknowledgments . . . 12

2 Overview **13**
- 2.1 Code organization . . . 13
- 2.2 Software architecture . . . 13

3 The Literate Programs **17**
- 3.1 The Core system . . . 17
  - 3.1.1 The kernel: `9pi` . . . 17
  - 3.1.2 The core library: `libc` . . . 17
  - 3.1.3 The shell: `rc` . . . 17
- 3.2 The development toolchain . . . 18
  - 3.2.1 The C compiler: `5c` . . . 18
  - 3.2.2 The assembler: `5a` . . . 18
  - 3.2.3 The linker: `5l` . . . 18
  - 3.2.4 The processor emulator: `5i` . . . 18
  - 3.2.5 The OCaml compiler: `ocamlc` and `ocamlrun` . . . 19
  - 3.2.6 The Lex and Yacc code generators: `olex` and `oyacc` . . . 19
- 3.3 The developer tools . . . 19
  - 3.3.1 The text editor: `efuns` . . . 19
  - 3.3.2 The build system: `mk` . . . 20
  - 3.3.3 The version control system: `ogit` . . . 20
  - 3.3.4 The debugger: `db` . . . 20
  - 3.3.5 The profiler: `prof` . . . 20
  - 3.3.6 The typesetting system: `troff` . . . 20
- 3.4 Graphics . . . 20
  - 3.4.1 The graphics stack: `draw` . . . 21
  - 3.4.2 The windowing system: `rio` . . . 21
  - 3.4.3 The graphical user interface toolkit: `libpanel` . . . 21
- 3.5 Networking . . . 21
  - 3.5.1 The network stack: `net` . . . 21

3.5.2 The web browser: mmm . . . 21
3.6 Utilities: cat, grep, ... 22
3.7 Applications . . . 23

4 Conclusion **24**

A Other Teaching Operating Systems **25**

B Bootstrapping from Scratch **27**
B.1 A basic computer . . . 28
B.2 A booting procedure . . . 28
B.3 Physical programs . . . 29
B.4 Phase 1: Machine code . . . 29
B.5 Phase 2: Assembly . . . 31
B.6 Phase 3: C . . . 34
B.7 Phase 4: OCaml . . . 35
B.8 Summary of Plan 9 ancestor programs . . . 36

C Literate Program Example: ed **38**
C.1 Introduction . . . 38
C.2 Motivations: mkenam . . . 38
C.3 Core data structures . . . 39
C.3.1 The backing store: tfname, tfile, and tline . . . 40
C.3.2 The target file: savedfile . . . 40
C.3.3 The lines as file offsets: zero . . . 40
C.3.4 Cursors: dot and dol . . . 41
C.3.5 Line input and output buffers: line and linebuf . . . 41
C.3.6 Other globals: count, col, etc. . . . 41
C.4 main() . . . 42
C.4.1 ed - and vflag . . . 43
C.4.2 ed -o and oflag . . . 43
C.4.3 mktemp() . . . 43
C.4.4 init() . . . 44
C.4.5 quit() . . . 45
C.5 Displaying and reading text . . . 45
C.5.1 Displaying text . . . 45
C.5.2 Reading text . . . 46
C.6 commands() interpreter loop . . . 48
C.7 reading a file: r . . . 49
C.7.1 Reading a filename() . . . 49
C.7.2 setwide() and squeeze() . . . 50
C.7.3 append() and getfile() . . . 51
C.7.4 putline() (simplified) . . . 52
C.7.5 exfile() . . . 54
C.8 writing a file: w . . . 54
C.8.1 putfile() . . . 55
C.8.2 getline() (simplified) . . . 55
C.8.3 Write and quit: wq . . . 56
C.9 Main commands . . . 56
C.9.1 printing lines: p . . . 56

C.9.2 printing remembered file: f . . . 58
C.9.3 printing line number: = . . . 58
C.9.4 append and insert: a, i . . . 58
C.9.5 quitting: q . . . 58
C.9.6 deleting lines: d . . . 59
C.9.7 changing lines: c . . . 59
C.9.8 moving and copying lines: m and t . . . 60
C.10 Command addresses . . . 62
C.10.1 Reading addr1 and addr2 . . . 63
C.10.2 address() . . . 64
C.10.3 Basic addresses: . and $ . . . 65
C.11 Search and replace . . . 65
C.11.1 Search as addresses: /re/ and ?re? . . . 66
C.11.2 Reading and compiling a regexp: compile() . . . 66
C.11.3 match() . . . 68
C.11.4 Reading and compiling a substitution: compsub() . . . 69
C.11.5 substitute: s . . . 70
C.11.6 Advanced substitutions . . . 72
C.11.7 global commands: g/re/cmd and v/re/cmd . . . 75
C.12 Advanced features . . . 77
C.12.1 Running a shell command: ! . . . 77
C.12.2 Advanced listing: l . . . 78
C.12.3 Advanced listing: n . . . 80
C.12.4 joining lines: j . . . 80
C.12.5 browse: b . . . 81
C.12.6 edit remembered file: e . . . 82
C.12.7 Append-only files . . . 82
C.13 Optimizations . . . 82
C.13.1 Optimized putline() . . . 83
C.13.2 getblock() . . . 83
C.13.3 Optimized getline() . . . 85
C.13.4 gdelete() . . . 86
C.14 Error management . . . 86
C.14.1 error() . . . 87
C.14.2 Notes/signals management . . . 88
C.15 Index . . . 89

D OCaml Literate Program Example: oed **90**
D.1 Introduction . . . 90
D.2 Core data structures . . . 90
D.2.1 Token . . . 90
D.2.2 Parser state . . . 90
D.2.3 Line addresses . . . 91
D.2.4 The environment . . . 91
D.2.5 The backing store: tfname, tfile, and tline . . . 92
D.2.6 The lines as file offsets: zero . . . 92
D.2.7 Cursors: dot and dol . . . 92
D.2.8 Other globals . . . 93
D.3 CLI.main() . . . 93
D.3.1 ed <file> and preloaded commands globp . . . 94

D.3.2 `ed -` and `vflag` . . . 94
D.3.3 `ed -o` and `oflag` . . . 95
D.3.4 `init()` . . . 95
D.3.5 `quit()` . . . 96
D.4 Displaying and reading text . . . 96
D.4.1 Displaying text . . . 97
D.4.2 Reading text . . . 97
D.5 Parsing commands . . . 98
D.6 `CLI.commands()` interpreter loop . . . 99
D.7 reading a file: `r` . . . 100
D.7.1 Reading a `filename()` . . . 101
D.7.2 `read()` . . . 101
D.7.3 `setwide()` and `squeeze()` . . . 102
D.7.4 `append()` and `getfile()` . . . 102
D.7.5 `putline()` . . . 103
D.7.6 `exfile()` . . . 104
D.8 writing a file: `w` . . . 104
D.8.1 `putfile()` . . . 105
D.8.2 `getline()` . . . 105
D.9 Main commands . . . 106
D.9.1 printing lines: `p` . . . 106
D.9.2 printing remembered file: `f` . . . 107
D.9.3 printing line number: `=` . . . 107
D.9.4 append and insert: `a`, `i` . . . 108
D.9.5 quitting: `q` . . . 108
D.9.6 deleting lines: `d` . . . 108
D.9.7 changing lines: `c` . . . 109
D.10 Command addresses . . . 109
D.10.1 Parsing addresses . . . 109
D.10.2 Evaluating addresses . . . 111
D.11 Search and replace . . . 111
D.11.1 Parsing regexps . . . 111
D.11.2 Search as addresses: `/re/` and `?re?` . . . 112
D.11.3 `match()` . . . 113
D.11.4 substitute: `s` . . . 113
D.11.5 Advanced substitutions . . . 114
D.11.6 global commands: `g/re/cmd` and `v/re/cmd` . . . 115
D.12 Advanced features . . . 116
D.12.1 Running a shell command: `!` . . . 116
D.12.2 `ed -r` and `rflag` restricted mode . . . 116
D.12.3 The caret . . . 117
D.12.4 The semicolon . . . 117
D.13 Error management and logging . . . 118
D.13.1 Legacy error management . . . 118
D.13.2 Improved error reporting and logging . . . 118
D.14 Debugging support . . . 119
D.15 Index . . . 119

References **119**

Chapter 1

Introduction

Principia Softwarica is a series of books explaining how things work in a computer by describing with full details all the source code of all the essential programs used by a programmer. Among those essential programs are the kernel, the shell, the windowing system, the compiler, the linker, the editor, or the debugger. Each program will be covered by a separate book.

The books not only describe the implementations of essential programs, *they are* the implementations of those programs. Indeed, each program in Principia Softwarica comes from a *literate program* [Knu92], which is a document containing both source code and documentation and where the code is organized and presented in a way to facilitate its comprehension. The actual code and the book are derived both automatically from this literate program. See Appendix C for a concrete example of literate program.

The goal of the report you are reading now is to introduce the series and to give a quick overview of the Principia Softwarica programs. They form together the foundation on top of which all applications can be built. Similar to *Principia Mathematica* [WR13], which is a series of books covering the foundations of mathematics, the goal of Principia Softwarica is to cover the fundamental programs. Those programs are mostly all *meta programs*, which are programs in which the input and/or output are other programs. For instance, the kernel is a program that manages other programs; the compiler is a program that generates other programs. Those programs are also sometimes referred as *system software*, in opposition to *application software* (e.g., spreadsheets, word processors, email clients), which I will not cover in Principia Softwarica.

1.1 Motivations

Why did I write those books? The main reason is that I have always been curious and always wanted to understand how things work under the hood, fully, to the smallest detail. Programs are now running the world; it is thus important to understand those programs, to be computer literate, and source code is what defines those programs.

There are already lots of books explaining how computers work, explaining the concepts, theories, and algorithms behind programs such as kernels or compilers. There are also a few books about debuggers. However, all those books rarely explain everything with full details, which is what source code is all about. There are a few books that include the whole source code of the program described, for instance, the books about Minix [Tan87], XINU [Com84], or LCC [FH95]. However, those books cover only a few essential programs, and mostly always either the kernel or the compiler. Moreover, they do not form a coherent set.

Enter Principia Softwarica, a set of books covering all essential programs, in a coherent way. In addition to the kernel and compiler, Principia Softwarica covers also the graphics stack, the networking stack, the windowing system, the assembler, the linker, and many other fundamental programs that have never been fully explained before to the best of my knowledge.

I want to demystify those programs by showing their code, and by showing that they are actually not that complicated. I hope to remove some of the mental barriers people have that prevent them from extending the

tools they use every day: text editors, compilers, even kernels. As a side effect, it will also maybe help people imagine better systems. Indeed, it can be very intimidating to invent something completely new if you have no clue that it is actually possible to build from scratch a complete operating system.

Another motivation for those books, in addition to satisfy my curiosity, as well as your curiosity, is that I think you are a better programmer if you understand how things work under the hood. In my opinion, you become a better C programmer when you understand roughly what code generates the C compiler. A good way to write more efficient code, or to avoid writing really slow code, is to have some ideas of the assembly code generated for your code by the compiler. In the same way, you can better use resources, for instance, memory, if you have some ideas about how the kernel manages for you these resources; you can better fix latency issues if you understand how the networking stack works. The Principia Softwarica books can complement the excellent book *Computer systems: a programmer's perspective* [BO10] by illustrating the many concepts this book introduces with concrete code.

I also think it is easier to debug programs if you better understand the environment in which those programs evolve, if you understand the whole software stack, and how things interact with each other. Indeed, to fully understand certain error messages, e.g., from the kernel, from the networking stack, or from the linker, it is very useful to have some ideas about what those programs do. Moreover, even if in almost all situations the bug or the performance issue is in your program, it could also be sometimes in a core library, in the kernel, in the compiler, or in the linker. It is very rare, but when those situations happen, if you have no clue about the environment in which your program runs, you will never be able to fix your problem.

Finally, by showing code written by great programmers, I hope you will also learn how to write better programs. In other engineering fields it is quite common for the students to learn from the work of the masters of their fields.

Here are a few questions I hope the Principia Softwarica books will answer:

- What happens when the user turns on the computer? Which program gets executed first? What does this program?
- What exactly happens when the user types `ls` in a terminal? What is the set of programs involved in such a command? What is the trace of this command through the different layers of the software stack, from the keyboard interrupt to the display of text glyphs on the screen?
- How does source code get compiled, assembled, linked, and finally loaded in memory? What is the memory image of a program? How does it relate to the original source code? How can a debugger display debugging information from a binary?
- How does a debugger work? Does it rely on special services from the processor or from the kernel to implement breakpoints?
- Which program contains the memory allocator? The kernel or The C library? How `malloc()` is implemented? How does garbage collection work?
- How certain graphical elements (e.g., rectangles, ellipses, characters) are rendered on the screen? How does the graphics card help? How does the kernel help? How things are intercepted by the windowing system to make sure applications can not draw in other windows?
- What happens when you open a connection to another machine, when you type a URL in your web browser? How can you achieve a reliable communication on an unreliable physical network?
- What does it take to port an entire operating system to a new machine, for instance, the Raspberry Pi[1]?

[1] https://www.raspberrypi.org/

1.2 The ideal teaching operating system

The question now is which actual source code to present? The code of the major operating systems (e.g., GNU/Linux, macOS, Windows) is gigantic with hundreds of millions lines of code (LOC). Here by operating system I mean not only the kernel but also the programs (the windowing system, the compiler, etc) that provide the platform for running application software. It is impossible to understand such large codebases.

I think there is hope though, that it is possible to understand fairly well everything, that it is possible to answer all of the questions mentioned in the previous section by focusing on the *essence* of those major operating systems. I think it is possible to design a teaching operating system with capabilities similar to the mainstream operating systems, but with a fraction of their code size. Here are the requirements for the ideal teaching operating system I want to use as the basis for the Principia Softwarica books:

- *Open source*: I want to show code, lots of code, and I want you to be able to play with such code, to modify it, so the ideal operating system must be open source.
- *Small*: I want programs that can be described in books of reasonable size. The program needs to be a bit minimalist.
- *As simple as possible, but not simpler*: I want small code, but I do not want to show toy code. For instance, ARM [Sea01] is a simpler architecture than x86 [Int86], so it makes sense to present the code of an ARM assembler rather than an x86 assembler, but I do not want to show a toy assembler for a toy architecture. In the same way I can present the code of a C compiler without certain advanced state-of-the-art optimizations, to simplify the presentation, but I do not want to show a toy compiler for a toy language.
- *Real*: It has to run on real machines (e.g., an x86 desktop, a Mac laptop, a Raspberry Pi), so you can play with it.
- *Complete*: It needs to not only have a kernel and a compiler but also a graphics stack, a networking stack, a windowing system, etc.
- *Coherent*: The whole set of programs must form a coherent set, so that the interactions between those programs can also be described, and described succinctly.
- *Self hosting*: I want to be able to improve, to recompile, and to run everything under the system itself.

Fortunately, this ideal teaching operating system already exists; it is Plan 9.

1.3 The Plan 9 operating system

Plan 9[2] is the successor of UNIX. It was designed from scratch by a small team of great programmers (Rob Pike, Dave Presotto, Phil Winterbottom), including the original creator of UNIX (Ken Thompson). Their goal was to redesign UNIX to better integrate graphics and networking, which both became popular after UNIX was originally invented. In some sense, Plan 9 is a kind of UNIX 2.0.

The choice of Plan 9 as the basis for the Principia Softwarica books may not be obvious, but it is in my opinion the simplest and at the same time fairly complete operating system. If you look at the screenshot of Plan 9 in action in Figure 1.1, you will see many features:

- A screen with basic graphics and multiple windows
- Multiple shells running independent commands at the same time

[2] plan9.bell-labs.com/plan9/

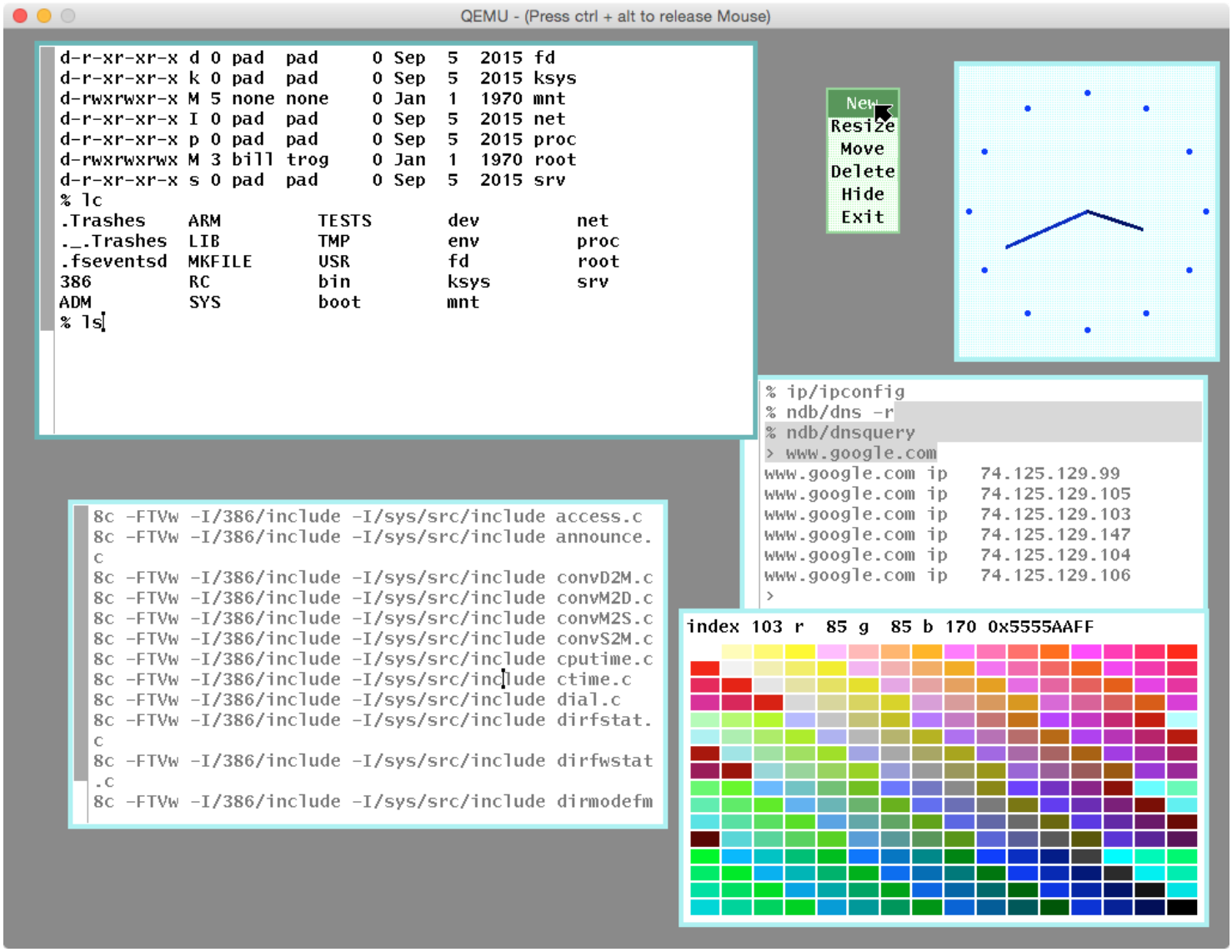


Figure 1.1: Plan 9 in action.

- A simple clock graphical application
- A simple program communicating through the network

By comparison, if you look at other systems you will see that Plan 9 *in essence* provides the same core services than GNU/Linux/Xorg, macOS, or Microsoft Windows, without certain bells and whistles, and with a significantly smaller codebase. Indeed, my fork of Plan 9 (whose source code is available at https://github.com/aryx/principia-softwarica/), which includes all the essential programs described in all the Principia Softwarica books, and even a few more programs, is less than 350 000 LOC.

The Plan 9 programs are minimalist but powerful, and their source code is simple and small. The creators of Plan 9 were not afraid to rethink everything, not just the kernel, even if it meant not being backward compatible with UNIX. This led to programs with more elegant designs, to the removal of many ugly corners in UNIX, and ultimately to less source code while still providing more services. For instance, thanks to a few novel ideas in the kernel, namely per-process namespace, user-space filesystems, and union mount, every application who wish to interact with the console can just use the uniform `/dev/cons` device file, whether this console is attached directly to the physical terminal, to a remote machine, or whether it is one of the terminal window of the graphical user interface. Thanks to this design, the Plan 9 windowing system `rio` could be implemented with only 8 800 LOC, including the code to emulate terminals in windows.

By comparison, Linux and X Window introduced the separate concepts of teletype `tty` devices and pseudo `pty` terminals. The code of `xterm`, which is just a terminal for X Window, not X Window itself, has already 88 000 LOC. This is partly because `xterm` carries the historical baggage of standards invented in the 70's (e.g., special control sequences for VT100 terminals).

Plan 9 contains all the essential programs used by a programmer. They form a coherent set because they were all designed from scratch by a small team of programmers.

1.4 The C language and ARM architecture

The goal of books such as *The Elements* [EucBC] or *Principia Mathematica* [WR13] is to describe the foundation of a field starting from a very small basis, for instance, a logic language with a small set of axioms and inference rules, on top of which all the rest can be derived or built. The book *The Elements of Computing Systems* [NS05] does a similar thing for computers. Starting only from the `nand` logic gate, the book builds gradually the `and`, `or`, and `not` logic gates, a multiplexer, a flip-flop, memory banks, an additioner, an arithmetic and logic unit, and finally a simple processor.

However, in Principia Softwarica we are interested in software, not hardware. Because most of the programs described in the Principia Softwarica books are written in C, our basis in some sense is the C programming language [KR88]. C is a fairly large language, with non trivial semantics, so our basis is unfortunately fairly large too. Note that one of the Principia Softwarica book describes a C compiler that targets the ARM [Sea01] architecture, so in principle our basis could be reduced to the ARM machine language, which is fairly simple. Another Principia Softwarica book describes an ARM emulator. However, both the C compiler and the ARM emulator are written in C itself, which brings us back to C as our basis.

A more elegant alternative, avoiding self-reference, would be to start from a simple machine and build a tower of increasingly powerful languages. Starting from raw binary machine code, we could gradually build more sophisticated languages through a series of *bootstrapping* steps. In fact, such a project partially exists[3], but with already six bootstrapping steps its author was still far away from a language like C. Appendix B outlines a similar (long) process to bootstrap Plan 9 from scratch.

Another alternative, chosen by Donald Knuth for his encyclopedic books *The Art of Computer Programming* [Knu73], was to pick as a basis a very simple computer he invented called MIX, and a very simple assembly language called MIXAL. Using assembly is maybe OK for describing algorithms, but I think it would not be productive for writing entire programs; this would lead to very long Principia Softwarica books.

I think that starting directly from the ARM machine, a real but fairly simple machine, and the C language, a higher level language than assembly, is maybe less elegant but more practical for Principia Softwarica.

1.5 Literate programming

I want to show source code because it is the ultimate explanation for what a program does. However, I think that showing pages and pages of listings in an appendix, as done for instance in the Minix book [Tan87], even when this appendix is preceded by documentation chapters, is not the best way to explain code. I think the code and its documentation should be mixed together, as done for instance in the Xinu book [Com84], so you do not have to switch back and forth between an appendix and multiple chapters.

Literate programming [Knu92] is a technique invented by Donald Knuth to make it easy to mix code and documentation in a document in order to better develop and better explain programs. Such documents are called *literate programs*. All Principia Softwarica programs are literate programs.

Note that literate programming is different from using API documentation generators such as javadoc[4] or

[3] http://homepage.ntlworld.com/edmund.grimley-evans/bcompiler.html

[4] http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html

doxygen[5]. Noweb[6], the literate programming tool I used, does not provide the same kind of services. Indeed, literate programming allows programmers to explain their code in the order they think the flow of their thoughts and their code would be best understood, rather than the order imposed by the compiler.

Literate programming allows, among other things, to explain the code piece by piece, with the possibility to present a high-level view first of the code, to switch between top-down and bottom-up explanations, and to separate concerns. For instance, the `Proc` data structure — which you will see in the KERNEL book [Pad14] is a data structure that represents some information about a process — is a huge structure with more than 90 fields. Many of those fields are used only for advanced features of the kernel. The C compiler imposes to define this structure in one place. Noweb allows to present this structure piece by piece, gradually, in different chapters. I can show first the code of the structure with the most important fields, and delay the exposition of other fields to advanced chapters. This greatly facilitates the understanding of the code, by not submerging you with too much details first.

In the same way, the `main()` function in most programs is rather large and mixes together many concerns: command line processing, error management, debugging output, optimizations, and usually a call to the main algorithm. Showing in one listing the whole function would hide behind noise this call to the main algorithm. The main flow of the program though is arguably the most important thing to understand first. Using literate programming, I can show code where the most important parts are highlighted, and where other concerns are hidden and presented later.

In fact, I spent lots of time during the writing of the Principia Softwarica books in transforming the Plan 9 programs in literate programs, and in reorganizing again and again the Plan 9 code to find the best way, the best order, the best separation of concerns in which I think you would more easily understand the code.

See Appendix C for a concrete example of literate program.

1.6 Getting started

To play with the different programs described in the Principia Softwarica books, I recommend to use my fork of Plan 9. See `https://www.principia-softwarica.org/getting-started.html`. Section 2.1 will explain later the directory structure of this Plan 9 repository.

To compile and install my fork of Plan 9, you will need a machine with a C compiler, a UNIX-like operating system, and a kernel that can write on a VFAT filesystem. So, GNU/Linux and macOS are possible host operating systems. Plan 9 is also a valid host as you can build Plan 9 under Plan 9.

The installation consists first in downloading my fork of Plan 9. Because the source code of Plan 9 uses a special dialect of C and some non-standard assembly, you can not use directly popular C compilers and assemblers such as `gcc` and `gas`. To compile Plan 9, you will also need to download my fork of Ken Thompson's C (cross) compilers called `kencc`. You can compile `kencc` using a regular C compiler (e.g., `gcc`) from your host operating system (e.g., from Linux or macOS). Then you can compile the Plan 9 kernel, the Plan 9 C standard library, and then the whole Plan 9 operating system with all its libraries and programs, by using the `kencc` compiler installed in the previous step. This will build from scratch a Plan 9 distribution.

Finally, you can run this Plan 9 distribution either under Qemu[7], which makes it easy to experiment, or by installing the distribution on a physical machine such as a Raspberry Pi.

1.7 Requirements

The Principia Softwarica books are not introductions to programming, computer science, or to any subfields of computer science. For instance, the book on the kernel is not an introduction to operating systems. Indeed,

[5] `http://www.stack.nl/~dimitri/doxygen/`
[6] `http://www.cs.tufts.edu/~nr/noweb/`
[7] `http://www.qemu.org`

I will assume you have already a rough idea of how a kernel works and so that you are already familiar with concepts such as virtual memory, critical regions, interrupts, or system calls. I will present with full details the source code of different programs, but I assume you already know most of the concepts, theories, and algorithms behind those programs. The Principia Softwarica books are there to cover the practice. I assume the readers of Principia Softwarica are students in computer science or programmers who desire to consolidate their knowledge by reading the ultimate computer science explanations: source code.

As said earlier, because most of the books are made of C source code, you will need a good knowledge of the C programming language [KR88]. Because most of the programs I describe are Plan 9 programs, and because Plan 9 has a lot in common with its ancestor UNIX, you will also need to be familiar with those systems. I recommend to read [KP84] for UNIX, and to read the Plan 9 tutorials [PPD+95, PPT+93] available in my Plan 9 repository.

1.8 Copyright

1.9 Acknowledgments

I would like to acknowledge of course Plan 9's authors who wrote in some sense most of the content of the Principia Softwarica books: Ken Thompson, Rob Pike, Dave Presotto, Phil Winterbottom, Russ Cox, and many other people from Bell Labs. I would like also to thank Xavier Leroy, Fabrice Le Fessant, Francoix Rouaix, and many other people from INRIA Rocquencourt who wrote the remaining programs I describe in Principia Softwarica.

Finally, I would like to thanks Donald Knuth for his early encouragments. It meant everything.

Chapter 2

Overview

Before jumping in the description of the different Principia Softwarica programs in the next chapter, I first give here an overview of how the code is organized, and how the different programs depend on each other.

2.1 Code organization

Table 2.1 presents short descriptions of the main directories in my fork of Plan 9, as well as the corresponding sections in this document in which the program associated with the directory is discussed.

A few Plan 9 programs have some architecture specific parts, with support for x86 and ARM in my fork of Plan 9. The LOC column in Table 2.1 accounts only for the code to support one of the architecture: the ARM. This is the only architecture-specific code I will show in the Principia Softwarica books. In the same way, even if some directories have a plural form, e.g., `editors/`, the LOC column accounts only for one variant of this program category, e.g., one of the editor, the one I chose to present in Principia Softwarica.

2.2 Software architecture

Many of the Principia Softwarica programs are mutually dependent on each other. Indeed, to run a compiler or an editor you need a kernel (and a shell), but to create this kernel in the first place you need an editor and a compiler. In a similar way the C compiler uses code from the core C library, but to create this library you need a C compiler. In fact, the C compiler is written in C itself, so there are even self-dependencies[1]

It is possible though to *layer* things by looking at how things are organized in memory. A first separation to make is between code running in *kernel space* and code running in *user space*, as shown in the left of Figure 2.1. Most of the code running in kernel space is in `kernel/`, as well as some code in `lib_graphics/` and `lib_networking/`. Some of the code in the C library (the memory pool library [2] and a few utility functions and globals) are used both in the kernel and in user programs, but they are the exceptions. The rest of the codebase runs in user space.

The boundary between user programs and the kernel is provided by the *system calls* application programming interface (API). The functions of this API are declared in `include/core/libc.h` (as well as many utility functions). The user-space part of the system calls are implemented in `lib_core/libc/9syscall/`. This directory contains one assembly file per system call and each of those files contains mostly the software interrupt instruction with a special value in one of the argument in order to be dispatched to the appropriate code in the kernel. The dispatcher kernel code is in `kernel/syscalls/`. Some of those system calls are process related (e.g., `rfork()`, `exec()`, `exit()` [3]), some are memory related (e.g., `brk()` [4]), and other are used for file input

[1] Appendix B describes how to solve those mutual and self-dependencies issues.
[2] `lib_core/libc/port/pool.c`
[3] with implementation in `kernel/processes/`
[4] with implementation in `kernel/memory`

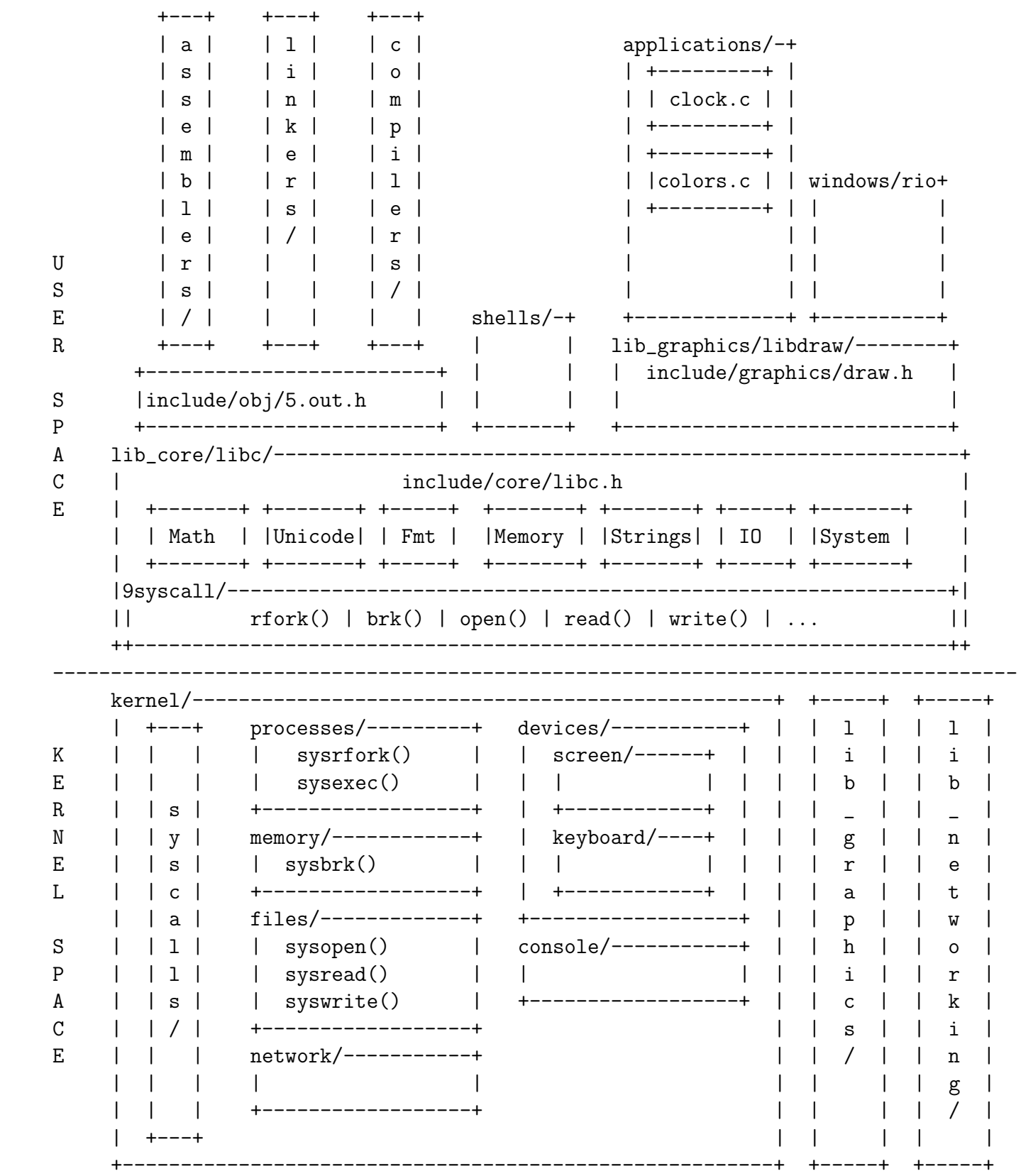


Figure 2.1: Software architecture of Plan 9.

Directory	Description	Section	LOC
`kernel/`	The Plan 9 basic kernel (for ARM and x86)	3.1.1	60 000
	+ graphics stack (kernel code)	3.4.1	10 000
	+ network stack (kernel code)	3.5.1	23 000
`include/`	The header files (e.g., `libc.h`)	3.1.2	5 500
`lib_core/`	The core C library (for ARM and x86)	3.1.2	21 500
`shells/`	The shell	3.1.3	7 500
`compilers/`	The C compiler (for ARM and x86)	3.2.1	25 000
`assemblers/`	The ARM and x86 assemblers	3.2.2	4 100
`linkers/`	The ARM and x86 linkers	3.2.3	7 900
`machine/`	The ARM emulator	3.2.4	4 400
`languages/`	The OCaml bytecode compiler/interpreter	3.2.5	30 000
`generators/`	The code generators Lex and Yacc	3.2.6	3 500
`editors/`	The editor	3.3.1	10 000
`builders/`	The build system	3.3.2	4 800
`debuggers/`	The debuggers and tracers	3.3.4	17 000
`profilers/`	The profilers	3.3.5	4 900
`lib_graphics/`	A large part of the graphics stack	3.4.1	16 000
`windows/`	The windowing system	3.4.2	10 500
`lib_networking/`	A small part of the network stack	3.5.1	1 000
`networking/`	Networking applications (clients and servers)	3.5.1	51 000
`browsers/`	The web browser	3.5.2	24 000
`utilities/`	Utilities such as `ls`, `cp`, `mv` `grep`, `gzip`, `tar`, `sed`, `diff`, `xargs`, `ps`, etc	3.6	16 000
Total			333 600

Table 2.1: Main source code directories of my Plan 9 repository.

and output (IO) (e.g., `open()`, `close()`, `read()`, `write()` [5]).

In fact, those IO system calls provide an extended way for programs to request services from the kernel by using the filesystem hierarchy. Indeed, one of the philosophy of UNIX is that *everything is a file*, including the devices. A process using the `/dev/cons` *device file* will trigger code in the kernel in `kernel/console/devcons.c`; this code will then trigger code that handles the keyboard device in `kernel/devices/keyboard/` (when reading from `/dev/cons`), or the screen device in `kernel/devices/screen/` (when writing to `/dev/cons`).

This philosophy was pushed even further under Plan 9 where *everything is a filesystem*. The graphics API for instance is accessible though many files under `/dev/draw/`, and triggers code in `kernel/devices/screen/devdraw.c` and `lib_graphics/`. In a similar way, the networking API is accessible through files under `/net/` and triggers code in `kernel/network/`.

A second separation to be made is between *library code* and *application code*. All the user programs in Plan 9 rely first on the core C library in `lib_core/libc/`, exposed via the `include/core/libc.h` header file; all programs are linked with `libc.a`. The other `lib_xxx/` directories contain other general-purpose functions and data structures, which are used by different programs. By using libraries, source code can be reused more easily.

The shell is a regular program using also the C library. The assembler, linker, and compiler rely on the C

[5] with implementation in `kernel/files/`

library as well as other header files, for instance, `include/obj/5.out.h`, which declares the set of ARM opcodes.

All the graphical applications, for instance, the clock, but also the windowing system, rely also on the `lib_graphics/lib_draw/` library, which is exposed in the `include/graphics/draw.h` header file. The graphics library is essentially a thin wrapper over the protocol used by the `/dev/draw/` device files. In the same way, the `lib_networking/libip/` library, exposed in the `include/net/ip.h` header file, is a thin wrapper over the protocol used by the `/net/` device files.

Chapter 3

The Literate Programs

I now switch to quick descriptions of the 20 programs, and so 20 books, composing the Principia Softwarica series. I organized those programs in six different groups.

3.1 The Core system

The first group, which I call the *core system*, is made of the minimal set of programs that are needed to reach the point where the programmer can interactively launch other programs. In the case of Plan 9, this minimal set is made of the kernel, the shell, and the standard C library.

The C library is necessary because it is used by the shell. Indeed, the C library provides the necessary bridge to call the kernel from user-space programs. The library provides also memory allocation routines and a few general utility functions. In fact, a small part of the C library is also used internally by the kernel.

3.1.1 The kernel: `9pi`

A *kernel* is a program that manages all the other programs. It is arguably the most important program; without the kernel no other program can run. The kernel provides the main abstractions of the computer and multiplex its resources (e.g., the processor, the memory, the input and output devices) to multiple programs at the same time.

The kernel is the biggest program and so the biggest book in the series. The Plan 9 kernel is called simply `9`, but I will present a variant of the kernel for the Raspberry Pi called `9pi`.

3.1.2 The core library: `libc`

The C library defines data structures and functions that are used by all the other programs (including the kernel). In the case of Plan 9, this library contains the memory allocation routines (`malloc()`, `free()`), the bridge to call the kernel (the system calls), the Unicode functions, and many other utility functions. Note that some of the code in the C library is written in assembly.

3.1.3 The shell: `rc`

A *shell* is a program that allows a user to launch other programs. It is the primary user interface of Plan 9, the so called *command-line interface* (we will see later another interface).

In Plan 9, and also in its ancestor UNIX, the shell is a regular user-space program; it is not part of the kernel. The user can actually change shell without changing the kernel.

The shell will be the first user program I will describe in the Principia Softwarica series. It illustrates many features provided by the kernel, because the shell is using internally many system calls (e.g., `rfork()`, `exec()`, `chdir()`, `pipe()`).

The Plan 9 shell is called `rc` (for `r`un `c`ommand).

3.2 The development toolchain

Once you have a terminal where you can launch programs, you will need tools to produce those binary programs from source code. The *development toolchain* is a set of tools working together to produce executables. This toolchain is made essentially of a compiler, an assembler, and a linker.

3.2.1 The C compiler: `5c`

A *compiler* is a program that transforms source code written in one high-level language (e.g., C), into another lower-level language (e.g., assembly). The C compiler is probably the most important program in Plan 9 after the kernel. Indeed, the Plan 9 kernel itself is written in C and so needs a C compiler to become executable by a machine. In fact, almost all programs in Principia Softwarica are written in C, including the compiler itself[1], so they all need the C compiler to become executables.

I will describe `5c`, a C compiler targeting ARM assembly. The `5` in `5c` comes from the Plan 9 convention to name architectures with a number or single letter (0 is MIPS, 5 is ARM, 8 is x86, etc). This is why the ARM assembler and linker I will describe later are called respectively `5a` and `5l`.

I chose the ARM architecture because it is one of the simplest architecture while being also one of the most used architecture in the world. Indeed, almost every phone contains an ARM processor. It is also the processor of the extremely cheap Raspberry Pi a machine used by many electronic hobbyists. Thus, ARM is a great candidate for my teaching purpose.

3.2.2 The assembler: `5a`

An *assembler* is a kind of compiler; it translates source code written in an assembly programming language into machine code, or into an object code close to machine code. Some of the kernel code, as well as code in the C library, are written in assembly.

To be consistent with the compiler, I will describe `5a`, the Plan 9 ARM assembler.

3.2.3 The linker: `5l`

A *linker* is a program taking as input multiple files called the *object files*, which contain object code generated either by the assembler or compiler, and creates the final executable. In theory, compiling, assembling, and linking could be done by a single program; the C compiler could take as input multiple C source files and produce directly an executable. However, from a software engineering point of view, it is better to separate concerns and have three separate tools for those three separate tasks.

I will describe `5l`, the Plan 9 ARM linker.

3.2.4 The processor emulator: `5i`

An *emulator* is a program that acts like a computer and can interpret another (binary) program. An emulator is not really a part of the development toolchain. However, because the assembler, linker, and compiler target an architecture, it is useful to describe the instruction set of an architecture (ISA) through its emulator. Moreover, an emulator can be extremely useful when transforming a compiler in a *cross compiler*, to test quickly the generated machine code.

I will describe `5i`, an ARM emulator, which we can view as a kind of interpreter for a low-level language, hence the `i`. In fact, `5i` can also act as a (slow) debugger and profiler.

[1] Appendix B describes how to solve this self-reference issue.

3.2.5 The OCaml compiler: `ocamlc` and `ocamlrun`

C is a great programming language, and a real improvement over assembly. It is arguably the best language to implement system programs such as kernels, virtual machines, just-in-time compilers, etc. C can also be used as a portable assembler; this makes C a great language to implement efficiently core libraries, which can even be used from programs written in different programming languages. For many applications though, especially applications without strong constraints on memory, speed, or latency, it can be far more productive for the programmer to use higher-level languages. Programming in C is indeed error-prone, with recurring bugs such as buffer overflows, segmentation faults, or security holes.

This is why a few programs in the Principia Softwarica series are written in the OCaml[2] programming language. OCaml is a statically-typed functional language, more expressive and less error-prone than C. It is almost impossible to have many of the bugs mentioned above while programming in OCaml. Just like I prefer, when possible, to describe programs written in C rather than assembly, because the resulting code is smaller and easier to understand, I also prefer, when possible, to describe programs written in OCaml rather than C, because the resulting code is also smaller and easier to understand.

For completeness, I need then to present also the code of an OCaml compiler or interpreter. Interestingly, OCaml uses a mixed approach. Indeed, the OCaml system includes first a compiler, `ocamlc`, written in OCaml itself, generating *bytecode* for a *virtual machine*, instead of object code for a real machine. Then, a bytecode interpreter, `ocamlrun`, written in C, implements this virtual machine and so can interpret the bytecode generated by the compiler. Any machine with a C compiler can then by transitivity execute OCaml (byte)code.

3.2.6 The Lex and Yacc code generators: `olex` and `oyacc`

Code generators are programs that generate automatically source code. Compilers are a kind of code generators as they generate assembly code. I will describe two more generators in Principia Softwarica: Lex and Yacc. They provide both a *domain specific language* (DSL) to help respectively scan and parse languages. They are used by a few other programs in Principia Softwarica: the assembler, the C compiler, the shell, and the build system. This is why I consider Lex and Yacc essential programs. The full understanding of the C compiler, for instance, would be incomplete if you could not understand the code generated from the C grammar specification by Yacc.

There are many clones of Lex and Yacc, written in many different languages. The `lex` and `yacc` programs included in Plan 9 are actually not really Plan 9 programs; they are the original `lex` and `yacc`, written in C, copied without much modification directly from UNIX. Because they were written a long time ago, their C source code is not as clear and modern as the code of the other Plan 9 programs. This is why I decided to present instead `olex` and `oyacc`, some Lex and Yacc clones written in OCaml.

3.3 The developer tools

The *developer tools* are a set of tools that are not strictly necessary to produce programs, like the software development toolchain I have described in the previous section, but which are useful in the software development process. Those tools are the text editor, the build system, the debugger, and the profiler.

3.3.1 The text editor: `efuns`

A *text editor* is a program to help read and write programs. It is perhaps the most important tool for a programmer. Indeed, even if the kernel and compiler are essentials to run and produce software, their source

[2] http://ocaml.org/

code was first entered in a computer by a programmer using a text editor. In fact, text editors are used to enter the code of the text editors themselves, leading to self-refence issues just like for a compiler[3].

The text editor is certainly also the program to which programmers are the most emotionally attached to[4]. In fact, this is the main reason I chose to not present a Plan 9 program, even though Plan 9 has a few text editors (`ed`, `sam`, and `acme`). Instead I will present an Emacs clone. Another deviation is that this program is not written in C but in OCaml. This clone is called `efuns`[5].

The text editor will be the first graphical program in the Principia Softwarica series. Indeed, until now all the programs I described were command-line programs. The graphics stack used by `efuns` will be described later.

3.3.2 The build system: `mk`

A *build system* is a program to help automate the compilation of programs by calling the appropriate tools from the development toolchain. The whole Plan 9 operating system can be built from scratch with one simple command, `mk all`, thanks to the build system called `mk` and a few configuration files (the `mkfile`s).

3.3.3 The version control system: `ogit`

3.3.4 The debugger: `db`

A *debugger* is a program that commands and inspects another program. Programming is so difficult that invariably we make mistakes when writing code. Having tools to help find those mistakes is essential, and the debugger is the most important of those tools. Even if many programmers, including great programmers [Sei09], use just `printf()` tracing instructions to debug their programs, I think a debugger can greatly accelerate the time it takes to find bugs. Debuggers are also great tools to help understand programs, especially programs written by other people.

The Plan 9 debugger is called simply `db`.

3.3.5 The profiler: `prof`

A *profiler* is a program that generates statistics about another program. It is mainly used to optimize code by spotting where the program spends most of its time. It can be useful also to find bugs because unexpected statistics can sometimes be good hints to fix code where the programmer was expecting different results.

3.3.6 The typesetting system: `troff`

3.4 Graphics

Up until now, I have mostly described command-line tools interacting with the programmer through simple text-based terminals. However, one of the most important inventions in computer science is the *graphical user interface* (GUI), introduced in the 1970's with the Xerox Alto [TML+79]. Even command-line text-based programs benefit from a graphical user interface as you can run multiple programs in different windows at the same time like in Figure 1.1 on page 9.

[3] See Appendix B for discussions on how to bootstrap an editor.

[4] https://en.wikipedia.org/wiki/Editor_war

[5] https://github.com/aryx/fork-efuns

3.4.1 The graphics stack: `draw`

A *graphics stack* provides the basis on top of which graphical applications can be built. GUI elements such as menus, windows, cursors, or texts are ultimately rendered on the screen using simple graphic operations provided by the graphics stack: lines, circles, rectangles, arcs, etc.

Under Plan 9, the graphics services are accessible through the `/dev/draw/` device directory, which is connected to the screen device driver in the kernel.

3.4.2 The windowing system: `rio`

One of the most important graphical applications is the *windowing system*. In some sense, a windowing system is an extension of the kernel and the shell; it is a program that also manages and launches other programs, which are represented visually by separate windows.

The Plan 9 windowing system is called `rio`. It is essentially a multiplexer of the `/dev/cons`, `/dev/mouse`, and `/dev/draw` device files. `rio` is a file server that internally uses the keyboard, mouse, and screen physical devices, and provides a virtual keyboard, virtual mouse, and virtual screen to its windows using views of those same device files (thanks to the per-process namespace feature in the kernel). In fact, an unusual feature of `rio` is that it can be run inside itself.

3.4.3 The graphical user interface toolkit: `libpanel`

3.5 Networking

Up until now, I have described programs that can run on isolated machines. I will now switch to programs that can communicate with each other on different machines.

Just like the graphical user interface, *networking* (and internetworking, also known as the Internet) has been one of the most important inventions in computer science. It has led ultimately to the creation of the Web where networking programs (e.g., web browsers and servers) made possible great things, for instance, the worldwide collaborative project Wikipedia.

3.5.1 The network stack: `net`

A *network stack* is a part of the kernel, usually fairly large, that provides the necessary abstractions for programs to communicate with each other on different machines, by using different protocols.

Under Plan 9, the network services are accessible through the `/net/` device directory.

3.5.2 The web browser: `mmm`

There are too many networking protocols and too many networking programs (clients and servers) to present. UUCP, Email, News, FTP, Telnet, IRC, or Gopher are all useful networking programs. However, I decided to focus on what I think is the most important one: the *Web browser*. You can actually use Email, News, FTP, and many other things by using just a Web browser. It has become so versatile that it can be considered almost an operating system on its own.

One of the reason Plan 9 did not get a wider adoption was that it did not include a complete Web browser. A few prototypes were written, but they were limited. Moreover, because Plan 9 is not fully compatible with UNIX, it was difficult to port Web browsers such as Firefox to Plan 9.

In Principia Softwarica, I will describe `mmm` [6], a Web browser written in OCaml. It is also more limited than popular Web browsers such as Firefox or Chrome, but it can handle popular websites such as Wikipedia. More importantly, its codebase is far easier to understand and present than the codebases of popular browsers.

[6] http://pauillac.inria.fr/mmm/

3.6 Utilities: `cat`, `grep`, ...

In addition to the programs I mentioned previously, a programmer very often uses small utilities to perform or automate certain tasks. The code of those utilities is usually pretty small because those utilities have often a single and simple function. One of the Principia Softwarica books will be dedicated to describe the code of a few of those utilities, the most important ones for the programmer.

File and directory utilities

The shell has very few *builtins* (e.g., `cd`); to create, delete, or modify files or directories, a programmer needs to use special programs. Those programs are essentially small wrappers around the file and directory system calls provided by the kernel.

Here are the file and directory utilities I will describe:

- `touch` and `mkdir`
- `cat` and `ls`
- `rm`, `cp`, and `mv`
- `chmod` and `chgrp`

Text processing utilities

Source code is stored in text files, which are made of a set of lines, which are a set of strings (words). It is thus normal that string-processing utilities are very often used by programmers to search, modify, or compare source code. Here are the string-processing utilities I will describe:

- `grep`: It is one of the most versatile and useful tool for a programmer. It can be used to find code using regular expressions.
- `sed`: It can be used to help refactor code.
- `diff`: It can be used to compare different versions of the same codebase.

Process utilities

Source code ultimately is transformed into binary programs, which ultimately become processes when run. A programmer needs a set of utilities to manipulate those processes. Under Plan 9, the `/proc` directory can be used to inspect and manipulate those processes. An alternative is to use command-line programs that are often small wrappers over `/proc`.

Here are a few process utilities I will describe:

- `ps` and `pstree`
- `kill`

Archive utilities

Once a program has been written, a programmer often wants to share it with the world. In this case, he can count on a few utilities to *package* and *compress* code.

Here are a few utilities used to help disseminate programs:

- `tar`
- `gzip`

3.7 Applications

I decided to limit Principia Softwarica to system programs, and to the system programs that are the most relevant to the programmer. I do not cover applications such as spreadsheets, word processors, calendars, email clients, or video games. However, I encourage other people to find applications with small codebases (using minimalist approaches), and to write and publish their literate programs.

Chapter 4

Conclusion

I hope the Principia Softwarica books will greatly consolidate your computer science knowledge, and give you a better and more complete picture of what is going on in your computer.

I think the Principia Softwarica programs form together the minimal foundation on top of which all applications can be built. Even though there are 20 books in the series, I still think it is the minimal foundation. Indeed, it is hard to remove any of those programs because they depend on each other. First, you need to rely on a kernel (hence the name), but the shell and the C library are also essential. However, because those programs are coded in C and assembly, you also need a C compiler and an assembler, and because source code is usually split in many files you also need a linker. The C compiler itself usually uses DSLs like Lex and Yacc. To write all this code in the first place you need an editor. Then, with so many source files you need a build system to automate and optimize the compilation process. Because the programs I just mentioned inevitably have bugs or non optimal parts, you will need a debugger and a profiler. Finally, nowadays it is inconceivable to not use a graphical user interface and to not work with multiple windows opened at the same time. In the same way it is also not conceivable to work in isolation; programmers collaborate with each other, especially via the Web. This means you need a graphical and networking stack as well as a windowing system and a Web browser. As I said earlier, it is hard to remove any of the programs from the series.

I hope those books will answer many of your questions, even those that seem very simple at first such as "What happens when the user type `ls` in a terminal window?". The answer to this question involves many software layers (the shell, the C library, the kernel, the graphics stack, and the windowing system) and lots of code.

The books in the series can be read mostly in any order. You do not have to read them all. I recommend to pick the program you are the most interested in, for instance, the one you are the most curious about because you have only a vague idea of how they are implemented, and read the corresponding book in the series.

Enjoy!

Appendix A

Other Teaching Operating Systems

Here are a few teaching operating systems that I considered for Principia Softwarica, but which I ultimately discarded:

- UNIX V6[1] (Ken Thompson et al.), fully commented in the classic book by John Lions [Lio77], or its modern incarnation xv6[2], are great resources to fully understand a UNIX kernel. However, this kernel is too simple; there is no support for graphics or networking for instance.
- XINU[3] (Douglas Comer), fully documented in two books [Com84, Com87], has a network stack, but the kernel is still too simple with no virtual memory for instance.
- Minix[4] (Andrew Tannenbaum et al.), also fully documented [Tan87], is fairly small, but it is just a kernel. Minix does not provide for instance its own windowing system; it relies instead on X Window, which is far more complicated than the Plan 9 windowing system.
- Hack[5] (Noam Nisan and Shimon Shocken) is a toy computer introduced in the excellent book *The Elements of Computing Systems* [NS05]. This book is great for understanding processors, assemblers, and even compilers, but the kernel part is really too simple.
- MMIX[6] (Donald Knuth) and its ancestor MIX are computers designed by Donald Knuth and used in his classic book series *The Art of Computer Programming* [Knu73]. Donald Knuth also wrote a book using literate programming, *MMIXware* [Knu99], to explain the full code of the MMIX simulator and assembler. However, similar to Hack, very few programs have been written for this machine. For instance, the book assumes the presence of a kernel called NNIX, but nobody has ever written it.
- STEPS[7] (Alan Kay et al.) is a project to reinvent from scratch programming. It has a far more ambitious goal than Principia Softwarica: write a full operating system in 20 000 LOC. It was unfortunately never finished.
- Oberon[8] (Niklaus Wirth et al.) is a kernel, compiler, and windowing system designed from scratch. It is a great operating system, very compact, and fully documented in a book [WG92]. However, it imposes strong restrictions on the programmer: only applications written in the Oberon programming language can be run. This simplifies many things, but operating systems like UNIX (and Plan 9) are more universal;

[1] http://minnie.tuhs.org/cgi-bin/utree.pl
[2] http://pdos.csail.mit.edu/6.828/2014/xv6.html
[3] http://minnie.tuhs.org/cgi-bin/utree.pl?file=Xinu7
[4] http://minnie.tuhs.org/cgi-bin/utree.pl?file=Minix1.1
[5] http://www.nand2tetris.org/
[6] http://www-cs-faculty.stanford.edu/~uno/mmix-news.html
[7] http://vpri.org/html/writings.php
[8] http://www.projectoberon.com/

they can run any program in any language, as long as the program can be interpreted or compiled into a binary.

- TempleOS[9] (Terry A. Davis) is an operating system single handedly created over a decade. It contains a kernel, a windowing system, a compiler for a dialect of C, and even some games. It has graphics capabilities but there is no network support.

Another candidate for Principia Softwarica was the combination of the GNU system[10], the Linux kernel[11], and the X Window graphical user interface Xorg[12]. However, GNU/Linux/Xorg together are far bigger than Plan 9. If you take the source code of the Linux kernel, the GNU C library (`glibc`), the `bash` shell, the GNU C compiler (`gcc`), the GNU assembler (`gas`) and linker (`ld`) part of the `binutils` package, the GNU Lex and Yacc clones (`flex` and `bison`), the Emacs editor, GNU `make`, the GNU debugger (`gdb`), the GNU profiler (`gprof`), and the X Window system (`Xorg`), you will get orders of magnitude more source code than Plan 9, even though Plan 9 provides in essence the same core services. In fact, almost all of the programs above use individually more source code than the whole Plan 9 system.

Of course, the Linux kernel contains thousands of specific device drivers, `gcc` handles a multitude of different architectures, and `Xorg` supports lots of graphic cards. All of those things could be discarded when presenting the core of those programs. However, their core is still far bigger than the equivalent core in Plan 9 programs.

[9] http://www.templeos.org/
[10] http://www.gnu.org/
[11] http://www.kernel.org/
[12] http://www.freedesktop.org

Appendix B

Bootstrapping from Scratch

It is currently fairly easy to install Plan 9 on a new machine, as explained in Section 1.6. This is because there are already in the world computers running operating systems that include executable C compilers and text editors, with standard formats for data as well as compatible storage devices. This is also because the source code of Plan 9 can easily be downloaded on the storage device of those computers through the network. All of this makes it easy to compile (or cross compile) Plan 9 on one machine and install it on the disk of a new machine that can be booted on. In fact, you can even use an emulator like Qemu and run Plan 9 from the same machine.

However, what if we had to start from scratch?

- What if there was no executable C compiler? As mentioned in Section 3.2.1, the C compiler I describe in Principia Softwarica is itself written in C. This self-reference leads to a *chicken and egg* problem: How was compiled the code of the first compiler? This is also true for OCaml, Lex, and Yacc, which are also written in themselves.

- What if there was no digital text of Plan 9 and no text editor? If the only representation of Plan 9 was the printed Principia Softwarica books, the source code of Plan 9 would have to be entered first in a computer. But without a text editor, how to enter and save text in a computer? How was entered the source code of the first text editor?

- What if there was no kernel? How to interact then with a new machine? If most programs are loaded into memory by the kernel, which can be seen as an interactive program loader, how was loaded in memory the first loader?

In fact, the answer to all those questions is a technique called *bootstrapping*. According to Wikipedia, "bootstrapping in general refers to the starting of a self-sustaining process that is supposed to proceed without external input". In our case, bootstrapping is a process where you start from something simple and gradually build something self-sustaining and more complex.

In the book *The Knowledge: How to Rebuild our World from Scratch* [Dar14], its author Lewis Dartnell imagines our world after an apocalypse and describes the key knowledge one needs to start rebuilding civilization from scratch. He does not reach though the computer and software age. What if after this apocalypse there was no more software? What if the only programs we had was the printed Principia Softwarica books retrieved from a time capsule? Would that be enough to bootstrap Plan 9? The kernel, compiler, and editor being mutually dependent on each other, it is difficult to imagine where to start. Which program to write first? For which machine? In which language? And how to enter this program in the machine?

In the following sections, I outline a bootstrapping process for Plan 9 where I start from scratch. I think it is an interesting intellectual exercise. By scratch I mean with no existing software, but because in Principia Softwarica we are interested in software and not hardware, I assume the existence of a basic computer though.

B.1 A basic computer

Software and hardware need each other in order to be useful, but hardware is the concrete starting point. Before executing programs, we first need a physical machine, a computer.

The main components of a basic computer are as follows:

- A processor: This is an interpreter for a simple low-level language where instructions are encoded in a binary format.
- Some memory: This contains data but also the code of programs. The *stored-program* concept is one of the major inventions of computer science.
- Input and output devices: This is used for interactivity with the user.

To learn how to build a simple processor and memory, I recommend to read [NS05] in which starting only from the `nand` logic gate, you learn how to build gradually the `and`, `or`, and `not` logic gates, a multiplexer, flip-flops, memory banks, an additioner, an arithmetic and logic unit (ALU), and finally a simple central processing unit (CPU).

Regarding the external devices, a basic computer typically has three:

- An input device, e.g., a keyboard, for reading data. The press of a key can trigger an interrupt in the processor that will trigger the execution of a special program. The value of the key could be read at a certain memory location (memory-mapped IO) or via a special instruction. Before keyboards, *switches* on *panels* or *punched cards* with card readers connected to the computer were the main ways to enter data in a computer.
- An output device, e.g., a screen, for displaying data. Again the device could be memory-mapped, in which case the writing of data at certain memory locations would trigger the display of special characters on the screen. Before screens, diods on panels or printers connected to the computer were the main ways to display data from the computer.
- A storage device, e.g., a magnetic tape with its tape drive, to permanently store data, for instance programs in executable forms and source forms. Again, access to this device could be provided by special instructions or by writing and reading certain codes at certain memory locations.

All those things can be built from scratch. They are not trivial to build, but they do not require any software; they are physical artifacts.

B.2 A booting procedure

Once you have a basic computer, an important question is what happens when a human presses the On/Off button of the computer? The initialization of a computer system is called *booting*, which is the shortened form of the word bootstrapping. The machine needs to initialize itself and execute the first program from memory. If most programs are loaded into memory by other programs already running on the computer, what is the mechanism to load the very first program?

There are multiple ways to load an initial program in memory. The very first computers used switches on panels where the initial configuration of memory could be manually toggled, or at least part of the memory could be toggled. Then the computer when turned on, by construction, would start to execute the code located at a specific memory address, e.g., `0x10`. This initial program, usually small, is called the *boot loader*. Its job is just to load a larger program stored on the main storage device: the kernel (which itself will load other programs).

Later, computers used punched cards and a card reader connected to the computer to load automatically their first program in memory. Modern computers use special read-only memory (ROM) chips to store the initial program. Such programs are also called *firmware*, because they are in the middle between the hard and soft.

In any case, all those techniques are similar: a program is hardwired at a persistent location (panel, punched card, tape, ROM) and loaded in memory at a specific address by the computer when the human presses a special button. The computer then jumps to this memory address and starts executing the program.

B.3 Physical programs

We have just seen different ways to enter programs in a computer without a text editor; switches and punched cards are ways to write data, and so also programs, in the physical world. Thanks to panels and card readers connected to the computer, those physical programs are made machine readable.

Different kinds of punched cards and card readers can be used to write different kinds of information:

- Different holes for different bits can be used to represent binary data such as machine instructions.
- Different holes for different letters or numbers can be used for textual data used by programs.
- Different holes for different instructions of specific languages can be used for the source code of languages such as assembly.

Note that a card reader (or a keypunch machine) could be built in such a way to behave like a primitive mechanical assembler. The holes in the punched card could correspond to different assembly mnemonic instructions and be transmitted to the computer as the assembled binary machine instructions.

Once those physical programs are loaded into memory, they can be stored back on magnetic tapes thanks to special computer instructions.

We now have everything we need to start producing software:

- A basic computer, which can be seen as a machine-language *physical interpreter*. This will allow us to bootstrap the compiler.
- A way to enter programs in the computer, with punched cards, a keypunch machine, and a card reader, which can be seen together as a *physical text (and binary) editor*. This will allow us to bootstrap the text editor.
- A simple booting procedure which will load the boot loader from a fixed *physical location*. This will allow us to bootstrap the kernel.

The following sections describe the process to bootstrap Plan 9 from scratch by writing a series of programs. The process is decomposed in different phases depending on the programming language used.

B.4 Phase 1: Machine code

`BOOT-M-CARD`

The very first program to write is a boot loader, which I call `BOOT-M-CARD`. `M` stands for machine code and `CARD` for punched card. The first program has to be written directly in machine code and entered in the computer via a punched card as there is no assembler and no text editor yet. The bits of the instructions of this program can be encoded as holes in the punched card. This card can then be inserted in the first position of the card reader and loaded at boot time by the computer according to its booting procedure.

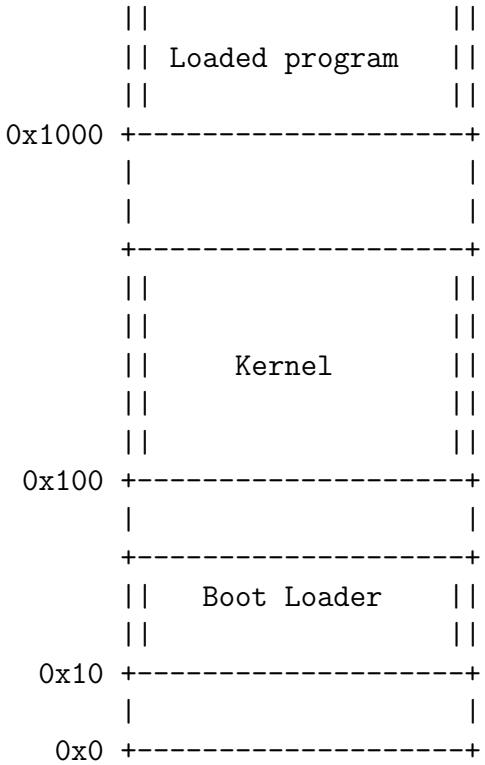
One card should be enough to encode a very simple boot loader. With just a few instructions, this program could load in memory the rest of the punched cards from the card deck (or data from the tape), and so bootstrap an hypothetical kernel. It must load this code after is own code though, to not overwrite itself, e.g., at address `0x100`, and then jump to this address.

To create `BOOT-M-CARD`, we could have created first `BOOT-ASM-PAPER`, the boot loader program written in assembly on paper. Then, thanks to a very powerful (but error-prone) computer and assembler, the human[1], we could have translated this program in binary and then generate `BOOT-M-CARD`.

KERNEL0-M-CARD

The next program to write is a simple kernel, which I call `KERNEL0-M-CARD`. It is essentially an interactive program loader. With this loader, you could type on the keyboard after a prompt the location of a program in order to get this program loaded in memory and jumped to.

The organization of different programs on the tape or in the card reader can be trivial: the different programs could be stored one after the other. The location of a program could then be specified simply by a letter and two numbers: the letter to select the card reader or tape drive, the two numbers for the start and end locations of the program on the tape or in the card deck. Moreover, by convention programs could be loaded at the memory address `0x1000` and executed from there. The last instruction of a program could be to jump to the address `0x100` to give back control to the kernel. The following diagram describes the memory layout after a program was loaded by the kernel:



There is no need yet for a real filesystem or an executable format. The code of this kernel can be very simple; it should require only a few punched cards. Those cards can then be placed just after the punched card of the boot loader in the cards deck.

ASSEMBLER0-M-CARD

The final program to write in machine code is a rudimentary assembler, which I call `ASSEMBLER0-M-CARD`. Again, we could write first `ASSEMBLER0-ASM-PAPER` and then translate this program in machine code by using a human assembler.

Because we do not have yet a text editor, this assembler when loaded could ask first a series of questions to the programmer, and then ask interactively to the programmer to enter via the keyboard the full content of

[1] Not just its brain, but also its powerful IO devices: the hands that can type on a keypunch machine.

the assembly program (without mistakes). The assembler could also take as input the content of an assembly program from cards or the tape, depending on the answer to the initial questions. Another question could be used to specify where to save the generated (assembled) machine-code program.

The punched cards of this assembler program can be put just after the cards of the kernel in the card deck. You can then count the number of cards in the deck to know the location numbers to enter in the prompt of the kernel to load this assembler program in memory.

B.5 Phase 2: Assembly

Thanks to `BOOT-M-CARD`, `KERNEL0-M-CARD`, and `ASSEMBLER0-M-CARD`, we can now enter via the keyboard programs in assembly, a major productivity improvement over punched cards and binary machine-code.

`EDITOR0-ASM0-KBD`

For improving even more productivity, the first assembly program to write is an editor, which I call `EDITOR0-ASM0-KBD`. It is written in the rudimentary assembly `ASM0` supported by the rudimentary assembler `ASSEMBLER0-M-CARD`, and bootstrapped via the keyboard input of the assembler. The assembler can then generate `EDITOR0-M-TAPE`, an executable editor, which can be loaded from tape.

Thanks to this editor, further programs can be entered via the keyboard, edited, and saved on tape. We can make mistake and fix the mistake easily. From now on I assume every programs will be entered via the text editor and saved on tape, so there is no need anymore for the `CARD`, `TAPE`, or `KBD` suffixes in the program names.

`ASSEMBLER1-ASM0`, `ASSEMBLER2-ASM1`, etc

Thanks to `ASSEMBLER0-M` (previously called `ASSEMBLER0-M-CARD`), we can now bootstrap an assembler written in itself: `ASSEMBLER1-ASM0`. In fact, we can write a series of increasingly powerful assemblers. Each assembler can add features, be assembled by the previous generation assembler, which will generate a new binary assembler, e.g., `ASSEMBLER1-M`, enabling now programmers to write assembly programs using those new features, including the assembler program itself. Hopefully at some point there will be no need for more features or optimizations and we will reach a fixpoint with an assembly language I call `ASM` from now on.

`EDITOR1-ASM`

Given this feature-rich assembly language `ASM` and assembler, we could rewrite the editor that was originally using the rudimentary assembly `ASM0`. I call `EDITOR1-ASM` this new editor. Again, we could write a series of increasingly powerful editors where each new feature added could be used to edit more quickly the code of the next version of the editor.

`KERNEL1-ASM`

We can now use a nice editor and program in a powerful assembly language. However, until this point we are still using the basic program loader `KERNEL0-M`. With this kernel, you need to enter the start and end locations on tape of a program to get it executed, which is inconvenient. Indeed, with the multitude of programs we previously created, you need to maintain a physical map of the tape to remember where the programs (in source and binary forms) are located. Moreover, the programs have to be stored contiguously on the tape. Finally, you must take care when using the editor to not create data that would overwrite another program.

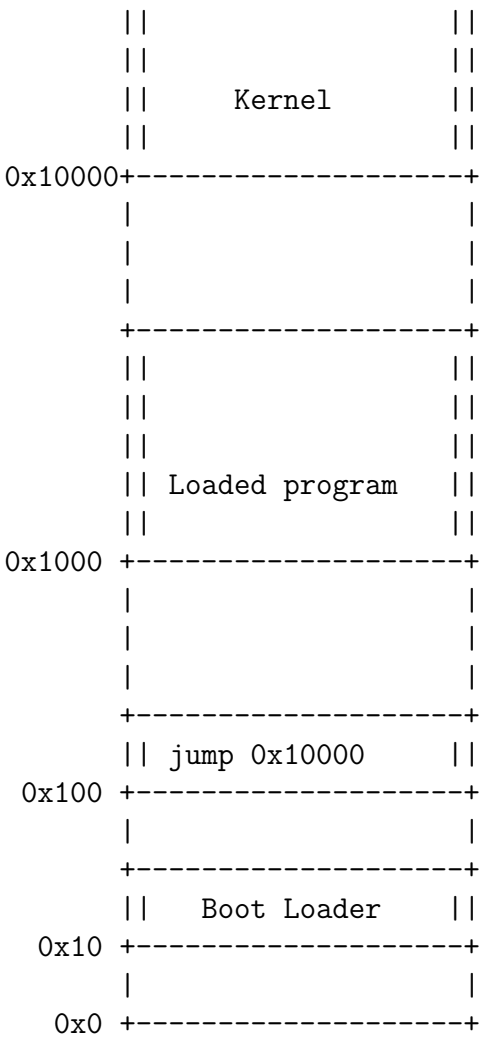
All of this management of data, currently manual and error-prone, can be done instead by the computer. I call `KERNEL1-ASM` a new kernel written in assembly with one important new feature: a *filesystem*. Transitioning to this new kernel can be facilitated by having two tape drives connected to the computer and two programs

FSINIT-ASM and CP-ASM, which can respectively initialize a new filesystem on a tape, and transfer data from one "raw" tape to another tape managed by a filesystem, and to give a filename to this data.

The start of a tape managed by a filesystem can still be reserved to contain the boot loader and the kernel. By preparing carefully the second tape with the boot loader, the new kernel, the filesystem with the previous programs (assembler, editor, kernel) in source and executable forms appropriately named, you could then switch the first tape with the second and boot the new kernel.

The interactive program loader in KERNEL1-ASM would now take after its prompt symbolic filenames instead of tape locations to execute commands. Note though that the copied text editor and assembler programs (EDITOR1-M and ASSEMBLER2-M) would still use tape locations to operate so you must take care when executing those commands to not corrupt the filesystem. One solution is to now use the second tape as a scratch and use CP-ASM to export and import data from this tape; the old editor and assembler could read and write data on this second tape without risking to corrupt the filesystem of the first tape.

Because the code required to manage a filesystem can be large, the kernel might not fit anymore in the original 0x100-0x1000 memory interval. After being loaded by the boot loader at 0x100, the kernel could copy itself in higher memory and just put at 0x100 a few instructions to jump back to higher memory. That way programs could still be loaded at 0x1000 without overwriting code from the kernel, as in the following diagram:



Note though that very large programs could still overwrite the kernel if their code size would reach the higher memory area where the kernel resides in. We will see later mechanisms to protect the kernel from such programs.

LINKER0-ASM, ASSEMBLER3-ASM-OBJ

Now that we have a filesystem, we can easily create many files. We can now split big programs in multiple files to separate concerns and also to factorize code among those programs in library files.

To create an executable from multiple files, we could create first a program `CAT-ASM` that would concatenate multiple files into one. We could then use the old program `ASSEMBLER2-M` on this single file (after it has been exported by `CP-ASM` on the scratch tape). An alternative is to create intermediate machine-code files, *object files*, and to create a *linker* I call `LINKER0-ASM` to produce the final executable from those object files. This program is more complicated than `CAT-ASM` but it enables *separate compilation*, which in the long term will save time.

To create those new object files we need to modify the assembler though and create `ASSEMBLER3-ASM-OBJ`. Moreover, we can use the opportunity to update also the interface of the assembler to now take filenames as arguments for input and output instead of tape locations.

LIB0-ASMs

For `ASSEMBLER3-ASM-OBJ` to work with filenames, code in the kernel related to the filesystem had to be copied and integrated in the assembler. Moreover, lots of code dealing with tapes, punched cards, and other devices had to be duplicated in different programs. This is unfortunate. Thanks to the linker we can now factorize code in one library source file and get the resulting object file linked in different programs.

I call `LIB0-ASMs` a core library written in assembly using multiple files (hence the `s` in `ASMs`). This library contains reusable code among all the previous programs. This library can provide functions to interact with devices, the filesystem, files, memory, etc. In some sense this library can also play the role of a kernel as it can abstract hardware and provide low-level functions to other programs. A system call is then simply a library call.

LINKER1-ASMs, ASSEMBLER4-ASMs, EDITOR2-ASMs

We can now reorganize the code of the previous assembly programs by splitting their code in multiple assembly files, to better separate concerns, and by removing the code that was factorized in the `LIB0-ASMs` library. I call those new programs `LINKER1-ASMs`, `ASSEMBLER4-ASMs`, and `EDITOR2-ASMs`. The interface of all those programs can also be changed to operate on filenames instead of tape locations.

KERNEL2-ASMs, LIB1-ASMs

`KERNEL1-ASM` has a convenient filesystem but its program loader is still rudimentary. You can load only one program at a time in memory (at `0x1000` by convention). You can load the editor, edit a file, quit, then load the assembler, wait it finishes, then run back the editor again, and so on. It would be nice to run the assembler command from the editor, or at least being able to switch between the editor and assembler without losing the state of the editor, or even better being able to edit files while a time-consuming assembling or linking job is running.

To do so requires a *multi-tasking* kernel where multiple programs loaded in memory at the same time called *processes* can execute concurrently. There are multiple ways to implement multi-tasking but the now dominant way is to write a *preemptive scheduler* using a *timer* interrupt and to leverage *virtual memory*. I call `KERNEL2-ASMs` a new kernel written in assembly using this technique. The first edition of UNIX was actually written in assembly[2].

Note that the timer and virtual memory requires hardware extensions to the basic computer I described in Section B.1. Thanks to virtual memory, multiple programs can still be loaded at the same (virtual) address, `0x1000`, as long as their physical addresses are different. This helps for the transition to the new kernel as old programs would still work with the new kernel. Those programs could also still call code from `LIB0-ASMs` to interact with devices, the filesystem, etc.

[2] https://github.com/c3x04/Unix-1st-Edition-jun72

The `LIB0-ASMs` library could be gradually extended to offer process related services, e.g., `fork()`, `exec()`, which would allow to program the scenario I mentioned before with the concurrent use of the editor and assembler. The program loader in the kernel could also be extended to provide advanced shell features such as pipes on the command line, job control, etc.

Protection rings are another hardware extension, often associated with virtual memory, which is very useful to have in a computer. They provide the mean to protect the kernel from regular programs and also to protect programs from each other when combined with virtual memory. In practice two rings are enough. The processor then can operate in two different modes:

- *Kernel mode*. In this mode all instructions are allowed, including the ones used to modify the virtual memory mapping, or the ones to interact with devices. Those instructions are called *privileged instructions*. Moreover, in kernel mode all memory accesses are allowed.

- *User mode*. In this mode many privileged instructions and access to certain memory area are disallowed. Such accesses cause *traps* in the kernel when those operations happens.

The only way to go from user mode to kernel mode (other than by causing a trap) is through a special instruction: the *software interrupt*. Calling this instruction is also known as performing a *system call* or *syscall*.

Thanks to protection rings and virtual memory, certain memory area can be marked as protected, e.g., the high memory area where resides the kernel, in which case programs can not mess anymore with the code of the kernel by overwriting its code. Moreover, we can gradually forbid programs to access directly devices such as the screen or the keyboard so that the kernel instead can mediate and control the use of those devices.

To transition to this safer model, we can gradually modify `LIB0-ASMs` and move routines using privileged instructions (dealing with memory, devices, the filesystem, processes, etc) to the kernel and give access to those routines via a system call (instead of a library call) to user programs. I call `LIB1-ASMs` this new library, which should be significantly smaller than `LIB0-ASMs`. Once all the privileged code has been migrated to the kernel, we can turn on the protection rings and forbid the execution of any privileged instructions in user programs.

`COMPILERC0-ASMs`

The final program to write in assembly will enable the transition to a higher-level language: C. It is a compiler written in assembly for a simple subset of C. I call this program `COMPILERC0-ASMs`.

B.6 Phase 3: C

Thanks to the previous programs, we now have a basic UNIX environment with a simple C compiler, all written in assembly. It is now time to rewrite this environment in a more succinct and readable way using C.

`COMPILERC1-C0`, `COMPILERC2-C1`, etc

Thanks to `COMPILERC0-M` we can now bootstrap a C compiler written in itself: `COMPILERC1-C0`. In fact, just like for the assembler and editor, we can write a series of increasingly powerful compilers: each compiler can add features, be compiled by the previous generation compiler generating a new binary compiler, e.g., `COMPILERC1-M`, enabling now programmers to write C programs using those new features, including the compiler program itself. Hopefully at some point there will be no need for more features and we will reach a fixpoint with a C language I call `C` from now on.

`KERNEL3-C/ASM`, `LIB2-C/ASM`

We can rewrite lots of the core library in C. Indeed, the C language being very flexible and powerful, many of the assembly code idioms can be expressed in C, and usually in a clearer and shorter way. However, some code has

to be kept in assembly, which is why I call this new library `LIB2-C/ASM`. Some low-level code, such as the system call instructions that allows to jump in the kernel, can not be expressed in C. Moreover, it can be preferable sometimes for heavily used routines to keep code written in highly-optimized assembly. Those routines though can be exposed to other programs with a C interface, e.g., in a `libc.h` header file. Those other programs can then be written entirely in C.

The kernel itself can also be rewritten in C and assembly in a program I call `KERNEL3-C/ASM`. The fourth edition of UNIX[3] was the edition where most programs, including the kernel, were rewritten in C. At the time, writing a kernel in a high-level language such a C was controversial.

`EDITOR3-C`, `LINKER2-C`, `ASSEMBLER5-C`

We can rewrite most of the previous assembly programs entirely in C: the editor (`EDITOR3-C`), the linker (`LINKER2-C`), and even the assembler can be rewritten in C (`ASSEMBLER5-C`).

`INTERPRETERBC-C`, `COMPILERML0-C`

We can now prepare the transition to an even higher-level language: OCaml. I call `COMPILERML0-C` a rudimentary compiler written in C for a subset of OCaml. Instead of generating object code, or assembly, like the C compiler does, the OCaml compiler uses a different approach. It generates instead *bytecode* instructions for a *virtual machine*. Then, a bytecode interpreter, which I call `INTERPRETERBC-C`, written in C, implements this virtual machine to run the bytecode generated by the compiler. This interpreter includes also the code for the OCaml runtime system: the *garbage collector* and the core OCaml libraries.

B.7 Phase 4: OCaml

Thanks to the previous programs we now have a nice UNIX environment with a simple OCaml compiler, all written in C (with a bit of assembly). It is now time to write or rewrite when it fits certain programs in OCaml.

`COMPILERML1-ML0`, `COMPILERML2-ML1`, etc

Thanks to `COMPILERML0-M` (and `INTERPRETERBC-M`) we can now bootstrap an OCaml compiler written in itself: `COMPILERML1-ML0`. In fact, just like for the assembler and C compiler, we can write a series of increasingly powerful OCaml compilers: each compiler can add features, compile itself generating a new binary compiler, e.g., `COMPILERML1-BC`, enabling now programmers to write OCaml programs using those new features. Hopefully at some point there will be no need for more features and we will reach a fixpoint with an OCaml language I call `ML` from now on.

`EDITOR4-ML`

Thanks to OCaml features such as garbage collection, closures, parametric polymorphism, algebraic data types, pattern matching, type inference, and exceptions, some code can be easier to write in OCaml than in C. I call `EDITOR4-ML` a new editor written in OCaml.

We could also rewrite the assembler, linker, and even C compiler in OCaml. It would make less sense though to rewrite the kernel or the core library in OCaml. C is better suited for low-level programs with strong constraints on speed, memory, and latency.

[3] http://minnie.tuhs.org/cgi-bin/utree.pl?file=V4

LEX-ML/LY, YACC-ML/LY

Algebraic data types and pattern matching are very useful OCaml features to implement algorithms working on trees such as the *abstract syntax trees* (ASTs) of programs. This makes OCaml a great language for writing compilers or more generally tools taking as input other programs. Lex and Yacc are two of those tools. They provide both domain specific languages to assist in writing respectively a scanner and a parser. We could implement Lex and Yacc using just OCaml, but because the Lex and Yacc programs require themselves a scanner and a parser, we can also use Lex and Yacc to program parts of Lex and Yacc. I call LEX-ML/LY and YACC-ML/LY a Lex and Yacc written in OCaml with a few parts (the scanner and parser) written in Lex and Yacc.

To solve the chicken and egg problem I usually wrote in the past first a compiler using a lower-level language. This compiler can then be used to bootstrap a compiler written in itself. For Lex and Yacc we can use a more direct strategy though. Indeed, because the lexical specifications and grammars of Lex and Yacc are pretty small, we can fairly easily run manually their algorithms to generate manually the resulting code: LEX0-ML and YACC0-ML. Those first versions, manually generated, do not have to be optimized. They can then be used on the original LEX-ML/LY and YACC-ML/LY to generate the optimized versions.

ASSEMBLER6-C/LY, COMPILERC3-C/LY, COMPILERML3-ML/LY

Lex and Yacc are fairly language independent; only the parts between curly braces, representing *actions*, are language dependent. It is fairly easy then to create a generic Lex and Yacc that can generate scanners and parsers for different languages, e.g. OCaml or C. Using this new Lex and Yacc we can rewrite parts of our past programs to finally get ASSEMBLER6-C/LY, COMPILERC3-C/LY, and COMPILERML3-ML/LY.

B.8 Summary of Plan 9 ancestor programs

I summarize here the list of programs that can be viewed as ancestors to programs explained in the Principia Softwarica series. Those programs would be necessary to bootstrap Plan 9 from scratch:

- Boot loader: BOOT-ASM (translated in BOOT-M-CARD by a human assembler). The Plan 9 kernel 9 described in Section 3.1.1 includes a boot loader fairly similar to BOOT-ASM.
- Kernels: KERNEL0-ASM (translated in KERNEL0-M-CARD by a human assembler), KERNEL1-ASM, KERNEL2-ASMs. This led finally to KERNEL3-C/ASM, which can be a close ancestor to 9 described in Section 3.1.1. We could derive KERNEL3-C/ASM from the source code of 9 by just removing many features.
- Core library: LIB0-ASMs, LIB1-ASMs. This led to LIB2-C/ASM, which can be a close ancestor to libc described in Section 3.1.2.
- C compilers: COMPILERC0-ASMs, COMPILERC1-C0, COMPILERC2-C1. This led finally to COMPILERC3-C/LY, which can be a close ancestor to 5c described in Section 3.2.1.
- Assemblers: ASSEMBLER0-ASM (translated in ASSEMBLER0-M-CARD by a human assembler), ASSEMBLER1-ASM0, ASSEMBLER2-ASM1, ASSEMBLER3-ASM-OBJ, ASSEMBLER4-ASMs, ASSEMBLER5-C. This led finally to ASSEMBLER6-C/ which can be a close ancestor to 5a described in Section 3.2.2.
- Linkers: LINKER0-ASM, LINKER1-ASMs, This led finally to LINKER2-C, which can be a close ancestor to 5l described in Section 3.2.3.
- OCaml compilers: COMPILERML0-C, COMPILERML1-ML0, COMPILERML2-ML1. This led finally to COMPILERML3-ML/ which can be a close ancestor to ocamlc described in Section 3.2.5. INTERPRETERBC-C can be a close ancestor to ocamlrun.

- Editors: `EDITOR0-ASM0`, `EDITOR1-ASM`, `EDITOR2-ASMs`, `EDITOR3-C`. This led finally to `EDITOR4-ML`, which can be a close ancestor to `efuns` described in Section 3.3.1.

Appendix C

Literate Program Example: `ed`

This appendix has two purposes. First, it is a worked example of a literate program—a complement to Section 1.5 of the main book, which presented literate programming in the abstract (Knuth, `noweb`, chunks, prose-and-code interleaving). Reading the source of a real program presented in literate form is the best way to see what literate programming actually buys you, and to learn the `noweb` conventions (`<<chunk name>>=`, `@`, and `{\tt{}code}` for inline code references) by example. Second, `ed` is a tutorial introduction to a real Plan 9 program: it is small enough (under 2000 lines of C) to fit in one appendix, yet old enough and famous enough that almost every modern editor descends from it in some way. Reading its source is a quick way to see how a non-trivial UNIX tool was structured in the 1970s.

C.1 Introduction

`ed`, short for "editor," is one of the oldest UNIX programs, originally written by Ken Thompson in the 1970s. It is a line editor: rather than displaying the whole file on screen like `vi`, Emacs, or Plan 9's `sam` and `acme`, `ed` works one line at a time, accepting commands from standard input and printing results to standard output. This makes it look primitive compared to modern screen editors, but it has a crucial advantage: it can be scripted. You can pipe a sequence of editing commands into `ed` from a shell script, something that is awkward or impossible with interactive editors.

Many ideas that seem to "belong" to other tools actually originated in `ed`: the `s/re/replacement/` syntax used by `sed`, the `g/re/p` command that gave `grep` its name ("global regular expression print"), and commands like `wq`, `a`, `i` that `vi` inherited directly.

While you can already create and modify files using `echo` and shell redirection (`>`, `>>`), `ed` offers far more power: searching, substituting, moving lines, and operating on ranges of lines.

For a tutorial introduction to `ed`, see Appendix 1 of "The UNIX Programming Environment" [KP84], or Kernighan's two tutorials from the Bell Labs technical reports.

C.2 Motivations: `mkenam`

Here is a real example of `ed` used as a code generator in the Plan 9 build system. The script below transforms a header file (`5.out.h`, containing assembler opcode definitions) into a C source file (`enam.c`, containing a string array of opcode names):

```
ed - ../5l/5.out.h <<'!'
v/^ A/d
1,$s/^ A/ "/
g/ .*$/s///
,s/,*$/",/
```

```
1i
char* anames[] =
{
.
$a
};
.
w enam.c
Q
!
```

To make the goal concrete, here is what a snippet of the input file `5.out.h` looks like (left) and what the output `enam.c` is supposed to look like (right):

```
5.out.h (input)                 enam.c (generated output)
---------------                 --------------------------
enum    As {                    char* anames[] =
    AXXX,                        {
    ANOP,                            "XXX",
    AAND,                            "NOP",
    AORR,                            "AND",
    AEOR,                            "ORR",
    AADD,                            "EOR",
    ASUB,                            "ADD",
    ...                              "SUB",
};                                   ...
                                };
```

The transformation: keep only lines starting with a tab followed by `A` (the opcode names), strip that prefix, wrap each name in double quotes followed by a comma, and add the `char* anames[]` array header and trailing brace.

The script uses `ed -` (quiet mode, suppressing character counts) and a shell here-document to feed commands. It first deletes lines not starting with `A` (`v/^ A/d`), then uses `s///` substitutions to reshape each remaining line into a quoted C string, and finally uses `i` and `a` to wrap the result with the array declaration. By the end of this chapter, you will understand every command in this script.

The key advantage of `ed` here is that the editing commands are plain text, easily embedded in a shell script or a `mkfile` rule. Try doing the same in Emacs or `vi`—you would need Emacs Lisp macros or complex key-sequence recordings, neither of which compose well with the shell.

Note that the Plan 9 version of `ed` is Unicode-ready, using `Rune` throughout for text processing. This adds some complexity to the code (conversions between `char` and `Rune`, separate buffers for byte I/O vs. character manipulation), but the overall structure remains simple and elegant.

C.3 Core data structures

The data structures in `ed` are radically different from modern editors. There is no linked list of lines, no rope, no gap buffer. In fact, the content of the file being edited is not even kept in memory. Instead, all text is stored in a temporary file on disk (`tfile`), and the only in-memory data structure is `zero`, an array of integer offsets into that temporary file—one offset per line. Editing operations like delete and move merely shuffle entries in `zero`; the temporary file itself only ever grows, never shrinks.

C.3.1 The backing store: `tfname`, `tfile`, and `tline`

The temporary file is `ed`'s backing store. `tfname` holds its path (e.g., `/tmp/e000042`), `tfile` is the open file descriptor, and `tline` tracks the current write offset—where the next line will be appended. Every operation that adds text (reading a file, appending user input) writes to `tfile` and records the offset in `zero`.

⟨globals ed.c 40a⟩≡ (? 0—1) 40b▷
```
char*   tfname; // temporary filename (/tmp/eXXXX)
```

⟨globals ed.c 40b⟩+≡ (? 0—1) ◁40a 40c▷
```
fdt tfile   = -1;
```

⟨globals ed.c 40c⟩+≡ (? 0—1) ◁40b 40e▷
```
// current write file offset in tfile;
int tline;
```

C.3.2 The target file: `savedfile`

`savedfile` remembers the name of the file being edited—the one specified on the command line or set by the `f` command. When you type `w`, this is the file that gets written.

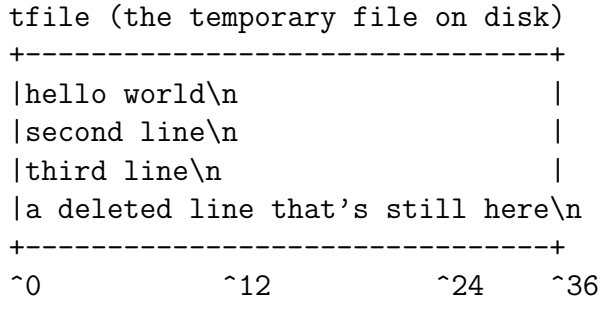
⟨constants ed.c 40d⟩≡ (? 0—1) 41d▷
```
FNSIZE  = 128,      /* file name */
```

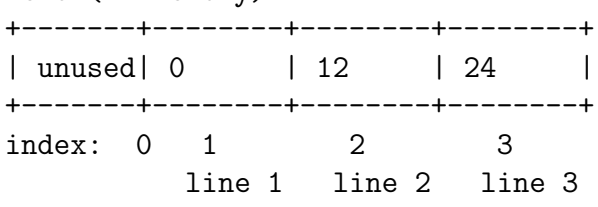
⟨globals ed.c 40e⟩+≡ (? 0—1) ◁40c 41a▷
```
// for w, r, f
char    savedfile[FNSIZE];
```

C.3.3 The lines as file offsets: `zero`

`zero` is the central data structure: a dynamically-growing array of `int` offsets into `tfile`, one per line. Lines are 1-indexed: `zero[1]` is the offset of line 1, `zero[2]` of line 2, etc. The entry `zero[0]` is unused (always 0), which is why the array is called "zero"—it starts at the zero-th slot which represents "before line 1." The crucial insight is that editing operations only modify this array of offsets, never the temporary file itself. When you delete a line with `d`, `ed` shifts entries in `zero` down but the text remains in `tfile`. When you read a file with `r`, `ed` appends text to `tfile` and inserts new offsets into `zero`. The temporary file only grows; it is never compacted.

Concretely, suppose the user has the following text in the buffer:





Three lines are visible to the user (the `$` address is 3), even though the temporary file on disk also contains a fourth line at offset 36 that was previously deleted. Editing operations shift entries within `zero`: deleting line 2 would move `zero[3]` (the offset 24) into `zero[2]`, leaving the actual text at offset 12 in `tfile` orphaned but harmless. This is why "ed never compacts the temp file"—deletes are O(1) in `zero` and zero work on disk.

⟨globals ed.c 41a⟩+≡ (? 0—1) ◁40e 41b▷
```
ulong   nlall = 128;
// growing_array<int>, initial size = (nlall+ 2+margin)*sizeof(int)
// where the ints are file offsets in tfname corresponding to different lines
int*    zero;
```

C.3.4 Cursors: `dot` and `dol`

`dot` and `dol` are pointers into the `zero` array. Their names match the ed command syntax: `dot` is the current line (what the user refers to as `.`), and `dol` is the last line (what the user refers to as `$`, "dollar"). Both are pointers, not indices: `dot - zero` gives the current line number, and `dol - zero` gives the total number of lines. When the buffer is empty, `dol == zero`.

⟨globals ed.c 41b⟩+≡ (? 0—1) ◁41a 41c▷
```
// ref<int> in zero array, current line pointer
int*    dot;
// ref<int> in zero array, last line pointer
int*    dol;
```

This is where the `1,$` address range from the `mkenam` script in Section C.2 comes from. The substitute command in that script means "from line 1 to `$` (the last line), substitute . . . "—and now we know that `$` is just the user spelling for `dol`. Similarly, the `.` in commands like `.s/old/new/` refers to `dot`.

C.3.5 Line input and output buffers: `line` and `linebuf`

There are two distinct line buffers, which can be confusing. `line` (70 bytes of `char`) is a small output buffer used by `putchr()X` to batch `write` system calls—characters accumulate in `line` and are flushed when a newline is encountered or the buffer is nearly full. `linebuf` (`LBSIZE` entries of `Rune`) is the main working buffer where a single line of edited text is held in Unicode form. Functions like `getfile()X` fill `linebuf` from the input file, `putline()X` writes it to `tfile`, and `getline()X` reads it back. The distinction between `char` for I/O and `Rune` for text manipulation runs throughout the code.

⟨globals ed.c 41c⟩+≡ (? 0—1) ◁41b 41e▷
```
char    line[70];
// ref<char> in line (mark end of line usually)
char*   linp    = line;
```

⟨constants ed.c 41d⟩+≡ (? 0—1) ◁40d 67b▷
```
LBSIZE  = 4096,     /* max line size */
```

⟨globals ed.c 41e⟩+≡ (? 0—1) ◁41c 41f▷
```
Rune    linebuf[LBSIZE];
```

C.3.6 Other globals: `count`, `col`, etc.

`count` is a general-purpose counter, most often used to track the number of characters read or written (displayed by `exfile()X` in verbose mode). `col` tracks the current output column, used by the `l` (list) command to wrap long lines.

⟨globals ed.c 41f⟩+≡ (? 0—1) ◁41e 42a▷
```
long count;
```

⟨globals ed.c 42a⟩+≡ (? 0—1) ◁41f 42b▷
```
int col;
```

C.4 main()

The `main` function initializes buffered console I/O, installs a signal handler, processes flags, then enters the main loop: `init()X`, `commands()X`, `quit()X`.

⟨globals ed.c 42b⟩+≡ (? 0—1) ◁42a 43a▷
```
// console buffered input
Biobuf  bcons;
```

⟨*function* `main`*(ed.c)* 42c⟩≡ (? 0—1)
```
void
main(int argc, char *argv[])
{
    char *p1, *p2;

    Binit(&bcons, STDIN, OREAD);
    notify(notifyf);

    ARGBEGIN {
    ⟨main()(ed.c) flags processing cases 43d⟩
    } ARGEND

    USED(argc);
    ⟨main()(ed.c) if - flag 43b⟩
    ⟨main()(ed.c) if oflag 43e⟩
    else if(*argv) {
        p1 = *argv;
        p2 = savedfile;
        while(*p2++ = *p1++)
            if(p2 >= &savedfile[sizeof(savedfile)])
                p2--;
        globp = L"r";
    }
    zero = malloc((nlall+5)*sizeof(int*)); // BUG should be sizeof(int)
    tfname = mktemp(template);

    init();
    ⟨main()(ed.c) before commands() 87a⟩
    commands();
    quit();
}
```

The `sizeof(int*)` in the `zero = malloc(...)` line is actually a bug—it should be `sizeof(int)`, since `zero` holds `int` file offsets, not pointers. The bug is harmless on architectures where pointers and ints have the same size (32-bit systems); on 64-bit systems, pointers are 8 bytes while `int` is still 4, so the allocation simply over-reserves memory without causing any incorrect behavior. Even Ken Thompson wrote slightly buggy C code.

The most interesting trick in `main` is `globp = L"r"`. When a filename argument is given (e.g., `ed foo.txt`), the filename is copied into `savedfile` and `globp` is set to the string `"r"`. Later, when `commands()X` calls `getchr()X`, it will read from `globp` before reading from the console, so the first command executed is automatically `r`—as if the user had typed `r foo.txt`. This elegant trick avoids duplicating file-reading logic in `main`.

C.4.1 `ed -` and `vflag`

By default `ed` is verbose (`vflag = true`): it prints character counts after `r` and `w` operations. The `-` flag (as in `ed - foo.txt`) suppresses this, which is what you want when `ed` is driven by a script (as in the `mkenam` example above).

⟨globals ed.c 43a⟩+≡ (? 0—1) ◁42b 43c▷
```
// verbose (a.k.a. interactive) mode
bool vflag   = true;
```

⟨main() *(ed.c) if* - *flag* 43b⟩≡ (42c)
```
if(*argv && (strcmp(*argv, "-") == ORD__EQ)) {
    argv++;
    vflag = false;
}
```

C.4.2 `ed -o` and `oflag`

The `-o` flag turns `ed` into a filter: it sets `savedfile` to `/fd/1` (Plan 9's path for stdout), so that `w` writes to standard output instead of a file. It also sets `globp = L"a"` to start in append mode immediately, and disables verbose output (since character counts would corrupt the output stream).

⟨globals ed.c 43c⟩+≡ (? 0—1) ◁43a 43f▷
```
// output to standard output (instead of editing a file). Useful
// when ed is used as a filter
bool oflag;
```

⟨main() *(ed.c) flags processing cases* 43d⟩≡ (42c)
```
case 'o':
    oflag = true;
    vflag = false;
    break;
```

⟨main() *(ed.c) if* `oflag` 43e⟩≡ (42c)
```
if(oflag) {
    p1 = "/fd/1";
    p2 = savedfile;
    while(*p2++ = *p1++)
        ;
    globp = L"a";
}
```

C.4.3 `mktemp()`

`mktemp()X` generates a unique temporary filename by replacing the trailing `X`'s in the template with the process ID digits. If the resulting file already exists, it tries replacing the first digit with `a`, `b`, etc.

⟨globals ed.c 43f⟩+≡ (? 0—1) ◁43c 44d▷
```
// in Linux pid can be very long, so better to have at least 6 X (was 5 before)
// the mkstemp man page recommends 6 X
char template[] = "/tmp/eXXXXXX";
```

⟨*function* `mktemp` *(ed.c)* 44a⟩≡ (? 0—1)
```
/// main -> <>
char*
mktemp(char *as)
{
    char *s;
    unsigned pid;
    int i;

    pid = getpid();
    s = as;
    while(*s++)
        ;
    s--;

    while(*--s == 'X') {
        *s = pid % 10 + '0';
        pid /= 10;
    }
    s++;

    i = 'a';
    while(access(as, 0) != -1) {
        if(i == 'z')
            return "/";
        *s = i++;
    }
    return as;
}
```

C.4.4 `init()`

`init()X` creates (or re-creates) the temporary file and resets `dot` and `dol` to `zero` (empty buffer). It is called both from `main` at startup and from the `e` command, which is why it closes the previous `tfile` first.

⟨*function* `init` *(ed.c)* 44b⟩≡ (? 0—1)
```
/// main | commands('e') -> <>
void
init(void)
{
    ⟨init() (ed.c) locals 75c⟩

    close(tfile);
    ⟨init() (ed.c) initializing globals 44c⟩
    if((tfile = create(tfname, ORDWR, 0600)) < 0){
        error_1(T);
        exits(nil);
    }
    dot = dol = zero;
}
```

⟨`init()` *(ed.c) initializing globals* 44c⟩≡ (44b) 71b▷
```
tline = 2;
```

`tline` starts at 2, not 0. This ensures that no valid line offset is ever 0 (used as "no line") or 1 (the low bit is stolen by the `global` and `mark` commands as a flag—see `global()X` and `k` later).

⟨`globals ed.c` 44d⟩+≡ (? 0—1) ◁43f 45a▷
```
char    T[] = "TMP";
```

C.4.5 quit()

quit()X implements the "are you sure?" safety net. If there are unsaved changes (fchange == true) and the buffer is non-empty, the first q prints ? and resets fchange, so a second q will actually exit. The Q command bypasses this by clearing fchange before calling quit()X.

⟨globals ed.c 45a⟩+≡ (? 0—1) ◁44d 45b▷
```
bool fchange;
```

⟨globals ed.c 45b⟩+≡ (? 0—1) ◁45a 46b▷
```
char    Q[] = "";
```

⟨*function* quit*(ed.c)* 45c⟩≡ (? 0—1)
```
/// main | commands('q') | ... -> <>
void
quit(void)
{
    if(vflag && fchange && dol!=zero) {
        fchange = false;
        error(Q);
    }
    remove(tfname);
    exits(nil);
}
```

C.5 Displaying and reading text

All text output in ed goes through a small buffering layer: putchr()X accumulates characters in the line buffer and flushes with a single write call when it sees a newline (or the buffer is nearly full). putst()X and putshst()X are wrappers for printing char strings and Rune strings respectively.

C.5.1 Displaying text

⟨*function* putstr*(ed.c)* 45d⟩≡ (? 0—1)
```
/// commands | getfile | error_1 | ... -> <>
void
putst(char *sp)
{
    Rune r;

    col = 0;
    for(;;) {
        sp += chartorune(&r, sp);
        if(r == 0)
            break;
        putchr(r);
    }
    putchr(L'\n');
}
```

putchr()X buffers output in line[] and flushes on newline or when the buffer is nearly full. The -5 margin in the comparison accounts for the worst case of runetochar(): a single rune can expand to up to 4 bytes in UTF-8, so the check must leave room for one full multi-byte character plus the possibility that lp already advanced past the check.

```
line[]:
+---+---+---+---+---+---+---+---+---+---+---+---+...---+---+---+---+---+---+---+
| h | e | l | l | o |   | w | o | r | l | d |   |     |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+...---+---+---+---+---+---+---+
^                                                   ^                         ^
line                                                linp                line+sizeof
                                                                        (line)-5
                                                     |<--- worst-case --->|
                                                        4-byte UTF-8 rune
                                                        + safety margin
```

`linp` tracks how full the buffer is. The flush check `lp >= &line[sizeof(line)-5]` reserves 5 bytes at the end of the buffer because: (1) UTF-8 runes can be up to 4 bytes each ("hello" μ "s" ends with a 2-byte rune that `runetochar()` writes after `lp` is checked), and (2) one extra byte for safety. Without this margin, a multi-byte rune arriving when the buffer is nearly full could overrun the array.

⟨*function* `putchr` *(ed.c)* 46a⟩≡ (? 0—1)

```
/// error | putst | ... -> <>
void
putchr(int ac)
{
    char *lp;
    int c;
    Rune rune;

    lp = linp;
    c = ac;
    ⟨putchr() if listf 78d⟩

    rune = c;
    lp += runetochar(lp, &rune);

    if(c == '\n' || lp >= &line[sizeof(line)-5]) {
        linp = line;
        write(oflag? STDERR: STDOUT, line, lp-line);
        return;
    }
    // else
    linp = lp;
}
```

Note the `oflag` check: when `-o` is active, the edited content goes to `STDOUT`, so diagnostic output from `putchr()X` is redirected to `STDERR` to avoid corrupting the data stream.

C.5.2 Reading text

Character input is layered through `getchr()X`, which checks three sources in order: (1) `peekc`, a one-character pushback used when a character needs to be "unread"; (2) `globp`, a programmed command string (e.g., `"r"` injected by `main`); (3) actual console input via `Bgetrune`. This layering is what makes the `globp` trick work—`commands()X` does not need to know whether input comes from the user or from a synthetic command.

⟨globals ed.c 46b⟩+≡ (? 0—1) ◁45b 47a▷

```
//option<char> (0 = None)
int lastc;
//option<char> (0 = None)
int peekc;
```

⟨globals ed.c 47a⟩+≡ (? 0—1) ◁46b 48c▷
```
// optional programmed command (e.g., "r", "a")
//option<Rune> or list<Rune> when used from global() ?
Rune*   globp;
```

⟨*function* getchr*(ed.c)* 47b⟩≡ (? 0—1)
```
int
getchr(void)
{
    if(lastc = peekc) {
        peekc = 0;
        return lastc;
    }
    // else
    if(globp) {
        if((lastc=*globp++) != 0)
            return lastc;
        // else
        globp = nil;
        return EOF;
    }
    // else
    lastc = Bgetrune(&bcons);
    return lastc;
}
```

`gettty()X` reads a line of user input (via `gety()X`) and checks for the special `.\backslash n` terminator that ends text-entry mode in commands like `a` and `i`. `gety()X` does the actual character-by-character reading into `linebuf`, stopping at a newline.

⟨*function* gettty*(ed.c)* 47c⟩≡ (? 0—1)
```
/// main -> commands('a') -> add -> <>
int
gettty(void)
{
    int rc;

    rc = gety();
    if(rc)
        return rc;
    // else
    if(linebuf[0] == '.' && linebuf[1] == '\0')
        return EOF;
    return 0; // OK_0 ?
}
```

⟨*function* gety*(ed.c)* 47d⟩≡ (? 0—1)
```
/// gettty -> <>
int
gety(void)
{
    int c;
    Rune *p = linebuf;
    ⟨gety() other locals 48a⟩
    for(;;) {
        c = getchr();
        if(c == '\n') {
            *p = 0;
            return 0;
        }
```

```
        // else
        if(c == EOF) {
            ⟨<<get() if gf 48b⟩
            return c;
        }
        // else
        if(c == 0)
            continue;
        // else
        *p++ = c;
        if(p >= &linebuf[LBSIZE-sizeof(Rune)])
            error(Q);
    }
}
```

⟨gety() *other locals* 48a⟩≡ (47d)
```
Rune *gf = globp;
```

⟨<<get() *if* gf 48b⟩≡ (47d)
```
if(gf)
    peekc = c;
```

C.6 commands() interpreter loop

`commands()X` is the main interpreter loop. It reads an optional address range into `addr1`/`addr2`, then dispatches on the command character via a `switch`. Each command ends with `continue` to loop back; falling through to `error(Q)` after the switch handles unknown commands.

⟨globals ed.c 48c⟩+≡ (? 0—1) ◁47a 49a▷
```
// ref<int>
int*    addr1;
// ref<int>
int*    addr2;
```

⟨*function* commands*(ed.c)* 48d⟩≡ (? 0—1)
```
void
commands(void)
{
    int c;
    ⟨commands() other locals 56a⟩

    for(;;) {
        ⟨commands() in for loop, if pflag 79c⟩
        ⟨commands() read addr1 and c via getchr 63d⟩
        switch(c) {
        ⟨commands() switch c cases (ed.c) 48e⟩
        }
        error(Q);
    }
}
```

⟨commands() *switch* c *cases (ed.c)* 48e⟩≡ (48d) 49c▷
```
case EOF:
    return;
```

C.7 reading a file: `r`

Reading a file is the first substantial command, and it involves several cooperating functions: `filename()X` parses the filename argument, `setwide()X` sets the address range to the whole buffer, `append()X` inserts lines by calling a callback in a loop, `getfile()X` (the callback) reads one line from the input file into `linebuf`, and `putline()X` writes `linebuf` to `tfile` and returns the offset to store in `zero`.

⟨globals ed.c 49a⟩+≡ (? 0—1) ◁48c 49b▷
```
char    file[FNSIZE];
```

⟨globals ed.c 49b⟩+≡ (? 0—1) ◁49a 54c▷
```
fdt io;
Biobuf  iobuf;
```

⟨commands() *switch* c *cases (ed.c)* 49c⟩+≡ (48d) ◁48e 54d▷
```
case 'r':
    // will set file[]
    filename(c);
caseread:
    if((io=open(file, OREAD)) < 0) {
        lastc = '\n';
        error(file);
    }
    ⟨commands() in r case if append only file 82c⟩
    Binit(&iobuf, io, OREAD);
    setwide();
    squeeze(0);
    c = (zero != dol);
    // getfile() will use iobuf
    append(getfile, addr2);
    exfile(OREAD); // will close io

    fchange = c;
    continue;
```

The `fchange = c` at the end is subtle: `c` was saved as `(zero != dol)` before the append, so `fchange` is set to true only if the buffer already had content. Reading the first file into an empty buffer does not count as a "change"—this is what lets you `q` without being warned after just opening a file.

C.7.1 Reading a `filename()`

`filename()X` reads a filename from input and stores it in `file`. If no filename is given (just a newline), it falls back to `savedfile`—this is how `r` with no argument re-reads the current file. It also updates `savedfile` when used from `e` or `f`, establishing the "remembered filename" for future operations.

⟨*function* filename*(ed.c)* 49d⟩≡ (? 0—1)
```
/// main -> commands('r') -> <>
void
filename(int comm)
{
    char *p1, *p2;
    Rune rune;
    int c;

    count = 0;
    c = getchr();
    ⟨filename() if c is newline or EOF 50b⟩
    // else
    ⟨filename() read a space and skip extra spaces 50a⟩
```

```
    p1 = file;
    do {
        if(p1 >= &file[sizeof(file)-6] || c == ' ' || c == EOF)
            error(Q);
        rune = c;
        p1 += runetochar(p1, &rune);
    } while((c=getchr()) != '\n');
    *p1 = '\0';

    ⟨filename() set savedfile depending on commands 50c⟩
}
```

⟨filename() *read a space and skip extra spaces* 50a⟩≡ (49d)

```
if(c != ' ')
    error(Q);
while((c=getchr()) == ' ')
    ;
if(c == '\n')
    error(Q);
```

⟨filename() *if* c *is newline or EOF* 50b⟩≡ (49d)

```
if(c == '\n' || c == EOF) {
    p1 = savedfile;
    if(*p1 == '\0' && comm != 'f')
        error(Q);
    p2 = file;
    while(*p2++ = *p1++)
        ;
    return;
}
```

⟨filename() *set* savedfile *depending on commands* 50c⟩≡ (49d)

```
if(savedfile[0] == '\0' || comm == 'e' || comm == 'f') {
    p1 = savedfile;
    p2 = file;
    while(*p1++ = *p2++)
        ;
}
```

C.7.2 setwide() and squeeze()

`setwide()X` sets `addr1`/`addr2` to span the whole buffer (line 1 to `dol`) when the user did not supply explicit addresses. `squeeze()X` validates that the address range is sane—called with 0 when an empty buffer is acceptable (e.g., `r`) or 1 when at least one line is required (e.g., `p`).

⟨*function* setwide*(ed.c)* 50d⟩≡ (? 0—1)

```
/// main -> commands('r') -> <>
void
setwide(void)
{
    if(!given) {
        addr1 = zero + (dol>zero);
        addr2 = dol;
    }
}
```

⟨*function* `squeeze` *(ed.c)* 51a⟩≡ (? 0—1)
```
/// main -> commands('r') -> <>
void
squeeze(int i)
{
    if(addr1 < zero+i || addr2 > dol || addr1 > addr2)
        error(Q);
}
```

C.7.3 `append()` and `getfile()`

`append()X` is the core insertion function. It takes a callback `f` (either `getfile()X` for reading files or `gettty()X` for user input) and an address `a` (where to insert after). For each line that `f` returns via `linebuf`, it calls `putline()X` to write it to `tfile`, then shifts all entries in `zero` above the insertion point up by one to make room for the new offset. This shift is the `while(a1 > rdot)` loop—the same kind of array insertion you see in any sorted-array implementation.

⟨*function* `append` *(ed.c)* 51b⟩≡ (? 0—1)
```
/// main -> commands('r') -> <>
int
append(int (*f)(void), int *a)
{
    //ref<int> in zero[]
    int *a1, *a2, *rdot;
    int nline = 0;
    // file offset in tfile for temporary line just added by putline()
    int tl;

    dot = a;
    // f() (e.g., getfile()) will modify linebuf
    while((*f)() == 0) {
        ⟨append() grow zero if zero too small 51c⟩
        // putline() will use linebuf
        tl = putline();
        nline++;

        // set zero[] indices
        a1 = ++dol;
        a2 = a1+1;
        rdot = ++dot;
        // shift existing line references up by one
        while(a1 > rdot)
            *--a2 = *--a1;
        // insert the new line reference in zero
        *rdot = tl;
    }
    return nline;
}
```

When `zero` is full, `realloc` may move it to a new address. Every pointer into the old array—`addr1`, `addr2`, `dol`, `dot`—must then be adjusted by the same delta (`a1 - zero`, i.e., new base minus old base). This is a classic C `realloc` pattern: compute the displacement once, then apply it to all pointers that referred to the old allocation.

⟨`append()` *grow* `zero` *if* `zero` *too small* 51c⟩≡ (51b)
```
if((dol-zero) >= nlall) {

    nlall += 512;
    a1 = realloc(zero, (nlall+5)*sizeof(int*)); // BUG: sizeof(int)
```

```
    if(a1 == nil) {
        error("MEM?");
        rescue();
    }
    tl = a1 - zero; /* relocate pointers */

    zero += tl;
    addr1 += tl;
    addr2 += tl;
    dol += tl;
    dot += tl;
}
```

`getfile()X` reads one line from `iobuf` into `linebuf`, counting characters in `count`. It replaces the terminating `\n` with a null rune (lines in `tfile` are null-terminated, not newline-terminated). It returns 0 on success or `EOF` when the file is exhausted.

⟨*function* `getfile` *(ed.c)* 52a⟩≡ (? 0—1)
```
/// main -> commands(c = 'r') -> append(<>, ...) -> <>
int
getfile(void)
{
    int c;
    Rune *lp;

    lp = linebuf;
    do {
        c = Bgetrune(&iobuf);
        ⟨getfile() if c negative, possibly return EOF 52b⟩
        // else
        ⟨getfile() if overflow linebuf 52c⟩
        // else
        *lp++ = c;
        count++;
    } while(c != '\n');
    lp[-1] = 0;
    return 0; // OK_0
}
```

⟨`getfile()` *if* `c` *negative, possibly return* `EOF` 52b⟩≡ (52a)
```
if(c < 0) {
    if(lp > linebuf) {
        putst("'\\n' appended");
        c = '\n';
    } else
        return EOF;
}
```

⟨`getfile()` *if overflow* `linebuf` 52c⟩≡ (52a)
```
if(lp >= &linebuf[LBSIZE]) {
    lastc = '\n';
    error(Q);
}
```

C.7.4 `putline()` (simplified)

Here is a simplified version of `putline()X` that shows the essential logic: seek to `tline` in `tfile`, write the contents of `linebuf`, advance `tline`, and return the old offset (which `append()X` stores in `zero`). The real

version (shown in the Optimizations section) uses block-based I/O for performance.

⟨putline() *simplified* 53⟩≡

```
int
putline(void)
{
    int n;
    long tl;

    tl = tline;                 // current temp-file write offset
    n = strlen(linebuf) + 1;    // include terminating '\0'

    if(tline + n >= ENDCORE)
        error("temp file overflow");

    seek(tfile, tline, 0);
    write(tfile, linebuf, n);

    tline += n;
    return tl;
}
```

A short example of how `zero` and `tfile` evolve. Suppose the user types:

```
r foo.txt   (file contains: "alpha\nbeta\ngamma\n")
2d          (delete line 2)
1a          (append after line 1)
delta       (the new text)
.
```

After `r` reads the three lines:

```
tfile (offsets):   0      6     11     17
                   +------+-----+------+
                   |alpha\|beta\|gamma\|
                   +------+-----+------+
zero:  [_, 0, 6, 11]    (3 lines, dot=3)
```

After `2d` (delete line 2: shift entries down in `zero`, do nothing on disk):

```
tfile:             unchanged (offset 6 'beta' is orphaned)
zero:  [_, 0, 11]       (2 lines, dot=2)
```

After `1a` then `delta` then `.` (append "delta" after line 1, `putline()` writes "delta n" to the end of `tfile` at offset 17, `append()` inserts the new offset into `zero`):

```
tfile:             0      6     11     17     23
                   +------+-----+------+------+
                   |alpha\|beta\|gamma\|delta\|
                   +------+-----+------+------+
                                       ^new
zero:  [_, 0, 17, 11]    (3 lines, dot=2)
```

Notice that `tfile` grew by one line (`delta`), but the "deleted" `beta` is still on disk—it just is no longer referenced by `zero`. This is why ed can do bounded edits in constant time but the temp file can become arbitrarily large during a long session.

C.7.5 `exfile()`

`exfile()X` ("exit file") finalizes an I/O operation: it flushes the buffer if writing, closes `io`, and in verbose mode prints the character count via `putd()X`.

⟨*function* `exfile` *(ed.c)* 54a⟩≡ (? 0—1)
```
/// main -> commands('r') -> <>
void
exfile(int om)
{

    if(om == OWRITE)
        if(Bflush(&iobuf) < 0)
            error(Q);
    close(io);
    io = -1;
    if(vflag) {
        putd();
        putchr(L'\n');
    }
}
```

⟨*function* `putd` *(ex.c)* 54b⟩≡ (? 0—1)
```
void
putd(void)
{
    int r;

    r = count%10;
    count /= 10;
    if(count)
        putd();
    putchr(r + L'0');
}
```

C.8 writing a file: `w`

Writing is the dual of reading. `putfile()X` iterates over the address range, fetches each line from `tfile` via `getline()X`, and writes it to the output file via `Bputrune`. The `W` (uppercase) variant opens the file in append mode.

⟨`globals ed.c` 54c⟩+≡ (? 0—1) ◁49b 63a▷
```
// write append
bool wrapp;
```

⟨`commands()` *switch* `c` *cases (ed.c)* 54d⟩+≡ (48d) ◁49c 56d▷
```
case 'W':
    wrapp = true;
    // Fallthrough
case 'w':
    setwide();
    squeeze(dol>zero);

    ⟨commands() when 'w' check for 'q' part1 56b⟩
    filename(c);

    if(!wrapp ||
      ((io = open(file, OWRITE)) == -1) ||
      ((seek(io, 0L, SEEK__END)) == -1))
```

```
        if((io = create(file, OWRITE, 0666)) < 0)
            error(file);

    Binit(&iobuf, io, OWRITE);
    wrapp = false;
    if(dol > zero)
        putfile();

    exfile(OWRITE);

    if(addr1<=zero+1 && addr2==dol)
        fchange = false;
    ⟨commands() when 'w' check for 'q' part2 56c⟩
    continue;
```

C.8.1 putfile()

⟨*function* putfile*(ed.c)* 55a⟩≡ (? 0—1)

```
  void
  putfile(void)
  {
      int *a1;
      Rune *lp;
      long c;

      a1 = addr1;
      do {
          lp = getline(*a1++);
          for(;;) {
              count++;
              c = *lp++;
              if(c == 0) {
                  if(Bputrune(&iobuf, '\n') < 0)
                      error(Q);
                  break;
              }
              if(Bputrune(&iobuf, c) < 0)
                  error(Q);
          }
      } while(a1 <= addr2);
      if(Bflush(&iobuf) < 0)
          error(Q);
  }
```

C.8.2 getline() (simplified)

`getline()X` is the dual of `putline()X`: given a line offset `tl` in `tfile`, it seeks to that position and reads characters into `linebuf` until it hits the null terminator. Like `putline()X`, the real version uses block-based I/O.

⟨getline() *simplified* 55b⟩≡

```
  char *
  getline(int tl)
  {
      int n;

      // seek to the line's offset in the temp file
```

```
    seek(tfile, tl, 0);

    // read bytes up to the next NUL
    n = 0;
    while (read(tfile, &linebuf[n], 1) == 1 && linebuf[n] != '\0')
        n++;

    linebuf[n] = '\0';
    return linebuf;
}
```

C.8.3 Write and quit: `wq`

The `w` command peeks at the next character to detect `wq` (write and quit) or `wQ` (write and force-quit). If the next character is neither `q` nor `Q`, it is pushed back with `peekc` so `filename()X` can read it.

⟨`commands()` *other locals* 56a⟩≡ (48d) 63b▷
```
int temp;
```

⟨`commands()` *when* `'w'` *check for* `'q'` *part1* 56b⟩≡ (54d)
```
temp = getchr();
if(temp != 'q' && temp != 'Q') {
    peekc = temp;
    temp = 0;
}
```

⟨`commands()` *when* `'w'` *check for* `'q'` *part2* 56c⟩≡ (54d)
```
if(temp == 'Q')
    fchange = false;
if(temp)
    quit();
```

C.9 Main commands

With the reading and writing machinery in place, we can now look at the basic editing commands. Most of them are short—often just a few lines calling the helper functions we have already seen.

C.9.1 printing lines: `p`

The `p` command prints lines in the address range. `printcom()X` iterates from `addr1` to `addr2`, calling `getline()X` to fetch each line from `tfile` and `putshst()X` to display it. Pressing just Enter (newline with no command) prints the next line—the `'\backslash n'` case below advances `dot` by one and prints.

⟨`commands()` *switch* `c` *cases (ed.c)* 56d⟩+≡ (48d) ◁54d 57e▷
```
⟨commands before 'p' case 78c⟩
case 'p':
case 'P':
    newline();
    printcom();
    continue;
```

⟨*function* `newline` *(ed.c)* 57a⟩≡ (? 0—1)

```
void
newline(void)
{
    int c;

    c = getchr();
    if(c == '\n' || c == EOF)
        return;
    ⟨newline() if special chars pln 79d⟩
    // else
    error(Q);
}
```

⟨*function* `printcom` *(ed.c)* 57b⟩≡ (? 0—1)

```
void
printcom(void)
{
    int *a1;

    nonzero();
    a1 = addr1;
    do {
        ⟨printcom() if listn 80d⟩
        putshst(getline(*a1++));
    } while(a1 <= addr2);
    dot = addr2;
    ⟨printcom() reset flags 80a⟩
}
```

⟨*function* `nonzero` *(ed.c)* 57c⟩≡ (? 0—1)

```
void
nonzero(void)
{
    squeeze(1);
}
```

⟨*function* `putshst` *(ed.c)* 57d⟩≡ (? 0—1)

```
void
putshst(Rune *sp)
{
    col = 0;
    while(*sp)
        putchr(*sp++);
    putchr(L'\n');
}
```

⟨`commands()` *switch* `c` *cases (ed.c)* 57e⟩+≡ (48d) ◁56d 58a▷

```
case '\n':
    if(a1==nil) {
        a1 = dot+1;
        addr2 = a1;
        addr1 = a1;
    }
    if(lastsep==';')
        addr1 = a1;

    printcom();
    continue;
```

C.9.2 printing remembered file: `f`

⟨`commands()` *switch* `c` *cases (ed.c)* 58a⟩+≡ (48d) ◁57e 58b▷

```
case 'f':
    setnoaddr();
    filename(c);
    putst(savedfile);
    continue;
```

C.9.3 printing line number: `=`

⟨`commands()` *switch* `c` *cases (ed.c)* 58b⟩+≡ (48d) ◁58a 58c▷

```
case '=':
    setwide();
    squeeze(0);
    newline();
    count = addr2 - zero;
    putd();
    putchr(L'\n');
    continue;
```

C.9.4 append and insert: `a`, `i`

Both `a` (append) and `i` (insert) use `add()X`, which calls `append(gettty, addr2)`. The difference is that `i` decrements `addr1`/`addr2` by one, so the new text goes *before* the current line instead of after it.

⟨`commands()` *switch* `c` *cases (ed.c)* 58c⟩+≡ (48d) ◁58b 58d▷

```
case 'a':
    add(0);
    continue;
```

⟨`commands()` *switch* `c` *cases (ed.c)* 58d⟩+≡ (48d) ◁58c 58f▷

```
case 'i':
    add(-1);
    continue;
```

⟨*function* `add`*(ed.c)* 58e⟩≡ (? 0—1)

```
void
add(int i)
{
    if(i && (given || dol > zero)) {
        addr1--;
        addr2--;
    }
    squeeze(0);
    newline();
    append(gettty, addr2);
}
```

C.9.5 quitting: `q`

⟨`commands()` *switch* `c` *cases (ed.c)* 58f⟩+≡ (48d) ◁58d 59b▷

```
case 'Q':
    fchange = false;
    // fallthrough:
case 'q':
    setnoaddr();
    newline();
    quit();
```

⟨*function* `setnoaddr` *(ed.c)* 59a⟩≡ (? 0—1)

```
void
setnoaddr(void)
{
    if(given)
        error(Q);
}
```

C.9.6 deleting lines: `d`

`rdelete()X` ("range delete") removes lines by shifting the `zero` entries above `addr2` down to overwrite the deleted range—the mirror image of the upward shift in `append()X`. The text remains in `tfile` (it is never reclaimed), only the offsets in `zero` change.

⟨`commands()` *switch* `c` *cases (ed.c)* 59b⟩+≡ (48d) ◁58f 59d▷

```
case 'd':
    nonzero();
    newline();
    rdelete(addr1, addr2);
    continue;
```

⟨*function* `rdelete` *(ed.c)* 59c⟩≡ (? 0—1)

```
void
rdelete(int *ad1, int *ad2)
{
    int *a1, *a2, *a3;

    a1 = ad1;
    a2 = ad2+1;
    a3 = dol;
    dol -= a2 - a1;
    do {
        *a1++ = *a2++;
    } while(a2 <= a3);
    a1 = ad1;
    if(a1 > dol)
        a1 = dol;
    dot = a1;
    fchange = true;
}
```

C.9.7 changing lines: `c`

The `c` (change) command is simply delete followed by append: it calls `rdelete()X` then `append(gettty, addr1-1)`.

⟨`commands()` *switch* `c` *cases (ed.c)* 59d⟩+≡ (48d) ◁59b 60a▷

```
case 'c':
    nonzero();
    newline();
    rdelete(addr1, addr2);
    append(gettty, addr1-1);
    continue;
```

C.9.8 moving and copying lines: `m` and `t`

The `m` (move) and `t` (copy, since `c` was taken) commands share the `move()X` function. For `t`, the lines are first duplicated to the end of the buffer via `append(getcopy, ...)`, then moved. The actual move uses a clever triple-reverse algorithm: to move a block from position A to position B, reverse the elements from A to the block, reverse the block, then reverse the whole range. This rearranges `zero` entries in-place without any temporary storage.

The triple-reverse trick is well known among C programmers but easy to forget. Here is how it rotates a block to the left in three reversal passes. Suppose `zero` looks like this and we want to move lines 4–5 (the block "DE") in front of line 2 (i.e., between line 1 and the current line 2):

```
start:            A B C D E F
                  1 2 3 4 5 6
                     ^   ^^^
                     adt ad1..ad2

reverse(adt, ad1):     reverse [B C] -> [C B]
                  A C B D E F

reverse(ad1, ad2):     reverse [D E] -> [E D]
                  A C B E D F

reverse(adt, ad2):     reverse [C B E D] -> [D E B C]
                  A D E B C F
                     ^^^
                     moved!
```

This is the same algorithm Brian Kernighan attributes to Doug McIlroy in *Programming Pearls*: rotating an array can be done in-place in $O(n)$ time and $O(1)$ extra space using three reversals, no temporary buffer required. Compare with the naive approach (allocate a temp array, copy, shift, copy back) which needs $O(n)$ extra memory. The opposite case (moving a block forward instead of backward) is symmetric: `move()X` picks one of the two branches based on whether the destination is before or after the source.

⟨`commands()` *switch* `c` *cases (ed.c)* 60a⟩+≡ (48d) ◁59d 60b▷

```
case 'm':
    move(0);
    continue;
```

⟨`commands()` *switch* `c` *cases (ed.c)* 60b⟩+≡ (48d) ◁60a 70a▷

```
case 't':
    move(1);
    continue;
```

⟨*function* `move` *(ed.c)* 60c⟩≡ (? 0—1)

```
void
move(int cflag)
{
    int *adt, *ad1, *ad2;

    nonzero();
    if((adt = address())==0)    /* address() guarantees addr is in range */
        error(Q);
    newline();

    if(cflag) {
        int *ozero, delta;
```

```
        ad1 = dol;
        ozero = zero;
        append(getcopy, ad1++);
        ad2 = dol;
        delta = zero - ozero;
        ad1 += delta;
        adt += delta;
    } else {
        ad2 = addr2;
        for(ad1 = addr1; ad1 <= ad2;)
            *ad1++ &= ~01;
        ad1 = addr1;
    }
    ad2++;
    if(adt<ad1) {
        dot = adt + (ad2-ad1);
        if((++adt)==ad1)
            return;
        reverse(adt, ad1);
        reverse(ad1, ad2);
        reverse(adt, ad2);
    } else
    if(adt >= ad2) {
        dot = adt++;
        reverse(ad1, ad2);
        reverse(ad2, adt);
        reverse(ad1, adt);
    } else
        error(Q);
    fchange = true;
}
```

⟨*function* `getcopy` *(ed.c)* 61a⟩≡ (? 0—1)

```
int
getcopy(void)
{
    if(addr1 > addr2)
        return EOF;
    getline(*addr1++);
    return 0;
}
```

⟨*function* `reverse` *(ed.c)* 61b⟩≡ (? 0—1)

```
void
reverse(int *a1, int *a2)
{
    int t;

    for(;;) {
        t = *--a2;
        if(a2 <= a1)
            return;
        *a2 = *a1;
        *a1++ = t;
    }
}
```

C.10 Command addresses

Until now I have glossed over how `addr1` and `addr2` are set. In `ed`, most commands can be preceded by one or two addresses: a line number (`3p`), a range (`1,5d`), the current line (`.p`), the last line (`$a`), or even a search (`/foo/p`). The address-parsing code runs before the command character is read, so every command benefits from the same rich addressing syntax.

Here is a cheat sheet of the address primitives and how they eventually show up in the code:

```
syntax      meaning                       handled by
------      --------------------------    ------------------
N           absolute line N               getnum() in address()
.           dot (current line)            case '.' in address()
$           dol (last line)               case '$' in address()
'x          line previously marked x      case '\'' (see 'k')
/re/        next line matching re         case '/'  in address()
?re?        previous line matching re     case '?'  in address()
A+N  A-N    A offset by N                 + and - operators
A,B         range (addr1=A, addr2=B)      top-level in commands()
A;B         range with side effect dot=A ';' case in commands()
```

All these forms feed a single recursive-descent parser. The two address separators `,` and `;` are parsed in `commands()X` itself; everything to the left or right of a separator is parsed by `address()X`.

A worked example makes this concrete. Suppose the user types `.+2,/foo/p` when `dot` points to line 5 and the next line matching `foo` is line 11. The parse proceeds as follows:

```
outer loop in commands():
  a1 = address()        -> parses ".+2"
     step 1: c='.'    -> a=dot (line 5), opcnt=1
     step 2: c='+'    -> sign=+, nextopand=2, loop
     step 3: c='2'    -> a += 2 (line 7), opcnt=2
     step 4: c=','    -> push back, return a (line 7)
  c = ','               -> separator, loop again
  addr1 = 7
  a1 = address()        -> parses "/foo/"
     step 1: c='/'    -> compile(/foo/); scan from dot
                           until match -> a=line 11
     step 2: c='p'    -> push back, return a (line 11)
  c = 'p'               -> not a separator, exit loop
  addr2 = 11; given = true
  switch(c='p') -> print lines 7..11
```

Two things are worth noticing. First, the address parser is strictly left-to-right with no precedence—every operator is applied in sequence, so `1+2-3` means `((1+2)-3)`, never `1+(2-3)`. Second, the search forms `/re/` and `?re?` are resolved *during* parsing: `address()X` actually scans the buffer and returns a concrete line pointer, which is why compiling the regexp happens inline in `address()` rather than later at execution time.

Returning to the `mkenam` script in Section C.2, you can now read the address parts of each line. The substitution line is two addresses (`1` and `$`) followed by the `s` command; the `v` line is a global-not-match command (no address part, matches operate on the whole buffer); `1i` is a single address (line 1) followed by `i` (insert before).

C.10.1 Reading addr1 and addr2

⟨globals ed.c 63a⟩+≡ (? 0—1) ◁54c 67a▷
```
bool given;
```

⟨commands() *other locals* 63b⟩+≡ (48d) ◁56a 63c▷
```
int *a1;
```

⟨commands() *other locals* 63c⟩+≡ (48d) ◁63b 82b▷
```
char lastsep; // '\n' or ',' or ';'
```

The address-parsing loop reads addresses separated by , or ;. If no addresses are given, addr1 = addr2 = dot (the current line). If only one address is given, both addr1 and addr2 point to it. The given flag records whether the user typed an explicit address, so commands like setwide()X can choose appropriate defaults.

⟨commands() *read* addr1 *and* c *via* getchr 63d⟩≡ (48d)
```
c = '\n';
addr1 = nil;

for(;;) {
    lastsep = c;
    a1 = address();
    c = getchr();

   if(c != ',' && c != ';')
        break;

    // else
    if(lastsep == ',')
        error(Q);
    if(a1 == nil) {
        a1 = zero+1; // line 1
        if(a1 > dol)
            a1--;
    }
    addr1 = a1;
    ⟨commands() in address parsing, if separator is ';' 63e⟩
}
⟨commands() after address parsing, use defaults if missing addresses 63f⟩
```

⟨commands() *in address parsing, if separator is* ';' 63e⟩≡ (63d)
```
if(c == ';')
    dot = a1;
```

⟨commands() *after address parsing, use defaults if missing addresses* 63f⟩≡ (63d)
```
if(lastsep != '\n' && a1 == nil)
    a1 = dol;

if((addr2=a1) == nil) {
    given = false;
    addr2 = dot;
} else
    given = true;

if(addr1 == nil)
    addr1 = addr2;
```

C.10.2 address()

`address()X` parses a single address expression. It starts from `dot` and applies offsets: a bare number is relative to line 0 (i.e., absolute), `.` and `$` set the base, `+` and `-` adjust the sign for subsequent operands, and `/re/` or `?re?` search forward or backward. The `default:` case handles the return: if no operator is pending, push the character back and return the computed address. The nested `do-do` structure is subtle. The inner loop just skips whitespace. The outer loop chains operators: each iteration consumes one operand (`.`, `$`, a number, or a search), and then loops back to see if another `+` or `-` follows. The variable `nextopand` tracks whether a trailing operator like `+` or `-` was seen without a following operand, in which case the `default:` branch adds an implicit `1` (so `3+` means `3+1`, i.e., line 4).

⟨*function* `address` *(ed.c)* 64a⟩≡ (? 0—1)

```
int*
address(void)
{
    int sign, *a, opcnt, nextopand, *b, c;

    nextopand = -1;
    sign = 1;
    opcnt = 0;
    a = dot;
    do {
        do {
            c = getchr();
        } while(c == ' ' || c == '\t');

        if(c >= '0' && c <= '9') {
            peekc = c;
            if(!opcnt)
                a = zero;
            a += sign*getnum();
        } else

        switch(c) {
        // will return in default: case
        ⟨address() (ed.c) switch c cases 64b⟩
        }

        sign = 1;
        opcnt++;
    } while(zero <= a && a <= dol);

    error(Q);
    return nil;
}
```

⟨`address()` *(ed.c) switch* `c` *cases* 64b⟩≡ (64a) 65b ▷

```
default:
    if(nextopand == opcnt) {
        a += sign;
        if(a < zero || dol < a)
            continue;       /* error(Q); */
    }

    if(c != '+' && c != '-' && c != '^') {
        peekc = c;
        if(opcnt == 0)
            a = nil;
```

```
        // finally returning!
        return a;
    }
    sign = 1;
    if(c != '+')
        sign = -sign;
    nextopand = ++opcnt;
    continue;
```

⟨*function* `getnum` *(ed.c)* 65a⟩≡ (? 0—1)

```
int
getnum(void)
{
    int r = 0;
    int c;

    for(;;) {
        c = getchr();
        if(c < '0' || c > '9')
            break;
        r = r*10 + (c-'0');
    }
    peekc = c;
    return r;
}
```

C.10.3 Basic addresses: . and $

⟨`address()` *(ed.c) switch* `c` *cases* 65b⟩+≡ (64a) ◁64b 66▷

```
case '$':
    a = dol;
    // Fallthrough
case '.':
    if(opcnt)
        error(Q);
    break;
```

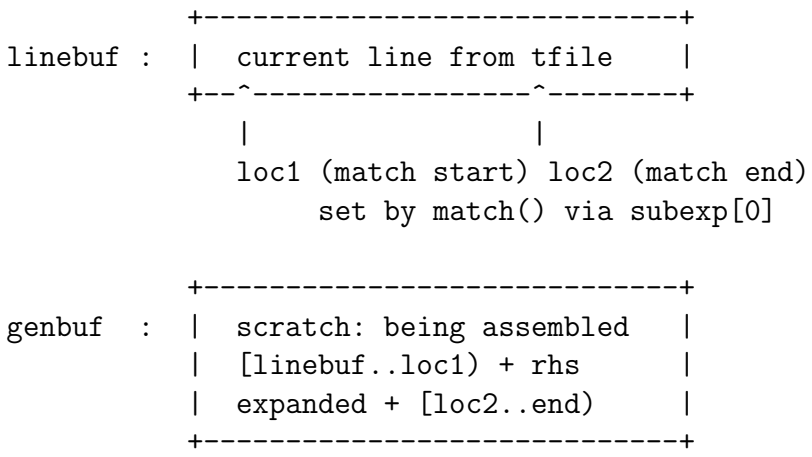
C.11 Search and replace

Search and replace is where `ed`'s real power lies. The `/re/` and `?re?` address forms search forward and backward, the `s` command performs substitutions, and the `g` command applies commands to all matching lines. The actual regular expression engine is in `libregexp` (see LIBCORE book [Pad16]); `ed` just reads and compiles patterns via `compile()X` and tests matches via `match()X`.

Before diving in, it helps to see the buffers and pointers that the substitute machinery juggles together. A single `s/re/rhs/` command touches five distinct pieces of state:

```
              +-----------------------------+
pattern -->|  libregexp Reprog  (NFA)    |   compiled RE
              +-----------------------------+

              +-----------------------------+
rhsbuf   :    |  replacement with ESCFLG    |   parsed by compsub()
              |  markers for \1..\9 and &   |
              +-----------------------------+
```



The flow of a substitution is always the same: (1) `getline()X` fills `linebuf` from `tfile`; (2) `match()X` runs `rregexec` on `linebuf` and records the hit as `loc1`/`loc2` pointers *into* `linebuf`; (3) `dosub()X` walks `linebuf` and `rhsbuf` together, emitting the result into `genbuf`, then copies `genbuf` back into `linebuf`; (4) `putline()X` writes `linebuf` to `tfile` and the new offset replaces the old one in `zero`. The two-buffer dance is what makes the `g` flag work: after `dosub()` the new `loc2` points into the updated `linebuf`, so a fresh `match(nil)` can resume scanning right where the previous hit ended.

C.11.1 Search as addresses: `/re/` and `?re?`

When `/re/` or `?re?` appears as an address, `address()X` compiles the pattern and scans through the buffer line by line, wrapping around from `dol` to `zero` (or vice versa for `?`), until a match is found or we return to where we started.

⟨`address()` *(ed.c) switch* `c` *cases* 66⟩+≡ (64a) ◁65b 75a▷

```
case '?':
    sign = -sign;
    // Fallthrough
case '/':
    compile(c);
    b = a;
    for(;;) {
        // direction
        a += sign;
        // wrap around
        if(a <= zero)
            a = dol;
        if(a > dol)
            a = zero;

        if(match(a))
            break;
        // reached back where we were without finding anything
        if(a == b)
            error(Q);
    }
    break;
```

C.11.2 Reading and compiling a regexp: `compile()`

`compile()X` reads a regular expression delimited by `eof` (typically `/` or `?`), handles backslash escapes, and passes the result to `regcomp()` from `libregexp`. If the delimiter is given with no pattern in between, the previously

compiled `pattern` is reused—this is how `//` means "repeat the last search."

⟨globals ed.c 67a⟩+≡ (? 0—1) ◁63a 68b▷
```
Reprog  *pattern;
```

⟨constants ed.c 67b⟩+≡ (? 0—1) ◁41d 68a▷
```
ESIZE   = 256,      /* max size of reg exp */
```

⟨*function* compile*(ed.c)* 67c⟩≡ (? 0—1)
```
/// commands('g' | 'v') -> global -> <>
void
compile(int eof)
{
    Rune c;
    char *ep;
    char expbuf[ESIZE];

    if((c = getchr()) == '\n') {
        peekc = c;
        c = eof;
    }
    if(c == eof) {
        if(!pattern)
            error(Q);
        return;
    }
    // else
    if(pattern) {
        free(pattern);
        pattern = nil;
    }
    ep = expbuf;
    do {
        if(c == '\\') {
            ⟨compile()(ed.c) sanity check ep inside expbuf 67d⟩
            // else
            ep += runetochar(ep, &c);
            if((c = getchr()) == '\n') {
                error(Q);
                return;
            }
        }
        ⟨compile()(ed.c) sanity check ep inside expbuf 67d⟩
        // else
        ep += runetochar(ep, &c);
    } while((c = getchr()) != eof && c != '\n');
    if(c == '\n')
        peekc = c;
    *ep = '\0';

    // lib_regexp call
    pattern = regcomp(expbuf);
}
```

⟨compile()*(ed.c) sanity check* ep *inside* expbuf 67d⟩≡ (67)
```
if(ep >= expbuf+sizeof(expbuf)) {
    error(Q);
    return;
}
```

C.11.3 match()

`match()X` tests whether the given line matches `pattern`. When `addr` is non-null, it calls `getline()X` to load the line into `linebuf` first; when `addr` is null, it retries the match from `loc2` (the end of the previous match on the same line), which is how `s///g` finds successive matches. On success, `loc1` and `loc2` bracket the matched region in `linebuf`.

⟨constants ed.c 68a⟩+≡ (? 0—1) ◁67b 69b▷
```
  MAXSUB  = 9,        /* max number of sub reg exp */
```

⟨globals ed.c 68b⟩+≡ (? 0—1) ◁67a 68f▷
```
  Resub   subexp[MAXSUB];
```

⟨*function* match*(ed.c)* 68c⟩≡ (? 0—1)
```
  bool
  match(int *addr)
  {
      ⟨match() (ed.c) return if no pattern 68d⟩
      if(addr){
          ⟨match() (ed.c) return if addr is zero 68e⟩
          subexp[0].s.rsp = getline(*addr);
      } else
          ⟨match() (ed.c) when null addr use loc2 not getline 68i⟩
      subexp[0].e.rep = nil;

      if(rregexec(pattern, linebuf, subexp, MAXSUB)) {
          ⟨match() (ed.c) set loc1 and loc2 with matched string 68g⟩
          return true;
      }
      // else
      ⟨match() (ed.c) reset loc1 and loc2 68h⟩
      return false;
  }
```

⟨match() *(ed.c) return if no* pattern 68d⟩≡ (68c)
```
  if(!pattern)
      return false;
```

⟨match() *(ed.c) return if* addr *is* zero 68e⟩≡ (68c)
```
  if(addr == zero)
      return false;
```

⟨globals ed.c 68f⟩+≡ (? 0—1) ◁68b 69a▷
```
  Rune*   loc1;
  Rune*   loc2;
```

⟨match() *(ed.c) set* loc1 *and* loc2 *with matched string* 68g⟩≡ (68c)
```
  loc1 = subexp[0].s.rsp;
  loc2 = subexp[0].e.rep;
```

⟨match() *(ed.c) reset* loc1 *and* loc2 68h⟩≡ (68c)
```
  loc1 = loc2 = nil;
```

⟨match() *(ed.c) when null* addr *use* loc2 *not* getline 68i⟩≡ (68c)
```
  subexp[0].s.rsp = loc2;
```

C.11.4 Reading and compiling a substitution: `compsub()`

`compsub()X` reads the full `s/re/replacement/` command. It first calls `compile()X` to handle the left-hand side (the pattern), then reads the right-hand side into `rhsbuf`, handling backslash escapes and the `ESCFLG` marker for group references like `\1`. It returns true if the trailing `g` flag is present (global substitution on the line).

⟨globals ed.c 69a⟩+≡ (? 0—1) ◁68f 71a▷
```
Rune    rhsbuf[LBSIZE/sizeof(Rune)];
```

⟨constants ed.c 69b⟩+≡ (? 0—1) ◁68a 76c▷
```
ESCFLG  = Runemax,  /* escape Rune - user defined code */
```

⟨*function* compsub*(ed.c)* 69c⟩≡ (? 0—1)
```
bool
compsub(void)
{
    int seof, c;
    Rune *p;

    seof = getchr();
    if(seof == '\n' || seof == ' ')
        error(Q);

    // read (and compile) the regexp (left hand side)
    compile(seof);

    // read the subst (right hand side)
    p = rhsbuf;
    for(;;) {
        c = getchr();
        ⟨compsub() (ed.c) if match group 73a⟩
        else
        ⟨compsub() (ed.c) if newline and no globp 69e⟩
        else
          if(c == seof)
              break;
        // else
        *p++ = c;
        <<[[compsub()]](ed.c) sanity check [[p]] inside [[rhsbuf]]>>=
    }
    *p = 0;
    ⟨compsub() (ed.c) peek to check for 'g' 73f⟩
    // else
    newline();
    return false;
}
```

⟨compsub() (ed.c) sanity check p inside rhsbuf 69d⟩≡ (73a)
```
if(p >= &rhsbuf[LBSIZE/sizeof(Rune)])
    error(Q);
```

⟨compsub() (ed.c) if newline and no globp 69e⟩≡ (69c)
```
if(c == '\n' && (!globp || !globp[0])) {
    peekc = c;
    pflag = true;
    break;
}
```

C.11.5 substitute: s

`substitute()X` is the most complex command. For each line in the address range, it calls `match()X` to find the pattern, then `dosub()X` to build the replacement in `genbuf`, copying everything before `loc1`, the replacement text (expanding `&` and `\1`–`\9` references), and everything after `loc2`. The result is copied back to `linebuf` and written to `tfile` via `putline()X`, replacing the old `zero` entry. With the `g` flag, the match/dosub loop repeats on the same line via `match(nil)`; the optional skip count (`s3/re/.../`) skips the first `n-1` matches.

⟨`commands()` *switch* `c` *cases (ed.c)* 70a⟩+≡ (48d) ◁60b 73e▷

```
case 's':
    nonzero();
    substitute(globp != nil);
    continue;
```

⟨*function* `substitute`*(ed.c)* 70b⟩≡ (? 0—1)

```
void
substitute(int inglob)
{
    int *a1;
    bool gsubf; // s/.../.../g  global subst
    int n = 0;
    ⟨substitute()(ed.c) other locals 74a⟩

    ⟨substitute()(ed.c) read optional n 73g⟩

    gsubf = compsub();

    for(a1 = addr1; a1 <= addr2; a1++) {
        // will internally set linebuf[]
        if(match(a1)){

            int *ozero;
            int m = n;

            do {
                int span = loc2-loc1;

                if(--m <= 0) {
                    // will modify linebuf[]
                    dosub();

                    if(!gsubf)
                        break;
                    // else
                    if(span == 0) { /* null RE match */
                        if(*loc2 == 0)
                            break;
                        loc2++;
                    }
                }
            } while(match(nil));

            if(m <= 0) {
                inglob |= 01;
                // will use linebuf[]
                subnewa = putline();
                *a1 &= ~01;
                ⟨substitute()(ed.c) after putline() if anymarks 75b⟩
                ⟨substitute()(ed.c) after putline() set subolda 73d⟩
                *a1 = subnewa;
```

```
            ⟨substitute()(ed.c) after putline() call append() 74b⟩
        }
    }
}
if(inglob == 0)
    error(Q);
}
```

⟨globals ed.c 71a⟩+≡ (? 0—1) ◁69a 71c▷

```
int subnewa;
```

⟨init()(ed.c) initializing globals 71b⟩+≡ (44b) ◁44c 74g▷

```
subnewa = 0;
```

⟨globals ed.c 71c⟩+≡ (? 0—1) ◁71a 73c▷

```
Rune    genbuf[LBSIZE];
```

`dosub()X` builds the replacement in `genbuf`: it copies `linebuf` up to `loc1` (the match start), expands the replacement string from `rhsbuf` (handling `&` for the whole match and `\1`–`\9` for subgroups via `place()X`), then appends the rest of `linebuf` after `loc2`. Finally it copies `genbuf` back to `linebuf`. The line `loc2 = sp - genbuf + linebuf` is the trickiest part: after assembly, `sp` points into `genbuf` where the tail (post-match text) was copied. This arithmetic converts that `genbuf` offset back into a `linebuf` pointer, so that the next call to `match(nil)` in the `g` flag loop will resume searching from the correct position in the updated `linebuf`.

A concrete trace of `s/world/earth/` on `linebuf = "hello world!"` shows the four phases of `dosub()X`:

```
initial state (after match() on linebuf):
  linebuf: "hello world!\0"
            ^      ^    ^
            |      loc1 loc2
            lp starts here
  rhsbuf:  "earth\0"
  genbuf:  (empty, sp at start)

phase 1: copy linebuf[..loc1) -> genbuf
  genbuf:  "hello "
                 ^sp

phase 2: expand rhsbuf into genbuf
  (no & or \N, so plain copy)
  genbuf:  "hello earth"
                       ^sp

phase 3: the famous pointer shuffle
  loc2 = sp - genbuf + linebuf
       = 11        - 0       + linebuf
       -> points into linebuf at offset 11 (the '!')
  This is where the next match(nil) would start.

phase 4: copy linebuf[loc2..] -> genbuf, then genbuf -> linebuf
  genbuf:  "hello earth!\0"
  linebuf: "hello earth!\0"
```

Phase 3 is subtle because `loc2` must end up pointing into the *new* `linebuf`, not the old one. At the moment of the assignment, `linebuf` still holds the pre-substitution text, but we know the new prefix ("hello earth") will occupy bytes `linebuf[0..11)` once phase 4 runs. So `sp-genbuf` (the length of that prefix) plus `linebuf` (the base) is the correct future location. It is a deliberate compute-first, copy-later trick—if the assignment were reordered after the final copy, `sp` would already have advanced and the arithmetic would break.

⟨*function* `dosub`*(ed.c)* 72a⟩≡ (? 0—1)

```
void
dosub(void)
{
    Rune *lp, *sp, *rp;
    int c, n;

    lp = linebuf;
    sp = genbuf;
    rp = rhsbuf;
    while(lp < loc1)
        *sp++ = *lp++;
    while(c = *rp++) {
        ⟨dosub()(ed.c) if c == '&' 72b⟩
        ⟨dosub()(ed.c) if match group 73b⟩
        // else
        *sp++ = c;
        if(sp >= &genbuf[LBSIZE])
            error(Q);
    }
    lp = loc2;
    loc2 = sp - genbuf + linebuf;
    while(*sp++ = *lp++)
        if(sp >= &genbuf[LBSIZE])
            error(Q);
    lp = linebuf;
    sp = genbuf;
    while(*lp++ = *sp++)
        ;
}
```

C.11.6 Advanced substitutions

Matched string reference: `s/.../...&.../`

⟨`dosub()`*(ed.c) if* `c == '&'` 72b⟩≡ (72a)

```
if(c == '&'){
    sp = place(sp, loc1, loc2);
    continue;
}
```

⟨*function* `place`*(ed.c)* 72c⟩≡ (? 0—1)

```
Rune*
place(Rune *sp, Rune *l1, Rune *l2)
{

    while(l1 < l2) {
        *sp++ = *l1++;
        if(sp >= &genbuf[LBSIZE])
            error(Q);
    }
    return sp;
}
```

Matched groups: `s/.../...\1.../`

⟨`compsub()` *(ed.c) if match group* 73a⟩≡ (69c)
```
if(c == '\\') {
    c = getchr();
    *p++ = ESCFLG;
    ⟨compsub() (ed.c) sanity check p inside rhsbuf 69d⟩
}
```

⟨`dosub()` *(ed.c) if match group* 73b⟩≡ (72a)
```
if(c == ESCFLG && (c = *rp++) >= '1' && c < MAXSUB+'0') {
    n = c-'0';
    if(subexp[n].s.rsp && subexp[n].e.rep) {
        sp = place(sp, subexp[n].s.rsp, subexp[n].e.rep);
        continue;
    }
    error(Q);
}
```

undo substitution: `u`

The `u` command provides a minimal undo: it remembers the previous `zero` entry (`subolda`) before the last substitution replaced it with `subnewa`, and swaps it back. This only undoes the most recent substitution on a single line—far from a general undo facility.

⟨`globals ed.c` 73c⟩+≡ (? 0—1) ◁71c 74e▷
```
int subolda;
```

⟨`substitute()` *(ed.c) after* `putline()` *set* `subolda` 73d⟩≡ (70b)
```
subolda = *a1;
```

⟨`commands()` *switch* `c` *cases (ed.c)* 73e⟩+≡ (48d) ◁70a 74d▷
```
case 'u':
    nonzero();
    newline();
    if((*addr2&~01) != subnewa)
        error(Q);
    *addr2 = subolda;
    dot = addr2;
    continue;
```

global subst: `s/.../.../g`

⟨`compsub()` *(ed.c) peek to check for* `'g'` 73f⟩≡ (69c)
```
peekc = getchr();
if(peekc == 'g') {
    peekc = 0;
    newline();
    return true;
}
```

Skipped matches: `s3/.../.../`

⟨`substitute()` *(ed.c) read optional* `n` 73g⟩≡ (70b)
```
n = getnum();   /* OK even if n==0 */
```

Append and subst

⟨`substitute()` *(ed.c) other locals* 74a⟩≡ (70b)
```
int *mp, nl;
```

⟨`substitute()` *(ed.c) after* `putline()` *call* `append()` 74b⟩≡ (70b)
```
ozero = zero;
nl = append(getsub, a1);
addr2 += nl;
nl += zero-ozero;
a1 += nl;
```

⟨*function* `getsub` *(ed.c)* 74c⟩≡ (? 0—1)
```
int
getsub(void)
{
    Rune *p1, *p2;

    p1 = linebuf;
    if((p2 = linebp) == 0)
        return EOF;
    while(*p1++ = *p2++)
        ;
    linebp = 0;
    return 0;
}
```

mark: `k`

The `k` command marks a line with a lowercase letter (`ka` through `kz`). The mark is stored in `names[c-'a']` as the line's `zero` entry (with the low bit cleared—since that bit is used by `global()X`). The `'` (apostrophe) address retrieves a marked line by scanning `zero` for the matching offset.

⟨`commands()` *switch* `c` *cases (ed.c)* 74d⟩+≡ (48d) ◁73e 76a▷
```
case 'k':
    nonzero();
    c = getchr();
    if(c < 'a' || c > 'z')
        error(Q);
    newline();
    names[c-'a'] = *addr2 & ~01;
    anymarks |= 01;
    continue;
```

⟨`globals ed.c` 74e⟩+≡ (? 0—1) ◁73c 74f▷
```
int names[26];
```

⟨`globals ed.c` 74f⟩+≡ (? 0—1) ◁74e 77b▷
```
int anymarks;
```

⟨`init()` *(ed.c) initializing globals* 74g⟩+≡ (44b) ◁71b 75d▷
```
anymarks = 0;
```

⟨address() *(ed.c) switch* c *cases* 75a⟩+≡ (64a) ◁66
```
case '\'':
    c = getchr();
    if(opcnt || c < 'a' || c > 'z')
        error(Q);
    a = zero;
    do {
        a++;
    } while(a <= dol && names[c-'a'] != (*a & ~01));
    break;
```

⟨substitute() *(ed.c) after* putline() *if* anymarks 75b⟩≡ (70b)
```
if(anymarks) {
    for(mp=names; mp<&names[26]; mp++)
        if(*mp == *a1)
            *mp = subnewa;
}
```

⟨init() *(ed.c) locals* 75c⟩≡ (44b)
```
int *markp;
```

⟨init() *(ed.c) initializing globals* 75d⟩+≡ (44b) ◁74g 84b▷
```
for(markp = names; markp < &names[26]; )
    *markp++ = 0;
```

C.11.7 global commands: g/re/cmd and v/re/cmd

The g/re/cmd command (and its inverse v/re/cmd) works in two phases. In phase 1, it scans all lines in the address range and marks those that match (or don't match, for v) the pattern by setting the low bit of their zero entry (*a1 |= 01—this is the "bit-stealing trick" that tline starting at 2 makes possible). In phase 2, it iterates over zero, and for each marked line, clears the mark, sets globp to the command string (stored in globuf), and calls commands()X recursively. The two-phase design is necessary because executing commands can change the buffer (e.g., g/re/d deletes lines), so we cannot match and execute at the same time. There is an important subtlety in the phase 2 loop: after each commands() call, a1 is reset to zero (line 2695) rather than just incrementing. This is because commands()X may call append()X, which may realloc the zero array, moving it to a new address. All saved pointers into the old array become dangling. Resetting a1 = zero picks up the (possibly new) base, and the loop re-scans from the beginning—this is safe because each executed line has its mark cleared, so it won't be processed again.

Visually, a g/foo/d on a four-line buffer goes through the following two phases. Bit 0 of each zero entry is stolen as the "matched" mark; I show it as M when set, . when clear:

```
initial:                 after phase 1 (mark):
zero[1] = 0x40           zero[1] = 0x40   .   "alpha"
zero[2] = 0x48           zero[2] = 0x49   M   "foobar"   <- matched
zero[3] = 0x56           zero[3] = 0x56   .   "beta"
zero[4] = 0x62           zero[4] = 0x63   M   "foo"      <- matched

phase 2 iteration:
  a1 = zero+1          .  skip
  a1 = zero+2          M  clear mark; dot=a1;
                           globp = "d\n"; commands();
                           (runs 'd' -> rdelete shifts entries down)
                           a1 = zero (restart from base)
```

```
a1 = zero+1          .  skip (old alpha)
a1 = zero+2          .  skip (was beta, shifted down)
a1 = zero+3          M  clear mark; delete; a1 = zero
a1 = zero+1          .  skip
a1 = zero+2          .  skip
dol is now zero+2    loop exits
```

Three consequences follow from this design. (1) The marks must live *inside* `zero` rather than in a parallel bitmap, because `rdelete()X` shifts `zero` entries around and a parallel bitmap would go out of sync instantly. Putting the mark in the low bit of the offset means delete, move, and substitute all carry marks along for free. (2) The low bit is safe to steal because `tline` starts at 2 and grows in multiples of 2 (aligned `Rune` boundaries), so legitimate offsets never have bit 0 set. (3) The restart-from-`zero` loop is $O(n^2)$ in the worst case (one full pass per matched line), which is why `gdelete()X` exists as a special case for `g/.../d`.

⟨`commands()` *switch* `c` *cases (ed.c)* 76a⟩+≡ (48d) ◁74d 76b▷

```
case 'g':
    global(1);
    continue;
```

⟨`commands()` *switch* `c` *cases (ed.c)* 76b⟩+≡ (48d) ◁76a 77a▷

```
case 'v':
    global(0);
    continue;
```

⟨`constants ed.c` 76c⟩+≡ (? 0—1) ◁69b 83c▷

```
GBSIZE  = 256,      /* max size of global command */
```

⟨*function* `global` *(ed.c)* 76d⟩≡ (? 0—1)

```
void
global(int k)
{
    Rune *gp, globuf[GBSIZE];
    int c, *a1;

    if(globp)
        error(Q);

    setwide();
    squeeze(dol > zero);

    // readding a '/' or '?'
    c = getchr();
    if(c == '\n')
        error(Q);

    // reading the whole pattern until corresponding ending '/' or '?'
    compile(c);

    gp = globuf;
    while((c=getchr()) != '\n') {
        if(c == EOF)
            error(Q);
        if(c == '\\') {
            c = getchr();
            if(c != '\n')
                *gp++ = '\\';
        }
        *gp++ = c;
        if(gp >= &globuf[GBSIZE-2])
```

```
            error(Q);
        }
        if(gp == globuf)
            *gp++ = 'p';
        *gp++ = '\n';
        *gp = 0;

        for(a1=zero; a1<=dol; a1++) {
            *a1 &= ~01;
            if(a1 >= addr1 && a1 <= addr2 && match(a1) == k)
                *a1 |= 01;
        }

        ⟨global() (ed.c) if g/.../d command, call optimized gdelete() 86a⟩

        for(a1=zero; a1<=dol; a1++) {
            if(*a1 & 01) {
                *a1 &= ~01;
                dot = a1;
                globp = globuf;
                // recurse!
                commands();
                a1 = zero; // zero may have grown and move, need update a1
            }
        }
    }
```

With this in hand, the origin of `grep` should now be obvious. The classic `ed` one-liner `g/re/p` means "for every line matching the regular expression `re`, execute the `p` command (print)"—i.e., global / regular expression / print. This was such a common idiom that Ken Thompson and the early UNIX folks pulled it out into its own standalone tool, named literally after the ed command: `g` - `re` - `p` → `grep`. The original `grep` manual page even said "`grep` is derived from a `g/re/p` command in the editor `ed`."

C.12 Advanced features

The previous sections covered the core editing commands, the addressing system, and search-and-replace. This section collects the rest: shell escapes (`!`), advanced display modes (`l` for unambiguous listing, `n` for numbered output), line joining (`j`), the `browse` paginator, re-editing the remembered file (`e`), and the special treatment of append-only files. None of these introduce new mechanisms—they are mostly small additions to the command dispatcher in `commands()X`.

C.12.1 Running a shell command: `!`

The `!` command forks a child process, executes the rest of the line via `/bin/rc -c`, and waits for it to finish. In verbose mode it prints `!` when the command completes. This is less useful on Plan 9 where you typically have multiple windows, but essential on a single-terminal system.

⟨commands() *switch* c *cases (ed.c)* 77a⟩+≡ (48d) ◁76b 80c▷

```
  case '!':
      callunix();
      continue;
```

⟨globals ed.c 77b⟩+≡ (? 0—1) ◁74f 78b▷

```
  bool waiting;
```

⟨*function* `callunix`*(ed.c)* 78a⟩≡ (? 0—1)
```
void
callunix(void)
{
    int c, pid;
    Rune rune;
    char buf[512];
    char *p;

    setnoaddr();

    p = buf;
    while((c=getchr()) != EOF && c != '\n')
        if(p < &buf[sizeof(buf) - 6]) {
            rune = c;
            p += runetochar(p, &rune);
        }
    *p = '\0';

    pid = fork();
    if(pid == 0) {
        // child
        execl("/bin/rc", "rc", "-c", buf, nil);
        // should not be reached
        exits("execl failed");
    }
    // else, parent
    waiting = true;
    while(waitpid() != pid)
        ;
    waiting = false;
    if(vflag)
        putst("!");
}
```

C.12.2 Advanced listing: `l`

The `l` (list) command is like `p` but makes non-printing characters visible: tabs become `\t`, backspaces `\b`, and other control or high characters are shown as `\xNNNN` hex escapes. Long lines are folded at column 72. The `listf` flag is checked inside `putchr()X` to trigger this special rendering.

⟨globals ed.c 78b⟩+≡ (? 0—1) ◁77b 79a▷
```
// for displaying special chars, 'l' list flag
bool listf;
```

⟨commands *before* 'p' *case* 78c⟩≡ (56d)
```
case 'l':
    listf = true;
    // fallthrough:
```

⟨putchr() *if* listf 78d⟩≡ (46a)
```
if(listf) {
    if(c == '\n') {
        if(linp != line && linp[-1] == ' ') {
            *lp++ = '\\';
            *lp++ = 'n';
        }
    } else {
        if(col > (72-6-2)) {
```

```
            col = 8;
            *lp++ = '\\';
            *lp++ = '\n';
            *lp++ = '\t';
        }
        col++;
        if(c=='\b' || c=='\t' || c=='\\') {
            *lp++ = '\\';
            if(c == '\b')
                c = 'b';
            else
            if(c == '\t')
                c = 't';
            col++;
        } else
        if(c<' ' || c>='\177') {
            *lp++ = '\\';
            *lp++ = 'x';
            *lp++ =  hex[c>>12];
            *lp++ =  hex[c>>8&0xF];
            *lp++ =  hex[c>>4&0xF];
            c     =  hex[c&0xF];
            col += 5;
        }
    }
}
```

⟨globals ed.c 79a⟩+≡ (? 0—1) ◁78b 79b▷

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

⟨globals ed.c 79b⟩+≡ (? 0—1) ◁79a 80b▷

```
bool pflag;
```

The `pflag` mechanism lets commands request an automatic print after execution. For example, `s/foo/bar/p` sets `pflag`, and at the top of the `commands()X` loop, if `pflag` is set, `dot` is printed before reading the next command. The suffixes `p`, `l`, and `n` can be appended to many commands (handled in `newline()X`).

⟨`commands()` *in for loop, if* `pflag` 79c⟩≡ (48d)

```
if(pflag) {
    pflag = false;
    addr1 = addr2 = dot;
    printcom();
}
```

⟨`newline()` *if special chars* `pln` 79d⟩≡ (57a)

```
if(c == 'p' || c == 'l' || c == 'n') {
    pflag = true;
    if(c == 'l')
        listf = true;
    else
    if(c == 'n')
        listn = true;
    c = getchr();
    if(c == '\n')
        return;
}
```

The `p` suffix (as in `3p` or `/foo/p`) is the same `p` that gave `grep` its name. The script form `g/re/p`—"global, regexp, print"—is just the `g` command from the previous section composed with the `p` suffix here.

⟨`printcom()` *reset flags* 80a⟩≡ (57b)
```
listf = false;
listn = false;
pflag = false;
```

C.12.3 Advanced listing: `n`

The `n` suffix prepends the line number before each printed line. It is useful when the user is working with addresses and wants to remember which line is which without typing `=` repeatedly. Like `l`, `n` is implemented as a flag set in `newline()X` and consumed by `printcom()X`.

⟨`globals ed.c` 80b⟩+≡ (? 0—1) ◁79b 81b▷
```
// 'n' flag
bool listn;
```

⟨`commands()` *switch* `c` *cases (ed.c)* 80c⟩+≡ (48d) ◁77a 80e▷
```
case 'n':
    listn = true;
    newline();
    printcom();
    continue;
```

⟨`printcom()` *if* `listn` 80d⟩≡ (57b)
```
if(listn) {
    count = a1-zero;
    putd();
    putchr(L'\t');
}
```

C.12.4 joining lines: `j`

The `j` command concatenates multiple lines into one. It copies all lines in the range into `genbuf` end-to-end, writes the result back as a single line via `putline()X`, then deletes the extra lines with `rdelete()X`.

⟨`commands()` *switch* `c` *cases (ed.c)* 80e⟩+≡ (48d) ◁80c 81a▷
```
case 'j':
    if(!given)
        addr2++;
    newline();
    join();
    continue;
```

⟨*function* `join`*(ed.c)* 80f⟩≡ (? 0—1)
```
void
join(void)
{
    Rune *gp, *lp;
    int *a1;

    nonzero();
    gp = genbuf;
    for(a1=addr1; a1<=addr2; a1++) {
        lp = getline(*a1);
        while(*gp = *lp++)
            if(gp++ >= &genbuf[LBSIZE-sizeof(Rune)])
```

```
                error(Q);
        }
        lp = linebuf;
        gp = genbuf;
        while(*lp++ = *gp++)
            ;
        *addr1 = putline();
        if(addr1 < addr2)
            rdelete(addr1+1, addr2);
        dot = addr1;
    }
```

C.12.5 browse: b

The b (browse) command is a paging facility: it prints bpagesize lines (default 20) starting from the current address. You can go forward or backward with b+ or b-, and change the page size with b30.

⟨commands() *switch* c *cases (ed.c)* 81a⟩+≡ (48d) ◁80e 82a▷

```
    case 'b':
        nonzero();
        browse();
        continue;
```

⟨globals ed.c 81b⟩+≡ (? 0—1) ◁80b 83a▷

```
    int bpagesize = 20;
```

⟨*function* browse*(ed.c)* 81c⟩≡ (? 0—1)

```
    void
    browse(void)
    {
        int forward, n;
        static int bformat, bnum; /* 0 */

        forward = 1;
        peekc = getchr();
        if(peekc != '\n'){
            if(peekc == '-' || peekc == '+') {
                if(peekc == '-')
                    forward = 0;
                getchr();
            }
            n = getnum();
            if(n > 0)
                bpagesize = n;
        }
        newline();
        if(pflag) {
            bformat = listf;
            bnum = listn;
        } else {
            listf = bformat;
            listn = bnum;
        }
        if(forward) {
            addr1 = addr2;
            addr2 += bpagesize;
            if(addr2 > dol)
                addr2 = dol;
        } else {
```

```
            addr1 = addr2-bpagesize;
            if(addr1 <= zero)
                addr1 = zero+1;
        }
        printcom();
    }
```

C.12.6 edit remembered file: `e`

The `e` command re-initializes the editor (calling `init()X` to reset `tfile` and `zero`) and reads a new file, effectively starting a fresh editing session. `E` is the force variant that skips the unsaved-changes check.

⟨`commands()` *switch* `c` *cases (ed.c)* 82a⟩+≡ (48d) ◁81a

```
    case 'E':
        fchange = false;
        c = 'e';
        // Fallthrough
    case 'e':
        setnoaddr();
        if(vflag && fchange) {
            fchange = false;
            error(Q);
        }
        filename(c);
        init();
        addr2 = zero;
        goto caseread;
```

C.12.7 Append-only files

Plan 9 supports append-only files (the `DMAPPEND` mode bit), used for log files that can only grow. `ed` checks for this flag when opening a file for reading and prints a warning, since writing back to such a file would fail.

⟨`commands()` *other locals* 82b⟩+≡ (48d) ◁63c

```
    Dir *d;
```

⟨`commands()` *in* `r` *case if append only file* 82c⟩≡ (49c)

```
    if((d = dirfstat(io)) != nil){
        if(d->mode & DMAPPEND)
            print("warning: %s is append only\n", file);
        free(d);
    }
```

C.13 Optimizations

The simplified `putline()X` and `getline()X` shown earlier do a `seek`+`write`/`read` for every single line. The real versions use a userland block cache: `getblock()X` maintains two `BLKSIZE`-byte buffers (`ibuff` for reads, `obuff` for writes) and only performs actual I/O when the requested block is not already cached. This was essential on early UNIX systems that lacked a kernel buffer cache, and it remains a significant optimization when editing large files.

C.13.1 Optimized `putline()`

The real `putline()X` writes `linebuf` to `tfile` through `getblock()X`, handling block boundaries: when the current block is full (`nleft == 0`), it advances to the next block. The offset arithmetic with `tline` uses block-aligned addresses.

⟨globals ed.c 83a⟩+≡ (? 0—1) ◁81b 83b▷
```
int nleft;
```

⟨globals ed.c 83b⟩+≡ (? 0—1) ◁83a 84a▷
```
Rune*   linebp;
```

⟨constants ed.c 83c⟩+≡ (? 0—1) ◁76c 83e▷
```
BLKSIZE = 4096,     /* block size in temp file */
```

⟨*function* `putline`*(ed.c)* 83d⟩≡ (? 0—1)
```
/// main -> commands('r') -> append -> <>
int
putline(void)
{
    Rune *lp, *bp;
    int nl, tl;

    fchange = true;
    lp = linebuf;
    tl = tline;

    bp = getblock(tl, OWRITE);
    nl = nleft;
    tl &= ~((BLKSIZE/sizeof(Rune))-1);
    while(*bp = *lp++) {
        if(*bp++ == '\n') {
            bp[-1] = 0;
            linebp = lp;
            break;
        }
        nl -= sizeof(Rune);
        if(nl == 0) {
            tl += BLKSIZE/sizeof(Rune);
            bp = getblock(tl, OWRITE);
            nl = nleft;
        }
    }

    nl = tline;
    tline += ((lp-linebuf) + 03) & 077776;
    return nl;
}
```

C.13.2 `getblock()`

⟨constants ed.c 83e⟩+≡ (? 0—1) ◁83c 88a▷
```
NBLK    = 8191,     /* max size of temp file */
```

`getblock()X` is the heart of the block cache. It converts a line offset `atl` into a block number and an offset within that block. If the requested block is already in `ibuff` (the read cache) or `obuff` (the write cache), it returns a pointer directly into the cached buffer—no I/O needed. Otherwise, it evicts the oldest cached block (flushing `ibuff` to disk via `blkio()X` if it was modified), loads the new block, and returns a pointer into it. The `ichanged` flag is the dirty bit for `ibuff`: when a write request (`iof == OWRITE`) hits the `ibuff` cache, `ichanged`

is set via `ichanged |= iof`. Later, if `ibuff` must be evicted for a read, it is flushed first. The `obuff` path has no such flag—it is always flushed on eviction because it is only used by writes, so it is always dirty.

```
tfile (on disk):
+-----------+-----------+-----------+-----------+-----...
|  block 0  |  block 1  |  block 2  |  block 3  |
|  4096 B   |  4096 B   |  4096 B   |  4096 B   |
+-----------+-----------+-----------+-----------+-----...

getblock(atl, iof):
    bno = atl / (BLKSIZE/sizeof(Rune))    -> which block
    off = (atl*sizeof(Rune)) & (BLKSIZE-1) -> byte offset within block

In-memory caches (ed.c globals):

    ibuff[4096]  <- iblock (currently cached read block)
    obuff[4096]  <- oblock (currently cached write block)
    ichanged                (dirty bit for ibuff)

Three cases:

    1. bno == iblock      -> cache hit on read buffer
                             return &ibuff[off]
                             if iof==OWRITE, set ichanged

    2. bno == oblock      -> cache hit on write buffer
                             return &obuff[off]

    3. miss               -> evict the appropriate buffer
                             (flush if ichanged or oblock>=0),
                             read or allocate the new block,
                             return its address
```

The split between `ibuff` and `obuff` avoids ping-pong: a typical `ed` session keeps reading lines from one part of `tfile` (cached in `ibuff`) while writing modified lines to the growing tail of `tfile` (cached in `obuff`). With a single buffer the cache would constantly be evicted as the program alternates reads and writes; with two, both stay warm.

⟨globals ed.c 84a⟩+≡ (? 0—1) ◁83b 87b▷

```
int iblock;
int oblock;
int ichanged;
```

⟨init() *(ed.c) initializing globals* 84b⟩+≡ (44b) ◁75d

```
iblock = -1;
oblock = -1;
ichanged = 0;
```

⟨*function* getblock*(ed.c)* 84c⟩≡ (? 0—1)

```
/// putline | getline -> <>
Rune*
getblock(int atl, int iof)
{
```

```
    int bno; // block number
    int off; // offset

    static uchar ibuff[BLKSIZE];
    static uchar obuff[BLKSIZE];

    bno = atl / (BLKSIZE/sizeof(Rune));
    /* &~3 so the ptr is aligned to 4 (?) */
    off = (atl*sizeof(Rune)) & (BLKSIZE-1) & ~3;
    if(bno >= NBLK) {
        lastc = '\n';
        error(T);
    }
    nleft = BLKSIZE - off;
    if(bno == iblock) {
        ichanged |= iof;
        return (Rune*)(ibuff+off);
    }
    if(bno == oblock)
        return (Rune*)(obuff+off);
    if(iof == OREAD) {
        if(ichanged)
            blkio(iblock, ibuff, write);
        ichanged = 0;
        iblock = bno;
        blkio(bno, ibuff, read);
        return (Rune*)(ibuff+off);
    }
    if(oblock >= 0)
        blkio(oblock, obuff, write);
    oblock = bno;
    return (Rune*)(obuff+off);
}
```

⟨*function* `blkio` *(ed.c)* 85a⟩≡ (? 0—1)

```
void
blkio(int b, uchar *buf, long (*iofcn)(int, void *, long))
{
    seek(tfile, b*BLKSIZE, SEEK__START);
    if((*iofcn)(tfile, buf, BLKSIZE) != BLKSIZE) {
        error(T);
    }
}
```

C.13.3 Optimized `getline()`

⟨*function* `getline` *(ed.c)* 85b⟩≡ (? 0—1)

```
/// printcom | putfile -> <>
Rune*
getline(int tl)
{
    Rune *lp, *bp;
    int nl;

    lp = linebuf;
    bp = getblock(tl, OREAD);
    nl = nleft;
    tl &= ~((BLKSIZE/sizeof(Rune)) - 1);
    while(*lp++ = *bp++) {
```

```
            nl -= sizeof(Rune);
            if(nl == 0) {
                tl += BLKSIZE/sizeof(Rune);
                bp = getblock(tl, OREAD);
                nl = nleft;
            }
        }
        return linebuf;
    }
```

C.13.4 gdelete()

The generic `g/re/cmd` loop calls `commands()X` once per marked line, which for `g/re/d` would result in $O(n^2)$ shifting in `rdelete()X`. `gdelete()X` special-cases this: it makes a single pass over `zero`, copying unmarked entries down and skipping marked ones, achieving O(n) deletion.

⟨`global()` *(ed.c) if* `g/.../d` *command, call optimized* `gdelete()` 86a⟩≡ (76d)

```
    /*
     * Special case: g/.../d (avoid n^2 algorithm)
     */
    if(globuf[0] == 'd' && globuf[1] == '\n' && globuf[2] == 0) {
        gdelete();
        return;
    }
```

⟨*function* `gdelete` *(ed.c)* 86b⟩≡ (? 0—1)

```
    void
    gdelete(void)
    {
        int *a1, *a2, *a3;

        a3 = dol;
        for(a1=zero; (*a1&01)==0; a1++)
            if(a1>=a3)
                return;
        for(a2=a1+1; a2<=a3;) {
            if(*a2 & 01) {
                a2++;
                dot = a1;
            } else
                *a1++ = *a2++;
        }
        dol = a1-1;
        if(dot > dol)
            dot = dol;
        fchange = true;
    }
```

C.14 Error management

`ed`'s error handling is based on `setjmp/longjmp`, which act as a poor man's exception mechanism. Before entering `commands()X`, `main` calls `setjmp(savej)` to save the execution context. When `error()X` is called anywhere in the program, it prints `?` followed by a short message, resets global state, and calls `longjmp(savej, 1)` to jump back to just before `commands()X`, ready to read the next command. This is why `ed`'s error messages are so famously cryptic—most errors just print `?` with an empty string.

This is an instance of a pattern that shows up in many command-interpreter loops across languages, just with different plumbing. The comparison is worth spelling out because the design problem (*how do I abort a deeply nested command and return to the prompt?*) is universal, but the language-level answers vary:

```
ed (C, 1970s)          setjmp/longjmp across a stack of fn calls
sh, bash               setjmp/longjmp (builtin traps use the same)
Python REPL            raise SystemExit / KeyboardInterrupt
OCaml (utop, orc)      exception / with; see \book{Editor} orc
Go                     panic/recover at the REPL boundary
Rust                   Result<T,E> bubbled up via '?'
Common Lisp            catch/throw or the condition system
```

What they all share is a single "frame" registered at the top of the command loop where control can land no matter how deeply the current command has descended. What differs is (a) whether the mechanism can run destructor-like cleanup on the way out (C `longjmp` cannot; Python and OCaml can); and (b) whether the abort is typed, so the handler can distinguish an internal bug from a user error. `ed` has neither: every error is just `longjmp(savej, 1)` with a string, and cleanup is manual (`error_1` resets the globals by hand). The price of this simplicity is paid in `error_1()` below, which has to enumerate every piece of transient state that might be dirty.

C.14.1 error()

⟨main() *(ed.c) before* commands() 87a⟩≡ (42c)
```
setjmp(savej);
```

⟨globals ed.c 87b⟩+≡ (? 0—1) ◁84a 88b▷
```
jmp_buf savej;
```

⟨*function* error *(ed.c)* 87c⟩≡ (? 0—1)
```
void
error(char *s)
{
    error_1(s);
    longjmp(savej, 1);
}
```

⟨*function* error_1 *(ed.c)* 87d⟩≡ (? 0—1)
```
void
error_1(char *s)
{
    int c;

    ⟨error_1() (ed.c)) reset globals 87e⟩
    putchr(L'?');
    putst(s);
}
```

⟨error_1() *(ed.c)) reset globals* 87e⟩≡ (87d)
```
wrapp = false;
listf = false;
listn = false;
count = 0;

seek(STDIN, 0, SEEK__END);
pflag = false;

if(globp)
```

```
        lastc = '\n';
    globp = nil;

    peekc = lastc;
    if(lastc)
        for(;;) {
            c = getchr();
            if(c == '\n' || c == EOF)
                break;
        }

    if(io > 0) {
        close(io);
        io = -1;
    }
```

⟨constants ed.c 88a⟩+≡ (? 0—1) ◁83e

```
    EOF = -1,
```

C.14.2 Notes/signals management

Plan 9 uses notes instead of UNIX signals. On an "interrupt" note, `ed` prints a newline and jumps back to the command loop via `notejmp` (the Plan 9 equivalent of `siglongjmp`). On a "hangup" note (terminal disconnected), it calls `rescue()X`, which saves the buffer to `ed.hup` before exiting—a safety net so you do not lose unsaved work.

The difference between Plan 9 notes and UNIX signals is worth spelling out because it affects what an `ed`-like program can do in the handler. Signals on UNIX are delivered asynchronously by the kernel, identified by a small integer (`SIGINT`, `SIGHUP`, ...), and the handler runs on the current stack with very few things it is allowed to touch (see the "async-signal-safe" list in `signal(7)`). Notes on Plan 9, by contrast, are *strings* (`"interrupt"`, `"hangup"`, `"sys: ..."`) that the kernel writes to `/proc/$pid/note`; the receiving process reads them out via its registered `notifyf()X` callback and decides what to do by calling `noted(NCONT)` (resume) or `noted(NDFLT)` (default action).

```
\unix signals              \plan notes
-----------------------    -------------------------------
int id (SIGINT, SIGHUP)    string ("interrupt", "hangup")
kernel async delivery      kernel writes /proc/$pid/note
handler on the same stack  handler via notified user thread
sigjmp_buf + siglongjmp    Label + notejmp (same idea)
SA_RESTART / EINTR dance   restartable by default; noted()
limited to async-safe fns  fewer restrictions in practice
```

For `ed` specifically, three things follow from being on Plan 9. First, the handler can `strcmp` the string instead of switching on an integer, which is why `notifyf()X` starts with `strcmp(s, "interrupt")`. Second, `notejmp` replaces `siglongjmp`: it restores the `Label` saved by `setjmp` and then tells the kernel the note has been handled. Third, the `rescuing` and `waiting` flags exist to short-circuit recursive notes: if an interrupt arrives while we are already saving `ed.hup`, or while `callunix` is blocked in `waitpid`, we do not want to restart the save or kill the child—we just call `noted(NCONT)` and resume.

⟨globals ed.c 88b⟩+≡ (? 0—1) ◁87b

```
    bool rescuing;
```

⟨*function* `notifyf` *(ed.c)* 89a⟩≡ (? 0—1)

```
void
notifyf(void *a, char *s)
{
    if(strcmp(s, "interrupt") == ORD__EQ){
        if(rescuing || waiting)
            noted(NCONT);
        putchr(L'\n');
        lastc = '\n';
        error_1(Q);
        notejmp(a, savej, 0);
    }
    if(strcmp(s, "hangup") == ORD__EQ){
        if(rescuing)
            noted(NDFLT);
        rescue();
    }
    fprint(STDERR, "ed: note: %s\n", s);
    abort();
}
```

⟨*function* `rescue` *(ed.c)* 89b⟩≡ (? 0—1)

```
void
rescue(void)
{
    rescuing = true;
    if(dol > zero) {
        addr1 = zero+1;
        addr2 = dol;
        io = create("ed.hup", OWRITE, 0666);
        if(io > 0){
            Binit(&iobuf, io, OWRITE);
            putfile();
        }
    }
    fchange = false;
    quit();
}
```

⟨*function* `regerror` *(ed.c)* 89c⟩≡ (? 0—1)

```
void
regerror(char *s)
{
    USED(s);
    error(Q);
}
```

⟨*function* `onquit` *(ed.c)* 89d⟩≡ (? 0—1)

```
void
onquit(int sig)
{
    USED(sig);
    quit();
}
```

C.15 Index

Appendix D

OCaml Literate Program Example: `oed`

D.1 Introduction

`oed` is an OCaml port of the Plan 9`ed` line editor (see the previous chapter). The overall architecture is the same—temporary file as backing store, `zero` array of offsets, same commands—but the OCaml version benefits from algebraic types, pattern matching, proper exceptions instead of `setjmp`/`longjmp`, and a capability-based security model (the `-r` restricted mode) inspired by GNU `ed`. Because OCaml is more expressive, the code is significantly shorter and many things that required explanation in C (pointer arithmetic, bit-stealing, buffer management) simply disappear.

D.2 Core data structures

Unlike the C version where command parsing was interleaved with execution, `oed` separates lexing and parsing into proper modules. Tokens, addresses, and the environment are all defined as OCaml types with `deriving show` for debugging.

D.2.1 Token

⟨*type* `Token.t` 90a⟩≡ (? 0—1)

```
type t =
  | Spaces | Newline | EOF

  (* letter or '=' or '!' for commands. ex: 'p' *)
  | Char of char

  (* start of address tokens. ex: "1,3" => [Int 1; Comma; Int 3] *)
  | Int of int
  | Dot | Dollar
  | Comma
  | Plus | Minus
  ⟨Token.t other cases 111d⟩
```

D.2.2 Parser state

The parser uses a simple peek/consume interface over a `Lexing.lexbuf`, with `globp` providing the same "virtual input" mechanism as in C (for preloaded commands like `"r"` or `"a"`).

⟨*type* `Parser.state` 90b⟩≡ (? ? 0—1)

```
type state = {
  stdin: Lexing.lexbuf;
```

```
    mutable lookahead : Token.t option;
    ⟨Parser.state other fields 94c⟩
  }
```

⟨*signature* `Parser.init` 91a⟩≡ (? 0—1)
```
  val init: <Cap.stdin; ..> -> state
```

⟨*function* `Parser.init` 91b⟩≡ (? 0—1)
```
  let init (caps : < Cap.stdin; ..>) : state =
    { stdin = Lexing.from_channel (Console.stdin caps);
      lookahead = None;
      globp = None;
    }
```

D.2.3 Line addresses

Addresses are represented as an algebraic type rather than being parsed and evaluated in one pass. This clean separation—parse into an AST, then evaluate—is a natural OCaml idiom that would be awkward in C.

⟨*type* `Address.t` 91c⟩≡ (? ? 0—1)
```
  (* An "address" is a way to specify a line number symbolically or literally *)
  type t =
    | Current (* '.' *)
    | Last    (* '$' *)
    | Line of int (* <n> *)
    | Relative of t * int (* -, +, ^ *)
    ⟨Address.t other cases 112f⟩
```

⟨*type* `Address.range` 91d⟩≡ (? ? 0—1)
```
  (* What is parsed before a command. For instance 1,3 will be parsed as
   * { addr1 = Some (Line 1); addr2 = Line 3; given = true; ...}.
   *)
  type range = {
    addr1 : t option;
    addr2 : t;
    given : bool;
    ⟨Address.range other fields 118b⟩
  }
```

D.2.4 The environment

Where the C version used global variables scattered throughout the file, `oed` packs them all into a single `Env.t` record that is threaded through every function. This makes dependencies explicit and would allow running multiple editor instances in the same process.

⟨*type* `Env.t` 91e⟩≡ (? 0—1)
```
  (* The globals *)
  type t = {
    ⟨Env.t in/out fields 91f⟩
    ⟨Env.t temporary file fields 92c⟩
    ⟨Env.t zero field 92f⟩
    ⟨Env.t cursor fields 93a⟩
    ⟨Env.t other fields 94a⟩
    ⟨Env.t flag fields 94f⟩
  }
```

⟨`Env.t` *in/out fields* 91f⟩≡ (91e) 92a ▷
```
  (* to read the user commands from (and also line input in 'a'/'i' modes) *)
  in_: Parser.state;
```

⟨Env.t *in/out fields* 92a⟩+≡ (91e) ◁91f
```
(* stdout unless oflag is set in which case it's stderr *)
out: Out_channel_.t;
```

D.2.5 The backing store: `tfname`, `tfile`, and `tline`

⟨*constant* Env.tfname 92b⟩≡ (? 0—1)
```
let tfname = Fpath.v "/tmp/oed.scratch"
```

⟨Env.t *temporary file fields* 92c⟩≡ (91e) 92d▷
```
(* This is the opened temporary tfname file; ed backing store!
 * Note that we can't use the usual {in/out}_channel OCaml types because we
 * need to both read and write in the temporary file, hence the use of the
 * more general Unix.file_descr.
 *)
tfile : Unix_.file_descr;
```

⟨Env.t *temporary file fields* 92d⟩+≡ (91e) ◁92c
```
(* current write file offset in tfile to append new lines *)
mutable tline : tfile_offset;
```

⟨*type* Env.tfile_offset 92e⟩≡ (? 0—1)
```
(* offset in tfname file content *)
type tfile_offset = Tfile_offset of int
```

D.2.6 The lines as file offsets: `zero`

The `zero` array uses a proper `offset_and_mark` record instead of the C version's bit-stealing trick (where the low bit of the integer offset doubled as a mark flag for `g/re/cmd`). This is cleaner and eliminates the need for `tline` to start at 2 (though it still does, for historical fidelity).

⟨Env.t *zero field* 92f⟩≡ (91e)
```
(* Growing array of line offsets in tfile. It is a 1-indexed array but the 0
 * entry is used as a sentinel. This array maps lineno -> tfile_offset.
 * The mark is used to remember a matching line in g/re/x operations.
 *)
mutable zero : offset_and_mark array;
```

⟨*type* Env.offset_and_mark 92g⟩≡ (? 0—1)
```
type offset_and_mark = {
  (* offset in tfile *)
  offset: tfile_offset;
  (* used by the 'g' or 'v' commands to mark matched lines *)
  mutable mark: bool;
}
```

⟨*constant* Env.no_line 92h⟩≡ (? 0—1)
```
let no_line = { offset = Tfile_offset 0; mark = false }
```

D.2.7 Cursors: `dot` and `dol`

In OCaml, `dot` and `dol` are plain integers (line numbers) rather than pointers into the array as in C. A `lineno` type alias documents intent.

⟨*type* Env.lineno 92i⟩≡ (? 0—1)
```
(* ed uses 1-indexed line numbers, but 0 is also used as a special value.
 * alt: call it cursor?
*)
type lineno = int
```

⟨Env.t *cursor fields* 93a⟩≡ (91e) 93b▷
```
(* index entried in zero *)
(* current line *)
mutable dot: lineno;
(* last line (dollar) *)
mutable dol: lineno;
```

D.2.8 Other globals

⟨Env.t *cursor fields* 93b⟩+≡ (91e) ◁93a
```
(* for 1,3p commands. See also Address.range, but here we have
 * concrete line number, not symbolic "addresses".
 *)
mutable addr1: lineno;
mutable addr2: lineno;
mutable given: bool;
```

D.3 CLI.main()

The entry point uses the capability system: `Cap.main` provides a set of capabilities (file I/O, process creation, etc.) that are threaded through the program and can be restricted by the `-r` flag.

⟨*toplevel* Main._1 93c⟩≡ (? 0—1)
```
let _ =
  Cap.main (fun (caps : Cap.all_caps) ->
     let argv = CapSys.argv caps in
     Exit.exit caps (Exit.catch (fun () -> CLI.main caps argv))
  )
```

⟨*signature* CLI.main 93d⟩≡ (? 0—1)
```
val main: <caps; ..> ->
  string array -> Exit.t
```

⟨*type* CLI.caps 93e⟩≡ (? ? 0—1)
```
type caps = <
    Cap.stdin; Cap.stdout; Cap.stderr;
    Cap.open_in; (* for 'r' *)
    Cap.open_out; (* for 'w' *)
    Cap.forkew; (* for '!' *)
  >
```

Here is the `mainX` skeleton. The `while true` loop replaces the C version's `setjmp/longjmp` error recovery: when an `Error.Error` exception is raised anywhere during command processing, it is caught here, the error message is printed, and the loop continues with the next command.

⟨*function* CLI.main 93f⟩≡ (? 0—1)
```
let main (caps : <caps; ..>) (argv : string array) : Exit.t =

  let args = ref [] in
  ⟨CLI.main() local flags 94d⟩

  let options = [
     ⟨CLI.main() local options elements 94e⟩
  ] |> Arg.align
  in
  (* may raise ExitCode *)
  Arg_.parse_argv caps argv options (fun t -> args := t::!args)
    (spf "usage: %s [options] [file]" argv.(0));
```

```
⟨CLI.main() setup logging after parsed argv 118h⟩
⟨CLI.main() restrict caps 116g⟩

let env : Env.t = Env.init caps !vflag !oflag !rflag in
⟨CLI.main() env adjustments 94b⟩

while true do
  (* when neither commands() nor quit() raise Error, then
   * quit() will proceed and raise Exit.ExitCode which
   * will exit this loop (and be caught in Main._)
   *)
  try (
      commands caps env;
      Commands.quit caps env;
  )
  with Error.Error s ->
      ⟨CLI.main() exn handler for Error.Error s 118e⟩
done;
(* should never reach *)
Exit.OK
```

D.3.1 `ed <file>` and preloaded commands `globp`

```
⟨Env.t other fields 94a⟩≡                                    (91e) 96e▷
  (* for 'w', 'r', 'f' *)
  mutable savedfile: Fpath.t option;
```

```
⟨CLI.main() env adjustments 94b⟩≡                                (93f)
  (match !args with
  | [] -> ()
  | [file] ->
      env.savedfile <- Some (Fpath.v file);
      env.in_.globp <- Some (Lexing.from_string "r")
  (* stricter: *)
  | _::_::_ -> failwith "too many arguments"
  );
  ⟨CLI.main() env adjustments if oflag 95f⟩
  ⟨CLI.main() debug env 119i⟩
```

```
⟨Parser.state other fields 94c⟩≡                                  (90b)
  (* for inserting "virtual" commands to process before stdin *)
  mutable globp: Lexing.lexbuf option;
```

D.3.2 `ed -` and `vflag`

```
⟨CLI.main() local flags 94d⟩≡                                (93f) 95a▷
  (* "verbose(interactive)" mode is set by default *)
  let vflag = ref true in
```

```
⟨CLI.main() local options elements 94e⟩≡                     (93f) 95b▷
  "-", Arg.Clear vflag,
  " non-interactive mode (opposite of verbose)";
```

```
⟨Env.t flag fields 94f⟩≡                                     (91e) 95c▷
  (* verbose (a.k.a. interactive) flag, cleared by 'ed -' *)
  vflag: bool;
```

D.3.3 `ed -o` and `oflag`

⟨`CLI.main()` *local flags* 95a⟩+≡ (93f) ◁94d 116d▷
```
(* when '-o', the command 'w' will write to stdout (useful for filters) *)
let oflag = ref false in
```

⟨`CLI.main()` *local* `options` *elements* 95b⟩+≡ (93f) ◁94e 116e▷
```
"-o", Arg.Set oflag,
" write buffer to standard output";
```

⟨`Env.t` *flag fields* 95c⟩+≡ (91e) ◁94f 116f▷
```
(* output flag, set by 'ed -o'.
 * used just in Out.putchr() so could be removed almost.
 *)
oflag: bool;
```

⟨`Env.init()` *set local* `savedfile` 95d⟩≡ (95g)
```
(* will be overwritten possibly in the caller by argv[1]
 * TODO: works on Linux? /fd/1 exists?
 *)
let savedfile = if oflag then Some (Fpath.v "/fd/1") else None in
```

⟨`Env.init()` *set local* `out` 95e⟩≡ (95g)
```
let out = if oflag then Console.stderr caps else Console.stdout caps in
```

⟨`CLI.main()` *env adjustments if* `oflag` 95f⟩≡ (94b)
```
if !oflag then env.in_.globp <- Some (Lexing.from_string "a");
```

D.3.4 `init()`

`Env.initX` creates the environment record. Unlike the C version, there is no separate `init()X` to re-create the temporary file on `e` commands—`oed` does not yet support the `e` command.

⟨*function* `Env.init` 95g⟩≡ (? 0—1)
```
let init (caps : < Cap.stdin; Cap.stdout; Cap.stderr; ..>)
     (vflag : bool) (oflag : bool) (rflag:  bool) : t =

  ⟨Env.init() set local out 95e⟩
  ⟨Env.init() set local savedfile 95d⟩
  {
    in_ = Parser.init caps;
    out;
    ⟨Env.init() tfile field 96a⟩
    ⟨Env.init() tline field 96b⟩

    zero = Array.make 10 no_line;
    dot = 0;
    dol = 0;
    addr1 = 0;
    addr2 = 0;
    given = false;

    savedfile;
    fchange = false;
    wrapp = false;
    count = 0;
    pflag = false;
    col = 0;

    vflag = if oflag then false else vflag;
```

```
    oflag;
    rflag;
  }
```

⟨Env.init() tfile *field* 96a⟩≡ (95g)
```
tfile =
  (try
    Unix.openfile !!tfname [ Unix.O_RDWR; Unix.O_CREAT ] 0o600
  with Unix.Unix_error (err, s1, s2) ->
    Logs.err (fun m -> m "%s %s %s" (Unix.error_message err) s1 s2);
    (* alt: just no try and rely on default exn and backtrace *)
    (* alt: call Out.putxxx funcs but mutual recursion *)
    output_string out "?TMP\n";
    (* ed was doing exits(nil) = exit 0 so we do the same *)
    raise (Exit.ExitCode 0)
   );
```

⟨Env.init() tline *field* 96b⟩≡ (95g)
```
(* sentinel value so that file offsets 0 and 1 are reserved and no
 * real line offsets in zero[] can have those values
 * TODO? mark is using a separate bool field now so we could
 * use Tfile_offset 1 too and so start at 1 (or even 0?) now.
 *)
tline = Tfile_offset 2;
```

D.3.5 quit()

⟨*signature* Commands.quit 96c⟩≡ (? 0—1)
```
(* 'q' (need open_out to remove Env.tfname from the filesystem) *)
val quit: < Cap.open_out; ..> -> Env.t -> unit
```

⟨*function* Commands.quit 96d⟩≡ (? 0—1)
```
(* 'q' *)
let quit (caps : <Cap.open_out; ..>) (e : Env.t) : unit =
  ⟨Commands.quit() check if modified buffer 96f⟩
  (* alt: could also Unix.close e.tfile *)
  FS.remove caps Env.tfname;
  raise (Exit.ExitCode 0)
```

⟨Env.t *other fields* 96e⟩+≡ (91e) ◁94a 97e▷
```
(* did the buffer changed (mostly tested with dol > 0) *)
mutable fchange: bool;
```

⟨Commands.quit() *check if modified buffer* 96f⟩≡ (96d)
```
if e.vflag && e.fchange && e.dol != 0 then begin
    (* so a second quit will actually quit *)
    e.fchange <- false;
    Error.e_warn "trying to quit with modified buffer"
end;
```

D.4 Displaying and reading text

The output layer is much simpler than in C: OCaml's standard `output_char`/`output_string` replace the hand-rolled buffering in `line[70]`. The `e.out` channel is either `stdout` or `stderr` depending on the `-o` flag.

D.4.1 Displaying text

⟨*signature* `Out.putchr` 97a⟩≡ (? 0—1)
```
val putchr: Env.t -> char -> unit
```

⟨*function* `Out.putchr` 97b⟩≡ (? 0—1)
```
let putchr (e : Env.t) (c : char) =
  (* TODO: if listf *)
  output_char e.out c;
  if c = '\n' then flush e.out
```

⟨*signature* `Out.putst` 97c⟩≡ (? 0—1)
```
val putst: Env.t -> string -> unit
```

⟨*function* `Out.putst` 97d⟩≡ (? 0—1)
```
(* pre: str should not contain '\n' ? *)
let putst (e : Env.t) (str : string) : unit =
  (* ugly? should set after putchr \n? also who uses col? *)
  e.col <- 0;
  (* iterate over str and call putchr to get a chance
   * for the listf code above
   *)
  String.iter (putchr e) str;
  putchr e '\n';
  ()
```

⟨`Env.t` *other fields* 97e⟩+≡ (91e) ◁96e 101d▷
```
(* ?? what functions rely on column number set? *)
mutable col: int;
```

D.4.2 Reading text

Reading is also simpler: `In.getyX` delegates to an `ocamllex` rule that reads until a newline, and `In.getttyX` wraps it with the `.` terminator check, returning `None` instead of `EOF`. The callback signature `unit -> string option` is the OCaml equivalent of the C function-pointer-returning-int pattern used by `append()X`.

⟨*signature* `In.gety` 97f⟩≡ (? 0—1)
```
(* return a line (without trailing '\n') *)
val gety :  Env.t -> string
```

⟨*function* `In.gety` 97g⟩≡ (? 0—1)
```
(* return a line (without trailing '\n') *)
let gety (e : Env.t) : string =
  Lexer.line e.in_.stdin
```

⟨*signature* `Lexer.line` 97h⟩≡ (? 0—1)
```
val line: Lexing.lexbuf -> string
```

⟨*function* `Lexer.line` 97i⟩≡ (? 0—1)
```
and line = parse
  | ([^ '\n' ]* as s) '\n' { s }
  | eof { failwith "eof in Lexer.line()" (* alt: None? *) }
```

⟨*signature* `In.gettty` 97j⟩≡ (? 0—1)
```
(* Used to read a set of lines from stdin until a single "." on a line
 * is entered marking the end of user text input.
 *)
val gettty : Env.t -> (unit -> string option)
```

⟨*function* In.gettty 98a⟩≡ (? 0—1)
```
(* Read a line from stdin. Return None when the user entered "." on a single
 * line meaning the end of interactive input.
 * This has a similar interface to getfile() so it can be passed to
 * append().
 *)
let gettty (e : Env.t) () : string option =
  let s = gety e in
  if s = "."
  then begin
    ⟨In.gettty() log end of input 119h⟩
    None
  end
  else Some s
```

D.5 Parsing commands

The lexer is written using `ocamllex`. `Parser.next_tokenX` checks `globp` first (for preloaded commands), falling back to the real `stdin` lexer—the same layering as `getchr()X` in C, but expressed more cleanly with `Option` types.

⟨*signature* Lexer.token 98b⟩≡ (? 0—1)
```
val token: Lexing.lexbuf -> Token.t
```

⟨*constant* Lexer.space 98c⟩≡ (? 0—1)
```
let space = [' ''\t']
```

⟨*constant* Lexer.letter 98d⟩≡ (? 0—1)
```
let letter = ['a'-'z''A'-'Z''_']
```

⟨*constant* Lexer.digit 98e⟩≡ (? 0—1)
```
let digit = ['0'-'9']
```

⟨*function* Lexer.token 98f⟩≡ (? 0—1)
```
rule token = parse
  | space+        { Spaces }
  | '\n'          { Newline }

  (* for the command *)
  | (letter | '=' | '!') as c   { Char c }

  ⟨Lexer.token() other cases 109b⟩
  | eof { EOF }
```

⟨*function* Parser.next_token 98g⟩≡ (? 0—1)
```
(* Do not use! this is internal! You should use peek() or consume() instead. *)
let next_token (st : state) : Token.t =
  let t =
    match st.globp with
    ⟨Parser.next_token() match globp cases 99d⟩
    | None -> Lexer.token st.stdin
  in
  ⟨Parser.next_token() debug token 119j⟩
  t
```

⟨*signature* Parser.peek 98h⟩≡ (? 0—1)
```
val peek : state -> Token.t
```

⟨*function* `Parser.peek` 99a⟩≡ (? 0—1)
```
let peek (st : state) : Token.t =
  match st.lookahead with
  | Some t -> t
  | None ->
      let t = next_token st in
      st.lookahead <- Some t;
      t
```

⟨*signature* `Parser.consume` 99b⟩≡ (? 0—1)
```
val consume: state -> Token.t
```

⟨*function* `Parser.consume` 99c⟩≡ (? 0—1)
```
let consume (st : state) : Token.t =
  match st.lookahead with
  | Some t -> st.lookahead <- None; t
  | None -> next_token st
```

⟨`Parser.next_token()` *match* `globp` *cases* 99d⟩≡ (98g)
```
| Some lexbuf ->
    let t = Lexer.token lexbuf in
    if t = T.EOF
    then st.globp <- None;
    t
```

D.6 `CLI.commands()` interpreter loop

The main loop first parses the address range into an `Address.range` AST, evaluates it to concrete line numbers, then dispatches on the command character via nested `match`. The C version used `continue` to loop and `error(Q)` as default; here the loop is a `while` and unknown commands raise a `failwith`.

⟨*function* `CLI.commands` 99e⟩≡ (? 0—1)
```
let rec commands (caps : < Cap.open_in; Cap.open_out; Cap.forkew; ..>)
                 (e : Env.t) : unit =
  ⟨CLI.commands() debug start 119l⟩
  let done_ = ref false in
  while not !done_ do
    ⟨CLI.commands(), at loop start, if pflag 107b⟩
    let range : Address.range = Address.parse_range e.in_ in
    let (addr1, addr2) = Address.eval_range e range in
    (* TODO: use range.set_dot! *)
    e.addr1 <- addr1;
    e.addr2 <- addr2;
    e.given <- range.given;

    (match Parser.consume e.in_ with
    | T.Char c ->
      (match c with
      (* inspecting *)
      ⟨CLI.commands() match c inspecting cases 106a⟩
      (* reading *)
      ⟨CLI.commands() match c reading case 100c⟩
      (* writing *)
      ⟨CLI.commands() match c writing cases 104e⟩
      (* modifying *)
      ⟨CLI.commands() match c modifying cases 108a⟩
      (* globals *)
      ⟨CLI.commands() match c global cases 115a⟩
```

```
      (* other *)
      ⟨CLI.commands() match c other cases 108e⟩

      | c -> failwith (spf "unsupported command '%c'" c)
      )
      (* ed: was doing error(Q) here but because ed relied on the commands
       * doing some 'continue' which we can't in OCaml so
       * better not use Error.e here
       *)

    | T.Newline ->
        ⟨CLI.commands() Newline case 107a⟩

    | T.EOF ->
       (* old: raise (Exit.ExitCode 0) but bad because we need to get to quit()
        * or to return to global() caller when nested call to commands().
        * TODO? raise (Exit.ExitCode 2) instead and check whether in nested
        * commands? so this would force people to use 'q' to quit cleanly
        * (so an error during quit in a script would fail and would
        * make the whole script fail when EOF is reached)
        * or raise EOF and capture it in global() ?
        *)
       done_ := true
    | t -> Parser.was_expecting_but_got "a letter" t
    )
  done;
  ⟨CLI.commands() debug end 119m⟩
```

⟨*signature* Address.parse_range 100a⟩≡ (? 0—1)
```
val parse_range: Parser.state -> range
```

⟨*signature* Address.eval_range 100b⟩≡ (? 0—1)
```
val eval_range: Env.t -> range -> Env.lineno * Env.lineno
```

D.7 reading a file: r

The reading pipeline is the same as in C: `In.filenameX` parses the filename, `Commands.readX` opens the file and calls `append e (Disk.getfile e chan) e.addr2`. The use of `FS.with_open_in` (a capability-aware file opener) ensures the file is properly closed even on errors.

⟨CLI.commands() *match* c *reading case* 100c⟩≡ (99e)
```
| 'r' ->
    let file : Fpath.t = In.filename e c in
    Commands.read caps e file
```

⟨*signature* In.filename 100d⟩≡ (? 0—1)
```
(* read a filename from stdin or from Env.savedfile otherwise *)
val filename: Env.t -> char (* 'f' or 'e' or 'r' or ? *) -> Fpath.t
```

⟨*signature* Commands.read 100e⟩≡ (? 0—1)
```
(* 'r' *)
val read: <Cap.open_in; ..> -> Env.t -> Fpath.t -> unit
```

D.7.1 Reading a `filename()`

⟨*function* `In.filename` 101a⟩≡ (? 0—1)
```
let filename (e : Env.t) (cmd : char) : Fpath.t =
  ⟨In.filename() reset count 101e⟩
  match Parser.consume e.in_ with
  | T.Newline | T.EOF ->
      (* no file specified, use maybe e.savedfile then *)
      (match e.savedfile with
      | Some file -> file
      | None when cmd <> 'f' -> Error.e_err "no savedfile and no filename given"
      | None -> failwith "TODO?? what does ed in that case?"
      )
  | T.Spaces ->
      let str = Lexer.filename e.in_.stdin in
      if str = ""
      then Parser.was_expecting "a non empty filename";
      (match Parser.consume e.in_ with
      | T.Newline ->
            let file = Fpath.v str in
            if e.savedfile = None || cmd = 'e' || cmd = 'f'
            then e.savedfile <- Some file;
            file
      | t -> Parser.was_expecting_but_got "a newline" t
      )
  | t -> Parser.was_expecting_but_got "a newline or space and filename" t
```

⟨*signature* `Lexer.filename` 101b⟩≡ (? 0—1)
```
val filename: Lexing.lexbuf -> string
```

⟨*function* `Lexer.filename` 101c⟩≡ (? 0—1)
```
and filename = parse
  | [^ '\n' ' ']* { Lexing.lexeme lexbuf }
  | eof { failwith "eof in Lexer.filename()" }
```

⟨`Env.t` *other fields* 101d⟩+≡ (91e) ◁97e 104f▷
```
(* count #chars read, or number of lines; displayed by Out.putd() *)
mutable count: int;
```

⟨`In.filename()` *reset* `count` 101e⟩≡ (101a)
```
(* alt: do it in the caller, clearer; will be incremented
 * when reading the file in getfile
 *)
e.count <- 0;
```

D.7.2 `read()`

⟨*function* `Commands.read` 101f⟩≡ (? 0—1)
```
(* 'r' *)
let read (caps : < Cap.open_in; .. >) (e : Env.t) (file : Fpath.t) : unit =
  ⟨Commands.read() restricted mode check 117d⟩
  try
    file |> FS.with_open_in caps (fun chan ->
        setwide e;
        squeeze e 0;
        let change = (e.dol != 0) in
        append e (Disk.getfile e chan) e.addr2 |> ignore;
        exfile e READ;
        e.fchange <- change;
```

```
    )
  with Sys_error str ->
    Logs.err (fun m -> m "Sys_error: %s" str);
    Error.e_legacy !!file
```

⟨*signature* Commands.setwide 102a⟩≡ (? 0—1)
```
(* helpers *)
val setwide: Env.t -> unit
```

⟨*signature* Commands.squeeze 102b⟩≡ (? 0—1)
```
val squeeze: Env.t -> int -> unit
```

⟨*signature* Disk.getfile 102c⟩≡ (? 0—1)
```
(* will return one line (without trailing '\n') or None when reached EOF *)
val getfile: Env.t -> Chan.i -> (unit -> string option)
```

⟨*signature* Commands.append 102d⟩≡ (? 0—1)
```
(* return number of lines added, but usually ignored by caller *)
val append: Env.t -> (unit -> string option) -> Env.lineno -> int
```

D.7.3 setwide() and squeeze()

⟨*function* Commands.setwide 102e⟩≡ (? 0—1)
```
let setwide (e : Env.t) : unit =
  if not e.given then begin
    e.addr1 <- if e.dol > 0 then 1 else 0;
    e.addr2 <- e.dol;
  end;
  ()
```

⟨*function* Commands.squeeze 102f⟩≡ (? 0—1)
```
let squeeze (e : Env.t) (i : lineno) : unit =
  if e.addr1 < i || e.addr2 > e.dol || e.addr1 > e.addr2
  then Error.e_warn "can't squeeze"
```

D.7.4 append() and getfile()

⟨*function* Commands.append 102g⟩≡ (? 0—1)
```
(* f can be Disk.getfile() or In.gettty() (or even getsub()?) *)
let append (e : Env.t) (f : unit -> string option) (addr : lineno) : int =
  e.dot <- addr;
  let nline = ref 0 in

  let rec aux () =
    match f () with
    | None -> (* EOF *) !nline
    | Some line ->
        ⟨Commands.append() grow zero if needed 103b⟩
        let tl = Disk.putline e line in
        incr nline;
        e.dol <- e.dol + 1;
        (* insert new line after e.dot *)
        e.dot <- e.dot + 1;

        (* shift zero entries from e.dol down to e.dot up one by one *)
        let a1 = ref e.dol in
        let a2 = ref (!a1 + 1) in
        while !a1 > e.dot do
```

```
          e.zero.(!a2) <- e.zero.(!a1);
          decr a2; decr a1;
        done;
        (* finally insert the new line reference in zero *)
        e.zero.(e.dot) <- {offset = tl; mark = false};
        aux ()
  in
  aux ()
```

⟨*signature* `Disk.putline` 103a⟩≡ (? 0—1)

```
(* store line (with added trailing '\n') in tfile and return its offset *)
val putline : Env.t -> string -> Env.tfile_offset
```

⟨`Commands.append()` *grow* `zero` *if needed* 103b⟩≡ (102g)

```
if e.dol + 2 >= Array.length e.zero
then begin
    let oldz = e.zero in
    let len = Array.length oldz in
    let newz  = Array.make (len + 512) Env.no_line in
    Array.blit oldz 0 newz 0 len;
    e.zero <- newz;
end;
```

⟨*function* `Disk.getfile` 103c⟩≡ (? 0—1)

```
(* will return one line (without trailing '\n') or None when reached EOF *)
let getfile (e : Env.t) (chan : Chan.i) () : string option =
  (* alt: use Stdlib.input_line which does some extra magic around newlines
   * and EOF we want, because ed also uniformize the lack of newline before EOF
   * (see the "\\n appended" message below),but we want to to match exactly what
   * ed does and display the same error message so we need to go lower level
   * than input_line and use input_char directly.
   *)
  try
    (* 's' will not have the trailing '\n' *)
    let s = input_line chan.ic in
    e.count <- e.count + String.length s + 1 (* to count the new line *);
    Some s
  with End_of_file -> None
```

D.7.5 `putline()`

The OCaml `Disk.putlineX` and `Disk.getlineX` are straightforward: no block cache, no `getblock()` indirection—just `Unix.lseek` followed by `Unix.write` or character-by-character `Unix.read`. The block-based I/O optimization from C was dropped; on modern systems the kernel buffer cache makes it unnecessary.

⟨*function* `Disk.putline` 103d⟩≡ (? 0—1)

```
(* store line in tfile and return its offset *)
let putline (e : Env.t) (line : string) : Env.tfile_offset =
  e.fchange <- true;
  let Tfile_offset old_tline = e.tline in
  Unix.lseek e.tfile old_tline Unix.SEEK_SET |> ignore;
  (* alt: could use a different terminator like '\0' in C but simpler to
   * use \n *)
  let line = line ^ "\n" in
  let len = String.length line in
  Unix.write e.tfile (Bytes.of_string line) 0 len |> ignore;
  e.tline <- Tfile_offset (old_tline + len);
  Tfile_offset old_tline
```

D.7.6 `exfile()`

⟨*type* `Commands.mode` 104a⟩≡ (? 0—1)
```
type mode = READ | WRITE
```

⟨*function* `Commands.exfile` 104b⟩≡ (? 0—1)
```
(* ed: "exit" file =~ close file *)
let exfile (e : Env.t) (_m : mode) : unit =
  (* ed: is using passed mode to flush if WRITE but no need in ocaml
   * because closing the channel will flush any remaining IO.
   *)
  if e.vflag then begin
      Out.putd e;
      Out.putchr e '\n';
  end
```

⟨*signature* `Out.putd` 104c⟩≡ (? 0—1)
```
(* will print Env.count *)
val putd: Env.t -> unit
```

⟨*function* `Out.putd` 104d⟩≡ (? 0—1)
```
(* display e.count *)
let rec putd (e : Env.t) : unit =
  let r = e.count mod 10 in
  e.count <- e.count / 10;
  if e.count > 0
  then putd e;
  putchr e (Char.chr (Char.code '0' + r))
```

D.8 writing a file: `w`

Writing follows the same structure as in C: `Disk.putfileX` iterates over the address range, calls `Disk.getlineX` for each line, and writes to the output channel. The OCaml `for` loop replaces the C `do/while` with pointer arithmetic.

⟨`CLI.commands()` *match* `c` *writing cases* 104e⟩≡ (99e)
```
| 'w' | 'W' ->
    if c = 'W' then e.wrapp <- true;
    (* TODO: if [wW][qQ] *)
    let file : Fpath.t = In.filename e c in
    Commands.write caps e file;
```

⟨`Env.t` *other fields* 104f⟩+≡ (91e) ◁101d 107c▷
```
(* write append, for 'W' *)
mutable wrapp : bool;
```

⟨*signature* `Commands.write` 104g⟩≡ (? 0—1)
```
(* 'w'*)
val write: <Cap.open_out; ..> -> Env.t -> Fpath.t -> unit
```

⟨*function* `Commands.write` 104h⟩≡ (? 0—1)
```
(* 'w' *)
let write (caps : < Cap.open_out; ..>) (e : Env.t) (file : Fpath.t) : unit =
  ⟨Commands.write() restricted mode check 117e⟩
  try
    file |> FS.with_open_out caps (fun chan ->
        (* TODO: when wq (or do in caller in CLI.ml) *)
        setwide e;
        squeeze e (if e.dol > 0 then 1 else 0);
```

```
        (* TODO: e.wrapp open without create mode? *)
        e.wrapp <- false;
        if e.dol > 0
        then Disk.putfile e chan;

        exfile e WRITE;
        if e.addr1 <= 1 && e.addr2 = e.dol
        then e.fchange <- false;
        (* TODO: when wq *)
    )
  with Sys_error str ->
    Logs.err (fun m -> m "Sys_error: %s" str);
    Error.e_legacy !!file
```

⟨*signature* Disk.putfile 105a⟩≡ (? 0—1)

```
(* dual of getfile() but this time writing all the lines, not just one *)
val putfile: Env.t -> Chan.o -> unit
```

D.8.1 putfile()

⟨*function* Disk.putfile 105b⟩≡ (? 0—1)

```
(* dual of getfile() but this time writing all the lines, not just one *)
let putfile (e : Env.t) (chan : Chan.o) : unit =
  for a1 = e.addr1 to e.addr2 do
    let l = getline e a1 ^ "\n" in
    e.count <- e.count + String.length l;
    output_string chan.oc l;
  done
```

⟨*signature* Disk.getline 105c⟩≡ (? 0—1)

```
(* retrieve line in tfile (without trailing '\n') *)
val getline: Env.t -> Env.lineno -> string
```

D.8.2 getline()

⟨*function* Disk.getline 105d⟩≡ (? 0—1)

```
(* dual of putline(), retrieve line in tfile (without trailing '\n')
 * ed: was taking an Env.tfile_offset but cleaner to take addr
 *)
let getline (e : Env.t) (addr : Env.lineno)  : string =
  let tl = e.zero.(addr).offset in
  let Tfile_offset offset = tl in
  Unix.lseek e.tfile offset Unix.SEEK_SET |> ignore;
  (* alt: Stdlib.input_line (Unix.in_channel_of_descr ...) but then
   * need to close it which unfortunately also close the file_descr so
   * we do our own adhoc input_line below.
   *)
  let bytes = Bytes.of_string " " in
  let rec aux acc =
    let n = Unix.read e.tfile bytes 0 1 in
    let c : char = Bytes.get bytes 0 in
    if n = 1 && c <> '\n'
    then aux (c::acc)
    (* no need to add the \n, putshst will add it *)
    else String_.of_chars (List.rev acc)
  in
  aux []
```

D.9 Main commands

The main commands are direct translations from C. Most are just a few lines calling the same helper functions.

D.9.1 printing lines: p

⟨CLI.commands() *match* c *inspecting cases* 106a⟩≡ (99e) 107d ▷
```
| 'p' | 'P' ->
    In.newline e;
    Commands.printcom e;
```

⟨*signature* Commands.printcom 106b⟩≡ (? 0—1)
```
(* 'p' *)
val printcom : Env.t -> unit
```

⟨*signature* In.newline 106c⟩≡ (? 0—1)
```
(* check the next token is a newline (or EOF) and consume it *)
val newline: Env.t -> unit
```

⟨*function* In.newline 106d⟩≡ (? 0—1)
```
let newline (e : Env.t) : unit =
  match Parser.consume e.in_ with
  | T.Newline -> ()
  (* tricky but is useful to treat EOF as a newline sometimes
   * like for globp "r" but sometimes not in g/re/x globp context so that
   * it is consumed by commands() leading to returning from commands().
   *)
  | T.EOF -> ()
  (* TODO: if special chars pln ? *)
  | t -> Parser.was_expecting_but_got "newline" t
```

⟨*function* Commands.printcom 106e⟩≡ (? 0—1)
```
(* 'p' *)
let printcom (e : Env.t) : unit =
  nonzero e;
  for a1 = e.addr1 to e.addr2 do
    (* TODO: if listn *)
    Out.putshst e (Disk.getline e a1);
  done;
  e.dot <- e.addr2;
  (* TODO: reset flags *)
  ()
```

⟨*signature* Out.putshst 106f⟩≡ (? 0—1)
```
(* ed: put "shell" string, legacy name, identical to putst for oed *)
val putshst : Env.t -> string -> unit
```

⟨*signature* Commands.nonzero 106g⟩≡ (? 0—1)
```
val nonzero: Env.t -> unit
```

⟨*function* Commands.nonzero 106h⟩≡ (? 0—1)
```
let nonzero (e : Env.t) =
  squeeze e 1
```

⟨*function* Out.putshst 106i⟩≡ (? 0—1)
```
(* origin: put shell string? *)
let putshst (e : Env.t) (str : string) : unit =
  (* no diff between rune and chars in oed *)
  putst e str
```

⟨CLI.commands() Newline *case* 107a⟩≡ (99e)
```
(* print when no command specified, as in 1\n *)

(* ed: was a1 == nil but simpler to look at given *)
if not range.given then begin
  (* so any subsequent newline will display a successive line *)
  let a1 = e.dot + 1 in
  e.addr2 <- a1;
  e.addr1 <- a1;
end;
(* note that printcom() will internally set e.dot to e.addr2
 * TODO: if lastsep = ';' *)
Commands.printcom e;
```

⟨CLI.commands(), *at loop start, if* pflag 107b⟩≡ (99e)
```
if e.pflag then begin
    e.pflag <- false;
    e.addr1 <- e.dot;
    e.addr2 <- e.dot;
    Commands.printcom e;
end;
```

⟨Env.t *other fields* 107c⟩+≡ (91e) ◁104f
```
(* set by ?? effect is to Out.printcom() in commands () before the next cmd *)
mutable pflag: bool;
```

D.9.2 printing remembered file: f

⟨CLI.commands() *match* c *inspecting cases* 107d⟩+≡ (99e) ◁106a 107g▷
```
| 'f' ->
   (* alt: move in Commands.file() *)
   Commands.setnoaddr e;
   let file : Fpath.t = In.filename e c in
   assert (e.savedfile = Some file);
   Out.putst e !!file;
```

⟨*signature* Commands.setnoaddr 107e⟩≡ (? 0—1)
```
val setnoaddr: Env.t -> unit
```

⟨*function* Commands.setnoaddr 107f⟩≡ (? 0—1)
```
let setnoaddr (e : Env.t) =
  if e.given
  then Error.e_err "setnoaddr ??"
```

D.9.3 printing line number: =

⟨CLI.commands() *match* c *inspecting cases* 107g⟩+≡ (99e) ◁107d 119n▷
```
| '=' ->
   (* alt: move in Commands.print_dot_line_number() *)
   Commands.setwide e;
   Commands.squeeze e 0;
   In.newline e;
   e.count <- e.addr2;
   Out.putd e;
   Out.putchr e '\n';
```

D.9.4 append and insert: a, i

⟨CLI.commands() *match* c *modifying cases* 108a⟩≡ (99e) 108b▷
```
| 'a' ->
    ⟨CLI.commands() in 'a' case, log append mode 119f⟩
    Commands.add e 0
```

⟨CLI.commands() *match* c *modifying cases* 108b⟩+≡ (99e) ◁108a 108f▷
```
| 'i' ->
    ⟨CLI.commands() in 'i' case, log insert mode 119g⟩
    Commands.add e (-1)
```

⟨*signature* Commands.add 108c⟩≡ (? 0—1)
```
(* 'a' and 'i' *)
val add: Env.t -> int -> unit
```

⟨*function* Commands.add 108d⟩≡ (? 0—1)
```
(* used for 'a' and 'i' *)
let add (e : Env.t) (i : int) =
  if i <> 0 && (e.given || e.dol > 0) then begin
     e.addr1 <- e.addr1 - 1;
     e.addr2 <- e.addr2 - 1;
  end;
  squeeze e 0;
  In.newline e;
  append e (In.gettty e) e.addr2 |> ignore
```

D.9.5 quitting: q

⟨CLI.commands() *match* c *other cases* 108e⟩≡ (99e) 116a▷
```
| 'q' | 'Q' ->
    if c = 'Q' then e.fchange <- false;
    Commands.setnoaddr e;
    In.newline e;
    Commands.quit caps e;
```

D.9.6 deleting lines: d

⟨CLI.commands() *match* c *modifying cases* 108f⟩+≡ (99e) ◁108b 109a▷
```
| 'd' ->
    Commands.nonzero e;
    In.newline e;
    Commands.rdelete e e.addr1 e.addr2;
```

⟨*signature* Commands.rdelete 108g⟩≡ (? 0—1)
```
(* 'd' and 'c' *)
val rdelete: Env.t -> Env.lineno -> Env.lineno -> unit
```

⟨*function* Commands.rdelete 108h⟩≡ (? 0—1)
```
(* used for 'r' and 'c' *)
let rdelete (e : Env.t) (ad1 : lineno) (ad2 : lineno) =
  let a1 = ref ad1 in
  let a2 = ref (ad2 + 1) in
  let a3 = e.dol in
  e.dol <- e.dol - (!a2 - !a1);
  let rec aux () =
    e.zero.(!a1) <- e.zero.(!a2);
    incr a1;
```

```
    incr a2;
    if !a2 <= a3
    then aux ()
  in
  aux ();
  a1 := ad1;
  if !a1 > e.dol then a1 := e.dol;
  e.dot <- !a1;
  e.fchange <- true
```

D.9.7 changing lines: c

⟨CLI.commands() *match* c *modifying cases* 109a⟩+≡ (99e) ◁108f 113f▷

```
| 'c' ->
   Commands.nonzero e;
   In.newline e;
   Commands.rdelete e e.addr1 e.addr2;
   Commands.append e (In.gettty e) (e.addr1 - 1) |> ignore;
```

D.10 Command addresses

Address parsing is split into two clean phases: `Address.parse_rangeX` builds an `Address.range` AST from tokens, then `Address.eval_rangeX` evaluates it against the current environment to produce concrete line numbers. In C, these two phases were tangled together in the `address()` function.

D.10.1 Parsing addresses

⟨Lexer.token() *other cases* 109b⟩≡ (98f) 112a▷

```
(* for the addresses *)
| digit+        { Int (int_of_string (Lexing.lexeme lexbuf)) }
| '.' { Dot } | '$' { Dollar }

| ',' { Comma }
| '+' { Plus } | '-' { Minus }
```

⟨*function* Address.parse_address 109c⟩≡ (? 0—1)

```
let parse_address (st : Parser.state) : t =
  let base =
    match P.peek st with
    | T.Plus | T.Minus | T.Caret ->
      (* implicit '.' for leading + - ^ *)
      Current
    | _ ->
        (match P.consume st with
        | T.Dot -> Current
        | T.Dollar -> Last
        | T.Int n -> Line n
        ⟨Address.parse_address() match consumed token other cases 112g⟩
        | _ -> P.was_expecting "valid address"
        )
  in
  parse_relatives base st
```

⟨*function* `Address.parse_relatives` 110a⟩≡ (? 0—1)

```
let rec parse_relatives (base : t) (st : Parser.state) : t =
  match P.peek st with
  | T.Plus | T.Minus | T.Caret ->
      let d = parse_delta st in
      (* recurse, one can have multiple ++ -- *)
      parse_relatives (Relative (base, d)) st
  | _ ->
      base
```

⟨*function* `Address.parse_delta` 110b⟩≡ (? 0—1)

```
let parse_delta (st : Parser.state) : int =
  match P.consume st with
  | T.Plus ->
      (match P.peek st with
       | T.Int n -> ignore (P.consume st); n
       | _ -> 1)
  | T.Minus ->
      (match P.peek st with
       | T.Int n -> ignore (P.consume st); -n
       | _ -> -1)
  ⟨Address.parse_delta() match consumed token other cases 117h⟩
  | _ ->
      P.was_expecting "relative operator"
```

⟨*function* `Address.parse_range` 110c⟩≡ (? 0—1)

```
let parse_range (st : Parser.state) : range =
  let t1 = P.peek st in
  (* optional first address *)
  let first : t option =
    ⟨Address.parse_range() compute optional first address 110d⟩
  in

  let t2 = P.peek st in
  match t2 with
  ⟨Address.parse_range() Comma or Semicolon case 110e⟩
  | _ ->
      (* single address or none *)
      (match first with
      | Some a ->
          { addr1 = None; addr2 = a; given = true; set_dot = false; }
      | None ->
          (* no addresses given, implicit "." (Current) *)
          { addr1 = None; addr2 = Current; given = false; set_dot = false; }
      )
```

⟨`Address.parse_range()` *compute optional* `first` *address* 110d⟩≡ (110c)

```
match t1 with
| T.Plus | T.Minus | T.Caret
| T.Dot | T.Dollar | T.Int _
| T.Mark _ | T.Slash _ | T.Question _ ->
    (* this will consume some tokens in st *)
    Some (parse_address st)
| T.Comma | T.Semicolon
| T.EOF | T.Spaces | T.Newline | T.Char _ ->
    None
```

⟨`Address.parse_range()` `Comma` *or* `Semicolon` *case* 110e⟩≡ (110c)

```
| T.Comma | T.Semicolon ->
    P.consume st |> ignore;
```

```
    let second =
      match P.peek st with
      | T.Plus | T.Minus | T.Caret
      | T.Dot | T.Dollar | T.Int _
      | T.Mark _ | T.Slash _ | T.Question _ ->
         parse_address st
      (* a missing second address default to '$' *)
      | T.Comma | T.Semicolon
      | T.EOF | T.Spaces | T.Newline | T.Char _ ->
         Last
    in
    {
       (* a missing first address in a range default to '1' so
        * a range like "," will be parsed as "1,$"
        *)
        addr1 = (match first with Some _ -> first | None -> Some (Line 1));
        addr2 = second;
        given = true;
        set_dot = (t2 = T.Semicolon);
       }
```

D.10.2 Evaluating addresses

⟨*function* `Address.eval_address` 111a⟩≡ (? 0—1)

```
let rec eval_address (e : Env.t) (adr : t) : Env.lineno =
  match adr with
  | Current -> e.dot
  | Last -> e.dol
  | Line n -> n
  | Relative (x, n) -> eval_address e x + n
  ⟨Address.eval_address() match adr other cases 112h⟩
```

⟨*function* `Address.eval_range` 111b⟩≡ (? 0—1)

```
let eval_range (e : Env.t) (r : range) : Env.lineno * Env.lineno =
  ⟨Address.eval_range debug range 119k⟩

  let addr2 = eval_address e r.addr2 in
  let addr1 =
    match r.addr1 with
    | None -> addr2
    | Some a -> eval_address e a
  in
  addr1, addr2
```

D.11 Search and replace

`oed` uses OCaml's `Str` library (via `Re_str`) for regular expressions, so there is no need for `compile()X` or `match()X` wrappers—just `Str.regexp` and `Str.search_forward`. The regexp itself is lexed by a dedicated `ocamllex` rule that handles the delimiter and backslash escapes.

⟨*type* `Regex.t` 111c⟩≡ (? 0—1)

```
type t = Str.regexp
```

D.11.1 Parsing regexps

⟨`Token.t` *other cases* 111d⟩≡ (90a) 114b▷

```
| Slash of string | Question of string
```

⟨Lexer.token() *other cases* 112a⟩+≡ (98f) ◁109b 114c▷
```
  | '/'              { Buffer.clear buf; Slash (regexp '/' lexbuf) }
  | '?'              { Buffer.clear buf; Question (regexp '?' lexbuf) }
```

⟨*signature* Lexer.buf 112b⟩≡ (? 0—1)
```
  (* ugly: should not be needed outside *)
  val buf: Buffer.t
```

⟨*constant* Lexer.buf 112c⟩≡ (? 0—1)
```
  let buf = Buffer.create 32
```

⟨*signature* Lexer.regexp 112d⟩≡ (? 0—1)
```
  val regexp: char -> Lexing.lexbuf -> string
```

⟨*function* Lexer.regexp 112e⟩≡ (? 0—1)
```
  and regexp delim = parse
    | '\\' (_ as c) {
        Buffer.add_char buf '\\'; Buffer.add_char buf c; regexp delim lexbuf
      }
    | '/' {
        if delim = '/'
        then Buffer.contents buf
        else begin Buffer.add_char buf '/'; regexp delim lexbuf end
       }
    | '?' {
        if delim = '?'
        then Buffer.contents buf
        else begin Buffer.add_char buf '?'; regexp delim lexbuf end
      }
    | '\n' | eof { failwith "Unterminated regular expression" }
    | _ as c { Buffer.add_char buf c; regexp delim lexbuf }
```

D.11.2 Search as addresses: /re/ and ?re?

⟨Address.t *other cases* 112f⟩≡ (91c) 114d▷
```
  | SearchFwd of string (* /.../ *)
  | SearchBwd of string (* ?...? *)
```

⟨Address.parse_address() *match consumed token other cases* 112g⟩≡ (109c) 114e▷
```
  | T.Slash r -> SearchFwd r
  | T.Question r -> SearchBwd r
```

⟨Address.eval_address() *match* adr *other cases* 112h⟩≡ (111a) 114f▷
```
  | SearchFwd _ | SearchBwd _ ->
      let dir, re_str =
        match adr with
        | SearchFwd re -> 1, re
        | SearchBwd re -> -1, re
        | _ -> raise (Impossible "cases matched above")
      in
      (* less: opti: use SearchFwd of regex instead of str *)
      let re = Str.regexp re_str in

      (* starting point *)
      (* TODO: ed: need to be `a` instead like in C ? *)
      let a = e.dot in
      let b = a in

      let rec aux (a : Env.lineno) : Env.lineno =
```

```
      let a = a + dir in
      ⟨Address.eval_address() when SearchXxx case, wrap a 113b⟩
      match () with
      | _ when Commands.match_ e re a -> a
      ⟨Address.eval_address() when SearchXxx case, when back to start 113c⟩
      | _ ->
          aux a
    in
    aux a
```

⟨*signature* Commands.match_ 113a⟩≡ (? 0—1)
```
val match_: Env.t -> Regex.t -> Env.lineno -> bool
```

⟨Address.eval_address() *when SearchXxx case, wrap* a 113b⟩≡ (112h)
```
let a =
  match () with
  (* wrap around start/end of buffer *)
  | _ when a <= 0 -> e.dol
  | _ when a > e.dol -> 1
  | _ -> a
in
```

⟨Address.eval_address() *when SearchXxx case, when back to start* 113c⟩≡ (112h)
```
(* back to starting point and nothing was found *)
| _ when a = b ->
    Error.e_warn (spf "search for %s had no match" re_str)
```

D.11.3 match()

⟨*function* Commands.match_ 113d⟩≡ (? 0—1)
```
let match_ (e : Env.t) (re : Regex.t) (addr : Env.lineno) : bool =
  let line = Disk.getline e addr in
  Regex.match_str re line
```

⟨*function* Regex.match_str 113e⟩≡ (? 0—1)
```
let match_str (re : t) (str: string) : bool =
  (* old: Str.string_match re line 0, but we need unanchored search *)
  try
    Str.search_forward re str 0 |> ignore;
    true
   with Not_found -> false
```

D.11.4 substitute: s

The substitute command is simpler than in C: OCaml's `String.sub` handles the before/matched/after splitting directly, without the manual `genbuf` shuffling of `dosub()X`. However, several features are still TODO: the `g` flag, `&` and `\1` in the replacement string, and the skip count.

⟨CLI.commands() *match* c *modifying cases* 113f⟩+≡ (99e) ◁109a
```
| 's' ->
    Commands.nonzero e;
    Commands.substitute e (e.in_.globp <> None);
```

⟨*signature* Commands.substitute 113g⟩≡ (? 0—1)
```
(* 's' *)
val substitute: Env.t -> bool (* inglob *) -> unit
```

⟨*function* `Commands.substitute` 114a⟩≡ (? 0—1)

```
(* used for 's' *)
let substitute (e : Env.t) (_inglob: bool) : unit =
  (* TODO: handle inglob *)
  match Parser.consume e.in_ with
  (* handle | Question re_str? does not really make sense. *)
  | Slash re_str ->
      let re = Str.regexp re_str in
      (* ugly: *)
      Buffer.clear Lexer.buf;
      let subst_str = Lexer.regexp '/' e.in_.stdin in
      (* TODO: parse 's/.../.../g' *)
      In.newline e;

      for a1 = e.addr1 to e.addr2 do
        let line = Disk.getline e a1 in
        (* TODO: handle 's/.../.../g' *)
        if Regex.match_str re line
        then begin
           let loc1 = Str.match_beginning () in
           let loc2 = Str.match_end () in
           let len = String.length line in

           let before = String.sub line 0 loc1 in
           let _matched = String.sub line loc1 (loc2 - loc1) in
           let after = String.sub line loc2 (len - loc2) in
           (* TODO: handle & and \1 in subst_str *)
           let newline = before ^ subst_str ^ after in

           let tl = Disk.putline e newline in
           (* TODO: mark? *)
           e.zero.(a1) <- { offset = tl; mark = false }
        end
      done

  | t -> Parser.was_expecting_but_got "a regexp" t
```

D.11.5 Advanced substitutions

mark: `k`

⟨`Token.t` *other cases* 114b⟩+≡ (90a) ◁111d 117f▷

```
| Mark of char
```

⟨`Lexer.token()` *other cases* 114c⟩+≡ (98f) ◁112a 117g▷

```
| '\'' ['a'-'z'] as s { Mark s.[1] }
```

⟨`Address.t` *other cases* 114d⟩+≡ (91c) ◁112f

```
| Mark of char (* \a *)
```

⟨`Address.parse_address()` *match consumed token other cases* 114e⟩+≡ (109c) ◁112g

```
| T.Mark c -> Mark c
```

⟨`Address.eval_address()` *match* `adr` *other cases* 114f⟩+≡ (111a) ◁112h

```
| Mark _ -> failwith "TODO: Mark"
```

D.11.6 global commands: g/re/cmd and v/re/cmd

The two-phase mark-then-execute design is preserved from C. Thanks to the `offset_and_mark` record, marking is just setting a boolean field (`e.zero.(a1).mark <- true`) rather than stealing bits from the offset integer. The recursive call to `commandsX` for each marked line is the same.

⟨`CLI.commands()` *match* `c` *global cases* 115a⟩≡ (99e)

```
| 'g' -> global caps e true
| 'v' -> global caps e false
```

⟨*function* `CLI.global` 115b⟩≡ (? 0—1)

```
(* g/re/cmd, v/re/cmd *)
and global caps (e : Env.t) (pos_or_neg : bool) : unit =

  e.in_.globp |> Option.iter (fun _ ->
      Error.e_err "global command already in"
  );
  Commands.setwide e;
  Commands.squeeze e (if e.dol > 0 then 1 else 0);

  match Parser.consume e.in_ with
  | Slash str ->
      let re = Str.regexp str in
      (* TODO: Lexer.line_or_escaped_lines *)
      let line = Lexer.line e.in_.stdin in
      let line = if line = "" then "p" else line in

      (* step1: marking the matching lines *)
      (* ed: was starting at 0, but then special case later in match()
       * so simpler to start at 1
       * old: I was recording in a local xs the list of matched lineno, but
       * it does not work because commands() in step2 below may modify
       * zero which would invalidate the matched lineno of step1
       *)
      for a1 = 1 to e.dol do
        if a1 >= e.addr1 && a1 <= e.addr2 &&
            Commands.match_ e re a1 = pos_or_neg then
          e.zero.(a1).mark <- true
      done;

      (* step2: processing the matching lines
       * old: for a1 = 1 to e.dol, but can't work because
       * commands() below might modify zero and delete lines in which case
       * dol will change dynamicaly.
       *)
      let a1 = ref 1 in
      while !a1 <= e.dol do
        if e.zero.(!a1).mark then begin
            e.zero.(!a1).mark <- false;
            e.dot <- !a1;
            (* ugly: need trailing \n otherwise T.EOF would be consumed
             * too early by In.Newline().
             *)
            e.in_.globp <- Some (Lexing.from_string (line ^ "\n"));
            (* recurse!! *)
            commands caps e;
            (* need to restart from scratch, but with one less marked line *)
            a1 := 0
        end;
        incr a1
```

```
      done

  | t -> Parser.was_expecting_but_got "a regexp" t
```

D.12 Advanced features

D.12.1 Running a shell command: `!`

The `!` command uses `CapSys.command` (a capability-wrapped version of `Sys.command`) instead of the C `fork`/`execl` pair. In restricted mode (`-r`), this capability is revoked.

⟨`CLI.commands()` *match* `c` *other cases* 116a⟩+≡ (99e) ◁108e
```
| '!' ->
    Commands.callunix caps e
```

⟨*signature* `Commands.callunix` 116b⟩≡ (? 0—1)
```
(* '!' *)
val callunix: <Cap.forkew; ..> -> Env.t -> unit
```

⟨*function* `Commands.callunix` 116c⟩≡ (? 0—1)
```
(* '! *)
let callunix (caps : <Cap.forkew; ..>) (e: Env.t) : unit =
  setnoaddr e;
  let s = In.gety e in
  ⟨Commands.callunix() restricted mode check 117c⟩
  (* ed: was calling rc -c but here we reuse (Cap)Sys.command which relies
   * on the default shell
   *)
  let _ret = CapSys.command caps s in
  if e.vflag
  then Out.putst e "!"
```

D.12.2 `ed -r` and `rflag` restricted mode

The `-r` flag is a new feature (inspired by GNU `ed`'s restricted mode) that limits what `oed` can do: no shell commands, and file access restricted to the current directory. This is implemented by wrapping the capabilities object: `restrict_capsX` returns a new object where `exec`, `open_in`, and `open_out` methods check the restriction before delegating. This is a good example of OCaml's object system used for capability-based security.

⟨`CLI.main()` *local flags* 116d⟩+≡ (93f) ◁95a 118f▷
```
(* new: restricted ed *)
let rflag = ref false in
```

⟨`CLI.main()` *local* `options` *elements* 116e⟩+≡ (93f) ◁95b 118g▷
```
"-r", Arg.Set rflag,
" restricted mode: no shell commands; edits limited to current directory";
```

⟨`Env.t` *flag fields* 116f⟩+≡ (91e) ◁95c
```
(* new: from GNU ed, "restrited mode" *)
rflag: bool;
```

⟨`CLI.main()` *restrict caps* 116g⟩≡ (93f)
```
let caps = restrict_caps !rflag caps in
```

⟨*function* CLI.restrict_caps 117a⟩≡ (? 0—1)
```
let restrict_caps rflag (x : < caps; ..>) =
  object
    method exec =
      if rflag then Error.e_err "!restricted mode on!"
      else x#exec
    method open_in file =
      if rflag && not (Fpath.is_seg file)
      then Error.e_err (spf "!restricted mode on, can't read %s!" file)
      else x#open_in file
    method open_out file =
      if rflag && not (Fpath.is_seg file) &&
         (* need open_out to delete tmp file in Commands.quit() *)
         not (file = !!Env.tfname)
      then Error.e_err (spf "!restricted mode on, can't write %s!" file)
      else x#open_out file
    method fork = x#fork
    method wait = x#wait
    method stdin = x#stdin
    method stdout = x#stdout
    method stderr = x#stderr
  end
```

⟨*function* Commands.file_in_current_dir 117b⟩≡ (? 0—1)
```
let file_in_current_dir (file : Fpath.t) : bool =
  (* alt: use Unix.realpath, follow symlink and compare to cwd *)
  Fpath.is_seg !!file
```

⟨Commands.callunix() *restricted mode check* 117c⟩≡ (116c)
```
(* needed only for ocaml-light who does not support method call
 * and so can't provide the dynamic caps of CLI.restrict_caps
 *)
if e.rflag
then Error.e_err "restricted mode on";
```

⟨Commands.read() *restricted mode check* 117d⟩≡ (101f)
```
(* ocaml-light: see callunix() comment below *)
if e.rflag && not (file_in_current_dir file)
then Error.e_err (spf "restricted mode on, can't access %s" !!file);
```

⟨Commands.write() *restricted mode check* 117e⟩≡ (104h)
```
(* ocaml-light: see callunix() comment below *)
if e.rflag && not (file_in_current_dir file)
then Error.e_err (spf "restricted mode on, can't access %s" !!file);
```

D.12.3 The caret

⟨Token.t *other cases* 117f⟩+≡ (90a) ◁114b 117i▷
```
| Caret
```

⟨Lexer.token() *other cases* 117g⟩+≡ (98f) ◁114c 118a▷
```
| '^' { Caret }
```

⟨Address.parse_delta() *match consumed token other cases* 117h⟩≡ (110b)
```
| T.Caret -> -1
```

D.12.4 The semicolon

⟨Token.t *other cases* 117i⟩+≡ (90a) ◁117f
```
| Semicolon
```

⟨Lexer.token() *other cases* 118a⟩+≡ (98f) ◁117g
```
| ';' { Semicolon }
```

⟨Address.range *other fields* 118b⟩≡ (91d)
```
set_dot : bool;
```

D.13 Error management and logging

The C `setjmp/longjmp` error recovery is replaced by OCaml exceptions. `Error.Error` is caught in the `while true` loop in `CLI.mainX`, which prints `?` and the message—exactly matching the original `ed` behavior. On top of this, `oed` adds proper logging via the `Logs` library: `Error.e_warn` and `Error.e_err` log a message at the appropriate level *and* raise `Error.Error` for backward compatibility. The `-verbose` and `-debug` flags control log verbosity.

D.13.1 Legacy error management

⟨*exception* Error.Error 118c⟩≡ (? 0—1)
```
exception Error of string
```

⟨*function* Error.e_legacy 118d⟩≡ (? 0—1)
```
(* the raise will effectively jump on the exn handler in CLI.main()
 * ed: we emulate the longjmp done in C.
 *)
let e_legacy s =
  raise (Error s)
```

⟨CLI.main() *exn handler for* Error.Error s 118e⟩≡ (93f)
```
(* ed: was in a separate error_1() function *)
(* TODO: reset globals too? *)
Out.putchr env '?';
Out.putst env s;
```

D.13.2 Improved error reporting and logging

⟨CLI.main() *local flags* 118f⟩+≡ (93f) ◁116d
```
let level = ref (Some Logs.Warning) in
```

⟨CLI.main() *local* options *elements* 118g⟩+≡ (93f) ◁116e
```
(* new: this is verbose *logging*, different from interactive mode *)
"-verbose", Arg.Unit (fun () -> level := Some Logs.Info),
" verbose logging mode";
"-debug", Arg.Unit (fun () -> level := Some Logs.Debug),
" debug logging mode";
"-quiet", Arg.Unit (fun () -> level := None),
" quiet logging mode";
```

⟨CLI.main() *setup logging after parsed argv* 118h⟩≡ (93f)
```
Logs_.setup !level ();
```

⟨*function* Error.e_warn 118i⟩≡ (? 0—1)
```
let e_warn s =
  Logs.warn (fun m -> m "%s" s);
  (* ed: to remain backward compatible *)
  raise (Error "")
```

⟨*function* `Error.e_err` 119a⟩≡ (? 0—1)
```
let e_err s =
  Logs.err (fun m -> m "%s" s);
  raise (Error "")
```

⟨*signature* `Parser.was_expecting` 119b⟩≡ (? 0—1)
```
(* internals that are used outside for now *)
val was_expecting: string -> 'a
```

⟨*signature* `Parser.was_expecting_but_got` 119c⟩≡ (? 0—1)
```
val was_expecting_but_got: string -> Token.t -> 'a
```

⟨*function* `Parser.was_expecting` 119d⟩≡ (? 0—1)
```
let was_expecting (expect : string) =
  Error.e_err (spf "was expecting %s" expect)
```

⟨*function* `Parser.was_expecting_but_got` 119e⟩≡ (? 0—1)
```
let was_expecting_but_got (expect : string) (tok : Token.t) =
  was_expecting (spf "%s, but got %s" expect (Token.show tok))
```

⟨`CLI.commands()` *in* `'a'` *case, log append mode* 119f⟩≡ (108a)
```
Logs.info (fun m -> m "append mode");
```

⟨`CLI.commands()` *in* `'i'` *case, log insert mode* 119g⟩≡ (108b)
```
Logs.info (fun m -> m "insert mode");
```

⟨`In.gettty()` *log end of input* 119h⟩≡ (98a)
```
Logs.info (fun m -> m "end of input, back to ed");
```

D.14 Debugging support

The `deriving show` annotations on all major types pay off here: `Env.show`, `Token.show`, and `Address.show_range` give free pretty-printing. The `X` command dumps the full internal state, including the raw content of `tfile`.

⟨`CLI.main()` *debug env* 119i⟩≡ (94b)
```
Logs.debug (fun m -> m "env = %s" (Env.show env));
```

⟨`Parser.next_token()` *debug token* 119j⟩≡ (98g)
```
Logs.debug (fun m -> m "tok = %s" (Token.show t));
```

⟨`Address.eval_range` *debug range* 119k⟩≡ (111b)
```
Logs.debug (fun m -> m "range = %s" (show_range r));
```

⟨`CLI.commands()` *debug start* 119l⟩≡ (99e)
```
Logs.debug (fun m -> m "commands ->");
```

⟨`CLI.commands()` *debug end* 119m⟩≡ (99e)
```
Logs.debug (fun m -> m "commands <-");
```

⟨`CLI.commands()` *match* `c` *inspecting cases* 119n⟩+≡ (99e) ◁107g
```
(* new: dumping internal state (useful for debugging) *)
| 'X' ->
    In.newline e;
    Unix.fsync e.tfile;
    Logs.app (fun m -> m "env = %s\ntfile content =\n%s"
                (Env.show e)
                (FS.cat caps Env.tfname |> String.concat "\n"));
```

D.15 Index

Bibliography

[BO10] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2010. cited page(s) 7

[Com84] Douglas Comer. *Operating System Design: The XINU Approach*. Prentice Hall, 1984. cited page(s) 6, 10, 25

[Com87] Douglas Comer. *Operating System Design: Volume II Internetworking with XINU*. Prentice Hall, 1987. cited page(s) 25

[Dar14] Lewis Dartnell. *The Knowledge: How to Rebuild Our World from Scratch*. Penguin Press, 2014. cited page(s) 27

[EucBC] Euclid. *The Elements*. 300 BC. cited page(s) 10

[FH95] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995. cited page(s) 6

[Int86] Intel. *Intel 80386 Programmer's Reference Manual*. Intel Corporation, 1986. cited page(s) 8

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 1, 2, 3*. Addison-Wesley, 1973. cited page(s) 10, 25

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 6, 10

[Knu99] Donald E. Knuth. *MMIXWare, a RISC Computer for the Third Millenium*. Springer, 1999. cited page(s) 25

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984. cited page(s) 12, 38

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 10, 12

[Lio77] John Lions. *Lions Commentary on UNIX 6th Edition, with Source Code*. Peer to Peer Communications, 1977. cited page(s) 25

[NS05] Noam Nisan and Shimon Shocken. *The Elements of Computing Systems*. MIT Press, 2005. cited page(s) 10, 25, 28

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 11

[Pad16] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 65

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. In *Computing Systems*, pages 221–254, 1995. Also available at docs/articles/9.ps. cited page(s) 12

[PPT+93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Operating Systems Review*, pages 72–76, 1993. Also available at docs/articles/names.ps. cited page(s) 12

[Sea01] David Seal. *ARM, Architecture Reference Manual*. Addison-Wesley, 2001. cited page(s) 8, 10

[Sei09] Peter Seibel. *Coders at Work, Reflections on the Craft of Programming*. Apress, 2009. cited page(s) 20

[Tan87] Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice Hall, 1987. cited page(s) 6, 10, 25

[TML+79] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A personal computer. Technical report, Xerox PARC, 1979. CSL-79-11. cited page(s) 20

[WG92] Niklaus Wirth and Jurg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992. cited page(s) 25

[WR13] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910, 1912, 1913. cited page(s) 6, 10