Principia Softwarica: The Plan 9 Utilities

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
many people

April 28, 2026

Contents

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of many utilities, especially utilities that are useful for the programmer such as `grep` and `sed`.

1.1 Motivations

Why utilities? Because they are the programmer's toolbox. The command-line interface is where most programmers spend a significant part of their time, composing small programs through pipes and scripts to accomplish tasks that would require hundreds of lines in a regular programming language. Kernighan and Plauger's *Software Tools* [KP76] was the first book to explain UNIX utilities by showing their implementation; Kernighan and Pike's *The UNIX Programming Environment* [KP84] continued in the same spirit, and their later *The Practice of Programming* [KP99] discusses many of the same design principles. This book follows their tradition, but with the Plan 9 versions of these tools.

The relevance of these tools has only grown with the rise of AI coding assistants. Tools like Claude Code operate by invoking `grep`, `sed`, `awk`, and shell scripts to explore and modify codebases—the same utilities described in this book. The UNIX philosophy of small, composable text-processing programs turns out to be an ideal interface for AI agents.

Moreover, many utilities are interesting programs in their own right: `grep` contains a regular expression engine, `diff` implements an elegant longest-common-subsequence algorithm, `awk` is a complete programming language, and `ed` is a line editor whose command language survives in `sed` and `vi` to this day. Kernighan and Plauger's *Software Tools* [KP76] and Kernighan and Pike's *The UNIX Programming Environment* [KP84] are classic books built around the same idea: the best way to learn programming is to study real, useful programs.

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

- How does `grep` search for patterns in files? How does a regular expression engine work?
- How does `diff` compute the differences between two files? What algorithm does it use?
- How does `awk` parse and execute a small programming language?
- How does `ed`, the line editor, manage a text buffer and apply editing commands?
- How does `tar` pack and unpack archives?
- What is the common structure shared by most utilities? How do they parse command-line flags and process input?

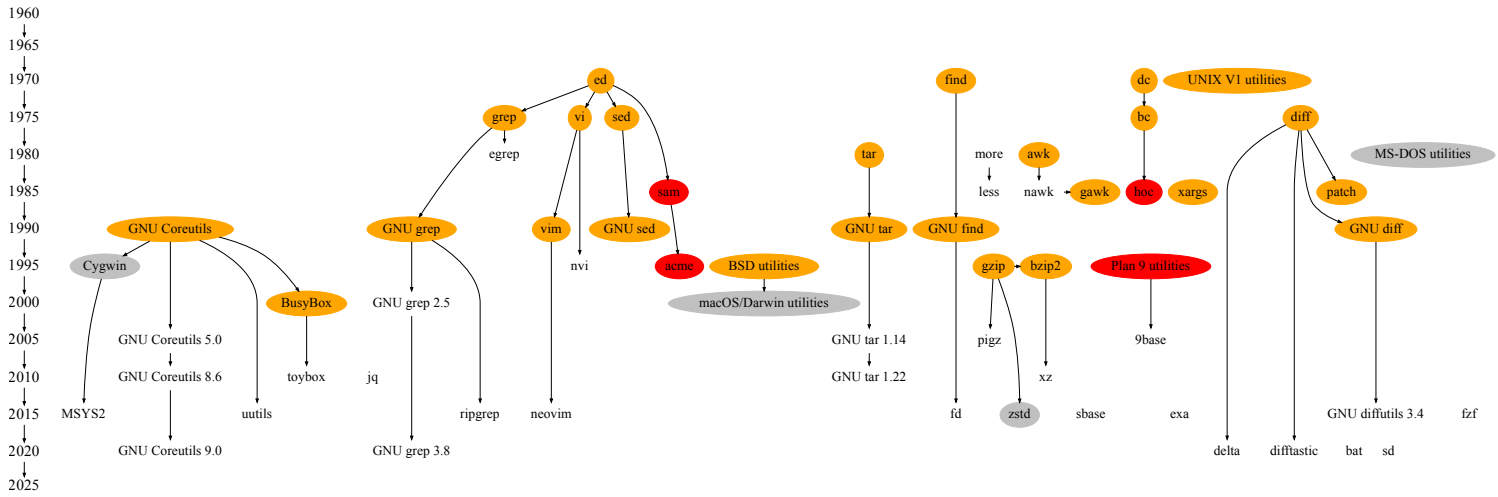


Figure 1.1: Unix utilities timeline

1.2 The Plan 9 utilities

I will explain in this book the code of the Plan 9 command-line utilities: file operations (`ls`, `cp`, `mkdir`), the line editor `ed`, pattern matching (`grep`, `sed`), text processing (`awk`), file comparison (`diff`, `cmp`), process management, archiving (`tar`, `ar`), and many others. Each utility is typically a few hundred lines of C. Like for most books in Principia Softwarica, I chose Plan 9 programs because they are simple, small, elegant, open source, and they form together a coherent set.

1.3 Other utilities

Here are a few alternative implementations of UNIX utilities:

- GNU coreutils[1] is the standard set of utilities on Linux. The GNU versions add many features (long options, internationalization, extensions) compared to the Plan 9 versions, at the cost of being much larger.
- BusyBox[2] packs hundreds of utilities into a single statically linked binary, aimed at embedded systems and Docker containers. About 256 000 LOC in total, though this includes far more than core utilities: a shell (`ash`), networking servers, and util-linux tools.
- Toybox[3] is a BSD-licensed BusyBox alternative, used in Android.
- sbase[4] is a suckless-style reimplementation in minimal C, in the same spirit as the Plan 9 utilities.
- Rust coreutils[5] is a rewrite in Rust, aiming for GNU compatibility. Despite using an arguably higher-level language, the project totals about 130 000 LOC—a measure of how much code GNU compatibility requires.

Figure 1.1 presents a timeline of major UNIX utility implementations. The Plan 9 utilities represent the best compromise for this book: they are direct descendants of the original UNIX tools, cleaned up over decades by the same people who wrote the originals (Ken Thompson, Rob Pike, Dennis Ritchie), and small enough to explain in full.

[1] https://www.gnu.org/software/coreutils/
[2] https://busybox.net/
[3] http://www.landley.net/toybox/
[4] https://core.suckless.org/sbase/
[5] https://github.com/uutils/coreutils

1.4 Requirements

Because this book is made of C source code, you will need a good knowledge of the C programming language [KR88] to understand it. The utilities covered here are generally short, so the code is usually more approachable than in the KERNEL book [Pad14] or the LIBCORE book [Pad16]. This book is not an introduction to UNIX or to command-line tools. I assume you are already familiar with basic shell usage: pipes, redirections, globbing, and invoking a command with options. If not, the classic introduction is Kernighan and Pike's *The Unix Programming Environment* [KP84].

If, while reading this book, you have specific questions on the command-line interface of one of the tools, I suggest you to consult the corresponding manual page in my Plan 9 repository. Section 1 (`docs/man/1/`) documents every tool covered here: for example, `docs/man/1/0intro` is the overview page (the analog of `docs/man/2/0intro` for the libraries), and individual pages such as `docs/man/1/ls`, `docs/man/1/grep`, `docs/man/1/ed`, `docs/man/1/dc`, `docs/man/1/cal`, `docs/man/1/dd`, and so on document each utility's flags and behavior. The top-level `docs/articles/` directory contains two papers relevant to the text-processing chapters: `docs/articles/utf.ps` ("Hello World", rendered in Greek and Japanese), which describes the UTF-8 support that pervades the Plan 9 string utilities, and `docs/articles/release4.ps` ("Plan 9 Release Notes for the Fourth Edition"), which lists what was new or changed in the utilities between editions.

1.5 Getting started

The utilities described in this book can be tested in several environments. Under the Plan 9 fork used in Principia Softwarica (see https://www.principia-softwarica.org/getting-started.html), they are available directly. They are also part of Goken9cc[6], where they can be compiled natively on Linux, macOS, and Windows using gcc or clang—similar to plan9port[7], Russ Cox's port of Plan 9 user-space tools. This means you can experiment with them on your regular machine without setting up a full Plan 9 system.

Here is a quick session under Plan 9:

```
$ echo hello world | wc -w
      2
$ ls /dev | grep cons
cons
consctl
$ echo 'hello world' | sed 's/world/plan9/'
hello plan9
$ echo '2+3' | bc
5
```

1.6 About this document

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

1.7 Copyright

[6] https://github.com/aryx/goken9cc
[7] https://9fans.github.io/plan9port/

1.8 Acknowledgments

Chapter 2

Overview

Before showing the source code of the Plan 9 utilities in the following chapters, I first give an overview of the general principles behind UNIX command-line tools, describe the common patterns they share, and explain how the code is organized.

2.1 Utilities principles

2.1.1 The UNIX philosophy

The UNIX philosophy, as articulated by Doug McIlroy, is: *write programs that do one thing and do it well; write programs to work together; write programs to handle text streams, because that is a universal interface.* This philosophy shapes every utility in this book. Each program is small and focused: `cat` concatenates files, `grep` searches for patterns, `wc` counts lines. Alone, each is trivial. Composed through pipes, they become powerful:

```
$ ls /sys/src | grep '\.c$' | wc -l
```

counts all C source files in a directory, using three programs that know nothing about each other.

McIlroy articulated this motto in the foreword to the 1978 Bell System Technical Journal issue on UNIX; Thompson and Ritchie had been practicing it since 1969 but never formally named it. The philosophy stands in deliberate contrast to the *integrated application* model that Windows and macOS popularized (Word, Excel, Outlook—large programs that do many things inside one process). The same idea resurfaced in the 2010s as *microservices*: each network service does one thing, communicates via APIs, and can be replaced independently—UNIX pipes scaled up to distributed systems.

The next three subsections unpack the three halves of McIlroy's motto in turn.

2.1.2 "Worse is better"

In 1989, the software engineer Richard Gabriel wrote an essay titled *"Worse Is Better"* that gave a name to the design philosophy behind UNIX and C. He contrasted two styles. The "MIT approach" (which he associated with Lisp and Multics) insists on *correctness* and *completeness*: every interface must handle every edge case, every abstraction must be clean, and shipping is delayed until the design is right. The "New Jersey approach" (which he associated with UNIX and C) insists on *simplicity*: the implementation must be simple even if the interface is slightly less powerful, and shipping a working 80% solution now beats shipping a perfect 100% solution never. The essay's provocative thesis is that the "worse" system—simpler, rougher, incomplete—wins in practice because it is easier to port, easier to understand, and easier to improve incrementally.

History has largely vindicated Gabriel's observation. UNIX displaced Multics; C displaced Lisp for systems work; TCP/IP displaced the OSI model; the web (built on simple HTTP and HTML) displaced every "correct" hypertext system that preceded it. The pattern repeats because *simplicity spreads*: a simple system runs on more

platforms, attracts more users, and accumulates improvements from more contributors, until the 80% solution evolves into a 90% or 95% solution that the "correct" alternative never reached because it never shipped. Plan 9 is arguably "worse is better" taken to its logical extreme: even simpler than UNIX, even smaller, even more uniform, with a 30-syscall kernel surface where Linux has 400-odd. Paradoxically, Plan 9 itself did not win—it broke backward compatibility with UNIX, which meant it could not inherit UNIX's installed base and spread the way UNIX had spread from Multics's. The irony is that "worse is better" requires being *compatible enough* to spread, and Plan 9 was too clean a break. But its ideas—UTF-8, goroutines, `/proc`, the Go toolchain—spread by being absorbed back into the systems that did win.

2.1.3 Text streams as a universal interface

The glue that makes utility composition possible is the choice of a *universal interchange format*. UNIX picked plain line-oriented text: ASCII (and later UTF-8) bytes separated by newlines, with no schema and no framing beyond what the data itself suggests. This decision is famously illustrated by Jon Bentley's *Programming Pearls* column from 1986, where he asked Donald Knuth and Doug McIlroy each to write a program that prints the N most-common words in a text file. Knuth's answer was a ten-page literate Pascal (WEB) program with a hand-built trie and its own *ad hoc* I/O routines. McIlroy's reply was six lines of pipeline:

```
tr -cs A-Za-z '\n' | tr A-Z a-z | sort | uniq -c | sort -rn | sed ${N}q
```

Every stage is one of the utilities in this book (`tr`, `sort`, `uniq`, `sed`), and the whole composition runs in constant memory, streams through the data, and is correct by inspection. The point is not that text streams are always the right representation: sometimes you need `xd` to dump binary bytes, or a purpose-built tool like `jq` for JSON. The point is that for the common case of tabular, line-oriented data, text is the lingua franca that lets programs written decades apart by different authors compose effortlessly—which is precisely what Windows PowerShell (which pipes typed objects) and nushell (which pipes typed tables) give up in exchange for stronger typing.

2.1.4 Pipes and composition

The mechanism that connects those text streams is the *pipe*: a kernel-buffered byte channel between two processes where one's standard output feeds the other's standard input. The idea was suggested by Doug McIlroy in a 1964 memo and sat unimplemented for almost a decade before Ken Thompson finally added the `pipe` system call to UNIX V3 in 1973, together with the shell's `'|'` syntax. The kernel buffer (a few kilobytes on historical UNIX, 64 KB on modern Linux) is what makes the scheme work. If the downstream reader is slower than the upstream writer, the writer blocks when the buffer fills, giving you *back-pressure* for free; if the reader is faster, it blocks on an empty buffer until more data arrives. And when the reader closes its end, the writer's next `write` receives `SIGPIPE`—by default terminating the process—which is why a pipeline shuts down promptly once a stage like `sed 10q` or `head` has decided it has seen enough.

The consequences of this one mechanism are far-reaching. A pipeline *streams* rather than buffers the whole dataset in memory, so it can process files larger than RAM. Each stage runs in its own process, so a pipeline on a multi-core machine is implicitly parallel at no cost to the programmer. And composition with `'|'` is mathematically just function composition with streams in place of values: `f | g | h` applies `h` first, then `g`, then `f`. This deeply simple primitive is also why modern distributed-data frameworks (Apache Beam, Kafka Streams, Flink, Spark Structured Streaming) are sometimes described as "UNIX pipes scaled out to clusters"—the operator-graph model they all share is, at bottom, the same idea Thompson shipped in 1973.

2.1.5 Tiny languages

Many of the utilities in this book are not just programs but *tiny domain-specific languages*. `grep` speaks regular expressions. `sed` speaks a stream-editing command language (`s/old/new/g`, `d`, `p`). `awk` speaks pattern-action rules with fields, variables, and arithmetic—essentially `sed` extended from text to numbers. `dc` speaks reverse

Polish notation; `bc` speaks algebraic expressions that compile down to `dc`. Even the document-processing tools are tiny languages: `pic` speaks a drawing language, `tbl` speaks table layout, `eqn` speaks mathematical notation, all feeding their output into `troff`. And the tools that *build* languages—`lex` for scanners, `yacc` for parsers—are themselves tiny languages whose input is a grammar and whose output is C code. UNIX is, at its heart, an ecosystem of tiny languages glued together by pipes and the shell.

The reason this works is that each tiny language is *deliberately incomplete*. `awk` does not need to read directories (`ls` does that), does not need to sort (`sort` does that), does not need to search (`grep` does that)—it only needs to compute on fields. Each language handles one domain, and the pipeline handles composition between domains. This is *separation of concerns* taken to its logical conclusion, and it only works because UNIX provides a universal composition mechanism (pipes) and a universal data format (text streams). Without both, every tiny language would need to be a full programming language—which is exactly what happened on operating systems that lacked pipes.

The natural counter-argument is: why not merge the tiny languages into one big one? Larry Wall did exactly that when he created Perl in 1987, combining `awk`'s field processing, `sed`'s regex substitution, and the shell's file-handling into a single scripting language. Perl succeeded massively in the 1990s (CGI scripting, system administration), and Python and Ruby later refined the same idea with cleaner syntax. But the tiny-tools philosophy quietly persisted: most shell scripts today still pipeline `grep`, `sed`, `awk`, and `sort` rather than rewriting everything in Python, because the pipeline is often shorter, clearer, and faster for the common case. In a recent twist, AI coding assistants like Claude Code and GitHub Copilot are bringing the tiny-tools philosophy to a new generation: when an agent needs to search, filter, or transform files, it reaches for `grep`, `sed`, `awk`, and `find` because they are fast, composable, and available on every machine—and in doing so, it teaches developers who might never have learned those tools on their own. The tension between "one big language" and "many tiny tools" is unresolved, and arguably unresolvable—it is the same trade-off as "monolith vs microservices" at a different scale.

2.1.6 The common utility structure

Most utilities in this book follow the same skeleton: parse command-line flags, then for each input file (or standard input if none is given), read the input, process it, and write the result to standard output. Errors go to standard error. This uniformity is what makes pipes and scripts work—each program can be a stage in a pipeline without any special adaptation—and it is a consequence, not a cause, of the three ideas above: because the universal interface is text (previous subsection), because composition happens through pipes (subsection before), and because each program does one thing (the philosophy subsection), the simplest possible program shape—read input, emit output, report errors on STDERR, exit with a status code—turns out to be all you need.

2.2 Command-line interfaces

Plan 9 utilities use single-letter flags (e.g., `-l`, `-n`) rather than GNU-style long options (`--recursive`, `--line-number`). This is more cryptic but also simpler, and it discourages adding too many flags—if you run out of letters, your program is probably too complex. The CLI parsing library in Plan 9 is minimal: just three macros in `libc.h` (`ARGBEGIN`, `ARGEND`, `ARGC`), compared to GNU's `getopt_long` with its option structs and tables. A typical usage pattern:

```
ARGBEGIN{
case 'n':
    nflag = true;
    break;
default:
    usage();
} ARGEND
```

The single-letter flag convention goes back to the earliest UNIX on the PDP-11 (1971): when your terminal runs at 110 baud, every keystroke costs time, so `-l` beats `--long`. GNU added long options in the 1980s (`--recursive`, `--verbose`) because complex programs like `gcc` and `tar` ran out of single letters. Modern CLIs increasingly use *subcommands* instead—`git add`, `docker run`, `kubectl get`—a pattern that treats the program as a namespace of operations rather than a bag of flags.

2.3 Code organization

Each utility is a standalone C program, typically between 50 and 500 lines, in its own source file under the appropriate directory. There are no shared libraries between utilities besides `libc` and `libbio`—each program is self-contained.

2.4 Software architecture

Every utility includes the same two Plan 9 headers: `u.h` for basic type definitions and `libc.h` for the system call interface and standard library functions. This is the minimal "include preamble" shared by all Plan 9 C programs.

⟨*plan9 includes* 14⟩≡ (476a 457 433 416b 333 267 266 264 262 261 235 234 228 227 225 224 192 185 179 178 177 176)

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

2.5 Book structure

The chapters are organized by function: file manipulation, search and replace, text processing, comparison, archiving, pipes, bytes, calculators, and miscellaneous. Within each chapter, simpler utilities come first (e.g., `cat` before `ls`), letting the reader build up familiarity with the common patterns before encountering the more complex programs.

Chapter 3

Files and Directories

This chapter covers the utilities that manipulate files and directories—the bread and butter of any operating system's command-line environment. Most of these programs are thin wrappers around one or two system calls, which makes them a good introduction to the Plan 9 system call interface.

The "thin wrapper" claim is concrete enough to spell out as a table. For each utility in this chapter, here is the system call (or two) that does the real work, with everything else being argument parsing, path canonicalization, and output formatting:

utility	real work	system call(s)
cat	copy bytes	read + write
ls	list directory entries	dirread / dirstat
touch	update timestamps	dirwstat / create
mkdir	create directory	create (mode | DMDIR)
rm	unlink file	remove
cp	copy bytes	open + read + write + create + close
mv	rename or copy+remove	rename (or fall back to cp + remove)
chmod	change file mode bits	dirwstat
chgrp	change group ownership	dirwstat
mtime	read modification time	dirstat
pwd	get working directory	fd2path / getwd
wc	count bytes/words/lines	read in a loop
du	recursive size	dirread + recurse

Two patterns fall out of staring at this column. First, an astonishing amount of the work is done by exactly one syscall, `dirstat` / `dirread` / `dirwstat`—they cover not just the obviously-stat-related tools but also `touch`, `chmod`, and `chgrp`, because in Plan 9 "change a file's metadata" is uniformly "call `dirwstat` with the field you want to update." On UNIX the same job needs separate `utimes`, `chmod`, and `chown` syscalls; Plan 9 collapses them into one. Second, the only "creative" tool in the table is `cp`, because it has to compose four primitives and decide between `rename` and `copy+remove` for the cross-device case—and even there the source is roughly 200 lines, the shortest non-trivial copy implementation in this book.

3.1 `cat`

Let's start with the simplest utility. `cat` (short for "concatenate") reads files sequentially and writes them to standard output. Despite its simplicity, it demonstrates the fundamental pattern shared by most utilities: if

arguments are given, process each one; otherwise, process standard input.

⟨*function* `main`*(cat.c)* 16a⟩≡ (176a)
```
void
main(int argc, char *argv[])
{
    int i;
    fdt f;

    argv0 = "cat";
    if(argc == 1)
        cat(STDIN, "<stdin>");
    else for(i=1; i<argc; i++){
        f = open(argv[i], OREAD);
        if(f < 0)
            sysfatal("can't open %s: %r", argv[i]);
        else{
            cat(f, argv[i]);
            close(f);
        }
    }
    exits(nil);
}
```

The actual copy loop is the classic `read`/`write` pair in a fixed-size buffer. Notice the careful error checking: a short write (where `write` returns less than `n`) is treated as a fatal error, and a negative `read` return is reported separately from end-of-file (which returns zero).

⟨*function* `cat` 16b⟩≡ (176a)
```
void
cat(fdt f, char *origin)
{
    char buf[8192];
    long n;

    while((n=read(f, buf, (long)sizeof buf))>0)
        if(write(STDOUT, buf, n)!=n)
            sysfatal("write error copying %s: %r", origin);
    if(n < 0)
        sysfatal("error reading %s: %r", origin);
}
```

3.2 `ls`

`ls` is the most complex utility in this chapter, at over 300 lines. The architecture is a two-phase design: first, collect all directory entries into a global `dirbuf` array of `NDir` structures, then sort and format them for output. This separation allows the sorting and formatting logic to be independent of the directory traversal.

⟨`main`*(ls.c) default case with usage* 16c⟩≡ (17d)
```
fprint(STDERR, "usage: ls [-dlmnpqrstuFQT] [file ...]\n");
exits("usage");
```

Each `NDir` pairs a `Dir` structure (returned by the kernel's `dirstat` or `dirreadall`) with an optional path prefix, used when listing files from multiple directories on the command line.

⟨*struct* `NDir` 16d⟩≡ (176b)
```
struct NDir
{
    // ref_own<Dir>
    Dir *d;
```

```
    // ?? for ls -lr ?
    char    *prefix;
};
```

⟨globals ls.c 17a⟩≡ (176b) 17b▷
```
// growing_array<ref_own<NDir> (size = ndirbuf, valid max = ndir)
NDir*   dirbuf;
int ndirbuf;
int ndir;
```

⟨globals ls.c 17b⟩+≡ (176b) ◁17a 17c▷
```
Biobuf  bin;
```

⟨globals ls.c 17c⟩+≡ (176b) ◁17b 21c▷
```
bool errs = false;
```

⟨*function* main*(ls.c)* 17d⟩≡ (176b)
```
void
main(int argc, char *argv[])
{
    int i;

    Binit(&bin, STDOUT, OWRITE);
    ARGBEGIN{
    ⟨main(ls.c) switch flag character cases 21b⟩
    default:
        ⟨main(ls.c) default case with usage 16c⟩
    }ARGEND

    doquote = needsrcquote;
    quotefmtinstall();
    fmtinstall('M', dirmodefmt);

    ⟨main(ls.c) if lflag 21e⟩
    if(argc == 0)
        errs = ls(".", false);
    else for(i=0; i<argc; i++)
        errs |= ls(argv[i], true);

    output();
    exits(errs? "errors" : nil);
}
```

Basic listing

The `ls` function fills the global `dirbuf` array. For directories, it calls `dirreadall` to get all entries at once, then stores them with the directory name as prefix (when listing multiple paths). For plain files, it stores the single `Dir` entry directly. Note that `output` is called between directories—this ensures entries from different directories appear in separate sorted groups, matching the behavior users expect from `ls dir1 dir2`.

⟨*function* ls 17e⟩≡ (176b)
```
error1
ls(char *s, bool multi)
{
    fdt fd;
    long i, n;
    char *p;
    Dir *db;
```

```
    db = dirstat(s);
    if(db == nil){
    error:
        fprint(STDERR, "ls: %s: %r\n", s);
        return ERROR_1;
    }
    if((db->qid.type&QTDIR) && !dflag){
        free(db);
        output();
        fd = open(s, OREAD);
        if(fd == -1)
            goto error;
        n = dirreadall(fd, &db);
        if(n < 0)
            goto error;
        xcleanname(s);
        growto(ndir+n);
        for(i=0; i<n; i++){
            dirbuf[ndir+i].d = db+i;
            dirbuf[ndir+i].prefix = multi? s : nil;
        }
        ndir += n;
        close(fd);
        // why output() here? will be done in main() anyway
        output();
    }else{
        growto(ndir+1);
        dirbuf[ndir].d = db;
        dirbuf[ndir].prefix = nil;
        xcleanname(s);
        p = utfrrune(s, '/');
        if(p){
            dirbuf[ndir].prefix = s;
            *p = 0;
        }
        ndir++;
        // no output() ? will be done in main
    }
    return OK_0;
}
```

⟨*function* `growto` *(ls.c)* 18a⟩≡ (176b)

```
void
growto(long n)
{
    if(n <= ndirbuf)
        return;
    ndirbuf = n;
    dirbuf=(NDir *)realloc(dirbuf, ndirbuf*sizeof(NDir));
    if(dirbuf == nil){
        fprint(STDERR, "ls: malloc fail\n");
        exits("malloc fail");
    }
}
```

⟨*function* `xcleanname` *(ls.c)* 18b⟩≡ (176b)

```
/*
 * Compress slashes, remove trailing slash.  Don't worry about . and ..
 */
char*
```

```
xcleanname(char *name)
{
    char *r, *w;

    for(r=w=name; *r; r++){
        if(*r=='/' && r>name && *(r-1)=='/')
            continue;
        if(w != r)
            *w = *r;
        w++;
    }
    *w = '\0';
    while(w-1>name && *(w-1)=='/')
        *--w = '\0';
    return name;
}
```

⟨*function* `output` *(ls.c)* 19a⟩≡ (176b)

```
void
output(void)
{
    int i;
    char buf[4096];
    char *s;

    if(!nflag)
        qsort(dirbuf, ndir, sizeof dirbuf[0], (int (*)(const void*, const void*))compar);

    for(i=0; i<ndir; i++)
        dowidths(dirbuf[i].d);

    for(i=0; i<ndir; i++) {
        if(!pflag && (s = dirbuf[i].prefix)) {
            if(strcmp(s, "/") ==0)  /* / is a special case */
                s = "";
            sprint(buf, "%s/%s", s, dirbuf[i].d->name);
            format(dirbuf[i].d, buf);
        } else
            format(dirbuf[i].d, dirbuf[i].d->name);
    }
    ndir = 0;
    Bflush(&bin);
}
```

⟨*function* `compar` *(ls.c)* 19b⟩≡ (176b)

```
ord
compar(const NDir *a, const NDir *b)
{
    ord i;
    Dir *ad, *bd;

    ad = a->d;
    bd = b->d;

    if(tflag){
        if(uflag)
            i = bd->atime-ad->atime;
        else
            i = bd->mtime-ad->mtime;
    }else{
```

```
        if(a->prefix && b->prefix){
            i = strcmp(a->prefix, b->prefix);
            if(i == ORD__EQ)
                i = strcmp(ad->name, bd->name);
        }else if(a->prefix){
            i = strcmp(a->prefix, bd->name);
            if(i == ORD__EQ)
                i = ORD__SUP;  /* a is longer than b */
        }else if(b->prefix){
            i = strcmp(ad->name, b->prefix);
            if(i == ORD__EQ)
                i = ORD__INF; /* b is longer than a */
        }else
            i = strcmp(ad->name, bd->name);
    }
    if(i == ORD__EQ)
        i = (a<b? ORD__INF : ORD__SUP);
    if(rflag)
        i = -i;
    return i;
}
```

⟨*function* `format` *(ls.c)* 20⟩≡ (176b)

```
void
format(Dir *db, char *name)
{
    int i;

    if(sflag)
        Bprint(&bin, "%*llud ",
            swidth, (db->length+1023)/1024);
    if(mflag){
        Bprint(&bin, "[%q] ", db->muid);
        for(i=2+strlen(db->muid); i<mwidth; i++)
            Bprint(&bin, " ");
    }
    if(qflag)
        Bprint(&bin, "(%.16llux %*lud %.2ux) ",
            db->qid.path,
            qwidth, db->qid.vers,
            db->qid.type);
    if(Tflag)
        Bprint(&bin, "%c ", (db->mode&DMTMP)? 't': '-');

    if(lflag)
        Bprint(&bin, "%M %C %*ud %*q %*q %*llud %s ",
            db->mode, db->type,
            vwidth, db->dev,
            -uwidth, db->uid,
            -gwidth, db->gid,
            lwidth, db->length,
            asciitime(uflag? db->atime: db->mtime));

    Bprint(&bin, Qflag? "%s%s\n": "%q%s\n", name, fileflag(db));
}
```

Advanced listings

`ls` supports 14 flags, each stored as a simple boolean. The flags control sorting (`-t` for time, `-r` for reverse, `-n`

to suppress sorting), output format (`-l` for long, `-s` for size in KB, `-q` for qid, `-m` for last modifier), and filtering (`-d` to not descend into directories, `-p` to suppress the path prefix, `-F` to append a type indicator).

⟨global flags ls.c 21a⟩≡ (176b)
```
// ??
bool dflag;
// ??
bool lflag;
// ??
bool mflag;
// ??
bool nflag;
// ??
bool pflag;
// ??
bool qflag;
// ??
bool Qflag;
// ??
bool rflag;
// ??
bool sflag;
// ??
bool tflag;
// ??
bool Tflag;
// ??
bool uflag;
// ??
bool Fflag;
```

⟨main*(ls.c) switch flag character cases* 21b⟩≡ (17d)
```
case 'F':   Fflag = true; break;
case 'd':   dflag = true; break;
case 'l':   lflag = true; break;
case 'm':   mflag = true; break;
case 'n':   nflag = true; break;
case 'p':   pflag = true; break;
case 'q':   qflag = true; break;
case 'Q':   Qflag = true; break;
case 'r':   rflag = true; break;
case 's':   sflag = true; break;
case 't':   tflag = true; break;
case 'T':   Tflag = true; break;
case 'u':   uflag = true; break;
```

⟨globals ls.c 21c⟩+≡ (176b) ◁17c 21d▷
```
int swidth;         /* max width of -s size */
int qwidth;         /* max width of -q version */
int vwidth;         /* max width of dev */
int uwidth;         /* max width of userid */
int mwidth;         /* max width of muid */
int lwidth;         /* max width of length */
int gwidth;         /* max width of groupid */
```

⟨globals ls.c 21d⟩+≡ (176b) ◁21c
```
ulong   clk;
```

⟨main*(ls.c) if* `lflag` 21e⟩≡ (17d)
```
if(lflag)
    clk = time(0);
```

⟨*function* `fileflag` *(ls.c)* 22a⟩≡ (176b)

```
char*
fileflag(Dir *db)
{
    if(!Fflag)
        return "";
    if(QTDIR & db->qid.type)
        return "/";
    if(0111 & db->mode)
        return "*";
    return "";
}
```

The long-listing output of `ls -l` is computed in two passes over the directory array. `dowidths` is the first pass: it walks every `Dir` and keeps a running maximum of the width each column would need; `formatX` is the second pass and actually prints each entry using those maxima as `%*ud` field widths. Seven per-column globals (`swidth`, `qwidth`, `vwidth`, `uwidth`, `mwidth`, `lwidth`, `gwidth`) hold those maxima, one per optional flag:

```
Dir 0: [ size=42 uid=bob   gid=staff  length=128  mtime=... ]
Dir 1: [ size=7  uid=alice gid=wheel  length=1024 mtime=... ]
Dir 2: [ size=1  uid=root  gid=sys    length=9    mtime=... ]
          \___/  \_____/   \_____/    \________/
            |       |         |          |
            |       |         |          v
dowidths:  swidth=2,uwidth=5,gwidth=5, lwidth=4
            |       |         |          |
            v       v         v          v
format:     "42 bob   staff   128 ..."
            " 7 alice wheel  1024 ..."
            " 1 root  sys       9 ..."
```

The reason for the split is that column widths cannot be known until the widest row has been seen, and Plan 9's `Bprint` has no deferred-flush or reflow machinery. The naive alternative would be to sprint each row into a string, remember them, and pad at the end—but that doubles the memory cost of `ls` on a large directory. The two-pass arrangement keeps each `Dir` live only once and never allocates a scratch row.

⟨*function* `dowidths` *(ls.c)* 22b⟩≡ (176b)

```
void
dowidths(Dir *db)
{
    char buf[256];
    int n;

    if(sflag) {
        n = sprint(buf, "%llud", (db->length+1023)/1024);
        if(n > swidth)
            swidth = n;
    }
    if(qflag) {
        n = sprint(buf, "%lud", db->qid.vers);
        if(n > qwidth)
            qwidth = n;
    }
    if(mflag) {
        n = snprint(buf, sizeof buf, "[%q]", db->muid);
        if(n > mwidth)
            mwidth = n;
```

```
    }
    if(lflag) {
        n = sprint(buf, "%ud", db->dev);
        if(n > vwidth)
            vwidth = n;
        n = sprint(buf, "%q", db->uid);
        if(n > uwidth)
            uwidth = n;
        n = sprint(buf, "%q", db->gid);
        if(n > gwidth)
            gwidth = n;
        n = sprint(buf, "%llud", db->length);
        if(n > lwidth)
            lwidth = n;
    }
}
```

⟨*function* `asciitime` *(ls.c)* 23a⟩≡ (176b)

```
char*
asciitime(long l)
{
    static char buf[32];
    char *t;

    t = ctime(l);
    /* 6 months in the past or a day in the future */
    if(l<clk-180L*24*60*60 || clk+24L*60*60<l){
        memmove(buf, t+4, 7);       /* month and day */
        memmove(buf+7, t+23, 5);        /* year */
    }else
        memmove(buf, t+4, 12);      /* skip day of week */
    buf[12] = '\0';
    return buf;
}
```

3.3 `touch`

`touch` updates a file's modification time, creating the file if it doesn't exist (unless `-c` is given). It demonstrates the `dirwstat` system call, which is Plan 9's way of modifying file metadata—the counterpart to UNIX's `utime` and `chmod` combined into a single operation.

⟨*function* `usage` *(touch.c)* 23b⟩≡ (177a)

```
void
usage(void)
{
    fprint(STDERR, "usage: touch [-c] [-t time] files\n");
    exits("usage");
}
```

⟨*global* `now` *(touch.c)* 23c⟩≡ (177a)

```
ulong now;
```

⟨*function* `main` *(touch.c)* 23d⟩≡ (177a)

```
void
main(int argc, char **argv)
{
    char *t, *s;
    bool nocreate = false;
```

```
    errorn status = OK_0;

    now = time(0);
    ARGBEGIN{
    case 't':
        t = EARGF(usage());
        now = strtoul(t, &s, 0);
        if(s == t || *s != '\0')
            usage();
        break;
    case 'c':
        nocreate = true;
        break;
    default:
        usage();
    }ARGEND

    if(!*argv)
        usage();
    while(*argv)
        status += touch(nocreate, *argv++);
    if(status)
        exits("touch");
    exits(nil);
}
```

The `touch` function first tries `dirwstat` to update the modification time of an existing file. `nulldir` initializes a `Dir` structure so that all fields are "don't care" (filled with `~0` values), and then only `mtime` is set to the desired time. If `dirwstat` fails (file doesn't exist) and `-c` was not given, `touch` creates the file with `OEXCL` to avoid overwriting an existing file that appeared between the failed `dirwstat` and the `create`.

⟨*function* `touch` 24⟩≡ (177a)

```
error1
touch(bool nocreate, char *name)
{
    Dir stbuff;
    fdt fd;

    nulldir(&stbuff);
    stbuff.mtime = now;
    if(dirwstat(name, &stbuff) >= 0)
        return OK_0;
    //else
    if(nocreate){
        fprint(STDERR, "touch: %s: cannot wstat: %r\n", name);
        return ERROR_1;
    }
    if((fd = create(name, OREAD|OEXCL, 0666)) < 0){
        fprint(STDERR, "touch: %s: cannot create: %r\n", name);
        return ERROR_1;
    }
    dirfwstat(fd, &stbuff);
    close(fd);
    return OK_0;
}
```

3.4 mkdir

`mkdir` creates directories using the `create` system call with the `DMDIR` flag—in Plan 9, there is no separate

`mkdir` system call.

⟨*function* `usage` *(mkdir.c)* 25a⟩≡ (177b)
```
void
usage(void)
{
    fprint(STDERR, "usage: mkdir [-p] [-m mode] dir...\n");
    exits("usage");
}
```

⟨*global* `mode` *(mkdir.c)* 25b⟩≡ (177b)
```
ulong mode = 0777L;
```

⟨*global* `e` *(mkdir.c)* 25c⟩≡ (177b)
```
// error
char *e;
```

⟨*function* `main` *(mkdir.c)* 25d⟩≡ (177b)
```
void
main(int argc, char *argv[])
{
    int i;
    bool pflag = false;
    char *m;

    ARGBEGIN{
    case 'm':
        m = ARGF();
        if(m == nil)
            usage();
        mode = strtoul(m, &m, 8);
        if(mode > 0777)
            usage();
        break;
    case 'p':
        pflag = true;
        break;
    default:
        usage();
    }ARGEND

    for(i=0; i<argc; i++){
        if(pflag)
            mkdirp(argv[i]);
        else
            makedir(argv[i]);
    }
    exits(e);
}
```

The `-p` flag triggers `mkdirp`, which walks the path from left to right, temporarily replacing each `/` with a null byte to create each intermediate directory. This is the same technique used by GNU `mkdir -p`. If any intermediate directory already exists, it is silently skipped.

⟨*function* `makedir` 25e⟩≡ (177b)
```
errorneg1
makedir(char *s)
{
    fdt f;

    if(access(s, AEXIST) == 0){
```

```
        fprint(STDERR, "mkdir: %s already exists\n", s);
        e = "error";
        return ERROR_NEG1;
    }
    f = create(s, OREAD, DMDIR | mode);
    if(f < 0){
        fprint(STDERR, "mkdir: can't create %s: %r\n", s);
        e = "error";
        return ERROR_NEG1;
    }
    close(f);
    return OK_0;
}
```

⟨*function* `mkdirp` 26a⟩≡ (177b)

```
void
mkdirp(char *s)
{
    char *p;

    for(p=strchr(s+1, '/'); p; p=strchr(p+1, '/')){
        *p = '\0';
        if(access(s, AEXIST) != 0 && makedir(s) == ERROR_NEG1)
            return;
        *p = '/';
    }
    if(access(s, AEXIST) != 0)
        makedir(s);
}
```

3.5 `rm`

`rm` first tries a simple `remove` system call on each argument. If that fails and `-r` is set, it checks whether the path is a directory and recurses into it with `rmdir_`.

⟨*global* `ignerr` *(rm.c)* 26b⟩≡ (177c)

```
// for 'rm -f'
bool ignerr = false;
```

⟨*global* `errbuf` *(rm.c)* 26c⟩≡ (177c)

```
char errbuf[ERRMAX];
```

⟨*function* `main` *(rm.c)* 26d⟩≡ (177c)

```
void
main(int argc, char *argv[])
{
    int i;
    // rm -r
    bool recurse = false;
    char *f;
    Dir *db;

    ARGBEGIN{
    case 'r':
        recurse = true;
        break;
    case 'f':
        ignerr = true;
        break;
```

```
    default:
        fprint(STDERR, "usage: rm [-fr] file ...\n");
        exits("usage");
    }ARGEND

    for(i=0; i<argc; i++){
        f = argv[i];
        if(remove(f) != ERROR_NEG1)
            continue;
        //else
        db = nil;
        if(recurse && (db=dirstat(f))!=nil && (db->qid.type&QTDIR))
            rmdir_(f);
        else
            err(f);
        free(db);
    }
    exits(errbuf);
}
```

Recursive deletion uses a two-pass strategy over the directory entries: the first pass tries to `remove` every entry (files and empty directories will succeed), marking successful removals by clearing the `QTDIR` bit. The second pass recurses into any remaining subdirectories that couldn't be removed in the first pass. This avoids unnecessary recursion for the common case of flat directories.

⟨*function* `rmdir` 27⟩≡ (177c)

```
/*
 * f is a non-empty directory. Remove its contents and then it.
 */
void
rmdir_(char *f)
{
    char *name;
    int i, j, n, ndir, nname;
    fdt fd;
    Dir *dirbuf;

    fd = open(f, OREAD);
    if(fd < 0){
        err(f);
        return;
    }
    //else
    n = dirreadall(fd, &dirbuf);
    close(fd);
    if(n < 0){
        err("dirreadall");
        return;
    }

    nname = strlen(f)+1+STATMAX+1;  /* plenty! */
    name = malloc(nname);
    if(name == nil){
        err("memory allocation");
        return;
    }

    ndir = 0;
    for(i=0; i<n; i++){
        snprint(name, nname, "%s/%s", f, dirbuf[i].name);
```

```
            if(remove(name) != ERROR_NEG1)
                dirbuf[i].qid.type = QTFILE;    /* so we won't recurse */
            else{
                if(dirbuf[i].qid.type & QTDIR)
                    ndir++;
                else
                    err(name);
            }
        }
        if(ndir)
            for(j=0; j<n; j++)
                if(dirbuf[j].qid.type & QTDIR){
                    snprint(name, nname, "%s/%s", f, dirbuf[j].name);
                    // recurse
                    rmdir_(name);
                }
        if(remove(f) == ERROR_NEG1)
            err(f);
        free(name);
        free(dirbuf);
    }
```

⟨*function* `err` *(rm.c)* 28a⟩≡ (177c)

```
    void
    err(char *f)
    {
        if(!ignerr){
            errbuf[0] = '\0';
            errstr(errbuf, sizeof errbuf);
            fprint(STDERR, "rm: %s: %s\n", f, errbuf);
        }
    }
```

3.6 cp

`cp` copies files. The interesting details are the `samefile` check (comparing qid paths and device numbers to prevent copying a file onto itself) and the optional metadata preservation flags: `-x` preserves time and mode, `-g` preserves group, and `-u` preserves owner (and implies `-g`).

⟨*global* `failed` *(cp.c)* 28b⟩≡ (178a)

```
    bool failed;
```

⟨*global flags(cp.c)* 28c⟩≡ (178a)

```
    // keep gid
    bool gflag = false;
    // keep uid (-u implies -g)
    bool uflag = false;
    // keep time and mode
    bool xflag = false;
```

⟨*label* `usage` *in* `main` *(cp.c)* 28d⟩≡ (29a)

```
    usage:
        fprint(STDERR, "usage:\tcp [-gux] fromfile tofile\n");
        fprint(STDERR, "\tcp [-x] fromfile ... todir\n");
        exits("usage");
```

⟨*function* `main`*(cp.c)* 29a⟩≡ (178a)

```
void
main(int argc, char *argv[])
{
    Dir *dirb;
    int i;
    bool todir = false;

    ARGBEGIN {
    case 'g':
        gflag = true;
        break;
    case 'u':
        uflag = true;
        gflag = true;
        break;
    case 'x':
        xflag = true;
        break;
    default:
        goto usage;
    } ARGEND

    if(argc < 2)
        goto usage;

    dirb = dirstat(argv[argc-1]);
    if(dirb!=nil && (dirb->mode&DMDIR))
        todir=true;
    if(argc>2 && !todir){
        fprint(STDERR, "cp: %s not a directory\n", argv[argc-1]);
        exits("bad usage");
    }

    for(i=0; i<argc-1; i++)
        copy(argv[i], argv[argc-1], todir);

    if(failed)
        exits("errors");
    exits(nil);
⟨label usage in main(cp.c) 28d⟩
}
```

⟨*function* `copy` 29b⟩≡ (178a)

```
void
copy(char *from, char *to, bool todir)
{
    Dir *dirb, dirt;
    char name[256];
    // fd from, fd to
    fdt fdf, fdt;
    int mode;

    if(todir){
        char *s, *elem;
        elem=s=from;
        while(*s++)
            if(s[-1]=='/')
                elem=s;
        sprint(name, "%s/%s", to, elem);
```

```
        to=name;
    }

    if((dirb=dirstat(from))==nil){
        fprint(STDERR,"cp: can't stat %s: %r\n", from);
        failed = true;
        return;
    }
    mode = dirb->mode;
    if(mode&DMDIR){
        fprint(STDERR, "cp: %s is a directory\n", from);
        free(dirb);
        failed = true;
        return;
    }
    if(samefile(dirb, from, to)){
        free(dirb);
        failed = true;
        return;
    }
    mode &= 0777;
    fdf=open(from, OREAD);
    if(fdf<0){
        fprint(STDERR, "cp: can't open %s: %r\n", from);
        free(dirb);
        failed = true;
        return;
    }
    fdt=create(to, OWRITE, mode);
    if(fdt<0){
        fprint(STDERR, "cp: can't create %s: %r\n", to);
        close(fdf);
        free(dirb);
        failed = true;
        return;
    }
    if(copy1(fdf, fdt, from, to)==OK_0 && (xflag || gflag || uflag)){
        nulldir(&dirt);
        if(xflag){
            dirt.mtime = dirb->mtime;
            dirt.mode = dirb->mode;
        }
        if(uflag)
            dirt.uid = dirb->uid;
        if(gflag)
            dirt.gid = dirb->gid;
        if(dirfwstat(fdt, &dirt) < 0)
            fprint(STDERR, "cp: warning: can't wstat %s: %r\n", to);
    }
    free(dirb);
    close(fdf);
    close(fdt);
}
```

⟨*constant* DEFB*(cp.c)* 30a⟩≡ (178a)

```
#define DEFB    (8*1024)
```

⟨*function* copy1 30b⟩≡ (178a)

```
errorneg1
copy1(fdt fdf, fdt fdt, char *from, char *to)
```

```
{
    char *buf;
    long n, n1, rcount;
    errorneg1 rv = OK_0;
    char err[ERRMAX];

    buf = malloc(DEFB);
    /* clear any residual error */
    err[0] = '\0';
    errstr(err, ERRMAX);

    // copy (read and write) as long as n read <= 0
    for(rcount=0;; rcount++) {
        n = read(fdf, buf, DEFB);
        if(n <= 0)
            break;
        n1 = write(fdt, buf, n);
        if(n1 != n) {
            fprint(STDERR, "cp: error writing %s: %r\n", to);
            failed = true;
            rv = ERROR_NEG1;
            break;
        }
    }
    if(n < 0) {
        fprint(2, "cp: error reading %s: %r\n", from);
        failed = true;
        rv = ERROR_NEG1;
    }
    free(buf);
    return rv;
}
```

⟨*function* `samefile`*(cp.c)* 31⟩≡ (178a)

```
bool
samefile(Dir *a, char *an, char *bn)
{
    Dir *b;
    bool ret = false;

    b=dirstat(bn);
    if(b != nil)
    if(b->qid.type==a->qid.type)
    if(b->qid.path==a->qid.path)
    if(b->qid.vers==a->qid.vers)
    if(b->dev==a->dev)
    if(b->type==a->type){
        fprint(STDERR, "cp: %s and %s are the same file\n", an, bn);
        ret = true;
    }
    free(b);
    return ret;
}
```

3.7 `mv`

`mv` has a subtle strategy. Plan 9 has no `rename` system call, so `mv` first tries to rename the file using `dirwstat` to change its name field—this only works if the source and destination are on the same file server directory. If that

fails (different directories or different servers), `mv` falls back to a full copy-then-remove. This fallback means `mv` across file servers works but is slow, while same-directory renames are instantaneous.

⟨*function* `main`*(mv.c)* 32a⟩≡ (178b)

```
void
main(int argc, char *argv[])
{
    int i;
    errorn failed = OK_0;
    Dir *dirto, *dirfrom;
    char *todir, *toelem;

    if(argc<3){
        fprint(STDERR, "usage: mv fromfile tofile\n");
        fprint(STDERR, "   mv fromfile ... todir\n");
        exits("bad usage");
    }

    /* prepass to canonicalise names before splitting, etc. */
    for(i=1; i < argc; i++)
        cleanname(argv[i]);

    if((dirto = dirstat(argv[argc-1])) != nil && (dirto->mode&DMDIR)){
        dirfrom = nil;
        if(argc == 3
        && (dirfrom = dirstat(argv[1])) != nil
        && (dirfrom->mode & DMDIR))
            split(argv[argc-1], &todir, &toelem); /* mv dir1 dir2 */
        else{               /* mv file... dir */
            todir = argv[argc-1];
            toelem = nil;       /* toelem will be fromelem */
        }
        free(dirfrom);
    }else
        split(argv[argc-1], &todir, &toelem);   /* mv file1 file2 */
    free(dirto);
    if(argc>3 && toelem != nil){
        fprint(STDERR, "mv: %s not a directory\n", argv[argc-1]);
        exits("bad usage");
    }

    for(i=1; i < argc-1; i++)
        if(mv(argv[i], todir, toelem) == ERROR_NEG1)
            failed++;
    if(failed)
        exits("failure");
    exits(nil);
}
```

⟨*function* `mv` 32b⟩≡ (178b)

```
errorneg1
mv(char *from, char *todir, char *toelem)
{
    errorneg1 stat;
    Dir *dirb;

    dirb = dirstat(from);
    if(dirb == nil){
        fprint(STDERR, "mv: can't stat %s: %r\n", from);
        return ERROR_NEG1;
```

```
    }
    stat = mv1(from, dirb, todir, toelem);
    free(dirb);
    return stat;
}
```

⟨*function* mv1 33⟩≡ (178b)

```
errorneg1
mv1(char *from, Dir *dirb, char *todir, char *toelem)
{
    // fd from, fd to
    fdt fdf, fdt;
    int i, j;
    errorneg1 stat;
    char toname[4096], fromname[4096];
    char *fromdir, *fromelem;
    Dir *dirt, null;

    strncpy(fromname, from, sizeof fromname);
    split(from, &fromdir, &fromelem);
    if(toelem == nil)
        toelem = fromelem;
    i = strlen(toelem);
    if(i==0){
        fprint(STDERR, "mv: null last name element moving %s\n", fromname);
        return ERROR_NEG1;
    }
    j = strlen(todir);
    if(i + j + 2 > sizeof toname){
        fprint(STDERR, "mv: path too big (max %d): %s/%s\n",
            sizeof toname, todir, toelem);
        return ERROR_NEG1;
    }
    memmove(toname, todir, j);
    toname[j] = '/';
    memmove(toname+j+1, toelem, i);
    toname[i+j+1] = '\0';

    if(samefile(fromdir, todir)){
        if(samefile(fromname, toname)){
            fprint(STDERR, "mv: %s and %s are the same\n",
                fromname, toname);
            return ERROR_NEG1;
        }

        /* remove target if present */
        dirt = dirstat(toname);
        if(dirt != nil) {
            hardremove(toname);
            free(dirt);
        }

        /* try wstat */
        nulldir(&null);
        null.name = toelem;
        if(dirwstat(fromname, &null) >= 0)
            return OK_0;
        if(dirb->mode & DMDIR){
            fprint(STDERR, "mv: can't rename directory %s: %r\n",
                fromname);
```

```
            return ERROR_NEG1;
        }
    }
    /*
     * Renaming won't work --- must copy
     */
    if(dirb->mode & DMDIR){
        fprint(STDERR, "mv: %s is a directory, not copied to %s\n",
            fromname, toname);
        return ERROR_NEG1;
    }
    fdf = open(fromname, OREAD);
    if(fdf < 0){
        fprint(STDERR, "mv: can't open %s: %r\n", fromname);
        return ERROR_NEG1;
    }

    dirt = dirstat(toname);
    if(dirt != nil && (dirt->mode & DMAPPEND))
        hardremove(toname);  /* because create() won't truncate file */
    free(dirt);

    fdt = create(toname, OWRITE, dirb->mode);
    if(fdt < 0){
        fprint(STDERR, "mv: can't create %s: %r\n", toname);
        close(fdf);
        return ERROR_NEG1;
    }
    stat = copy1(fdf, fdt, fromname, toname);
    close(fdf);

    if(stat >= 0){
        nulldir(&null);
        null.mtime = dirb->mtime;
        null.mode = dirb->mode;
        dirfwstat(fdt, &null);  /* ignore errors; e.g. user none always fails */
        if(remove(fromname) < 0){
            fprint(STDERR, "mv: can't remove %s: %r\n", fromname);
            stat = ERROR_NEG1;
        }
    }
    close(fdt);
    return stat;
}
```

⟨*function* copy1*(mv.c)* 34⟩≡ (178b)

```
errorneg1
copy1(fdt fdf, fdt fdt, char *from, char *to)
{
    char buf[8192];
    long n, n1;

    while ((n = read(fdf, buf, sizeof buf)) > 0) {
        n1 = write(fdt, buf, n);
        if(n1 != n){
            fprint(STDERR, "mv: error writing %s: %r\n", to);
            return ERROR_NEG1;
        }
    }
    if(n < 0){
```

```
        fprint(STDERR, "mv: error reading %s: %r\n", from);
        return ERROR_NEG1;
    }
    return OK_0;
}
```

⟨*function* `split` *(mv.c)* 35a⟩≡ (178b)

```
void
split(char *name, char **pdir, char **pelem)
{
    char *s;

    s = utfrrune(name, '/');
    if(s){
        *s = '\0';
        *pelem = s+1;
        *pdir = name;
    }else if(strcmp(name, "..") == ORD__EQ){
        *pdir = "..";
        *pelem = ".";
    }else{
        *pdir = ".";
        *pelem = name;
    }
}
```

⟨*function* `samefile` *(mv.c)* 35b⟩≡ (178b)

```
bool
samefile(char *a, char *b)
{
    Dir *da, *db;
    bool ret;

    if(strcmp(a, b) == ORD__EQ)
        return true;
    da = dirstat(a);
    db = dirstat(b);
    ret = (da != nil && db != nil &&
        da->qid.type==db->qid.type &&
        da->qid.path==db->qid.path &&
        da->qid.vers==db->qid.vers &&
        da->dev==db->dev &&
        da->type==db->type);
    free(da);
    free(db);
    return ret;
}
```

⟨*function* `hardremove` *(mv.c)* 35c⟩≡ (178b)

```
void
hardremove(char *a)
{
    if(remove(a) == ERROR_NEG1){
        fprint(STDERR, "mv: can't remove %s: %r\n", a);
        exits("mv");
    }
    //????
    while(remove(a) != ERROR_NEG1)
        ;
}
```

3.8 chmod

`chmod` accepts either an octal mode or a symbolic specification like `ugw+`. The symbolic parser in `parsemode` builds a mask (which bits to change) and a mode (what to set them to), then applies them with the expression `(old & ~mask) | (mode & mask)`. Note the Plan 9-specific mode bits: `DMAPPEND` (append-only), `DMEXCL` (exclusive-use), and `DMTMP` (temporary), which have no direct UNIX counterpart.

⟨macros chmod.c 36a⟩≡ (178c)
```
#define U(x) (x<<6)
#define G(x) (x<<3)
#define O(x) (x)
#define A(x) (U(x)|G(x)|O(x))

#define DMRWE (DMREAD|DMWRITE|DMEXEC)
```

⟨*function* main*(chmod.c)* 36b⟩≡ (178c)
```
void
main(int argc, char *argv[])
{
    int i;
    Dir *dir, ndir;
    ulong mode, mask;
    char *p;

    if(argc < 3){
        fprint(STDERR, "usage: chmod 0777 file ... or chmod [who]op[rwxalt] file ...\n");
        exits("usage");
    }
    mode = strtol(argv[1], &p, 8);
    if(*p == '\0')
        mask = A(DMRWE);
    else if(parsemode(argv[1], &mask, &mode) == ERROR_0){
        fprint(STDERR, "chmod: bad mode: %s\n", argv[1]);
        exits("mode");
    }
    nulldir(&ndir);
    for(i=2; i<argc; i++){
        dir = dirstat(argv[i]);
        if(dir == nil){
            fprint(STDERR, "chmod: can't stat %s: %r\n", argv[i]);
            continue;
        }
        ndir.mode = (dir->mode & ~mask) | (mode & mask);
        free(dir);
        if(dirwstat(argv[i], &ndir)==-1){
            fprint(STDERR, "chmod: can't wstat %s: %r\n", argv[i]);
            continue;
        }
    }
    exits(nil);
}
```

⟨*function* parsemode*(chmod.c)* 36c⟩≡ (178c)
```
error0
parsemode(char *spec, ulong *pmask, ulong *pmode)
{
    ulong mode, mask;
    bool done;
    int op;
    char *s;
```

```
    s = spec;
    mask = DMAPPEND | DMEXCL | DMTMP;
    for(done=false; !done; ){
        switch(*s){
        case 'u':
            mask |= U(DMRWE); break;
        case 'g':
            mask |= G(DMRWE); break;
        case 'o':
            mask |= O(DMRWE); break;
        case 'a':
            mask |= A(DMRWE); break;
        case 0:
            return ERROR_0;
        default:
            done = true;
        }
        if(!done)
            s++;
    }
    if(s == spec)
        mask |= A(DMRWE);
    op = *s++;
    if(op != '+' && op != '-' && op != '=')
        return ERROR_0;
    mode = 0;
    for(; *s ; s++){
        switch(*s){
        case 'r':
            mode |= A(DMREAD); break;
        case 'w':
            mode |= A(DMWRITE); break;
        case 'x':
            mode |= A(DMEXEC); break;
        case 'a':
            mode |= DMAPPEND; break;
        case 'l':
            mode |= DMEXCL; break;
        case 't':
            mode |= DMTMP; break;
        default:
            return ERROR_0;
        }
    }
    if(*s != 0)
        return ERROR_0;
    if(op == '+' || op == '-')
        mask &= mode;
    if(op == '-')
        mode = ~mode;
    *pmask = mask;
    *pmode = mode;
    return OK_1;
}
```

3.9 chgrp

⟨*global* `uflag` *(chgrp.c)* 38a⟩≡ (179a)
```
// change uid of file (instead of gid)
bool uflag;
```

⟨*function* `main` *(chgrp.c)* 38b⟩≡ (179a)
```
void
main(int argc, char *argv[])
{
    int i;
    Dir dir;
    char *group;
    char *errs = nil;

    ARGBEGIN {
    default:
    usage:
        fprint(STDERR, "usage: chgrp [ -uo ] group file ....\n");
        exits("usage");
        return;
    case 'u':
    case 'o':
        uflag = true;
        break;
    } ARGEND
    if(argc < 1)
        goto usage;

    group = argv[0];
    for(i=1; i<argc; i++){
        nulldir(&dir);
        if(uflag)
            dir.uid = group;
        else
            dir.gid = group;
        if(dirwstat(argv[i], &dir) == ERROR_NEG1) {
            fprint(STDERR, "chgrp: can't wstat %s: %r\n", argv[i]);
            errs = "can't wstat";
            continue;
        }
    }
    exits(errs);
}
```

3.10 mtime

⟨*function* `usage` *(mtime.c)* 38c⟩≡ (179b)
```
void
usage(void)
{
    fprint(STDERR, "usage: mtime file...\n");
    exits("usage");
}
```

⟨*function* `main` *(mtime.c)* 38d⟩≡ (179b)
```
void
main(int argc, char **argv)
```

```
{
    bool errors = false;
    int i;
    Dir *d;

    ARGBEGIN{
    default:
        usage();
    }ARGEND

    for(i=0; i<argc; i++){
        if((d = dirstat(argv[i])) == nil){
            fprint(STDERR, "stat %s: %r\n", argv[i]);
            errors = true;
        }else{
            print("%11lud %s\n", d->mtime, argv[i]);
            free(d);
        }
    }
    exits(errors ? "errors" : nil);
}
```

3.11 pwd

⟨*function* `main`*(pwd.c)* 39a⟩≡

```
void
main(int argc, char *argv[])
{
    char pathname[512];

    USED(argc, argv);
    if(getwd(pathname, sizeof(pathname)) == ERROR_0) {
        fprint(STDERR, "pwd: %r\n");
        exits("getwd");
    }
    // else
    print("%s\n", pathname);
    exits(nil);
}
```

3.12 wc

`wc` counts lines, words, runes, bad runes, and characters. The Plan 9 version adds `-r` (rune count) and `-b` (bad rune count) to the traditional UNIX `-lwc` flags, reflecting Plan 9's native UTF-8 support. The counting loop uses `Bgetrune` from the buffered I/O library to decode UTF-8 one rune at a time, and the word-counting state machine toggles between `Space` and `Word` on whitespace transitions.

⟨`globals wc.c` 39b⟩≡ (179c)

```
/* flags, per-file counts, and total counts */
static bool pline, pword, prune, pbadr, pchar;
static uvlong nline, nword, nrune, nbadr, nchar;
static uvlong tnline, tnword, tnrune, tnbadr, tnchar;
```

⟨*function* main*(wc.c)* 40a⟩≡ (179c)

```
void
main(int argc, char *argv[])
{
    char *sts = nil;
    Biobuf sin, *bin;
    int i;

    ARGBEGIN {
    case 'l': pline = true; break;
    case 'w': pword = true; break;
    case 'c': pchar = true; break;
    // plan9 new: -r for runes
    case 'r': prune = true; break;
    // ??
    case 'b': pbadr = true; break;
    default:
        fprint(STDERR, "Usage: %s [-lwrbc] [file ...]\n", argv0);
        exits("usage");
    } ARGEND
    if(!pline && !pword && !prune && !pbadr && !pchar){
        // defaults
        pline = true;
        pword = true;
        pchar = true;
    }
    if(argc == 0){
        Binit(&sin, STDIN, OREAD);
        wc(&sin);
        report(nline, nword, nrune, nbadr, nchar, nil);
        Bterm(&sin);
    }else{
        for(i = 0; i < argc; i++){
            bin = Bopen(argv[i], OREAD);
            if(bin == nil){
                perror(argv[i]);
                sts = "can't open";
                continue;
            }
            wc(bin);
            report(nline, nword, nrune, nbadr, nchar, argv[i]);
            Bterm(bin);
        }
        if(argc>1)
            report(tnline, tnword, tnrune, tnbadr, tnchar, "total");
    }
    exits(sts);
}
```

⟨enum wc.c 40b⟩≡ (179c)

```
enum{Space, Word};
```

⟨*function* wc 40c⟩≡ (179c)

```
static void
wc(Biobuf *bin)
{
    int where;
    long r;

    nline = 0;
```

```
    nword = 0;
    nrune = 0;
    nbadr = 0;
    where = Space;
    while ((long)(r = Bgetrune(bin)) >= 0) {
        nrune++;
        if(r == Runeerror) {
            nbadr++;
            continue;
        }
        if(r == '\n')
            nline++;
        if(where == Word){
            if(isspacerune(r))
                where = Space;
        }else
            if(isspacerune(r) == 0){
                where = Word;
                nword++;
            }
    }
    nchar = Boffset(bin);
    tnline += nline;
    tnword += nword;
    tnrune += nrune;
    tnbadr += nbadr;
    tnchar += nchar;
}
```

⟨*function* `report` *(wc.c)* 41⟩≡ (179c)

```
static void
report(uvlong nline, uvlong nword, uvlong nrune, uvlong nbadr, uvlong nchar, char *fname)
{
    char line[1024], *s, *e;

    s = line;
    e = line + sizeof line;
    line[0] = '\0';
    if(pline)
        s = seprint(s, e, " %7llud", nline);
    if(pword)
        s = seprint(s, e, " %7llud", nword);
    if(prune)
        s = seprint(s, e, " %7llud", nrune);
    if(pbadr)
        s = seprint(s, e, " %7llud", nbadr);
    if(pchar)
        s = seprint(s, e, " %7llud", nchar);
    if(fname != nil)
        seprint(s, e, " %s",   fname);
    print("%s\n", line+1);
}
```

3.13 du

Chapter 4

The Line Editor `ed`

Chapter 5

Search and Replace

This chapter covers the search and replace utilities, from the foundational `grep` to `sed` and `tr`. Even with modern IDEs offering semantic search, these tools remain indispensable for quick searches across large codebases and for scripted text transformations.

5.1 `grep`

The name `grep` comes from the `ed` command `g/re/p` ("globally search for a regular expression and print"). The Plan 9`grep` has its own built-in regexp engine rather than using `libregexp`—it includes a yacc grammar for parsing regular expressions and a DFA-based matching engine for speed. The `-e` and `-f` flags allow specifying patterns on the command line or reading them from a file, a design that influenced tools like Semgrep decades later.

⟨*function* `usage`*(grep)* 43a⟩≡ (332)

```
void
usage(void)
{
    fprint(STDERR, "usage: grep [-%s] [-e pattern] [-f patternfile] [file ...]\n", validflags);
    exits("usage");
}
```

⟨*constant* `validflags`*(grep)* 43b⟩≡ (332)

```
char *validflags = "bchiLlnsv";
```

⟨*enum* `grep_flags` 43c⟩≡ (333)

```
enum
{
    /* count */
    Cflag       = 1<<0,
    /* do not print file name in output */
    Hflag       = 1<<1,
    /* fold upper-lower */
    Iflag       = 1<<2,
    /* print only name of file if any match */
    Llflag      = 1<<3,
    /* print only name of file if any non match */
    LLflag      = 1<<4,
    /* count only */
    Nflag       = 1<<5,
    /* status only */
    Sflag       = 1<<6,
    /* inverse match */
    Vflag       = 1<<7,
```

```
    /* dont buffer output */
    Bflag       = 1<<8
};
```

5.1.1 Data structures

The regexp is represented as a linked list of `Re` nodes, each tagged with a type. `Tclass` matches a character range (`lo` to `hi`), `Tor` represents alternation, `Talt` represents a branch point (used for repetition), `Tbegin` and `Tend` anchor to line boundaries, and `Tcase` is an optimization that dispatches on the first byte through a 256-entry lookup table. The `Re2` structure is a pair of `beg`/`end` pointers used during parsing—the `end` pointer collects unresolved `next` links that `patchnext` will later fill in, similar to backpatching in a compiler.

⟨*struct* `Re` *(grep)* 44a⟩≡ (333)

```
struct  Re
{
    // enum<Re_type>
    uchar   type;
    ushort  gen;
    union
    {
        Re* alt;      /* Talt */
        Re**  cases;  /* case */
        struct        /* class */
        {
            Rune    lo;
            Rune    hi;
        };
        Rune    val;  /* char */
    };
    // ref_own ?
    Re* next;
};
```

⟨*enum* `Re_type` 44b⟩≡ (333)

```
enum Re_type
{
    Talt        = 1,
    Tbegin,
    Tcase,
    Tclass,
    Tend,
    Tor,
};
```

⟨*struct* `State` *(grep)* 44c⟩≡ (333)

```
struct  State
{
    int count;
    int match;
    // ??
    Re**    re;

    // extra
    State*  linkleft;
    State*  linkright;
    // ??
    State*  next[256];
};
```

⟨*struct* `Re2` *(grep)* 45a⟩≡ (333)
```
struct  Re2
{
    // ref_own<Re>
    Re* beg;
    // ref_own<Re>
    Re* end;
};
```

⟨*globals grep* 45b⟩≡ (334b) 45d▷
```
State*  state0;
// Top regexp?
Re2 topre;
```

The `U` union overlays two uses of the same 32 KB buffer: during parsing, it holds the character class string being constructed by the lexer; during matching, its `pre`/`buf` pair implements a sliding window over the input. When a line spans multiple reads, the end of the previous buffer is shifted into `pre` so the full matching line can be printed.

⟨*union* `U` *(grep)* 45c⟩≡ (333)
```
union U
{
    char    string[16*1024];
    struct
    {
        /*
         * if a line requires multiple reads, we keep shifting
         * buf down into pre and then do another read into
         * buf.  so you'll get the last 16-32k of the matching line.
         * if pre were smaller than buf you'd get a suffix of the
         * line with a hole cut out.
         */
        uchar   pre[16*1024];   /* to save to previous '\n' */
        uchar   buf[16*1024];   /* input buffer */
    };
};
```

⟨*globals grep* 45d⟩+≡ (334b) ◁45b 45e▷
```
union U u;
```

5.1.2 `main()`

Flags are stored in a 256-entry `char` array indexed by the flag character itself—a common Plan 9 idiom that avoids a separate boolean for each flag.

⟨*globals grep* 45e⟩+≡ (334b) ◁45d 45f▷
```
char    flags[256];
```

⟨*globals grep* 45f⟩+≡ (334b) ◁45e 45g▷
```
Biobuf  bout;
```

⟨*globals grep* 45g⟩+≡ (334b) ◁45f 45h▷
```
// option<ref_own<Biobuf>> for -f
Biobuf* rein;
```

⟨*globals grep* 45h⟩+≡ (334b) ◁45g 47a▷
```
// option<string>, for -f
char    *filename;
long    lineno;
```

⟨*function* main*(grep)* 46⟩≡ (332)

```
void
main(int argc, char *argv[])
{
    int i;
    int status;

    ARGBEGIN {
    default:
        if(utfrune(validflags, ARGC()) == nil)
            usage();
        flags[ARGC()]++;
        break;

    case 'e':
        flags['e']++;
        lineno = 0;
        str2top(EARGF(usage()));
        break;

    case 'f':
        flags['f']++;
        filename = EARGF(usage());
        rein = Bopen(filename, OREAD);
        if(rein == nil) {
            fprint(STDERR, "grep: can't open %s: %r\n", filename);
            exits("open");
        }
        lineno = 1;
        str2top(filename);
        break;
    } ARGEND

    if(flags['f'] == 0 && flags['e'] == 0) {
        if(argc <= 0)
            usage();
        str2top(argv[0]);
        argc--;
        argv++;
    }

    follow = mal(maxfollow*sizeof(*follow));
    state0 = initstate_(topre.beg);

    Binit(&bout, STDOUT, OWRITE);
    switch(argc) {
    case 0:
        status = search(nil, 0);
        break;
    case 1:
        status = search(argv[0], 0);
        break;
    default:
        status = 0;
        for(i=0; i<argc; i++)
            status |= search(argv[i], Hflag);
        break;
    }
    if(status)
        exits(nil);
```

```
    exits("no matches");
}
```

⟨*globals grep* 47a⟩+≡ (334b) ◁45h 49d▷

```
char*   input;
char    *pattern;
ushort  gen;
```

`str2top` compiles a pattern string into the regexp graph. It calls `yyparse` (the yacc-generated parser), which builds a `Re2` graph from the input. Multiple patterns (from repeated `-e` or newline-separated patterns in `-f`) are combined with `re2or`, so the engine matches any of them.

⟨*function* `str2top` *(grep)* 47b⟩≡ (334c)

```
void
str2top(char *p)
{
    Re2 oldtop;

    oldtop = topre;
    input = p;
    if (*p == '\0')
        yyerror("empty pattern");   /* can't be a file name here */
    if (!flags['f'])
        pattern = p;

    topre.beg = nil;
    topre.end = nil;

    yyparse();

    gen++;
    if(topre.beg == nil)
        yyerror("syntax");
    if(oldtop.beg)
        topre = re2or(oldtop, topre);
}
```

⟨*function* `error` *(grep)* 47c⟩≡ (334c)

```
void
error(char *s)
{
    fprint(STDERR, "grep: internal error: %s\n", s);
    exits(s);
}
```

⟨*function* `yyerror` *(grep)* 47d⟩≡ (49a)

```
void
yyerror(char *e, ...)
{
    va_list args;

    fprint(STDERR, "grep: ");
    if(filename)
        fprint(STDERR, "%s:%ld: ", filename, lineno);
    else if (pattern)
        fprint(STDERR, "%s: ", pattern);
    va_start(args, e);
    vfprint(STDERR, e, args);
    va_end(args);
    fprint(STDERR, "\n");
    exits("syntax");
}
```

`grep` uses a custom bump allocator (`mal`) that grabs memory from `sbrk` in large hunks and hands it out sequentially. Since `grep` never frees anything—it allocates the regexp graph, builds the DFA states, and then exits—this is both faster and simpler than `malloc`.

⟨*function* `mal` *(grep)* 48a⟩≡ (334c)

```
void*
mal(int n)
{
    static char *s;
    static int m = 0;
    void *v;

    n = (n+3) & ~3;
    if(m < n) {
        if(n > Nhunk) {
            v = sbrk(n);
            memset(v, 0, n);
            return v;
        }
        s = sbrk(Nhunk);
        m = Nhunk;
    }
    v = s;
    s += n;
    m -= n;
    memset(v, 0, n);
    return v;
}
```

⟨*function* `sal` *(grep)* 48b⟩≡ (334c)

```
State*
sal(int n)
{
    State *s;

    s = mal(sizeof(*s));
//  s->next = mal(256*sizeof(*s->next));
    s->count = n;
    s->re = mal(n*sizeof(*state0->re));
    return s;
}
```

⟨*function* `ral` *(grep)* 48c⟩≡ (334c)

```
Re*
ral(int type)
{
    Re *r;

    r = mal(sizeof(*r));
    r->type = type;
    maxfollow++;
    return r;
}
```

5.1.3 Regexp lexer

The regexp parser is split into a hand-written lexer (`yylex`) and a yacc grammar. The lexer handles escape sequences, character classes (delimited by `[]`), and special characters like `*`, `|`, `^`, and `$`. The `literal` flag (set

by a leading * in the grammar) disables metacharacter interpretation, treating everything as a literal character.

⟨*grep/grep.y* 49a⟩≡
```
%{
#include    "grep.h"
%}

⟨grep grammar union decl 49b⟩
⟨grep grammar type decl 51b⟩
⟨grep grammar token decl 49c⟩
%%
⟨grep grammar 51c⟩
%%

⟨function yyerror(grep) 47d⟩
⟨function yylex(grep) 49f⟩
```

⟨*grep grammar union decl* 49b⟩≡ (49a)
```
%union
{
    int   val;
    char* str;
    Re2   re;
}
```

⟨*grep grammar* token *decl* 49c⟩≡ (49a)
```
%token  <str>   LCLASS
%token  <val>   LCHAR
%token      LLPAREN LRPAREN LALT LSTAR LPLUS LQUES
%token      LBEGIN LEND LDOT
%token      LBAD LNEWLINE
```

⟨*globals grep* 49d⟩+≡ (334b) ◁47a 49e▷
```
// option<int> | EOF (-1)? (None = 0)
int peekc;
```

⟨*globals grep* 49e⟩+≡ (334b) ◁49d 334a▷
```
bool literal;
```

⟨*function* yylex*(grep)* 49f⟩≡ (49a)
```
long
yylex(void)
{
    char *q, *eq;
    int c, s;

    if(peekc) {
        s = peekc;
        peekc = 0;
        return s;
    }
    c = getrec();
    if(literal) {
        if(c != 0 && c != '\n') {
            yylval.val = c;
            return LCHAR;
        }
        literal = false;
    }
    switch(c) {
    default:
```

```
        yylval.val = c;
        s = LCHAR;
        break;
    case '\\':
        c = getrec();
        yylval.val = c;
        s = LCHAR;
        if(c == '\n')
            s = LNEWLINE;
        break;
    case '[':
        goto getclass;
    case '(':
        s = LLPAREN;
        break;
    case ')':
        s = LRPAREN;
        break;
    case '|':
        s = LALT;
        break;
    case '*':
        s = LSTAR;
        break;
    case '+':
        s = LPLUS;
        break;
    case '?':
        s = LQUES;
        break;
    case '^':
        s = LBEGIN;
        break;
    case '$':
        s = LEND;
        break;
    case '.':
        s = LDOT;
        break;
    case 0:
        peekc = -1;
    case '\n':
        s = LNEWLINE;
        break;
    }
    return s;

getclass:
    q = u.string;
    eq = q + nelem(u.string) - 5;
    c = getrec();
    if(c == '^') {
        q[0] = '^';
        q[1] = '\n';
        q[2] = '-';
        q[3] = '\n';
        q += 4;
        c = getrec();
    }
    for(;;) {
```

```
            if(q >= eq)
                error("class too long");
            if(c == ']' || c == 0)
                break;
            if(c == '\\') {
                *q++ = c;
                c = getrec();
                if(c == 0)
                    break;
            }
            *q++ = c;
            c = getrec();
        }
        *q = 0;
        if(c == 0)
            return LBAD;
        yylval.str = u.string;
        return LCLASS;
    }
```

⟨*function* `getrec` *(grep)* 51a⟩≡ (334c)

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

        if(flags['f']) {
            c = Bgetc(rein);
            if(c <= 0)
                return 0;
        } else
            c = *input++ & 0xff;
        if(flags['i'] && c >= 'A' && c <= 'Z')
            c += 'a'-'A';
        if(c == '\n')
            lineno++;
        return c;
    }
```

5.1.4 Regexp grammar

The grammar builds the regexp graph bottom-up using the `Re2` pair of begin/end pointers. Concatenation (`re2cat`) patches the end of the first subexpression to the beginning of the second. Alternation (`re2or`) creates a `Tor` node whose `next` and `alt` pointers lead to the two branches. Kleene star (`re2star`) creates a `Talt` loop back to the subexpression's beginning. The top-level `prog` rule wraps the user's pattern in `.*pattern.*[^\n]` so that `grep` can find the pattern anywhere within a line.

⟨*grep grammar type decl* 51b⟩≡ (49a)

```
    %type   <re>    expr prog
    %type   <re>    expr0 expr1 expr2 expr3 expr4
```

⟨*grep grammar* 51c⟩≡ (49a)

```
    prog:   /* empty */
        {
            yyerror("empty pattern");
        }
    |   expr newlines
        {
```

```
        $$.beg = ral(Tend);
        $$.end = $$.beg;
        $$ = re2cat(re2star(re2or(re2char(0x00, '\n'-1), re2char('\n'+1, 0xff))), $$);
        $$ = re2cat($1, $$);
        $$ = re2cat(re2star(re2char(0x00, 0xff)), $$);
        topre = $$;
    }

expr:
    expr0
|   expr newlines expr0 { $$ = re2or($1, $3); }

expr0:
    expr1
|   LSTAR { literal = true; } expr1 { $$ = $3; }

expr1:
    expr2
|   expr1 LALT expr2     { $$ = re2or($1, $3); }

expr2:
    expr3
|   expr2 expr3  { $$ = re2cat($1, $2); }

expr3:
    expr4
|   expr3 LSTAR { $$ = re2star($1); }
|   expr3 LPLUS
    {
        $$.beg = ral(Talt);
        patchnext($1.end, $$.beg);
        $$.beg->alt = $1.beg;
        $$.end = $$.beg;
        $$.beg = $1.beg;
    }
|   expr3 LQUES
    {
        $$.beg = ral(Talt);
        $$.beg->alt = $1.beg;
        $$.end = $1.end;
        appendnext($$.end,  $$.beg);
    }

expr4:
    LCHAR
    {
        $$.beg = ral(Tclass);
        $$.beg->lo = $1;
        $$.beg->hi = $1;
        $$.end = $$.beg;
    }
|   LBEGIN
    {
        $$.beg = ral(Tbegin);
        $$.end = $$.beg;
    }
|   LEND
    {
        $$.beg = ral(Tend);
        $$.end = $$.beg;
```

```
    }
|   LDOT   { $$ = re2class("^\n"); }
|   LCLASS { $$ = re2class($1); }
|   LLPAREN expr1 LRPAREN { $$ = $2; }

newlines:
    LNEWLINE
|   newlines LNEWLINE
```

⟨*function* re2cat *(grep)* 53a⟩≡ (334c)

```
Re2
re2cat(Re2 a, Re2 b)
{
    Re2 c;

    c.beg = a.beg;
    c.end = b.end;
    patchnext(a.end, b.beg);
    return c;
}
```

⟨*function* patchnext *(grep)* 53b⟩≡ (334c)

```
void
patchnext(Re *a, Re *b)
{
    Re *n;

    while(a) {
        n = a->next;
        a->next = b;
        a = n;
    }
}
```

⟨*function* re2star *(grep)* 53c⟩≡ (334c)

```
Re2
re2star(Re2 a)
{
    Re2 c;

    c.beg = ral(Talt);
    c.beg->alt = a.beg;
    patchnext(a.end, c.beg);
    c.end = c.beg;
    return c;
}
```

⟨*function* re2or *(grep)* 53d⟩≡ (334c)

```
Re2
re2or(Re2 a, Re2 b)
{
    Re2 c;

    c.beg = ral(Tor);
    c.beg->alt = b.beg;
    c.beg->next = a.beg;
    c.end = b.end;
    appendnext(c.end,  a.end);
    return c;
}
```

⟨*function* `appendnext` *(grep)* 54a⟩≡ (334c)

```
void
appendnext(Re *a, Re *b)
{
    Re *n;

    while(n = a->next)
        a = n;
    a->next = b;
}
```

⟨*function* `re2char` *(grep)* 54b⟩≡ (334c)

```
Re2
re2char(int c0, int c1)
{
    Re2 c;

    c.beg = ral(Tclass);
    c.beg->lo = c0 & 0xff;
    c.beg->hi = c1 & 0xff;
    c.end = c.beg;
    return c;
}
```

5.1.5 `initstate_()`

`initstate_` converts the parsed regexp graph into the initial DFA state. It first calls `addcase`, which optimizes large alternations by building a 256-entry dispatch table (a `Tcase` node) when the number of alternatives exceeds `Caselim`. This turns character-class matching from a linear scan into a constant-time table lookup—critical for patterns like `[a-zA-Z]`.

⟨*function* `initstate` *(grep)* 54c⟩≡ (332)

```
State*
initstate_(Re *r)
{
    State *s;
    int i;

    addcase(r);
    if(flags['1'])
        reprint("r", r);
    nfollow = 0;
    gen++;
    fol1(r, Cbegin);
    follow[nfollow++] = r;
    qsort(follow, nfollow, sizeof(*follow), fcmp);

    s = sal(nfollow);
    for(i=0; i<nfollow; i++)
        s->re[i] = follow[i];
    return s;
}
```

⟨*function* `addcase` *(grep)* 54d⟩≡ (334c)

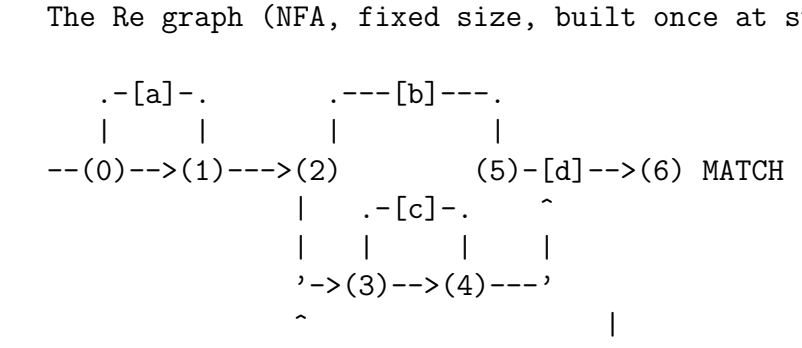
```
Re*
addcase(Re *r)
{
    int i, n;
    Re *a;
```

```
    if(r->gen == gen)
        return r;
    r->gen = gen;
    switch(r->type) {
    default:
        error("addcase");

    case Tor:
        n = countor(r);
        if(n >= Caselim) {
            a = ral(Tcase);
            a->cases = mal(256*sizeof(*a->cases));
            case1(a, r);
            for(i=0; i<256; i++)
                if(a->cases[i]) {
                    r = a->cases[i];
                    if(countor(r) < n)
                        a->cases[i] = addcase(r);
                }
            return a;
        }
        return r;

    case Talt:
        r->next = addcase(r->next);
        r->alt = addcase(r->alt);
        return r;

    case Tbegin:
    case Tend:
    case Tclass:
        return r;
    }
}
```

5.1.6 `search()`

The search loop is the performance-critical core. It reads raw bytes into the buffer and steps through them one at a time, advancing the DFA state via the `next` transition table. When a state's `next` entry is null, `increment` computes and caches the transition lazily—this is DFA construction on demand, avoiding the exponential blowup of building the full DFA up front.

The lazy DFA trick is worth a worked example, because it explains what otherwise looks like a contradiction: regex engines are typically described as either "slow Thompson NFA" or "fast precomputed DFA," and Plan 9`grep` manages to be both. Suppose we compile the pattern `a(b|c)*d` into the `Re` graph (this is what `yyparse` produces from the grammar):



```
                |                  |
                '------------------'  (loop back via Talt for *)

   Each node is a Re struct (Tclass for [a], [b], [c], [d];
   Tor between b/c; Talt for the * loop).
```

A pure Thompson NFA simulator would, for each input byte, walk *every* reachable Re node in parallel and follow the right edges—correct but slow because it re-does the same walk on every input. A pure DFA approach precomputes a State for *every reachable subset* of NFA nodes ahead of time, which is fast at search time but can blow up exponentially (the classic "2^n states" problem). Plan 9`grep` takes the middle road:

```
   The State graph (DFA, grown lazily as input arrives):

   state0 = { node 0 }                  -- starting state
                              ^
                              | input 'a'
                              v
   state1 = { node 1, 2, 3, 5 }         -- closure after 'a'
                              ^
                              | input 'b'
                              v
   state2 = { node 4, 2, 3, 5 }         -- closure after 'a' 'b'
                              ^
                              | input 'd'
                              v
   state3 = { node 6 }                  -- MATCH

   state->next[256] is the transition table; state->re is
   the set of NFA nodes this state represents.
```

At startup, only `state0` exists. The first time the search loop reads an `'a'` in this state, `increment(state0,'a')` computes the closure—"which NFA nodes can we be in after consuming an `'a'`?"—allocates a new `State` for the result, and stores the pointer in `state0->next['a']`. Every subsequent `'a'` in this state takes the cached transition in O(1). Bytes that never appear in the input never trigger a `State` allocation, so the DFA stays bounded by the *actually observed* input alphabet, not the full 256-byte range. This is why grepping a megabyte of ASCII text builds at most a few dozen states even for a hairy pattern, and why the search loop in `search()` can afford to be a tight `s = s->next[c]` dispatch with no parsing logic at all. The loop is duplicated: one copy for normal matching and one for case-insensitive (`-i`) matching with inline lowercasing. This duplication avoids a per-character branch on the `-i` flag, which matters when scanning millions of bytes. The use of `goto` instead of function calls keeps everything in a tight loop with no call overhead.

⟨*function* `search`*(grep)* 56⟩≡ (332)

```
int
search(char *file, int flag)
{
    State *s, *ns;
    fdt fid;
    int c, eof, nl, empty;
    long count, lineno, n;
    uchar *elp, *lp, *bol;

    if(file == nil) {
        file = "stdin";
        fid = STDIN;
```

```
        flag |= Bflag;
    } else
        fid = open(file, OREAD);

    if(fid < 0) {
        fprint(STDERR, "grep: can't open %s: %r\n", file);
        return 0;
    }

    if(flags['b'])
        flag ^= Bflag;      /* dont buffer output */
    if(flags['c'])
        flag |= Cflag;      /* count */
    if(flags['h'])
        flag &= ~Hflag;     /* do not print file name in output */
    if(flags['i'])
        flag |= Iflag;      /* fold upper-lower */
    if(flags['l'])
        flag |= Llflag;     /* print only name of file if any match */
    if(flags['L'])
        flag |= LLflag;     /* print only name of file if any non match */
    if(flags['n'])
        flag |= Nflag;      /* count only */
    if(flags['s'])
        flag |= Sflag;      /* status only */
    if(flags['v'])
        flag |= Vflag;      /* inverse match */

    s = state0;
    lineno = 0;
    count = 0;
    eof = 0;
    empty = 1;
    nl = 0;
    lp = u.buf;
    bol = lp;

loop0:
    n = lp-bol;
    if(n > sizeof(u.pre))
        n = sizeof(u.pre);
    memmove(u.buf-n, bol, n);
    bol = u.buf-n;
    n = read(fid, u.buf, sizeof(u.buf));
    /* if file has no final newline, simulate one to emit matches to last line */
    if(n > 0) {
        empty = 0;
        nl = u.buf[n-1]=='\n';
    } else {
        if(n < 0){
            fprint(STDERR, "grep: read error on %s: %r\n", file);
            return count != 0;
        }
        if(!eof && !nl && !empty) {
            u.buf[0] = '\n';
            n = 1;
            eof = 1;
        }
    }
    if(n <= 0) {
```

```
        close(fid);
        if(flag & Cflag) {
            if(flag & Hflag)
                Bprint(&bout, "%s:", file);
            Bprint(&bout, "%ld\n", count);
        }
        if(((flag&Llflag) && count != 0) || ((flag&LLflag) && count == 0))
            Bprint(&bout, "%s\n", file);
        Bflush(&bout);
        return count != 0;
    }
    lp = u.buf;
    elp = lp+n;
    if(flag & Iflag)
        goto loopi;

/*
 * normal character loop
 */
loop:
    c = *lp;
    ns = s->next[c];
    if(ns == 0) {
        increment(s, c);
        goto loop;
    }
//  if(flags['2'])
//      if(s->match)
//          print("%d: %.2x**\n", s, c);
//      else
//          print("%d: %.2x\n", s, c);
    lp++;
    s = ns;
    if(c == '\n') {
        lineno++;
        if(!!s->match == !(flag&Vflag)) {
            count++;
            if(flag & (Cflag|Sflag|Llflag|LLflag))
                goto cont;
            if(flag & Hflag)
                Bprint(&bout, "%s:", file);
            if(flag & Nflag)
                Bprint(&bout, "%ld: ", lineno);
            /* suppress extra newline at EOF unless we are labeling matches with file name */
            Bwrite(&bout, bol, lp-bol-(eof && !(flag&Hflag)));
            if(flag & Bflag)
                Bflush(&bout);
        }
        if((lineno & Flshcnt) == 0)
            Bflush(&bout);
    cont:
        bol = lp;
    }
    if(lp != elp)
        goto loop;
    goto loop0;

/*
 * character loop for -i flag
 * for speed
```

```
 */
loopi:
    c = *lp;
    if(c >= 'A' && c <= 'Z')
        c += 'a'-'A';
    ns = s->next[c];
    if(ns == 0) {
        increment(s, c);
        goto loopi;
    }
    lp++;
    s = ns;
    if(c == '\n') {
        lineno++;
        if(!!s->match == !(flag&Vflag)) {
            count++;
            if(flag & (Cflag|Sflag|Llflag|LLflag))
                goto conti;
            if(flag & Hflag)
                Bprint(&bout, "%s:", file);
            if(flag & Nflag)
                Bprint(&bout, "%ld: ", lineno);
            /* suppress extra newline at EOF unless we are labeling matches with file name */
            Bwrite(&bout, bol, lp-bol-(eof && !(flag&Hflag)));
            if(flag & Bflag)
                Bflush(&bout);
        }
        if((lineno & Flshcnt) == 0)
            Bflush(&bout);
    conti:
        bol = lp;
    }
    if(lp != elp)
        goto loopi;
    goto loop0;
}
```

⟨*function* countor *(grep)* 59a⟩≡ (334c)

```
int
countor(Re *r)
{
    int n;

    n = 0;
loop:
    switch(r->type) {
    case Tor:
        n += countor(r->alt);
        r = r->next;
        goto loop;
    case Tclass:
        return n + r->hi - r->lo + 1;
    }
    return n;
}
```

⟨*function* oralloc *(grep)* 59b⟩≡ (334c)

```
Re*
oralloc(int t, Re *r, Re *b)
{
```

```
    Re *a;

    if(b == 0)
        return r;
    a = ral(t);
    a->alt = r;
    a->next = b;
    return a;
}
```

⟨*function* `case1` *(grep)* 60a⟩≡ (334c)

```
void
case1(Re *c, Re *r)
{
    int n;

loop:
    switch(r->type) {
    case Tor:
        case1(c, r->alt);
        r = r->next;
        goto loop;

    case Tclass:    /* add to character */
        for(n=r->lo; n<=r->hi; n++)
            c->cases[n] = oralloc(Tor, r->next, c->cases[n]);
        break;

    default:    /* add everything unknown to next */
        c->next = oralloc(Talt, r, c->next);
        break;
    }
}
```

5.1.7 Advanced features

Debugging with `-1`

⟨*function* `reprint` *(grep)* 60b⟩≡ (334c)

```
void
reprint(char *s, Re *r)
{
    print("%s:\n", s);
    gen++;
    reprint1(r);
    print("\n\n");
}
```

⟨*function* `reprint1` *(grep)* 60c⟩≡ (334c)

```
void
reprint1(Re *a)
{
    int i, j;

loop:
    if(a == 0)
        return;
    if(a->gen == gen)
        return;
```

```
    a->gen = gen;
    print("%p: ", a);
    switch(a->type) {
    default:
        print("type %d\n", a->type);
        error("print1 type");

    case Tcase:
        print("case ->%p\n", a->next);
        for(i=0; i<256; i++)
            if(a->cases[i]) {
                for(j=i+1; j<256; j++)
                    if(a->cases[i] != a->cases[j])
                        break;
                print(" [%.2x-%.2x] ->%p\n", i, j-1, a->cases[i]);
                i = j-1;
            }
        for(i=0; i<256; i++)
            reprint1(a->cases[i]);
        break;

    case Tbegin:
        print("^ ->%p\n", a->next);
        break;

    case Tend:
        print("$ ->%p\n", a->next);
        break;

    case Tclass:
        print("[%.2x-%.2x] ->%p\n", a->lo, a->hi, a->next);
        break;

    case Tor:
    case Talt:
        print("| %p ->%p\n", a->alt, a->next);
        reprint1(a->alt);
        break;
    }
    a = a->next;
    goto loop;
}
```

5.2 tr

5.3 sed

5.4 find

5.5 which

Chapter 6

Text processing: `awk`

This is Brian Kernighan's "one true awk," the reference implementation by one of the language's three creators (Aho, Weinberger, Kernighan). Unlike the other utilities in this book, `awk` is a complete programming language with variables, arrays, functions, and control flow. It bridges the gap between simple line-oriented tools like `grep`/`sed` and full programming languages like C or Perl (which `awk` directly influenced). The implementation follows the classic interpreter pattern: a yacc grammar parses the program into an AST of `Node` trees, then `run` walks the tree to execute it.

6.1 Introduction

`awk` occupies an unusual niche in the UNIX toolbox: it is the smallest tool in this book that is also a complete programming language with variables, arrays, functions, regular expressions, control flow, and dynamic typing. Most of the other utilities you have seen in this book do one thing; `awk` does anything you can express in its little language. Yet the implementation is roughly 6 000 lines of C—less than `grep` would be if you counted `libregexp` separately.

Where does `awk` sit in the spectrum of text-processing tools? Each tool in the table below adds expressive power along one specific axis, with the cost being implementation size:

```
tool       data model         control      LOC (orig)
--------   ---------------    ----------   ----------
tr         single byte        none         300
cut        column             none         300
sed        line + 2 buffers   goto/branch  1500
grep       line               none         600
awk        line + fields +    if/while/    6000
           assoc arrays       for/func
perl       arbitrary objects + everything  200000+
python     arbitrary objects + everything  1M+
```

The interesting jump is from `sed` to `awk`: by adding *associative arrays* and *user-defined functions*, `awk` crosses the line from "advanced filter" to "programming language." Larry Wall famously called Perl "a cleaned-up `awk`"—if you trace the history forward, `awk` → Perl → Ruby → Python's regex/os modules covers most of the "Unix scripting" lineage. The fact that the original `awk` paper [**?**] predates Perl by a decade and Python by 15 years makes it the true ancestor of every text-mostly scripting language.

`awk`'s programming model is also distinctive enough to be worth spelling out. An `awk` program is a list of *pattern–action* pairs, plus optional `BEGIN` and `END` blocks. The implicit main loop is:

```
run BEGIN block
```

```
for each line in the input:
    split it on FS into $1, $2, ..., $NF
    for each pattern-action pair:
        if pattern matches the current line:
            run the action
run END block
```

That implicit loop is what makes one-liners short: `awk -F: '$3>1000{print $1}' /etc/passwd` prints user names whose UID exceeds 1000, with no `for`, no `open`, no `split` in the user code—the language already knows the program will be "do this for every line." This is the same trade-off `sed` makes (one buffer, one pass), pushed one notch further up the expressive-power axis. Patterns can be regular expressions, range pairs, or arbitrary boolean expressions; if the pattern is omitted, the action runs on every line; if the action is omitted, the pattern's matching-or-not becomes a default `print` of the line.

6.2 Data structures

`awk` has *three* major data structures: `Cell` for runtime values, `Node` for parse trees, and `Array` for associative arrays. The interesting one is `Cell`, because it is what gives `awk` its peculiar dual-typed semantics where `$1` can be both the string `"42"` and the number `42` simultaneously.

6.2.1 `Cell`

A `Cell` is the awk-level value: a string, a number, a function, an array, or a variable holding any of those. The key trick is that the `tval` field is a bitset rather than an enum—a Cell can carry `NUM | STR` simultaneously, meaning "this value has both a string form and a numeric form, and both are currently consistent." This is what lets `$1=="42"` and `$1==42` both work without an explicit cast.

```
tval bits         meaning
---------------   --------------------------------
NUM               fnval is the canonical form
STR               sval is the canonical form
NUM | STR         both are present and consistent
                  (after a comparison forces conversion)
STR | DONTFREE    sval points to literal text in the
                  program, do not free
ARR               sval points to an Array (assoc. table)
FCN               sval points to a Node (function body)
```

The state transitions are: a freshly-read field `$1` starts as `STR`; the first arithmetic that touches it sets the `NUM` bit and computes `fnval` via `atof`; assigning a new value clears both bits and re-sets one. Comparisons are where the famous awk surprise lives: if both operands have `NUM` set, they compare as numbers, otherwise as strings, which is why `"10" < "9"` is true (lexicographic) but `10 < 9` is false (numeric). The dual-typing is *the* thing that makes awk one-liners short and awk gotchas long.

6.2.2 Symbol table

6.3 `main()`

The `main` function parses command-line options manually (not using Plan 9's `ARGBEGIN`—this code uses standard C conventions since it predates and is independent of the Plan 9 library). After processing flags and initializing

the symbol table, it calls `yyparse` to compile the program, then `run(winner)` to execute the resulting AST. The `winner` variable holds the root `PROGRAM` node, which links the `BEGIN` block, the main pattern-action list, and the `END` block.

⟨*function* `main` *(awk)* 64⟩≡ (415)

```
int main(int argc, char *argv[])
{
    char *fs = NULL, *marg;
    int temp;

    cmdname = argv[0];
    if (argc == 1) {
        fprintf(stderr, "Usage: %s [-F fieldsep] [-mf n] [-mr n] [-v var=value] [-f programfile | 'program'] [f
        exit(1);
    }
    signal(SIGFPE, fpecatch);
    yyin = NULL;
    symtab = makesymtab(NSYMTAB);
    while (argc > 1 && argv[1][0] == '-' && argv[1][1] != '\0') {
        if (strcmp(argv[1], "--") == 0) {   /* explicit end of args */
            argc--;
            argv++;
            break;
        }
        switch (argv[1][1]) {
        case 's':
            if (strcmp(argv[1], "-safe") == 0)
                safe = 1;
            break;
        case 'f':   /* next argument is program filename */
            argc--;
            argv++;
            if (argc <= 1)
                FATAL("no program filename");
            pfile[npfile++] = argv[1];
            break;
        case 'F':   /* set field separator */
            if (argv[1][2] != 0) {  /* arg is -Fsomething */
                if (argv[1][2] == 't' && argv[1][3] == 0)   /* wart: t=>\t */
                    fs = "\t";
                else if (argv[1][2] != 0)
                    fs = &argv[1][2];
            } else {        /* arg is -F something */
                argc--; argv++;
                if (argc > 1 && argv[1][0] == 't' && argv[1][1] == 0)   /* wart: t=>\t */
                    fs = "\t";
                else if (argc > 1 && argv[1][0] != 0)
                    fs = &argv[1][0];
            }
            if (fs == NULL || *fs == '\0')
                WARNING("field separator FS is empty");
            break;
        case 'v':   /* -v a=1 to be done NOW.  one -v for each */
            if (argv[1][2] == '\0' && --argc > 1 && isclvar((++argv)[1]))
                setclvar(argv[1]);
            break;
        case 'm':   /* more memory: -mr=record, -mf=fields */
                /* no longer needed */
            marg = argv[1];
            if (argv[1][3])
```

```
            temp = atoi(&argv[1][3]);
        else {
            argv++; argc--;
            temp = atoi(&argv[1][0]);
        }
        switch (marg[2]) {
        case 'r':   recsize = temp; break;
        case 'f':   nfields = temp; break;
        default: FATAL("unknown option %s\n", marg);
        }
        break;
    case 'd':
        dbg = atoi(&argv[1][2]);
        if (dbg == 0)
            dbg = 1;
        printf("awk %s\n", version);
        break;
    case 'V':   /* added for exptools "standard" */
        printf("awk %s\n", version);
        exit(0);
        break;
    default:
        WARNING("unknown option %s ignored", argv[1]);
        break;
    }
    argc--;
    argv++;
}
/* argv[1] is now the first argument */
if (npfile == 0) {  /* no -f; first argument is program */
    if (argc <= 1) {
        if (dbg)
            exit(0);
        FATAL("no program given");
    }
       dprintf( ("program = |%s|\n", argv[1]) );
    lexprog = argv[1];
    argc--;
    argv++;
}

recinit(recsize);
syminit();
compile_time = 1;
argv[0] = cmdname;  /* put prog name at front of arglist */
   dprintf( ("argc=%d, argv[0]=%s\n", argc, argv[0]) );
arginit(argc, argv);
if (!safe)
    envinit(environ);
yyparse();
if (fs)
    *FS = qstring(fs, '\0');
   dprintf( ("errorflag=%d\n", errorflag) );
if (errorflag == 0) {
    compile_time = 0;
    run(winner);
} else
    bracecheck();
return(errorflag);
}
```

⟨*awkgram.y prelude* 66a⟩≡ (343b)
```
#include <stdio.h>
#include <string.h>
#include "awk.h"

#define makedfa(a,b)    compre(a)

void checkdup(Node *list, Cell *item);
int yywrap(void) { return(1); }

Node    *beginloc = 0;
Node    *endloc = 0;
int infunc  = 0;    /* = 1 if in arglist or body of func */
int inloop  = 0;    /* = 1 if in while, for, do */
char    *curfname = 0;  /* current function name */
Node    *arglist = 0;   /* list of args for current function */
```

6.4 Lexer

The lexer handles `awk`'s C-like syntax with a few twists: newlines are significant tokens (they can terminate statements), `\\` followed by a newline is a line continuation, and the lexer must distinguish between `/` as division and `/` as a regex delimiter using context (the `reg` flag). The bracket/brace/parenthesis counters detect unbalanced delimiters early.

⟨*union directive awkgram.y* 66b⟩≡ (343b)
```
%union {
    Node    *p;
    Cell    *cp;
    int i;
    char    *s;
}
```

⟨*token directives awkgram.y* 66c⟩≡ (343b)
```
%token  <i> FIRSTTOKEN  /* must be first */
%token  <p> PROGRAM PASTAT PASTAT2 XBEGIN XEND
%token  <i> NL ',' '{' '(' '|' ';' '/' ')' '}' '[' ']'
%token  <i> ARRAY
%token  <i> MATCH NOTMATCH MATCHOP
%token  <i> FINAL DOT ALL CCL NCCL CHAR OR STAR QUEST PLUS
%token  <i> AND BOR APPEND EQ GE GT LE LT NE IN
%token  <i> ARG BLTIN BREAK CLOSE CONTINUE DELETE DO EXIT FOR FUNC
%token  <i> SUB GSUB IF INDEX LSUBSTR MATCHFCN NEXT NEXTFILE
%token  <i> ADD MINUS MULT DIVIDE MOD
%token  <i> ASSIGN ASGNOP ADDEQ SUBEQ MULTEQ DIVEQ MODEQ POWEQ
%token  <i> PRINT PRINTF SPRINTF
%token  <p> ELSE INTEST CONDEXPR
%token  <i> POSTINCR PREINCR POSTDECR PREDECR
%token  <cp>    VAR IVAR VARNF CALL NUMBER STRING
%token  <s> REGEXPR
```

⟨*function* `yylex`*(awk)* 66d⟩≡ (344)
```
int yylex(void)
{
    int c;
    static char *buf = 0;
    static int bufsize = 500;

    if (buf == 0 && (buf = (char *) malloc(bufsize)) == NULL)
```

```
    FATAL( "out of space in yylex" );
if (sc) {
    sc = 0;
    RET('}');
}
if (reg) {
    reg = 0;
    return regexpr();
}
for (;;) {
    c = gettok(&buf, &bufsize);
    if (c == 0)
        return 0;
    if (isalpha(c) || c == '_')
        return word(buf);
    if (isdigit(c) || c == '.') {
        yylval.cp = setsymtab(buf, tostring(buf), atof(buf), CON|NUM, symtab);
        /* should this also have STR set? */
        RET(NUMBER);
    }

    yylval.i = c;
    switch (c) {
    case '\n':  /* {EOL} */
        RET(NL);
    case '\r':  /* assume \n is coming */
    case ' ':   /* {WS}+ */
    case '\t':
        break;
    case '#':   /* #.* strip comments */
        while ((c = input()) != '\n' && c != 0)
            ;
        unput(c);
        break;
    case ';':
        RET(';');
    case '\\':
        if (peek() == '\n') {
            input();
        } else if (peek() == '\r') {
            input(); input();   /* \n */
            lineno++;
        } else {
            RET(c);
        }
        break;
    case '&':
        if (peek() == '&') {
            input(); RET(AND);
        } else
            RET('&');
    case '|':
        if (peek() == '|') {
            input(); RET(BOR);
        } else
            RET('|');
    case '!':
        if (peek() == '=') {
            input(); yylval.i = NE; RET(NE);
        } else if (peek() == '~') {
```

```
        input(); yylval.i = NOTMATCH; RET(MATCHOP);
    } else
        RET(NOT);
case '~':
    yylval.i = MATCH;
    RET(MATCHOP);
case '<':
    if (peek() == '=') {
        input(); yylval.i = LE; RET(LE);
    } else {
        yylval.i = LT; RET(LT);
    }
case '=':
    if (peek() == '=') {
        input(); yylval.i = EQ; RET(EQ);
    } else {
        yylval.i = ASSIGN; RET(ASGNOP);
    }
case '>':
    if (peek() == '=') {
        input(); yylval.i = GE; RET(GE);
    } else if (peek() == '>') {
        input(); yylval.i = APPEND; RET(APPEND);
    } else {
        yylval.i = GT; RET(GT);
    }
case '+':
    if (peek() == '+') {
        input(); yylval.i = INCR; RET(INCR);
    } else if (peek() == '=') {
        input(); yylval.i = ADDEQ; RET(ASGNOP);
    } else
        RET('+');
case '-':
    if (peek() == '-') {
        input(); yylval.i = DECR; RET(DECR);
    } else if (peek() == '=') {
        input(); yylval.i = SUBEQ; RET(ASGNOP);
    } else
        RET('-');
case '*':
    if (peek() == '=') {    /* *= */
        input(); yylval.i = MULTEQ; RET(ASGNOP);
    } else if (peek() == '*') { /* ** or **= */
        input();    /* eat 2nd * */
        if (peek() == '=') {
            input(); yylval.i = POWEQ; RET(ASGNOP);
        } else {
            RET(POWER);
        }
    } else
        RET('*');
case '/':
    RET('/');
case '%':
    if (peek() == '=') {
        input(); yylval.i = MODEQ; RET(ASGNOP);
    } else
        RET('%');
case '^':
```

```
            if (peek() == '=') {
                input(); yylval.i = POWEQ; RET(ASGNOP);
            } else
                RET(POWER);

        case '$':
            /* BUG: awkward, if not wrong */
            c = gettok(&buf, &bufsize);
            if (c == '(' || c == '[' || (infunc && isarg(buf) >= 0)) {
                unputstr(buf);
                RET(INDIRECT);
            } else if (isalpha(c)) {
                if (strcmp(buf, "NF") == 0) {   /* very special */
                    unputstr("(NF)");
                    RET(INDIRECT);
                }
                yylval.cp = setsymtab(buf, "", 0.0, STR|NUM, symtab);
                RET(IVAR);
            } else {
                unputstr(buf);
                RET(INDIRECT);
            }

        case '}':
            if (--bracecnt < 0)
                SYNTAX( "extra }" );
            sc = 1;
            RET(';');
        case ']':
            if (--brackcnt < 0)
                SYNTAX( "extra ]" );
            RET(']');
        case ')':
            if (--parencnt < 0)
                SYNTAX( "extra )" );
            RET(')');
        case '{':
            bracecnt++;
            RET('{');
        case '[':
            brackcnt++;
            RET('[');
        case '(':
            parencnt++;
            RET('(');

        case '"':
            return string();    /* BUG: should be like tran.c ? */

        default:
            RET(c);
        }
    }
}
```

6.5 Grammar

The `awk` grammar is a full expression language with C-style operator precedence. Pattern-action rules (`pa_stat`) are the core: a pattern optionally followed by a brace-delimited action. When the action is omitted, the default is to print the current record. `BEGIN` and `END` blocks are collected separately into `beginloc` and `endloc` lists. Note the `CAT` (string concatenation) operator, which has no explicit syntax—two adjacent expressions are concatenated, a feature unique to `awk` among C-family languages.

⟨*type directives awkgram.y* 70a⟩≡ (343b)

```
%type   <p> pas pattern ppattern plist pplist patlist prarg term re
%type   <p> pa_pat pa_stat pa_stats
%type   <s> reg_expr
%type   <p> simple_stmt opt_simple_stmt stmt stmtlist
%type   <p> var varname funcname varlist
%type   <p> for if else while
%type   <i> do st
%type   <i> pst opt_pst lbrace rbrace rparen comma nl opt_nl and bor
%type   <i> subop print
```

⟨*priority directives awkgram.y* 70b⟩≡ (343b)

```
%right  ASGNOP
%right  '?'
%right  ':'
%left   BOR
%left   AND
%left   GETLINE
%nonassoc APPEND EQ GE GT LE LT NE MATCHOP IN '|'
%left   ARG BLTIN BREAK CALL CLOSE CONTINUE DELETE DO EXIT FOR FUNC
%left   GSUB IF INDEX LSUBSTR MATCHFCN NEXT NUMBER
%left   PRINT PRINTF RETURN SPLIT SPRINTF STRING SUB SUBSTR
%left   REGEXPR VAR VARNF IVAR WHILE '('
%left   CAT
%left   '+' '-'
%left   '*' '/' '%'
%left   NOT UMINUS
%right  POWER
%right  DECR INCR
%left   INDIRECT
%token  LASTTOKEN   /* must be last */
```

⟨*grammar awkgram.y* 70c⟩≡ (343b)

```
program:
      pas   { if (errorflag==0)
            winner = (Node *)stat3(PROGRAM, beginloc, $1, endloc); }
    | error { yyclearin; bracecheck(); SYNTAX("bailing out"); }
    ;

and:
      AND | and NL
    ;

bor:
      BOR | bor NL
    ;

comma:
      ',' | comma NL
    ;

do:
```

```
      DO | do NL
    ;

else:
      ELSE | else NL
    ;

for:
      FOR '(' opt_simple_stmt ';' opt_nl pattern ';' opt_nl opt_simple_stmt rparen {inloop++;} stmt
        { --inloop; $$ = stat4(FOR, $3, notnull($6), $9, $12); }
    | FOR '(' opt_simple_stmt ';'  ';' opt_nl opt_simple_stmt rparen {inloop++;} stmt
        { --inloop; $$ = stat4(FOR, $3, NIL, $7, $10); }
    | FOR '(' varname IN varname rparen {inloop++;} stmt
        { --inloop; $$ = stat3(IN, $3, makearr($5), $8); }
    ;

funcname:
      VAR   { setfname($1); }
    | CALL  { setfname($1); }
    ;

if:
      IF '(' pattern rparen     { $$ = notnull($3); }
    ;

lbrace:
      '{' | lbrace NL
    ;

nl:
      NL | nl NL
    ;

opt_nl:
      /* empty */   { $$ = 0; }
    | nl
    ;

opt_pst:
      /* empty */   { $$ = 0; }
    | pst
    ;


opt_simple_stmt:
      /* empty */           { $$ = 0; }
    | simple_stmt
    ;

pas:
      opt_pst           { $$ = 0; }
    | opt_pst pa_stats opt_pst  { $$ = $2; }
    ;

pa_pat:
      pattern   { $$ = notnull($1); }
    ;

pa_stat:
      pa_pat            { $$ = stat2(PASTAT, $1, stat2(PRINT, rectonode(), NIL)); }
```

```
    | pa_pat lbrace stmtlist '}'    { $$ = stat2(PASTAT, $1, $3); }
    | pa_pat ',' pa_pat     { $$ = pa2stat($1, $3, stat2(PRINT, rectonode(), NIL)); }
    | pa_pat ',' pa_pat lbrace stmtlist '}' { $$ = pa2stat($1, $3, $5); }
    | lbrace stmtlist '}'       { $$ = stat2(PASTAT, NIL, $2); }
    | XBEGIN lbrace stmtlist '}'
        { beginloc = linkum(beginloc, $3); $$ = 0; }
    | XEND lbrace stmtlist '}'
        { endloc = linkum(endloc, $3); $$ = 0; }
    | FUNC funcname '(' varlist rparen {infunc++;} lbrace stmtlist '}'
        { infunc--; curfname=0; defn((Cell *)$2, $4, $8); $$ = 0; }
    ;

pa_stats:
      pa_stat
    | pa_stats opt_pst pa_stat  { $$ = linkum($1, $3); }
    ;

patlist:
      pattern
    | patlist comma pattern     { $$ = linkum($1, $3); }
    ;

ppattern:
      var ASGNOP ppattern       { $$ = op2($2, $1, $3); }
    | ppattern '?' ppattern ':' ppattern %prec '?'
        { $$ = op3(CONDEXPR, notnull($1), $3, $5); }
    | ppattern bor ppattern %prec BOR
        { $$ = op2(BOR, notnull($1), notnull($3)); }
    | ppattern and ppattern %prec AND
        { $$ = op2(AND, notnull($1), notnull($3)); }
    | ppattern MATCHOP reg_expr { $$ = op3($2, NIL, $1, (Node*)makedfa($3, 0)); }
    | ppattern MATCHOP ppattern
        { if (constnode($3))
            $$ = op3($2, NIL, $1, (Node*)makedfa(strnode($3), 0));
          else
            $$ = op3($2, (Node *)1, $1, $3); }
    | ppattern IN varname       { $$ = op2(INTEST, $1, makearr($3)); }
    | '(' plist ')' IN varname  { $$ = op2(INTEST, $2, makearr($5)); }
    | ppattern term %prec CAT   { $$ = op2(CAT, $1, $2); }
    | re
    | term
    ;

pattern:
      var ASGNOP pattern        { $$ = op2($2, $1, $3); }
    | pattern '?' pattern ':' pattern %prec '?'
        { $$ = op3(CONDEXPR, notnull($1), $3, $5); }
    | pattern bor pattern %prec BOR
        { $$ = op2(BOR, notnull($1), notnull($3)); }
    | pattern and pattern %prec AND
        { $$ = op2(AND, notnull($1), notnull($3)); }
    | pattern EQ pattern        { $$ = op2($2, $1, $3); }
    | pattern GE pattern        { $$ = op2($2, $1, $3); }
    | pattern GT pattern        { $$ = op2($2, $1, $3); }
    | pattern LE pattern        { $$ = op2($2, $1, $3); }
    | pattern LT pattern        { $$ = op2($2, $1, $3); }
    | pattern NE pattern        { $$ = op2($2, $1, $3); }
    | pattern MATCHOP reg_expr  { $$ = op3($2, NIL, $1, (Node*)makedfa($3, 0)); }
    | pattern MATCHOP pattern
        { if (constnode($3))
```

```
            $$ = op3($2, NIL, $1, (Node*)makedfa(strnode($3), 0));
          else
            $$ = op3($2, (Node *)1, $1, $3); }
    | pattern IN varname        { $$ = op2(INTEST, $1, makearr($3)); }
    | '(' plist ')' IN varname  { $$ = op2(INTEST, $2, makearr($5)); }
    | pattern '|' GETLINE var   {
          if (safe) SYNTAX("cmd | getline is unsafe");
          else $$ = op3(GETLINE, $4, itonp($2), $1); }
    | pattern '|' GETLINE       {
          if (safe) SYNTAX("cmd | getline is unsafe");
          else $$ = op3(GETLINE, (Node*)0, itonp($2), $1); }
    | pattern term %prec CAT    { $$ = op2(CAT, $1, $2); }
    | re
    | term
    ;

plist:
      pattern comma pattern     { $$ = linkum($1, $3); }
    | plist comma pattern       { $$ = linkum($1, $3); }
    ;

pplist:
      ppattern
    | pplist comma ppattern     { $$ = linkum($1, $3); }
    ;

prarg:
      /* empty */           { $$ = rectonode(); }
    | pplist
    | '(' plist ')'         { $$ = $2; }
    ;

print:
      PRINT | PRINTF
    ;

pst:
      NL | ';' | pst NL | pst ';'
    ;

rbrace:
      '}' | rbrace NL
    ;

re:
       reg_expr
        { $$ = op3(MATCH, NIL, rectonode(), (Node*)makedfa($1, 0)); }
    | NOT re    { $$ = op1(NOT, notnull($2)); }
    ;

reg_expr:
      '/' {startreg();} REGEXPR '/'     { $$ = $3; }
    ;

rparen:
      ')' | rparen NL
    ;

simple_stmt:
      print prarg '|' term      {
```

```
            if (safe) SYNTAX("print | is unsafe");
            else $$ = stat3($1, $2, itonp($3), $4); }
    | print prarg APPEND term   {
            if (safe) SYNTAX("print >> is unsafe");
            else $$ = stat3($1, $2, itonp($3), $4); }
    | print prarg GT term       {
            if (safe) SYNTAX("print > is unsafe");
            else $$ = stat3($1, $2, itonp($3), $4); }
    | print prarg           { $$ = stat3($1, $2, NIL, NIL); }
    | DELETE varname '[' patlist ']' { $$ = stat2(DELETE, makearr($2), $4); }
    | DELETE varname         { $$ = stat2(DELETE, makearr($2), 0); }
    | pattern           { $$ = exptostat($1); }
    | error             { yyclearin; SYNTAX("illegal statement"); }
    ;

st:
      nl
    | ';' opt_nl
    ;

stmt:
      BREAK st      { if (!inloop) SYNTAX("break illegal outside of loops");
                $$ = stat1(BREAK, NIL); }
    | CLOSE pattern st  { $$ = stat1(CLOSE, $2); }
    | CONTINUE st       {  if (!inloop) SYNTAX("continue illegal outside of loops");
                $$ = stat1(CONTINUE, NIL); }
    | do {inloop++;} stmt {--inloop;} WHILE '(' pattern ')' st
       { $$ = stat2(DO, $3, notnull($7)); }
    | EXIT pattern st   { $$ = stat1(EXIT, $2); }
    | EXIT st       { $$ = stat1(EXIT, NIL); }
    | for
    | if stmt else stmt { $$ = stat3(IF, $1, $2, $4); }
    | if stmt       { $$ = stat3(IF, $1, $2, NIL); }
    | lbrace stmtlist rbrace { $$ = $2; }
    | NEXT st   { if (infunc)
                SYNTAX("next is illegal inside a function");
              $$ = stat1(NEXT, NIL); }
    | NEXTFILE st   { if (infunc)
                SYNTAX("nextfile is illegal inside a function");
              $$ = stat1(NEXTFILE, NIL); }
    | RETURN pattern st { $$ = stat1(RETURN, $2); }
    | RETURN st     { $$ = stat1(RETURN, NIL); }
    | simple_stmt st
    | while {inloop++;} stmt    { --inloop; $$ = stat2(WHILE, $1, $3); }
    | ';' opt_nl        { $$ = 0; }
    ;

stmtlist:
      stmt
    | stmtlist stmt     { $$ = linkum($1, $2); }
    ;

subop:
      SUB | GSUB
    ;

term:
      term '/' ASGNOP term      { $$ = op2(DIVEQ, $1, $4); }
    | term '+' term         { $$ = op2(ADD, $1, $3); }
    | term '-' term         { $$ = op2(MINUS, $1, $3); }
```

```
| term '*' term         { $$ = op2(MULT, $1, $3); }
| term '/' term         { $$ = op2(DIVIDE, $1, $3); }
| term '%' term         { $$ = op2(MOD, $1, $3); }
| term POWER term       { $$ = op2(POWER, $1, $3); }
| '-' term %prec UMINUS     { $$ = op1(UMINUS, $2); }
| '+' term %prec UMINUS     { $$ = $2; }
| NOT term %prec UMINUS     { $$ = op1(NOT, notnull($2)); }
| BLTIN '(' ')'         { $$ = op2(BLTIN, itonp($1), rectonode()); }
| BLTIN '(' patlist ')'     { $$ = op2(BLTIN, itonp($1), $3); }
| BLTIN             { $$ = op2(BLTIN, itonp($1), rectonode()); }
| CALL '(' ')'          { $$ = op2(CALL, celltonode($1,CVAR), NIL); }
| CALL '(' patlist ')'      { $$ = op2(CALL, celltonode($1,CVAR), $3); }
| DECR var          { $$ = op1(PREDECR, $2); }
| INCR var          { $$ = op1(PREINCR, $2); }
| var DECR          { $$ = op1(POSTDECR, $1); }
| var INCR          { $$ = op1(POSTINCR, $1); }
| GETLINE var LT term       { $$ = op3(GETLINE, $2, itonp($3), $4); }
| GETLINE LT term       { $$ = op3(GETLINE, NIL, itonp($2), $3); }
| GETLINE var           { $$ = op3(GETLINE, $2, NIL, NIL); }
| GETLINE           { $$ = op3(GETLINE, NIL, NIL, NIL); }
| INDEX '(' pattern comma pattern ')'
    { $$ = op2(INDEX, $3, $5); }
| INDEX '(' pattern comma reg_expr ')'
    { SYNTAX("index() doesn't permit regular expressions");
      $$ = op2(INDEX, $3, (Node*)$5); }
| '(' pattern ')'       { $$ = $2; }
| MATCHFCN '(' pattern comma reg_expr ')'
    { $$ = op3(MATCHFCN, NIL, $3, (Node*)makedfa($5, 1)); }
| MATCHFCN '(' pattern comma pattern ')'
    { if (constnode($5))
        $$ = op3(MATCHFCN, NIL, $3, (Node*)makedfa(strnode($5), 1));
      else
        $$ = op3(MATCHFCN, (Node *)1, $3, $5); }
| NUMBER            { $$ = celltonode($1, CCON); }
| SPLIT '(' pattern comma varname comma pattern ')'     /* string */
    { $$ = op4(SPLIT, $3, makearr($5), $7, (Node*)STRING); }
| SPLIT '(' pattern comma varname comma reg_expr ')'    /* const /regexp/ */
    { $$ = op4(SPLIT, $3, makearr($5), (Node*)makedfa($7, 1), (Node *)REGEXPR); }
| SPLIT '(' pattern comma varname ')'
    { $$ = op4(SPLIT, $3, makearr($5), NIL, (Node*)STRING); }  /* default */
| SPRINTF '(' patlist ')'   { $$ = op1($1, $3); }
| STRING            { $$ = celltonode($1, CCON); }
| subop '(' reg_expr comma pattern ')'
    { $$ = op4($1, NIL, (Node*)makedfa($3, 1), $5, rectonode()); }
| subop '(' pattern comma pattern ')'
    { if (constnode($3))
        $$ = op4($1, NIL, (Node*)makedfa(strnode($3), 1), $5, rectonode());
      else
        $$ = op4($1, (Node *)1, $3, $5, rectonode()); }
| subop '(' reg_expr comma pattern comma var ')'
    { $$ = op4($1, NIL, (Node*)makedfa($3, 1), $5, $7); }
| subop '(' pattern comma pattern comma var ')'
    { if (constnode($3))
        $$ = op4($1, NIL, (Node*)makedfa(strnode($3), 1), $5, $7);
      else
        $$ = op4($1, (Node *)1, $3, $5, $7); }
| SUBSTR '(' pattern comma pattern comma pattern ')'
    { $$ = op3(SUBSTR, $3, $5, $7); }
| SUBSTR '(' pattern comma pattern ')'
    { $$ = op3(SUBSTR, $3, $5, NIL); }
```

```
    | var
    ;

var:
      varname
    | varname '[' patlist ']'   { $$ = op2(ARRAY, makearr($1), $3); }
    | IVAR              { $$ = op1(INDIRECT, celltonode($1, CVAR)); }
    | INDIRECT term         { $$ = op1(INDIRECT, $2); }
    ;

varlist:
      /* nothing */     { arglist = $$ = 0; }
    | VAR           { arglist = $$ = celltonode($1,CVAR); }
    | varlist comma VAR {
            checkdup($1, $3);
            arglist = $$ = linkum($1,celltonode($3,CVAR)); }
    ;

varname:
      VAR           { $$ = celltonode($1, CVAR); }
    | ARG           { $$ = op1(ARG, itonp($1)); }
    | VARNF         { $$ = op1(VARNF, (Node *) $1); }
    ;


while:
      WHILE '(' pattern rparen  { $$ = notnull($3); }
    ;
```

6.6 Interpreter

6.7 awk builtins

Chapter 7

Compare

Comparison utilities are essential programmer tools: `cmp` for byte-level comparison (useful in regression tests that snapshot output), `comm` for comparing sorted lists, and `diff` for the line-by-line differences that underpin every version control system.

7.1 `cmp`

`cmp` compares two files byte by byte. The main loop reads both files into 64 KB buffers and uses `memcmp` for fast bulk comparison. Only when `memcmp` detects a difference does it fall into the slower byte-by-byte loop to report the exact position. The `-s` flag suppresses all output (useful in scripts that only check the exit status), and `-l` prints every differing byte position.

⟨*function* `usage` *(cmp.c)* 77a⟩≡ (224)
```
static void
usage(void)
{
    print("usage: cmp [-lLs] file1 file2 [offset1 [offset2] ]\n");
    exits("usage");
}
```

⟨*global flags (cmp.c)* 77b⟩≡ (224)
```
// print nothing for differing files (shutup) but set exit status
bool sflag = false;
// print byte number (decimal) and differing bytes (hexa)
bool lflag = false;
// print the line number of the first differing bytes
bool Lflag = false;
```

⟨*constant* `BUF` *(cmp.c)* 77c⟩≡ (224)
```
#define     BUF     65536
```

⟨`main` *locals (cmp.c)* 77d⟩≡ (78b) 77e▷
```
char *name1, *name2;
fdt f1, f2;
uchar buf1[BUF], buf2[BUF];
uchar *b1s, *b1e, *b2s, *b2e;
```

⟨`main` *locals (cmp.c)* 77e⟩+≡ (78b) ◁77d 77f▷
```
int n, i;
```

⟨`main` *locals (cmp.c)* 77f⟩+≡ (78b) ◁77e 78a▷
```
uchar *p, *q;
```

⟨main *locals (cmp.c)* 78a⟩+≡ (78b) ◁77f

```
// ??
vlong nc = 1;
// line
vlong l = 1;
```

The read loop maintains two independent 64 KB sliding windows, one per file, each described by a (start, end) pointer pair into its `BUF`-byte backing buffer. The pairs are not kept in lockstep: if one file has 40 KB left in its window and the other has only 5 KB, the next `memcmp` only covers 5 KB, both starts advance by 5 KB, and the next iteration refills whichever window emptied first. The "is empty, refill" check at the top of each iteration treats the buffer as a ring: when `b1s` reaches `&buf1[BUF]` the pointer wraps back to `buf1` before the `read`. The layout for a concrete two-iteration trace on two files looks like:

```
buf1 [ . . . . . . . . . . . . . . . . . . ]  BUF=65536
       ^b1s                          ^b1e     window = 40 KB
buf2 [ . . . . . . . . . . . . . . . . . . ]
       ^b2s        ^b2e                       window =  5 KB
       |----n=5KB--|
       memcmp(b1s, b2s, n)   -> equal
       b1s += 5KB, b2s += 5KB   (b2 window now empty)

iter 2:
buf1 [ . . . . . . . . . . . . . . . . . . ]
                   ^b1s             ^b1e     window = 35 KB
buf2 [ . . . . . . . . . . . . . . . . . . ]
       ^b2s                               ^b2e   refilled 64 KB
       |----n=35KB----|
```

Two windows rather than one shared buffer is the key point: `read` on one file can return short without forcing a resync on the other, and `memcmp` on the overlap is fast enough that the whole loop is I/O-bound rather than CPU-bound. Only on a mismatch does the code fall into the slow per-byte `for(p=b1s,q=b2s...)` reporting loop, which also maintains the line counter `l` for the `-L` flag.

⟨*function* `main`*(cmp.c)* 78b⟩≡ (224)

```
void
main(int argc, char *argv[])
{
    ⟨main locals (cmp.c) 77d⟩

    ARGBEGIN{
    case 's':   sflag = true; break;
    case 'l':   lflag = true; break;
    case 'L':   Lflag = true; break;
    default:    usage();
    }ARGEND

    if(argc < 2 || argc > 4)
        usage();

    if((f1 = open(name1 = *argv++, OREAD)) == -1){
        if(!sflag) perror(name1);
        exits("open");
    }
    if((f2 = open(name2 = *argv++, OREAD)) == -1){
        if(!sflag) perror(name2);
        exits("open");
```

```
}
argv = seekoff(f1, name1, argv);
argv = seekoff(f2, name2, argv);
if(*argv)
    usage();

b1s = b1e = buf1;
b2s = b2e = buf2;

for(;;){
    if(b1s >= b1e){
        if(b1s >= &buf1[BUF])
            b1s = buf1;
        n = read(f1, b1s,  &buf1[BUF] - b1s);
        b1e = b1s + n;
    }
    if(b2s >= b2e){
        if(b2s >= &buf2[BUF])
            b2s = buf2;
        n = read(f2, b2s,  &buf2[BUF] - b2s);
        b2e = b2s + n;
    }
    n = b2e - b2s;
    if(n > b1e - b1s)
        n = b1e - b1s;
    if(n <= 0)
        break;

    if(memcmp((void *)b1s, (void *)b2s, n) != 0){
        if(sflag)
            exits("differ");
        for(p = b1s, q = b2s, i = 0; i < n; p++, q++, i++) {
            if(*p == '\n')
                l++;
            if(*p != *q){
                if(!lflag){
                    print("%s %s differ: char %lld",
                        name1, name2, nc+i);
                    print(Lflag?" line %lld\n":"\n", l);
                    exits("differ");
                }
                print("%6lld 0x%.2x 0x%.2x\n", nc+i, *p, *q);
            }
        }
    }
    if(Lflag)
        for(p = b1s; p < b1e;)
            if(*p++ == '\n')
                l++;
    nc += n;
    b1s += n;
    b2s += n;
} // end for (;;)

if (b1e - b1s < 0 || b2e - b2s < 0) {
    if (!sflag) {
        if (b1e - b1s < 0)
            print("error on %s after %lld bytes\n",
                name1, nc-1);
        if (b2e - b2s < 0)
```

```
                print("error on %s after %lld bytes\n",
                    name2, nc-1);
        }
        exits("read error");
    }

    // files are the same, exit 0
    if(b1e - b1s == b2e - b2s)
        exits(nil);

    if(!sflag)
        print("EOF on %s after %lld bytes\n",
            (b1e - b1s > b2e - b2s)? name2 : name1, nc-1);
    exits("EOF");
}
```

⟨*function* `seekoff` *(cmp.c)* 80a⟩≡ (224)

```
char **
seekoff(fdt fd, char *name, char **argv)
{
    vlong o;

    if(*argv){
        if (!isascii(**argv) || !isdigit(**argv))
            usage();
        o = strtoll(*argv++, nil, 0);
        if(seek(fd, o, SEEK__START) < 0){
            if(!sflag) fprint(STDERR, "cmp: %s: seek by %lld: %r\n",
                name, o);
            exits("seek");
        }
    }
    return argv;
}
```

7.2 comm

`comm` compares two sorted files and produces three columns: lines only in file 1, lines only in file 2, and lines in both. The flags `-1`, `-2`, `-3` suppress the corresponding column. The implementation is a simple merge of two sorted streams: compare the current lines, output the smaller one to its column, and advance that file.

⟨`main` *label* `usage` *(comm.c)* 80b⟩≡ (81b)

```
Usage:
    fprint(STDERR, "usage: comm [-123] file1 file2\n");
    exits("usage");
```

⟨global flags comm.c 80c⟩≡ (225a)

```
bool one;
bool two;
bool three;
```

⟨globals comm.c 80d⟩≡ (225a) 80e▷

```
char    *ldr[3];
```

⟨globals comm.c 80e⟩+≡ (225a) ◁80d

```
Biobuf *ib1;
Biobuf *ib2;
```

⟨*constant* LB*(comm.c)* 81a⟩≡ (225a)

```
#define LB 2048
```

The main loop is the textbook two-way merge of sorted streams, but with the twist that equal lines are output to a *third* column rather than dropped. The leader strings `ldr[0..2]` are “”, one tab, two tabs, so that the three columns line up when output is plain text; the `-1`/`-2` flags rewrite those leaders to shorter strings so the remaining columns shift left. A concrete run of `comm file1 file2` on `apple,cherry,date` versus `banana,cherry,elderberry` produces:

```
lb1     lb2         compare   action      output line
------  ----------  --------  ----------  ------------------
apple   banana      1         wr(lb1,1)   "apple"
cherry  banana      2         wr(lb2,2)   "\tbanana"
cherry  cherry      0         wr(lb1,3)   "\t\tcherry"
date    elderberry  1         wr(lb1,1)   "date"
(EOF)   elderberry  -         copy(ib2,2) "\telderberry"
```

The `compare` function returns 1/2/0 instead of the usual negative/positive/zero convention because those values double as direct indices into `ldr` and as the column argument to `wr`; the trailing `copy` helper drains whichever file still has lines when the other hits EOF. Notice also that after a match (`case 0`) *both* files are advanced, whereas in the unequal cases only the smaller side moves—the standard merge invariant.

⟨*function* main*(comm.c)* 81b⟩≡ (225a)

```
void
main(int argc, char *argv[])
{
    int l;
    char    lb1[LB],lb2[LB];

    ldr[0] = "";
    ldr[1] = "\t";
    ldr[2] = "\t\t";
    l = 1;

    ARGBEGIN{
    case '1':
        if(!one) {
            one = true;
            ldr[1][0] = '\0';
            ldr[2][l--] = '\0';
        }
        break;

    case '2':
        if(!two) {
            two = true;
            ldr[2][l--] = '\0';
        }
        break;

    case '3':
        three = true;
        break;

    default:
        goto Usage;

    }ARGEND
```

```
    if(argc < 2) {
    ⟨main label usage (comm.c) 80b⟩
    }

    ib1 = openfil(argv[0]);
    ib2 = openfil(argv[1]);

    if(rd(ib1,lb1) < 0){
        if(rd(ib2,lb2) < 0)
            exits(nil);
        copy(ib2,lb2,2);
    }
    if(rd(ib2,lb2) < 0)
        copy(ib1,lb1,1);

    for(;;){
        switch(compare(lb1,lb2)) {
        case 0:
            wr(lb1,3);
            if(rd(ib1,lb1) < 0) {
                if(rd(ib2,lb2) < 0)
                    exits(nil);
                copy(ib2,lb2,2);
            }
            if(rd(ib2,lb2) < 0)
                copy(ib1,lb1,1);
            continue;

        case 1:
            wr(lb1,1);
            if(rd(ib1,lb1) < 0)
                copy(ib2,lb2,2);
            continue;

        case 2:
            wr(lb2,2);
            if(rd(ib2,lb2) < 0)
                copy(ib1,lb1,1);
            continue;
        }
    }
}
```

⟨*function* openfil*(comm.c)* 82⟩≡ (225a)

```
Biobuf*
openfil(char *s)
{
    Biobuf *b;

    if(s[0]=='-' && s[1]=='\0')
        s = "/fd/0";
    b = Bopen(s, OREAD);
    if(b)
        return b;
    fprint(STDERR, "comm: cannot open %s: %r\n",s);
    exits("open");
    // never reached
    return nil;   /* shut up ken */
}
```

⟨*function* `rd`*(comm.c)* 83a⟩≡ (225a)

```
int
rd(Biobuf *file, char *buf)
{
    int i, c;

    i = 0;
    while((c = Bgetc(file)) != Beof) {
        *buf = c;
        if(c == '\n' || i > LB-2) {
            *buf = '\0';
            return 0;
        }
        i++;
        buf++;
    }
    return -1;
}
```

⟨*function* `compare`*(comm.c)* 83b⟩≡ (225a)

```
int
compare(char *a, char *b)
{
    while(*a == *b){
        if(*a == '\0')
            return 0;
        a++;
        b++;
    }
    if(*a < *b)
        return 1;
    return 2;
}
```

⟨*function* `wr`*(comm.c)* 83c⟩≡ (225a)

```
void
wr(char *str, int n)
{

    switch(n){
        case 1:
            if(one)
                return;
            break;

        case 2:
            if(two)
                return;
            break;

        case 3:
            if(three)
                return;
    }
    print("%s%s\n", ldr[n-1],str);
}
```

⟨*function* `copy`*(comm.c)* 83d⟩≡ (225a)

```
void
copy(Biobuf *ibuf, char *lbuf, int n)
```

```
{
    do
        wr(lbuf,n);
    while(rd(ibuf,lbuf) >= 0);
    exits(nil);
}
```

7.3 diff

7.4 patch

Chapter 8

Processes

This chapter covers the two standard process-inspection utilities in Plan 9: `ps`, which prints a one-line summary of every running process, and `pstree`, which prints the same information as a parent-child tree. Every multi-user operating system needs a way to ask "what is the machine doing right now?"—both for users (who own those processes and need to inspect or kill them) and for administrators (who need to find runaway processes, debug deadlocks, and check resource usage). The job has three parts: enumerate the currently running processes, retrieve metadata about each one (name, user, CPU time, memory, parent PID, state), and format the result for a human.

The reason `ps` and `pstree` fit together in under 200 lines of C is that Plan 9 exposes process state through the per-process directories under `/proc`: `/proc/1234/status`, `/proc/1234/args`, `/proc/1234/segment`, and so on, each a plain text file. Listing processes is therefore just listing the numeric subdirectories of `/proc` and reading a little text from each—no bespoke system call, no `sysctl` call returning an array of `struct kinfo_proc`s as on FreeBSD and macOS, no `NtQuerySystemInformation` with schema-typed structs as on Windows, and no scraping of `/dev/kmem` as on pre-PROC BSD systems. The same property is what makes shell one-liners like `cat /proc/*/status` practical in Plan 9: if you can read a file, you can build `ps`. The interactive cousins `top`, `htop`, `btop`, `glances`, and Windows Task Manager do the same information-gathering on a timer and redraw a TUI (or GUI) instead of printing a single snapshot.

The chapter walks both utilities in turn. `ps` is the simpler of the two—open `/proc`, list its numeric entries, read each `status` and `args` file, and print a line of output. `pstree` does the same information-gathering but then builds a parent-child tree by following each process's parent-PID field, and finally traverses the tree depth-first to print an indented view. The depth-first walk is the only nontrivial piece of code in either program.

8.1 `ps`

8.2 `pstree`

Chapter 9

Archiving

Before version control systems like git, open source software was distributed as `.tar.gz` archives. Even today, `tar` remains fundamental for packaging, backups, and deployment. This chapter also covers `gzip` for compression.

9.1 `tar`

`tar` ("tape archiver") concatenates files into a single archive stream. The format is simple: each file is preceded by a 512-byte header containing the file name, size, mode, and checksum, followed by the file data padded to a 512-byte boundary. Despite originating from tape backup workflows (hence the name), this format became the universal distribution format for UNIX software.

⟨*function* `usage` *(tar.c)* 86a⟩≡ (416b)
```
static void
usage(void)
{
    fprint(STDERR, "usage: %s {crtx}[PRTfgikmpsuvz] [archive] [file1 file2...]\n",
        argv0);
    exits("usage");
}
```

9.1.1 Data structures

`Verb` and flags

⟨*enum* `Verb` 86b⟩≡ (416b)
```
enum _Verb { None, Toc, Xtract, Replace };
```

⟨*global* `verb` *(tar.c)* 86c⟩≡ (416a)
```
// enum<Verb>
static int verb;
```

⟨global flags tar.c 86d⟩≡ (416b) 93a▷
```
static bool docreate;
static bool verbose;
static bool docompress;
```

⟨globals tar.c 86e⟩≡ (416b) 91h▷
```
static char *usefile;
```

⟨constants tar.c 86f⟩≡ (416b) 87a▷
```
Binsize = 0x80,     /* flag in size[0], from gnu: positive binary size */
Binnegsz = 0xff,    /* flag in size[0]: negative binary size */

Nblock = 40,        /* maximum blocksize */
Debug = 0,
```

`Hdr`

The `Hdr` union overlays raw 512-byte data with the named header fields. All numeric fields (mode, uid, size, mtime) are stored as ASCII octal strings—a design choice that makes archives human-readable with `cat` but limits the original format to files under 8 GB. The `Binsize` flag in `size[0]` signals a non-standard "base 256" encoding for larger files, a GNU extension. The POSIX ustar extension adds `magic`, `uname`, `gname`, and `prefix` fields, allowing longer path names (up to 255 characters by combining `prefix` and `name`).

⟨constants tar.c 87a⟩+≡ (416b) ◁86f 91g▷
```
Tblock = 512,
Namsiz = 100,
Maxpfx = 155,       /* from POSIX */
```

⟨*union* `Hdr` *(tar.c)* 87b⟩≡ (416b)
```
typedef union {
    uchar   data[Tblock];
    struct {
        char    name[Namsiz];
        char    mode[8];
        char    uid[8];
        char    gid[8];
        char    size[12];
        char    mtime[12];
        char    chksum[8];
        // enum<LinkFlag>
        char    linkflag;
        char    linkname[Namsiz];

        /* rest are defined by POSIX's ustar format; see p1003.2b */
        char    magic[6];   /* "ustar" */
        char    version[2];
        char    uname[32];
        char    gname[32];
        char    devmajor[8];
        char    devminor[8];
        char    prefix[Maxpfx]; /* if non-null, path= prefix "/" name */
    };
} Hdr;
```

⟨*function* `isustar` *(tar.c)* 87c⟩≡ (416b)
```
static bool
isustar(Hdr *hp)
{
    return strcmp(hp->magic, "ustar") == ORD__EQ;
}
```

⟨*enum* `LinkFlag` *(tar.c)* 87d⟩≡ (416b)
```
enum {
    LF_PLAIN1 = '\0',
    LF_PLAIN2 = '0',
    LF_LINK =   '1',
    LF_SYMLINK1 =   '2',
    LF_SYMLINK2 =   's',        /* 4BSD used this */
    LF_CHR =    '3',
    LF_BLK =    '4',
    LF_DIR =    '5',
    LF_FIFO =   '6',
    LF_CONTIG = '7',
    /* 'A' - 'Z' are reserved for custom implementations */
};
```

⟨*macros* `islinkxxx`*(tar.c)* 88a⟩≡ (416b)
```
#define islink(lf)  (isreallink(lf) || issymlink(lf))
#define isreallink(lf)  ((lf) == LF_LINK)
#define issymlink(lf)   ((lf) == LF_SYMLINK1 || (lf) == LF_SYMLINK2)
```

⟨*function* `mkhdr`*(tar.c)* 88b⟩≡ (416b)
```
static int
mkhdr(Hdr *hp, Dir *dir, char *file)
{
    int r;

    /*
     * some of these fields run together, so we format them left-to-right
     * and don't use snprint.
     */
    sprint(hp->mode, "%6lo ", dir->mode & 0777);
    sprint(hp->uid, "%6o ", aruid);
    sprint(hp->gid, "%6o ", argid);
    if (dir->length >= (Off)1<<32) {
        static bool printed;

        if (!printed) {
            printed = true;
            fprint(STDERR, "%s: storing large sizes in \"base 256\"\n", argv0);
        }
        hp->size[0] = Binsize;
        /* emit so-called 'base 256' representation of size */
        putbe((uchar *)hp->size+1, dir->length, sizeof hp->size - 2);
        hp->size[sizeof hp->size - 1] = ' ';
    } else
        sprint(hp->size, "%11lluo ", dir->length);
    sprint(hp->mtime, "%11luo ", dir->mtime);
    hp->linkflag = (dir->mode&DMDIR? LF_DIR: LF_PLAIN1);
    r = putfullname(hp, file);
    if (posix) {
        strncpy(hp->magic, "ustar", sizeof hp->magic);
        strncpy(hp->version, "00", sizeof hp->version);
        strncpy(hp->uname, dir->uid, sizeof hp->uname);
        strncpy(hp->gname, dir->gid, sizeof hp->gname);
    }
    sprint(hp->chksum, "%6luo", chksum(hp));
    return r;
}
```

⟨*function* `readhdr`*(tar.c)* 88c⟩≡ (416b)
```
static Hdr *
readhdr(int ar)
{
    long hdrcksum;
    Hdr *hp;

    hp = getblkrd(ar, Alldata);
    if (hp == nil)
        sysfatal("unexpected EOF instead of archive header in %s",
            arname);
    if (eotar(hp))          /* end-of-archive block? */
        return nil;

    hdrcksum = parsecksum(hp->chksum, name(hp));
    if (hdrcksum == -1 || chksum(hp) != hdrcksum) {
```

```
        if (!resync)
            sysfatal("bad archive header checksum in %s: "
                "name %.100s...; expected %#luo got %#luo",
                arname, hp->name, hdrcksum, chksum(hp));
        fprint(STDERR, "%s: skipping past archive header with bad checksum in %s...",
            argv0, arname);
        do {
            hp = getblkrd(ar, Alldata);
            if (hp == nil)
                sysfatal("unexpected EOF looking for archive header in %s",
                    arname);
            hdrcksum = parsecksum(hp->chksum, name(hp));
        } while (hdrcksum == -1 || chksum(hp) != hdrcksum);
        fprint(STDERR, "found %s\n", name(hp));
    }
    nexthdr += Tblock*(1 + BYTES2TBLKS(arsize(hp)));
    return hp;
}
```

Compress

⟨*struct* Compress*(tar.c)* 89a⟩≡ (416b)

```
typedef struct {
    char    *comp;
    char    *decomp;
    char    *sfx[4];
} Compress;
```

⟨*constant* comps*(tar.c)* 89b⟩≡ (416b)

```
static Compress comps[] = {
    "gzip",     "gunzip",       { ".tar.gz", ".tgz" },  /* default */
    "compress", "uncompress",   { ".tar.Z",  ".tz" },
    "bzip2",    "bunzip2",      { ".tar.bz", ".tbz", ".tar.bz2",".tbz2" },
};
```

⟨*function* compmethod*(tar.c)* 89c⟩≡ (416b)

```
static Compress *
compmethod(char *name)
{
    int i, nmlen, sfxlen;
    Compress *cp;

    if (name != nil) {
        nmlen = strlen(name);
        for (cp = comps; cp < comps + nelem(comps); cp++)
            for (i = 0; i < nelem(cp->sfx) && cp->sfx[i]; i++) {
                sfxlen = strlen(cp->sfx[i]);
                if (nmlen > sfxlen &&
                    strcmp(cp->sfx[i], name+nmlen-sfxlen) == ORD__EQ)
                    return cp;
            }
    }
    // suffix not found, then return first entry (default)
    return docompress? comps: nil;
}
```

Pushstate

⟨*struct* Pushstate*(tar.c)* 90a⟩≡ (416b)
```
typedef struct {
    int kid;
    fdt fd; /* original fd */
    fdt rfd;    /* replacement fd */
    int input;
    int open;
} Pushstate;
```

9.1.2 main()

⟨*function* main*(tar.c)* 90b⟩≡ (416b)
```
void
main(int argc, char *argv[])
{
    bool errflg = false;
    char *ret = nil;

    //XXX: fmtinstall('M', dirmodefmt);

    TARGBEGIN {
    ⟨main() switch character cases (tar.c) 90c⟩
    default:
        fprint(STDERR, "tar: unknown letter %C\n", TARGC());
        errflg = true;
        break;
    } TARGEND

    if (argc < 0 || errflg)
        usage();

    initblks();

    switch (verb) {
    case Toc:
    case Xtract:
        ret = extract(argv);
        break;
    case Replace:
        if (getwd(origdir, sizeof origdir) == nil)
            strcpy(origdir, "/tmp");
        ret = replace(argv);
        break;
    default:
        usage();
        break;
    }
    exits(ret);
}
```

⟨main() *switch character cases (tar.c)* 90c⟩≡ (90b) 91a▷
```
case 'c':
    docreate = true;
    verb = Replace;
    break;
```

⟨main() *switch character cases (tar.c)* 91a⟩+≡ (90b) ◁90c 91b▷
```
case 't':
    verb = Toc;
    break;
```

⟨main() *switch character cases (tar.c)* 91b⟩+≡ (90b) ◁91a 91c▷
```
case 'x':
    verb = Xtract;
    break;
```

⟨main() *switch character cases (tar.c)* 91c⟩+≡ (90b) ◁91b 91d▷
```
case 'r':
    verb = Replace;
    break;
```

⟨main() *switch character cases (tar.c)* 91d⟩+≡ (90b) ◁91c 91e▷
```
case 'v':
    verbose = true;
    break;
```

⟨main() *switch character cases (tar.c)* 91e⟩+≡ (90b) ◁91d 91f▷
```
case 'f':
    usefile = arname = EARGF(usage());
    break;
```

⟨main() *switch character cases (tar.c)* 91f⟩+≡ (90b) ◁91e 93b▷
```
case 'z':
    docompress = true;
    break;
```

⟨constants tar.c 91g⟩+≡ (416b) ◁87a 92a▷
```
Dblock = 20,        /* default blocksize */
```

⟨globals tar.c 91h⟩+≡ (416b) ◁86e 94l▷
```
// array<ref_own<Hdr>> (size = nblock)
static Hdr *tpblk;
static int nblock = Dblock;
static Hdr *endblk;
```

⟨*function* initblks*(tar.c)* 91i⟩≡ (416b)
```
static void
initblks(void)
{
    free(tpblk);
    tpblk = malloc(Tblock * nblock);
    assert(tpblk != nil);
    endblk = tpblk + nblock;
}
```

⟨*macros* TARGxxx*(tar.c)* 91j⟩≡ (416b)
```
/*
 * modified versions of those in libc.h; scans only the first arg for
 * keyletters and options.
 */
#define TARGBEGIN {\
    (argv0 || (argv0 = *argv)), argv++, argc--;\
    if (argv[0]) {\
        char *_args, *_argt;\
        Rune _argc;\
        _args = &argv[0][0];\
```

```
        _argc = 0;\
        while(*_args && (_args += chartorune(&_argc, _args)))\
            switch(_argc)
#define TARGEND SET(_argt); USED(_argt);USED(_argc);USED(_args); \
    argc--, argv++; } \
    USED(argv); USED(argc); }
#define TARGC() (_argc)
```

9.1.3 extract()

⟨constants tar.c 92a⟩+≡ (416b) ◁91g
```
Maxname = Namsiz + 1 + Maxpfx,
```

⟨*function* extract*(tar.c)* 92b⟩≡ (416b)
```
static char *
extract(char **argv)
{
    fdt ar;
    char *longname;
    char msg[Maxname + 40];
    Compress *comp;
    Hdr *hp;
    Pushstate ps;

    /* open archive to be read */
    if (usefile)
        ar = open(usefile, OREAD);
    else
        ar = STDIN;

    /* push decompression filter if requested or extension is known */
    comp = compmethod(usefile);
    if (comp)
        ar = push(ar, comp->decomp, Input, &ps);
    if (ar < 0)
        sysfatal("can't open archive %s: %r", usefile);

    while ((hp = readhdr(ar)) != nil) {
        longname = name(hp);
        if (match(longname, argv))
            extract1(ar, hp, longname);
        else {
            snprint(msg, sizeof msg, "extracting %s", longname);
            skip(ar, hp, msg);
        }
    }

    if (comp)
        return pushclose(&ps);
    if (ar > STDERR)
        close(ar);
    return nil;
}
```

9.1.4 replace()

⟨*function* replace*(tar.c)* 92c⟩≡ (416b)
```
static char *
```

```
replace(char **argv)
{
    int i, ar;
    char *arg;
    Compress *comp = nil;
    Pushstate ps;

    /* open archive to be updated */
    if (usefile && docreate)
        ar = create(usefile, OWRITE, 0666);
    else if (usefile) {
        if (docompress)
            sysfatal("cannot update compressed archive");
        ar = open(usefile, ORDWR);
    } else
        ar = STDOUT;

    /* push compression filter, if requested */
    if (docompress) {
        comp = compmethod(usefile);
        if (comp)
            ar = push(ar, comp->comp, Output, &ps);
    }
    if (ar < 0)
        sysfatal("can't open archive %s: %r", usefile);

    if (usefile && !docreate)
        skiptoend(ar);

    for (i = 0; argv[i] != nil; i++) {
        arg = argv[i];
        cleanname(arg);
        if (strcmp(arg, "..") == 0 || strncmp(arg, "../", 3) == 0)
            fprint(2, "%s: name starting with .. is a bad idea\n",
                argv0);
        addtoar(ar, arg, arg);
        chdir(origdir);     /* for correctness & profiling */
    }

    /* write end-of-archive marker */
    getblkz(ar);
    putblk(ar);
    getblkz(ar);
    putlastblk(ar);

    if (comp)
        return pushclose(&ps);
    if (ar > STDERR)
        close(ar);
    return nil;
}
```

9.1.5 Advanced features

⟨global flags tar.c 93a⟩+≡ (416b) ◁86d 94b▷

```
static bool ignerrs;     /* flag: ignore i/o errors if possible */
```

⟨main() *switch character cases (tar.c)* 93b⟩+≡ (90b) ◁91f 94a▷

```
case 'i':
```

```
    ignerrs = true;
    break;
```

⟨main() *switch character cases (tar.c)* 94a⟩+≡ (90b) ◁93b 94c▷
```
case 'k':
    keepexisting++;
    break;
```

⟨global flags tar.c 94b⟩+≡ (416b) ◁93a 94e▷
```
static bool settime;
```

⟨main() *switch character cases (tar.c)* 94c⟩+≡ (90b) ◁94a 94d▷
```
case 'm':   /* compatibility */
    settime = false;
    break;
```

⟨main() *switch character cases (tar.c)* 94d⟩+≡ (90b) ◁94c 94f▷
```
case 'T':
    settime = true;
    break;
```

⟨global flags tar.c 94e⟩+≡ (416b) ◁94b 94h▷
```
static bool posix = true;
```

⟨main() *switch character cases (tar.c)* 94f⟩+≡ (90b) ◁94d 94g▷
```
case 'p':
    posix = true;
    break;
```

⟨main() *switch character cases (tar.c)* 94g⟩+≡ (90b) ◁94f 94i▷
```
case 'P':
    posix = false;
    break;
```

⟨global flags tar.c 94h⟩+≡ (416b) ◁94e 94j▷
```
static bool relative = true;
```

⟨main() *switch character cases (tar.c)* 94i⟩+≡ (90b) ◁94g 94k▷
```
case 'R':
    relative = false;
    break;
```

⟨global flags tar.c 94j⟩+≡ (416b) ◁94h
```
static bool resync;
```

⟨main() *switch character cases (tar.c)* 94k⟩+≡ (90b) ◁94i 94m▷
```
case 's':
    resync = true;
    break;
```

⟨globals tar.c 94l⟩+≡ (416b) ◁91h 94n▷
```
static int aruid;
```

⟨main() *switch character cases (tar.c)* 94m⟩+≡ (90b) ◁94k 95a▷
```
case 'u':
    aruid = strtoul(EARGF(usage()), nil, 0);
    break;
```

⟨globals tar.c 94n⟩+≡ (416b) ◁94l 416a▷
```
static int argid;
```

⟨main() *switch character cases (tar.c)* 95a⟩+≡ (90b) ◁94m 95b▷
```
  case 'g':
      argid = strtoul(EARGF(usage()), nil, 0);
      break;
```

⟨main() *switch character cases (tar.c)* 95b⟩+≡ (90b) ◁95a
```
  case '-':
      break;
```

9.2 gzip

`gzip` is a thin CLI wrapper around `libflate`'s deflate implementation (see LIBCORE book [Pad16]). The gzip format (RFC 1952) is simple: a fixed header with magic bytes, a compression method (always deflate), optional file name and timestamp, the compressed data, and a trailing CRC-32 plus original size. The `gzip` function writes this header byte by byte, calls `deflate` to compress the data, then appends the CRC and size footer. The `crcread` callback computes the CRC incrementally as data is read, avoiding a separate pass over the input.

⟨*function* `usage` *(gzip.c)* 95c⟩≡ (433b)
```
  void
  usage(void)
  {
      fprint(STDERR, "usage: gzip [-vcD] [-1-9] [file ...]\n");
      exits("usage");
  }
```

⟨global flags gzip.c 95d⟩≡ (433b)
```
  // -v
  static  bool verbose;
  // -D
  static  bool debug;
```

⟨globals gzip.c 95e⟩≡ (433b) 95f▷
```
  static  Biobuf  bout;
  // compression level, default = -6
  static  int level;
```

⟨globals gzip.c 95f⟩+≡ (433b) ◁95e 99a▷
```
  static  ulong   *crctab;
```

⟨*gzip constants* 95g⟩≡ (432) 97a▷
```
  GZCRCPOLY   = 0xedb88320UL,
```

⟨*function* `main` *(gzip.c)* 95h⟩≡ (433b)
```
  void
  main(int argc, char *argv[])
  {
      int i;
      // enum<FlateError> (OK = 0) and then error0 (OK = 1)
      int ok;
      // -c
      bool stdout = false;

      level = 6;
      ARGBEGIN{
      case 'v':
          verbose = true;
          break;
      case 'D':
```

```
        debug = true;
        break;
    case 'c':
        stdout = true;
        break;
    case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        level = ARGC() - '0';
        break;
    default:
        usage();
        break;
    }ARGEND

    crctab = mkcrctab(GZCRCPOLY);
    ok = deflateinit();
    if(ok != FlateOk)
        sysfatal("deflateinit failed: %s", flateerr(ok));

    if(argc == 0){
        Binit(&bout, STDOUT, OWRITE);
        ok = gzip(nil, time(0), STDIN, &bout);
        Bterm(&bout);
    }else{
        ok = OK_1;
        for(i = 0; i < argc; i++)
            ok &= gzipf(argv[i], stdout);
    }
    exits(ok ? nil: "errors");
}
```

⟨*function* gzipf 96⟩≡ (433b)

```
static error0
gzipf(char *file, bool stdout)
{
    Dir *dir;
    char ofile[256], *f, *s;
    fdt ifd, ofd;
    int ok;

    ifd = open(file, OREAD);
    if(ifd < 0){
        fprint(STDERR, "gzip: can't open %s: %r\n", file);
        return ERROR_0;
    }
    dir = dirfstat(ifd);
    if(dir == nil){
        fprint(STDERR, "gzip: can't stat %s: %r\n", file);
        close(ifd);
        return ERROR_0;
    }
    if(dir->mode & DMDIR){
        fprint(STDERR, "gzip: can't compress a directory\n");
        close(ifd);
        free(dir);
        return ERROR_0;
    }

    if(stdout){
        ofd = STDOUT;
```

```
            strcpy(ofile, "<stdout>");
        }else{
            f = strrchr(file, '/');
            if(f != nil)
                f++;
            else
                f = file;
            s = strrchr(f, '.');
            if(s != nil && s != ofile && strcmp(s, ".tar") == ORD__EQ){
                *s = '\0';
                snprint(ofile, sizeof(ofile), "%s.tgz", f);
            }else
                snprint(ofile, sizeof(ofile), "%s.gz", f);
            ofd = create(ofile, OWRITE, 0666);
            if(ofd < 0){
                fprint(STDERR, "gzip: can't open %s: %r\n", ofile);
                close(ifd);
                return ERROR_0;
            }
        }

        if(verbose)
            fprint(STDERR, "compressing %s to %s\n", file, ofile);

        Binit(&bout, ofd, OWRITE);
        ok = gzip(file, dir->mtime, ifd, &bout);
        if(!ok || Bflush(&bout) < 0){
            fprint(STDERR, "gzip: error writing %s: %r\n", ofile);
            if(!stdout)
                remove(ofile);
        }
        Bterm(&bout);
        free(dir);
        close(ifd);
        close(ofd);
        return ok;
    }
```

⟨*gzip constants* 97a⟩+≡ (432) ◁95g 97b▷

```
    GZMAGIC1    = 0x1f,
    GZMAGIC2    = 0x8b,
```

⟨*gzip constants* 97b⟩+≡ (432) ◁97a

```
    GZDEFLATE   = 8,
```

⟨*gzip flags* 97c⟩≡ (432)

```
    GZFTEXT     = 1 << 0,       /* file is text */
    GZFHCRC     = 1 << 1,       /* crc of header included */
    GZFEXTRA    = 1 << 2,       /* extra header included */
    GZFNAME     = 1 << 3,       /* name of file included */
    GZFCOMMENT  = 1 << 4,       /* header comment included */
    GZFMASK     = (1 << 5) -1,      /* mask of specified bits */
```

⟨*gzip fs type* 97d⟩≡ (432) 98a▷

```
    GZOSFAT     = 0,            /* FAT file system */
    GZOSAMIGA   = 1,            /* Amiga */
    GZOSVMS     = 2,            /* VMS or OpenVMS */
    GZOSUNIX    = 3,            /* Unix */
    GZOSVMCMS   = 4,            /* VM/CMS */
    GZOSATARI   = 5,            /* Atari TOS */
```

```
  GZOSHPFS    = 6,            /* HPFS file system */
  GZOSMAC     = 7,            /* Macintosh */
  GZOSZSYS    = 8,            /* Z-System */
  GZOSCPM     = 9,            /* CP/M */
  GZOSTOPS20  = 10,           /* TOPS-20 */
  GZOSNTFS    = 11,           /* NTFS file system */
  GZOSQDOS    = 12,           /* QDOS */
  GZOSACORN   = 13,           /* Acorn RISCOS */
  GZOSUNK     = 255,
```

⟨*gzip fs type* 98a⟩+≡ (432) ◁97d

```
  GZOSINFERNO = GZOSUNIX,
```

⟨*function* gzip 98b⟩≡ (433b)

```
  static int
  gzip(char *file, long mtime, fdt ifd, Biobuf *bout)
  {
      int flags, err;

      flags = 0;
      Bputc(bout, GZMAGIC1);
      Bputc(bout, GZMAGIC2);
      Bputc(bout, GZDEFLATE);

      if(file != nil)
          flags |= GZFNAME;
      Bputc(bout, flags);

      Bputc(bout, mtime);
      Bputc(bout, mtime>>8);
      Bputc(bout, mtime>>16);
      Bputc(bout, mtime>>24);

      Bputc(bout, 0);
      Bputc(bout, GZOSINFERNO);

      if(flags & GZFNAME)
          Bwrite(bout, file, strlen(file)+1);

      crc = 0;
      eof = 0;
      totr = 0;
      err = deflate(bout, gzwrite, (void*)ifd, crcread, level, debug);
      if(err != FlateOk){
          fprint(STDERR, "gzip: deflate failed: %s\n", flateerr(err));
          return ERROR_0;
      }

      Bputc(bout, crc);
      Bputc(bout, crc>>8);
      Bputc(bout, crc>>16);
      Bputc(bout, crc>>24);

      Bputc(bout, totr);
      Bputc(bout, totr>>8);
      Bputc(bout, totr>>16);
      Bputc(bout, totr>>24);

      return OK_1;
  }
```

⟨globals gzip.c 99a⟩+≡ (433b) ◁95f
```
static  ulong   crc;
static  bool    eof;
static  ulong   totr;
```

⟨*function* crcread*(gzip.c)* 99b⟩≡ (433b)
```
static int
crcread(void *fd, void *buf, int n)
{
    int nr, m;

    nr = 0;
    for(; !eof && n > 0; n -= m){
        m = read((int)(uintptr)fd, (char*)buf+nr, n);
        if(m <= 0){
            eof = true;
            if(m < 0)
                return -1;
            break;
        }
        nr += m;
    }
    crc = blockcrc(crctab, crc, buf, nr);
    totr += nr;
    return nr;
}
```

⟨*function* gzwrite*(gzip.c)* 99c⟩≡ (433b)
```
static int
gzwrite(void *bout, void *buf, int n)
{
    if(n != Bwrite(bout, buf, n)){
        eof = true;
        return -1;
    }
    return n;
}
```

9.3 gunzip

⟨*function* usage*(gunzip.c)* 99d⟩≡ (433a)
```
void
usage(void)
{
    fprint(STDERR, "usage: gunzip [-ctvTD] [file ....]\n");
    exits("usage");
}
```

⟨globals flags gunzip.c 99e⟩≡ (433a)
```
// -v
static  bool verbose;
// -D
static  bool debug;
// -t ??
static  bool table;
// -T ??
static  bool settimes;
```

⟨globals gunzip.c 100a⟩≡ (433a) 100b▷
```
static  char    *infile;
static  Biobuf  bin;
```

⟨globals gunzip.c 100b⟩+≡ (433a) ◁100a 100c▷
```
static  ulong   *crctab;
```

⟨globals gunzip.c 100c⟩+≡ (433a) ◁100b 100e▷
```
static  vlong   gzok;
static  jmp_buf zjmp;
static  char    *delfile;
```

⟨*function* error*(gunzip.c)* 100d⟩≡ (433a)
```
static void
error(char *fmt, ...)
{
    va_list arg;

    if(gzok)
        fprint(STDERR, "gunzip: %s: corrupted data after byte %lld ignored\n", infile, gzok);
    else{
        fprint(STDERR, "gunzip: ");
        if(infile)
            fprint(STDERR, "%s: ", infile);
        va_start(arg, fmt);
        vfprint(STDERR, fmt, arg);
        va_end(arg);
        fprint(STDERR, "\n");

        if(delfile != nil){
            fprint(STDERR, "gunzip: removing output file %s\n", delfile);
            remove(delfile);
            delfile = nil;
        }
    }

    longjmp(zjmp, 1);
}
```

⟨globals gunzip.c 100e⟩+≡ (433a) ◁100c 101a▷
```
static  ulong   crc;
static  ulong   wlen;
```

⟨*struct* GZHead 100f⟩≡ (433a)
```
struct GZHead
{
    ulong   mtime;
    char    *file;
};
```

⟨*function* main*(gunzip.c)* 100g⟩≡ (433a)
```
void
main(int argc, char *argv[])
{
    int i;
    // enum<FlateError> (OK = 0) and then error0 (OK = 1, hmmm)
    int ok;
    // -c
    bool stdout;
```

```
    stdout = false;
    ARGBEGIN{
    case 'D':
        debug = true;
        break;
    case 'c':
        stdout = true;
        break;
    case 't':
        table = true;
        break;
    case 'T':
        settimes = true;
        break;
    case 'v':
        verbose = true;
        break;
    default:
        usage();
        break;
    }ARGEND

    crctab = mkcrctab(GZCRCPOLY);
    ok = inflateinit();
    if(ok != FlateOk)
        sysfatal("inflateinit failed: %s", flateerr(ok));

    if(argc == 0){
        Binit(&bin, STDIN, OREAD);
        settimes = false;
        infile = "<stdin>";
        ok = gunzip(STDOUT, "<stdout>", &bin);
    }else{
        ok = OK_1;
        if(stdout)
            settimes = false;
        for(i = 0; i < argc; i++)
            ok &= gunzipf(argv[i], stdout);
    }

    exits(ok ? nil: "errors");
}
```

⟨globals gunzip.c 101a⟩+≡ (433a) ◁100e

```
// error in writing (write is bad)
static  bool wbad;
```

⟨*function* gunzipf 101b⟩≡ (433a)

```
static error0
gunzipf(char *file, bool stdout)
{
    char ofile[256], *s;
    fdt ofd, ifd;
    int ok;

    infile = file;
    ifd = open(file, OREAD);
    if(ifd < 0){
        fprint(STDERR, "gunzip: can't open %s: %r\n", file);
        return ERROR_0;
```

```
}

Binit(&bin, ifd, OREAD);
if(Bgetc(&bin) != GZMAGIC1 || Bgetc(&bin) != GZMAGIC2 || Bgetc(&bin) != GZDEFLATE){
    fprint(STDERR, "gunzip: %s is not a gzip deflate file\n", file);
    Bterm(&bin);
    close(ifd);
    return ERROR_0;
}
Bungetc(&bin);
Bungetc(&bin);
Bungetc(&bin);

if(table)
    ofd = -1;
else if(stdout){
    ofd = STDOUT;
    strcpy(ofile, "<stdout>");
}else{
    s = strrchr(file, '/');
    if(s != nil)
        s++;
    else
        s = file;
    strecpy(ofile, ofile+sizeof ofile, s);
    s = strrchr(ofile, '.');
    if(s != nil && s != ofile && strcmp(s, ".gz") == ORD__EQ)
        *s = '\0';
    else if(s != nil && strcmp(s, ".tgz") == ORD__EQ)
        strcpy(s, ".tar");
    else if(strcmp(file, ofile) == ORD__EQ){
        fprint(STDERR, "gunzip: can't overwrite %s\n", file);
        Bterm(&bin);
        close(ifd);
        return ERROR_0;
    }

    ofd = create(ofile, OWRITE, 0666);
    if(ofd < 0){
        fprint(STDERR, "gunzip: can't create %s: %r\n", ofile);
        Bterm(&bin);
        close(ifd);
        return ERROR_0;
    }
    delfile = ofile;
}

wbad = false;
// back go gunzip
ok = gunzip(ofd, ofile, &bin);
Bterm(&bin);
close(ifd);
if(wbad){
    fprint(STDERR, "gunzip: can't write %s: %r\n", ofile);
    if(delfile)
        remove(delfile);
}
delfile = nil;
if(!stdout && ofd >= 0)
    close(ofd);
```

```
    return ok;
}
```

⟨*function* gunzip 103a⟩≡ (433a)

```
static error0
gunzip(fdt ofd, char *ofile, Biobuf *bin)
{
    Dir *d;
    GZHead h;
    int err;

    h.file = nil;
    gzok = 0;
    for(;;){
        if(Bgetc(bin) < 0)
            return OK_1;
        // else
        Bungetc(bin);

        if(setjmp(zjmp))
            return ERROR_0;
        header(bin, &h);
        gzok = 0;

        wlen = 0;
        crc = 0;

        if(!table && verbose)
            fprint(STDERR, "extracting %s to %s\n", h.file, ofile);

        // call to libflate library
        err = inflate((void*)ofd, crcwrite, bin, (int(*)(void*))Bgetc);
        if(err != FlateOk)
            error("inflate failed: %s", flateerr(err));

        trailer(bin, wlen);

        if(table){
            if(verbose)
                print("%-32s %10ld %s", h.file, wlen, ctime(h.mtime));
            else
                print("%s\n", h.file);
        }else if(settimes && h.mtime && (d=dirfstat(ofd)) != nil){
            d->mtime = h.mtime;
            dirfwstat(ofd, d);
            free(d);
        }

        free(h.file);
        h.file = nil;
        gzok = Boffset(bin);
    }
}
```

⟨*function* crcwrite*(gunzip.c)* 103b⟩≡ (433a)

```
static int
crcwrite(void *out, void *buf, int n)
{
    fdt fd;
    int nw;
```

```
    wlen += n;
    // blockcrc in libflate
    crc = blockcrc(crctab, crc, buf, n);
    fd = (int)(uintptr)out;
    if(fd < 0)
        return n;
    nw = write(fd, buf, n);
    if(nw != n)
        wbad = true;
    return nw;
}
```

⟨*function* header*(gunzip.c)* 104⟩≡ (433a)

```
static void
header(Biobuf *bin, GZHead *h)
{
    char *s;
    int i, c, flag, ns, nsa;

    if(get1(bin) != GZMAGIC1 || get1(bin) != GZMAGIC2)
        error("bad gzip file magic");
    if(get1(bin) != GZDEFLATE)
        error("unknown compression type");

    flag = get1(bin);
    if(flag & ~(GZFTEXT|GZFEXTRA|GZFNAME|GZFCOMMENT|GZFHCRC))
        fprint(STDERR, "gunzip: reserved flags set, data may not be decompressed correctly\n");

    /* mod time */
    h->mtime = get4(bin);

    /* extra flags */
    get1(bin);

    /* OS type */
    get1(bin);

    if(flag & GZFEXTRA)
        for(i=get1(bin); i>0; i--)
            get1(bin);

    /* name */
    if(flag & GZFNAME){
        nsa = 32;
        ns = 0;
        s = malloc(nsa);
        if(s == nil)
            error("out of memory");
        while((c = get1(bin)) != 0){
            s[ns++] = c;
            if(ns >= nsa){
                nsa += 32;
                s = realloc(s, nsa);
                if(s == nil)
                    error("out of memory");
            }
        }
        s[ns] = '\0';
        h->file = s;
```

```
    }else
        h->file = strdup("<unnamed file>");

    /* comment */
    if(flag & GZFCOMMENT)
        while(get1(bin) != 0)
            ;

    /* crc16 */
    if(flag & GZFHCRC){
        get1(bin);
        get1(bin);
    }
}
```

⟨*function* `trailer` *(gunzip.c)* 105a⟩≡ (433a)

```
static void
trailer(Biobuf *bin, long wlen)
{
    ulong tcrc;
    long len;

    tcrc = get4(bin);
    if(tcrc != crc)
        error("crc mismatch");

    len = get4(bin);

    if(len != wlen)
        error("bad output length: expected %lud got %lud", wlen, len);
}
```

⟨*function* `get1` *(gunzip.c)* 105b⟩≡ (433a)

```
static int
get1(Biobuf *b)
{
    int c;

    c = Bgetc(b);
    if(c < 0)
        error("unexpected eof reading file information");
    return c;
}
```

⟨*function* `get4` *(gunzip.c)* 105c⟩≡ (433a)

```
static ulong
get4(Biobuf *b)
{
    ulong v;
    int i, c;

    v = 0;
    for(i = 0; i < 4; i++){
        c = Bgetc(b);
        if(c < 0)
            error("unexpected eof reading file information");
        v |= c << (i * 8);
    }
    return v;
}
```

Chapter 10

Time

This chapter covers `date` and `cal`, two simple utilities dealing with time. `date` prints or converts timestamps; `cal` prints a calendar for a given month or year.

10.1 `date`

⟨*global flags(date.c)* 106a⟩≡ (235a)
```
// UTC time, which is mostly the same than GMT
bool uflg;
// number of seconds since the (UNIX) epoch, 1/1/1970:00:00
bool nflg;
```

⟨*function* `main`*(date.c)* 106b⟩≡ (235a)
```
void
main(int argc, char *argv[])
{
    ulong now;

    ARGBEGIN{
    case 'n':   nflg = true; break;
    case 'u':   uflg = true; break;
    default:
       fprint(STDERR, "usage: date [-un] [seconds]\n");
       exits("usage");
    }ARGEND

    if(argc == 1)
        now = strtoul(*argv, nil, 0);
    else
        now = time(0);

    if(nflg)
        print("%ld\n", now);
    else if(uflg)
        print("%s", asctime(gmtime(now)));
    else
        print("%s", ctime(now));

    exits(nil);
}
```

10.2 cal

`cal` computes calendars using the `jan1` function, which calculates the day of the week for January 1 of any year using Gregorian and Julian calendar rules. The code even handles the 1752 calendar reform, when September lost 11 days as Britain switched from the Julian to the Gregorian calendar.

The 1752 hack is worth slowing down on, because running `cal 9 1752` on any UNIX-derived system literally prints a September with eleven missing days—a piece of historical weirdness baked into the calendar code:

```
$ cal 9 1752
     September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

The story: until 1582, the entire Christian world used the *Julian* calendar, which assumes a year is exactly 365.25 days. The real solar year is shorter by about 11 minutes, so by the 1500s the Julian calendar had drifted about 10 days ahead of the actual seasons. Pope Gregory XIII fixed this in 1582 by (a) skipping 10 days that October so the spring equinox landed back on March 21, and (b) changing the leap-year rule: century years are leap years *only* if divisible by 400. That is the famous `if (year%4==0 && (year%100!=0 || year%400==0))` test every programmer eventually writes.

Catholic countries adopted the new calendar immediately; Protestant Britain and its colonies dragged their feet for 170 years. By 1752 the gap had grown from 10 to 11 days, so when Britain finally switched, Wednesday September 2 1752 was followed directly by Thursday September 14 1752—eleven days that historically did not exist on British soil, including British North America (this is why some early American historical records have peculiar gaps). `jan1()X` encodes this in three rules:

```
d = 4 + y + (y+3)/4              ; Julian: leap day every 4 years
if y > 1800:                     ; switch to proper Gregorian rule
    d -= (y-1701)/100            ;   subtract century years
    d += (y-1601)/400            ;   add back the /400 ones
if y > 1752:                     ; the British switch
    d += 3                       ;   shift by 11-day gap
                                 ;   (modulo 7 = 4 weekdays + 4)
return d % 7
```

And `cal()`[109] makes September 1752 explicitly short: when it detects `mon[m]==19` (its sentinel for "the short September") and the print loop reaches day 3, it jumps to day 14 and extends the month length so the loop ends at the right place. This is where the magic `mon[9] = 19` in `cal()` comes from—it is not a typo, it is the encoded length of September 1752 specifically. POSIX `cal` inherits this behaviour from Plan 9/Berkeley `cal`, which inherited it from Research UNIX, which inherited it from ... a real piece of British political history.

⟨constants cal.c 107⟩≡ (235b)

```
char    dayw[] =
{
    " S  M Tu  W Th  F  S"
};
char    *smon[] =
{
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December",
};
char    mon[] =
```

```
{
    0,
    31, 29, 31, 30,
    31, 30, 31, 31,
    30, 31, 30, 31,
};
```

⟨globals cal.c 108a⟩≡ (235b)
```
char    string[432];
Biobuf  bout;
```

⟨*function* main*(cal.c)* 108b⟩≡ (235b)
```
void
main(int argc, char *argv[])
{
    int y, m;
    int i, j;

    if(argc > 3) {
        fprint(STDERR, "usage: cal [month] [year]\n");
        exits("usage");
    }
    Binit(&bout, STDOUT, OWRITE);

/*
 * no arg, print current month
 */
    if(argc == 1) {
        m = curmo();
        y = curyr();
        goto xshort;
    }

/*
 * one arg
 *  if looks like a month, print month
 *  else print year
 */
    if(argc == 2) {
        y = number(argv[1]);
        if(y < 0)
            y = -y;
        if(y >= 1 && y <= 12) {
            m = y;
            y = curyr();
            goto xshort;
        }
        goto xlong;
    }

/*
 * two arg, month and year
 */
    m = number(argv[1]);
    if(m < 0)
        m = -m;
    y = number(argv[2]);
    goto xshort;

/*
```

```
 *  print out just month
 */
xshort:
    if(m < 1 || m > 12)
        goto badarg;
    if(y < 1 || y > 9999)
        goto badarg;
    Bprint(&bout, "   %s %ud\n", smon[m-1], y);
    Bprint(&bout, "%s\n", dayw);
    cal(m, y, string, 24);
    for(i=0; i<6*24; i+=24)
        pstr(string+i, 24);
    exits(nil);

/*
 *  print out complete year
 */
xlong:
    y = number(argv[1]);
    if(y<1 || y>9999)
        goto badarg;
    Bprint(&bout, "\n\n\n");
    Bprint(&bout, "                                %ud\n", y);
    Bprint(&bout, "\n");
    for(i=0; i<12; i+=3) {
        for(j=0; j<6*72; j++)
            string[j] = '\0';
        Bprint(&bout, "         %.3s", smon[i]);
        Bprint(&bout, "                     %.3s", smon[i+1]);
        Bprint(&bout, "                     %.3s\n", smon[i+2]);
        Bprint(&bout, "%s   %s   %s\n", dayw, dayw, dayw);
        cal(i+1, y, string, 72);
        cal(i+2, y, string+23, 72);
        cal(i+3, y, string+46, 72);
        for(j=0; j<6*72; j+=72)
            pstr(string+j, 72);
    }
    Bprint(&bout, "\n\n\n");
    exits(nil);

badarg:
    Bprint(&bout, "cal: bad argument\n");
}
```

⟨*function* cal 109⟩≡ (235b)

```
void
cal(int m, int y, char *p, int w)
{
    int d, i;
    char *s;

    s = p;
    d = jan1(y);
    mon[2] = 29;
    mon[9] = 30;

    switch((jan1(y+1)+7-d)%7) {

    /*
     *  non-leap year
```

```
     */
    case 1:
        mon[2] = 28;
        break;

    /*
     *  1752
     */
    default:
        mon[9] = 19;
        break;

    /*
     *  leap year
     */
    case 2:
        ;
    }
    for(i=1; i<m; i++)
        d += mon[i];
    d %= 7;
    s += 3*d;
    for(i=1; i<=mon[m]; i++) {
        if(i==3 && mon[m]==19) {
            i += 11;
            mon[m] += 11;
        }
        if(i > 9)
            *s = i/10+'0';
        s++;
        *s++ = i%10+'0';
        s++;
        if(++d == 7) {
            d = 0;
            s = p+w;
            p = s;
        }
    }
}
```

⟨*function* jan1*(cal.c)* 110⟩≡ (235b)

```
/*
 *  return day of the week
 *  of jan 1 of given year
 */
int
jan1(int yr)
{
    int y, d;

/*
 *  normal gregorian calendar
 *  one extra day per four years
 */

    y = yr;
    d = 4+y+(y+3)/4;

/*
 *  julian calendar
```

```
     *  regular gregorian
     *  less three days per 400
     */

        if(y > 1800) {
            d -= (y-1701)/100;
            d += (y-1601)/400;
        }

    /*
     *  great calendar changeover instant
     */

        if(y > 1752)
            d += 3;

        return d%7;
    }
```

⟨*function* curmo*(cal.c)* 111a⟩≡ (235b)

```
    /*
     * system dependent
     * get current month and year
     */
    int
    curmo(void)
    {
        Tm *tm;

        tm = localtime(time(0));
        return tm->mon+1;
    }
```

⟨*function* curyr*(cal.c)* 111b⟩≡ (235b)

```
    int
    curyr(void)
    {
        Tm *tm;

        tm = localtime(time(0));
        return tm->year+1900;
    }
```

⟨*struct* dict*(cal.c)* 111c⟩≡ (235b)

```
    struct
    {
        char*   word;
        int val;
    } dict[] =
    {
        "jan",      1,
        "january",  1,
        "feb",      2,
        "february", 2,
        "mar",      3,
        "march",    3,
        "apr",      4,
        "april",    4,
        "may",      5,
        "jun",      6,
```

```
    "june",     6,
    "jul",      7,
    "july",     7,
    "aug",      8,
    "august",   8,
    "sep",      9,
    "sept",     9,
    "september",    9,
    "oct",      10,
    "october",  10,
    "nov",      11,
    "november", 11,
    "dec",      12,
    "december", 12,
    0
};
```

⟨*function* number*(cal.c)* 112a⟩≡ (235b)

```
/*
 * convert to a number.
 * if its a dictionary word,
 * return negative  number
 */
int
number(char *str)
{
    int n, c;
    char *s;

    for(n=0; s=dict[n].word; n++)
        if(strcmp(s, str) == 0)
            return -dict[n].val;
    n = 0;
    s = str;
    while(c = *s++) {
        if(c<'0' || c>'9')
            return 0;
        n = n*10 + c-'0';
    }
    return n;
}
```

⟨*function* pstr*(cal.c)* 112b⟩≡ (235b)

```
void
pstr(char *str, int n)
{
    int i;
    char *s;

    s = str;
    i = n;
    while(i--)
        if(*s++ == '\0')
            s[-1] = ' ';
    i = n+1;
    while(i--)
        if(*--s != ' ')
            break;
    s[1] = '\0';
    Bprint(&bout, "%s\n", str);
}
```

Chapter 11

Pipes

These utilities are designed to be stages in a pipeline: `tee` duplicates a stream, `p` paginates it, and `mc` columnates it. They embody the UNIX philosophy of small programs composed through pipes.

11.1 `tee`

`tee` reads from standard input and writes to both standard output and one or more files simultaneously—named after a T-shaped pipe fitting. The implementation is a simple read loop that writes each buffer to every open file descriptor.

⟨*global flags(tee.c)* 113a⟩≡ (234a)
```
// append mode
bool aflag;
// deprecated
bool uflag;
```

⟨*global* `openf` *(tee.c)* 113b⟩≡ (234a)
```
fdt *openf;
```

⟨*global* `in` *(tee.c)* 113c⟩≡ (234a)
```
// read buffer
char in[8192];
```

⟨*function* `main` *(tee.c)* 113d⟩≡ (234a)
```
void
main(int argc, char **argv)
{
    int i;
    int r, n;

    ARGBEGIN {
    case 'a':
        aflag = true;
        break;

    case 'i':
        atnotify(intignore, true); // register
        break;

    case 'u':
        uflag = true;
        /* uflag is ignored and undocumented; it's a relic from Unix */
        break;
```

```
    default:
        fprint(STDERR, "usage: tee [-ai] [file ...]\n");
        exits("usage");
    } ARGEND

    openf = malloc((1+argc)*sizeof(int));
    if(openf == nil)
        sysfatal("out of memory: %r");

    n = 0;
    while(*argv) {
        if(aflag) {
            openf[n] = open(argv[0], OWRITE);
            if(openf[n] < 0)
                openf[n] = create(argv[0], OWRITE, 0666);
            seek(openf[n], 0L, SEEK__END);
        } else
            openf[n] = create(argv[0], OWRITE, 0666);
        if(openf[n] < 0) {
            fprint(STDERR, "tee: cannot open %s: %r\n", argv[0]);
        } else
            n++;
        argv++;
    }
    openf[n++] = STDOUT;

    for(;;) {
        r = read(STDIN, in, sizeof in);
        if(r <= 0)
            exits(nil);
        for(i=0; i<n; i++)
            write(openf[i], in, r);
    }
}
```

⟨*function* `intignore` *(tee.c)* 114a⟩≡ (234a)

```
bool
intignore(void *a, char *msg)
{
    USED(a);
    if(strcmp(msg, "interrupt") == ORD__EQ)
        return true;
    return false;
}
```

11.2 paginate: `p` (`more`, `less`)

`p` is Plan 9's pager, equivalent to UNIX's `more`. It displays output one screenful at a time, reading interactive commands from `/dev/cons` (or `/dev/tty` on UNIX). This dual-input design—data from stdin, commands from the console—is why `p` opens `/dev/cons` directly rather than reading commands from stdin.

⟨*constant* `DEF` *(p.c)* 114b⟩≡ (227)

```
#define DEF 22  /* lines in chunk: 3*DEF == 66, #lines per nroff page */
```

⟨globals p.c 114c⟩≡ (227)

```
Biobuf *cons;
Biobuf bout;

int pglen = DEF;
```

⟨*function* `main`*(p.c)* 115a⟩≡ (227)

```
void
main(int argc, char *argv[])
{
    int n;
    fdt f;

    // plan9 uses /dev/cons and Linux /dev/tty (both virtual devices)
    if((cons = Bopen("/dev/cons", OREAD)) == nil) {
        if((cons = Bopen("/dev/tty", OREAD)) == nil) {
            fprint(STDERR, "p: can't open /dev/cons or /dev/tty\n");
            exits("missing /dev/cons or /dev/tty");
        }
    }
    Binit(&bout, STDOUT, OWRITE);
    n = 0;
    while(argc > 1) {
        --argc; argv++;
        if(*argv[0] == '-'){
            pglen = atoi(&argv[0][1]);
            if(pglen <= 0)
                pglen = DEF;
        } else {
            n++;
            f = open(argv[0], OREAD);
            if(f < 0){
                fprint(STDERR, "p: can't open %s - %r\n", argv[0]);
                continue;
            }
            printfile(f);
            close(f);
        }
    }
    if(n == 0)
        printfile(STDIN);
    exits(nil);
}
```

⟨*function* `printfile`*(p.c)* 115b⟩≡ (227)

```
void
printfile(fdt f)
{
    int i, j, n;
    char *s, *cmd;
    Biobuf *b;

    b = malloc(sizeof(Biobuf));
    Binit(b, f, OREAD);
    for(;;){
        for(i=1; i <= pglen; i++) {
            s = Brdline(b, '\n');
            if(s == 0){
                n = Blinelen(b);
                if(n > 0)   /* line too long for Brdline */
                    for(j=0; j<n; j++)
                        Bputc(&bout, Bgetc(b));
                else{       /* true EOF */
                    free(b);
                    return;
                }
```

```
            }else{
                Bwrite(&bout, s, Blinelen(b)-1);
                if(i < pglen)
                    Bwrite(&bout, "\n", 1);
            }
        }
        Bflush(&bout);
        getcmd:
        cmd = Brdline(cons, '\n');
        if(cmd == nil || *cmd == 'q')
            exits(nil);
        cmd[Blinelen(cons)-1] = 0;
        if(*cmd == '!'){
            if(fork() == 0){
                dup(Bfildes(cons), 0);
                execl("/bin/rc", "rc", "-c", cmd+1, nil);
            }
            waitpid();
            goto getcmd;
        }
    }
}
```

11.3 columnate: `mc`

`mc` (multi-column) reformats a list of items into columns, filling top-to-bottom then left-to-right. It computes the maximum word width, divides the terminal width by that to determine how many columns fit, then prints each row by sampling every `nlines`-th word. This is the utility behind the multi-column output of commands like `ls | mc` in Plan 9.

⟨constants mc.c 116a⟩≡ (225b) 116f▷
```
#define WIDTH           80
#define TAB 4
```

⟨globals mc.c 116b⟩≡ (225b) 116e▷
```
int linewidth=WIDTH;
```

⟨global flags mc.c 116c⟩≡ (225b) 116d▷
```
// ??
bool colonflag = false;
```

⟨global flags mc.c 116d⟩+≡ (225b) ◁116c
```
// set to true in acme
bool tabflag = false;  /* -t flag turned off forever */
```

⟨globals mc.c 116e⟩+≡ (225b) ◁116b 116g▷
```
Biobuf  bin;
Biobuf  bout;
```

⟨constants mc.c 116f⟩+≡ (225b) ◁116a
```
#define ALLOC_QUANTA        4096
#define WORD_ALLOC_QUANTA   1024
```

⟨globals mc.c 116g⟩+≡ (225b) ◁116e 118b▷
```
Rune *cbuf, *cbufp;
Rune **word;
```

⟨*function* main*(mc.c)* 117⟩≡ (225b)

```
void
main(int argc, char *argv[])
{
    int i;
    bool lineset = false;
    fdt ifd;

    Binit(&bout, STDOUT, OWRITE);

    // Why no ARGBEGIN/ARGEND?
    while(argc > 1 && argv[1][0] == '-'){
        --argc; argv++;
        switch(argv[0][1]){
        case '\0':
            colonflag = true;
            break;
        // useful only for acme
        case 't':
            tabflag = false;
            break;
        default:
            linewidth = atoi(&argv[0][1]);
            if(linewidth <= 1)
                linewidth = WIDTH;
            lineset = true;
            break;
        }
    }

    if(!lineset){
        getwidth();
        if(linewidth <= 1){
            linewidth = WIDTH;
            font = nil;
        }
    }

    cbuf = cbufp = malloc(ALLOC_QUANTA*(sizeof *cbuf));
    word = malloc(WORD_ALLOC_QUANTA*(sizeof *word));
    if(word == nil || cbuf == nil)
        error("out of memory");

    if(argc == 1)
        readbuf(STDIN);
    else{
        for(i = 1; i < argc; i++){
            if((ifd = open(*++argv, OREAD)) == -1)
                fprint(STDERR, "mc: can't open %s (%r)\n", *argv);
            else{
                readbuf(ifd);
                Bflush(&bin);
                close(ifd);
            }
        }
    }
    columnate();
    exits(nil);
}
```

⟨*function* `error` *(mc.c)* 118a⟩≡ (225b)

```
void
error(char *s)
{
    fprint(STDERR, "mc: %s\n", s);
    exits(s);
}
```

⟨globals mc.c 118b⟩+≡ (225b) ◁116g 119d▷

```
int nalloc=ALLOC_QUANTA;
int nchars=0;
```

⟨*function* `readbuf` *(mc.c)* 118c⟩≡ (225b)

```
void
readbuf(fdt fd)
{
    int lastwascolon = 0;
    int linesiz = 0;
    long c;

    Binit(&bin, fd, OREAD);
    do{
        if(nchars++ >= nalloc)
            morechars();
        *cbufp++ = c = Bgetrune(&bin);
        linesiz++;
        if(c == '\t') {
            cbufp[-1] = L' ';
            while(linesiz%TAB != 0) {
                if(nchars++ >= nalloc)
                    morechars();
                *cbufp++ = L' ';
                linesiz++;
            }
        }
        if(colonflag && c == ':')
            lastwascolon++;
        else if(lastwascolon){
            if(c == '\n'){
                --nchars;   /* skip newline */
                *cbufp = L'\0';
                while(nchars > 0 && cbuf[--nchars] != '\n')
                    ;
                if(nchars)
                    nchars++;
                columnate();
                if (nchars)
                    Bputc(&bout, '\n');
                Bprint(&bout, "%S", cbuf+nchars);
                nchars = 0;
                cbufp = cbuf;
            }
            lastwascolon = 0;
        }
        if(c == '\n')
            linesiz = 0;
    }while(c >= 0);
}
```

⟨*function* `morechars`*(mc.c)* 119a⟩≡ (225b)
```
void
morechars(void)
{
    nalloc += ALLOC_QUANTA;
    if((cbuf = realloc(cbuf, nalloc*sizeof(*cbuf))) == nil)
        error("out of memory");
    cbufp = cbuf+nchars-1;
}
```

⟨*function* `getwidth`*(mc.c)(unix)* 119b⟩≡ (225b)
```
void getwidth(void)
{
}
```

⟨*function* `wordwidth`*(mc.c)(unix)* 119c⟩≡ (225b)
```
int
wordwidth(Rune *w, int nw)
{
    return nw;
}
```

⟨`globals mc.c` 119d⟩+≡ (225b) ◁118b 120a▷
```
int nwords=0;
int mintab=1;
int maxwidth=0;
```

The inner loop uses a strided sampling trick that is worth unpacking because it is not obvious at first read. Given `nwords` items, a maximum word width of `maxwidth`, and a terminal width of `linewidth`, the function computes `words_per_line = linewidth/maxwidth` and `nlines = ceil(nwords/words_per_line)`, then prints row `i` by picking words at indices `i, i+nlines, i+2*nlines, ....` The stride `nlines` gives the top-to-bottom fill order that programmers expect from `ls`: alphabetical neighbours sit in the same column, not the same row. Concretely, with 10 words and a 24-column terminal where the widest word is 7 characters (`words_per_line = 3`, `nlines = 4`):

```
word[] = a b c d e f g h i j

print layout (row i, then word[j] for j = i, i+4, i+8):
                      col0   col1   col2
            row 0:    a      e      i
            row 1:    b      f      j
            row 2:    c      g
            row 3:    d      h
```

Had the code instead walked `word` linearly with stride 1 and wrapped at `words_per_line` the columns would read `a b c | d e f | g h i | j`—row-major order, which is what GNU `ls` does by default but which splits alphabetically adjacent names across columns. The stride trick achieves column-major ordering without any transposition step: `word` stays a single flat array and the indexing formula is all that differs. The padding loop at the bottom of the inner `for` handles the last column specially: the final word on each row gets no trailing padding, so short last columns don't leave a rectangle of spaces that confuses downstream tools.

⟨*function* `columnate`*(mc.c)* 119e⟩≡ (225b)
```
void
columnate(void)
{
    int i, j;
    int words_per_line;
```

```
    int nlines;
    int col;
    int endcol;


    scanwords();
    if(nwords==0)
        return;
    maxwidth = nexttab(maxwidth+mintab-1);
    words_per_line = linewidth/maxwidth;
    if(words_per_line <= 0)
        words_per_line = 1;
    nlines=(nwords+words_per_line-1)/words_per_line;
    for(i = 0; i < nlines; i++){
        col = endcol = 0;
        for(j = i; j < nwords; j += nlines){
            endcol += maxwidth;
            Bprint(&bout, "%S", word[j]);
            col += wordwidth(word[j], runestrlen(word[j]));
            if(j+nlines < nwords){
                if(tabflag) {
                    while(col < endcol){
                        Bputc(&bout, '\t');
                        col = nexttab(col);
                    }
                }else{
                    while(col < endcol){
                        Bputc(&bout, ' ');
                        col++;
                    }
                }
            }
        }
        Bputc(&bout, '\n');
    }
}
```

⟨globals mc.c 120a⟩+≡ (225b) ◁119d 121a▷

```
int nwalloc=WORD_ALLOC_QUANTA;
```

⟨*function* scanwords*(mc.c)* 120b⟩≡ (225b)

```
void
scanwords(void)
{
    Rune *p, *q;
    int i, w;

    nwords=0;
    maxwidth=0;
    for(p = q = cbuf, i = 0; i < nchars; i++){
        if(*p++ == L'\n'){
            if(nwords >= nwalloc){
                nwalloc += WORD_ALLOC_QUANTA;
                if((word = realloc(word, nwalloc*sizeof(*word)))==0)
                    error("out of memory");
            }
            word[nwords++] = q;
            p[-1] = L'\0';
            w = wordwidth(q, p-q-1);
            if(w > maxwidth)
```

```
                maxwidth = w;
            q = p;
        }
    }
}
```

⟨globals mc.c 121a⟩+≡ (225b) ◁120a
```
int tabwidth=0;
```

⟨*function* nexttab*(mc.c)* 121b⟩≡ (225b)
```
int
nexttab(int col)
{
    if(tabwidth){
        col += tabwidth;
        col -= col%tabwidth;
        return col;
    }
    return col+1;
}
```

11.4 sort

`sort` is one of the most important UNIX utilities—combined with `uniq`, `join`, and `comm`, it turns the shell into a crude relational database. The ability to sort arbitrarily large files by multiple keys, merge pre-sorted files, and eliminate duplicates makes it a fundamental building block for data processing pipelines.

Unix `sort` is an *external* sort: it is designed to handle input larger than RAM, which is the case that in-memory `qsort` cannot serve. The classical structure has three phases, forming a small pipeline inside the one process:

```
phase 1: read input in chunks that fit in memory,
         qsort each chunk in place, write it out as
         a temporary ‘‘run’’ file

        input  --->  run_0  run_1  run_2  ...  run_k
        (tens of GB)    (each ~1 MB, sorted)

phase 2: k-way merge the run files using a priority
         queue (binary heap) keyed on the current head
         line of each run

        run_0   --\
        run_1     \
        run_2      >---(min-heap of k heads)---> output
        ...       /
        run_k   --/

phase 3: if k exceeds the file-descriptor budget or the
         heap gets too wide, merge runs in passes:
         merge 16 runs at a time into intermediate runs,
         repeat log_16(k) passes until one run remains
```

The first phase is the "replacement-selection" or simple quicksort version depending on the implementation; the second phase is the reason people still teach priority queues in CS courses; the third phase is why old Unix `sort` manpages talk about `-T` (temp directory) and why on a busy system `sort huge.log` can generate a conspicuous amount of disk traffic even though the final output looks like a single stream. The Plan 9 source for `sort` is not reproduced in this book—it is one of the larger utilities and the algorithm is standard—but it is useful to remember that the friendly one-liner `sort file | uniq -c | sort -rn` hides an external merge, a linear scan, and a second external merge behind two pipes.

Basic sort

Advanced sort

11.5 `uniq`

`uniq` removes consecutive duplicate lines from sorted input. The modes are: `-d` to print only duplicates, `-u` to print only unique lines, and `-c` to prefix each line with its repeat count. The comparison uses a two-buffer ping-pong: read into `b1`, then `b2`, compare, and output whichever buffer has a new unique value.

⟨globals uniq.c 122a⟩≡ (266) 122b▷
```
int fields  = 0;
int letters = 0;
// 'u' or 'd' or 'c' or 's' (meaning??)
char    mode;
```

⟨globals uniq.c 122b⟩+≡ (266) ◁122a 122c▷
```
int linec   = 0;
```

⟨globals uniq.c 122c⟩+≡ (266) ◁122b 122d▷
```
bool uniq;
```

⟨globals uniq.c 122d⟩+≡ (266) ◁122c 122e▷
```
char    *b1, *b2;
long    bsize = SIZE;
```

⟨globals uniq.c 122e⟩+≡ (266) ◁122d
```
Biobuf  fin;
Biobuf  fout;
```

⟨*constant* SIZE*(uniq.c)* 122f⟩≡ (266)
```
#define SIZE    8000
```

⟨*function* main*(uniq.c)* 122g⟩≡ (266)
```
void
main(int argc, char *argv[])
{
    fdt f = STDIN;

    argv0 = argv[0]; // use??
    b1 = malloc(bsize);
    b2 = malloc(bsize);
    while(argc > 1) {
        if(*argv[1] == '-') {
            if(isdigit(argv[1][1]))
                fields = atoi(&argv[1][1]);
            else
                mode = argv[1][1];
            argc--;
```

```
            argv++;
            continue;
        }
        if(*argv[1] == '+') {
            letters = atoi(&argv[1][1]);
            argc--;
            argv++;
            continue;
        }
        f = open(argv[1], OREAD);
        if(f < 0)
            sysfatal("cannot open %s", argv[1]);
        break;
    }
    if(argc > 2)
        sysfatal("unexpected argument %s", argv[2]);
    Binit(&fin, f, OREAD);
    Binit(&fout, STDOUT, OWRITE);

    if(gline(b1))
        exits(nil);

    for(;;) {
        linec++;
        if(gline(b2)) {
            pline(b1);
            exits(nil);
        }
        if(!equal(b1, b2)) {
            pline(b1);
            linec = 0;
            do {
                linec++;
                if(gline(b1)) {
                    pline(b2);
                    exits(nil);
                }
            } while(equal(b2, b1));
            pline(b2);
            linec = 0;
        }
    }
}
```

⟨*function* `gline` *(uniq.c)* 123⟩≡ (266)

```
bool
gline(char *buf)
{
    int len;
    char *p;

    p = Brdline(&fin, '\n');
    if(p == nil)
        return true;
    len = Blinelen(&fin);
    if(len >= bsize-1)
        sysfatal("line too long");
    memmove(buf, p, len);
    buf[len-1] = '\0';
    return false;
```

```
}
```

⟨*function* `pline` *(uniq.c)* 124a⟩≡ (266)

```
void
pline(char *buf)
{
    switch(mode) {

    case 'u':
        if(uniq) {
            uniq = false;
            return;
        }
        break;

    case 'd':
        if(uniq)
            break;
        return;

    case 'c':
        Bprint(&fout, "%4d ", linec);
    }
    uniq = false;
    Bprint(&fout, "%s\n", buf);
}
```

⟨*function* `equal` *(uniq.c)* 124b⟩≡ (266)

```
bool
equal(char *b1, char *b2)
{
    char c;

    if(fields || letters) {
        b1 = skip(b1);
        b2 = skip(b2);
    }
    for(;;) {
        c = *b1++;
        if(c != *b2++) {
            if(c == '\0' && mode == 's')
                return true;
            return false;
        }
        if(c == '\0') {
            uniq = true;
            return true;
        }
    }
}
```

⟨*function* `skip` *(uniq.c)* 124c⟩≡ (266)

```
char*
skip(char *s)
{
    int nf, nl;

    nf = nl = 0;
    while(nf++ < fields) {
        while(*s == ' ' || *s == '\t')
```

```
            s++;
        while(!(*s == ' ' || *s == '\t' || *s == '\0') )
            s++;
    }
    while(nl++ < letters && *s != '\0')
            s++;
    return s;
}
```

11.6 xargs

`xargs` reads arguments from standard input and executes a command with those arguments—essential for turning the output of one command into arguments for another (`find | grep .c | xargs ls -l`). The `-p` flag enables parallel execution by forking multiple child processes, and `-n` controls how many input lines are batched per invocation.

⟨*function* `usage` *(xargs.c)* 125a⟩≡ (234b)

```
void
usage(void)
{
    fprint(STDERR, "usage: xargs [ -n lines ] [ -p procs ] args ...\n");
    exits("usage");
}
```

⟨*function* `main` *(xargs.c)* 125b⟩≡ (234b)

```
void
main(int argc, char **argv)
{
    int lines = 10;
    int procs = 1;
    int i, j, run;
    char **nargv, **args, **p;
    static Biobuf bp;

    ARGBEGIN {
    case 'n': lines = atoi(EARGF(usage())); break;
    case 'p': procs = atoi(EARGF(usage())); break;
    default: usage();
    } ARGEND;

    if(argc < 1)
        usage();
    // else

    nargv = malloc(sizeof(char *) * (argc + lines + 1));
    if(nargv == nil)
        sysfatal("malloc: %r");
    memcpy(nargv, argv, sizeof(char *) * argc);
    args = nargv + argc;

    if(Binit(&bp, STDIN, OREAD) < 0)
        sysfatal("Binit: %r");

    //PAD: Blethal(&bp, nil); only in 9front
    atexit(dowait);

    for(j = 0, run = 1; run; j++){
        if(j >= procs)
```

```
            waitpid();
        memset(args, 0, sizeof(char *) * (lines + 1));
        for(i = 0; i < lines; i++)
            if((args[i] = Brdstr(&bp, '\n', 1)) == nil){
                if(i == 0)
                    exits(nil);
                run = 0;
                break;
            }

        switch(fork()){
        case ERROR_NEG1:
            sysfatal("fork: %r");
        // child case
        case 0:
            exec(*nargv, nargv);
            // here if ??
            if(**nargv != '/' && strncmp(*nargv, "./", 2) != 0 &&
                    strncmp(*nargv, "../", 3) != 0){
                *nargv = smprint("/bin/%s", *nargv);
                // try again
                exec(*nargv, nargv);
            }
            sysfatal("exec: %r");
        }
        // else, parent
        for(p = args; *p; p++)
            free(*p);
    }
    exits(nil);
}
```

⟨*function* `dowait` *(xargs.c)* 126⟩≡ (234b)

```
void
dowait(void)
{
    while(waitpid() != ERROR_NEG1)
        ;
}
```

Chapter 12

Bytes

This chapter covers utilities that work at the byte level: hex dumpers, the byte-level copy tool `dd`, and `split` for breaking large files into pieces.

The four tools in this chapter exist because text-oriented utilities like `cat` and `grep` become useless the moment you point them at a binary file: control bytes corrupt the terminal, multi-byte UTF-8 sequences get split, and there is no notion of a "line" to count. The byte-level tools fill this gap with four different views of the same raw bytes:

```
tool    role                        canonical example output
-----   -------------------------   --------------------------
xd      hex+ASCII side-by-side      0000000  48 65 6c 6c 6f
                                            20 77 6f 72 6c 64
                                            0a Hello world.
od      octal/hex/decimal/char      0000000   062510 066154
                                            032157 067127
dd      copy bytes with optional    dd if=/dev/zero bs=1k
        block size, count, offset   count=4 of=/tmp/foo
split   break a file into named     split -n 1000 huge.log
        pieces                      -> xaa xab xac ...
```

Each tool is more or less the simplest possible thing it can be. `xd` and `od` are read-loops with a hex-formatted `printf`; `dd` is a read-into-buffer / write-from-buffer loop with explicit block sizes (its arcane `if=/of=/bs=/count=` syntax is a relic of its 1970s job, copying between magnetic tapes whose blocks had to match exactly); `split` reads lines and rotates output files when the count threshold is reached. None has any internal state worth diagramming. The interesting one is `split`'s `-e` regex mode, which lets you split a log file every time a pattern appears—this is the byte-level cousin of `csplit` from GNU coreutils, and the only `split` feature that goes beyond "copy N lines, then close and open the next file."

Both `xd` and `od` read bytes in fixed-size chunks (typically 16 bytes per display line) and emit three aligned fields per line: a running byte offset on the left, the 16 bytes rendered in the chosen radix in the middle, and, for `xd`, the same 16 bytes as printable ASCII on the right (dots for non-printables). The alignment is what makes the output useful for eyeballing binary structures—bit-field boundaries, magic numbers, length-prefixed strings—so the formatting is fixed rather than content-dependent:

```
offset      hex bytes (2 chars + space = 3 per byte)          ascii
--------   -----------------------------------------------   ----------------
0000000    7f 45 4c 46  02 01 01 00  00 00 00 00  00 00 00 00  .ELF............
0000010    02 00 3e 00  01 00 00 00  b0 04 40 00  00 00 00 00  ..>...........@.
0000020    40 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  @...............
```

```
column:    0          11         22         33         44  48             64
           |          |          |          |          |   |              |
           offset     byte 0     byte 4     byte 8     byte 12             ascii end
           (7 chars)  (12 chars) (12 chars) (12 chars) (12 chars, no sep)  (16 chars)
```

The split of 16 bytes into 4 groups of 4 with an extra space between groups is pure ergonomics: 32-bit words become visible at a glance, which is the most common thing one is actually hunting for in a hex dump (pointers, lengths, opcodes). `od` by default uses *octal* 16-bit words rather than hex bytes—same skeleton, different field width and radix—which is a historical relic of the PDP-11 where machine instructions were 16 bits and octal was the native way to read them. Modern users almost always run `od -c` or `od -x` or `od -A x` to get something closer to `xd`. Either tool's *source* is dominated by the loop that pads and aligns these fields; the actual "dumping" is a trivial loop over `read` buffers.

12.1 xd

12.2 od

12.3 dd

12.4 split

`split` divides a file into pieces of `n` lines each (default 1000), naming them `xaa`, `xab`, etc. The `-e` flag enables regex-based splitting: a new file starts whenever a line matches the pattern, and if the regex contains a capture group, that group becomes the output file name.

⟨*function* `usage` *(split.c)* 128a⟩≡ (192a)
```
void
usage(void)
{
    fprint(STDERR, "usage: split [-n num] [-e exp] [-f stem] [-s suff] [-x] [-i] [file]\n");
    exits("usage");
}
```

⟨`main` *(split.c) locals* 128b⟩≡ (129g) 128c▷
```
int n = 1000;
```

⟨`main` *(split.c) locals* 128c⟩+≡ (129g) ◁128b 129d▷
```
// ??
bool xflag = false;
// ??
bool iflag = false;
```

⟨`globals split.c` 128d⟩≡ (192a) 128e▷
```
char    *stem = "x";
```

⟨`globals split.c` 128e⟩+≡ (192a) ◁128d 129b▷
```
char    *suffix = "";
```

⟨main(*split.c*) *switch flag character cases* 129a⟩≡ (129g)
```
case 'l':
case 'n':
    n=atoi(EARGF(usage()));
    break;
case 'e':
    pattern = strdup(EARGF(usage()));
    break;
case 'f':
    stem = strdup(EARGF(usage()));
    break;
case 's':
    suffix = strdup(EARGF(usage()));
    break;
case 'x':
    xflag = true;
    break;
case 'i':
    iflag = true;
    break;
```

⟨globals split.c 129b⟩+≡ (192a) ◁128e 129c▷
```
char    digit[] = "0123456789";
char    suff[] = "aa";
char    name[200];
```

⟨globals split.c 129c⟩+≡ (192a) ◁129b 131a▷
```
Biobuf  bout;
Biobuf  *output = &bout;
```

⟨main(*split.c*) *locals* 129d⟩+≡ (129g) ◁128c 129e▷
```
Biobuf bin;
Biobuf *b = &bin;
```

⟨main(*split.c*) *locals* 129e⟩+≡ (129g) ◁129d 129f▷
```
char *line;
```

⟨main(*split.c*) *locals* 129f⟩+≡ (129g) ◁129e 131b▷
```
char buf[256];
```

⟨*function* main(*split.c*) 129g⟩≡ (192a)
```
void
main(int argc, char *argv[])
{
    ⟨main(split.c) locals 128b⟩

    ARGBEGIN {
    ⟨main(split.c) switch flag character cases 129a⟩
    default:
        usage();
        break;

    } ARGEND;

    if(argc < 0 || argc > 1)
        usage();

    if(argc != 0) {
        b = Bopen(argv[0], OREAD);
        if(b == nil) {
```

```
            fprint(STDERR, "split: can't open %s: %r\n", argv[0]);
            exits("open");
        }
    } else
        Binit(b, STDIN, OREAD);

    ⟨main() (split.c) if pattern 131c⟩
    else {
        int linecnt = n;

        while((line=Brdline(b,'\n')) != 0) {
            if(++linecnt > n) {
                nextfile();
                linecnt = 1;
            }
            Bwrite(output, line, Blinelen(b));
        }

        /*
         * in case we didn't end with a newline, tack whatever's
         * left onto the last file
         */
        while((n = Bread(b, buf, sizeof(buf))) > 0)
            Bwrite(output, buf, n);
    }
    if(b != nil)
        Bterm(b);
    exits(nil);
}
```

Basic usage

⟨*function* nextfile *(split.c)* 130a⟩≡ (192a)

```
bool
nextfile(void)
{
    static bool canopen = true;

    if(suff[0] > 'z') {
        if(canopen)
            fprint(STDERR, "split: file %szz not split\n",stem);
        canopen = false;
    } else {
        snprint(name, sizeof name, "%s%s", stem, suff);
        if(++suff[1] > 'z')
            suff[1] = 'a', ++suff[0];
        openf();
    }
    return canopen;
}
```

⟨*function* openf *(split.c)* 130b⟩≡ (192a)

```
void
openf(void)
{
    static fdt fd = 0;

    Bflush(output);
    Bterm(output);
```

```
    if(fd > 0)
        close(fd);
    fd = create(name,OWRITE,0666);
    if(fd < 0) {
        fprint(STDERR, "grep: can't create %s: %r\n", name);
        exits("create");
    }
    Binit(output, fd, OWRITE);
}
```

Advanced usage

⟨globals split.c 131a⟩+≡ (192a) ◁129c

```
char *pattern = nil;
```

⟨main*(split.c) locals* 131b⟩+≡ (129g) ◁129f

```
Reprog *exp;
```

⟨main()*(split.c) if* pattern 131c⟩≡ (129g)

```
if(pattern) {
    Resub match[2];

    if(!(exp = regcomp(iflag? fold(pattern, strlen(pattern)):
        pattern)))
        badexp();
    memset(match, 0, sizeof match);
    matchfile(match);
    while((line=Brdline(b,'\n')) != 0) {
        memset(match, 0, sizeof match);
        line[Blinelen(b)-1] = 0;
        if(regexec(exp, iflag? fold(line, Blinelen(b)-1): line,
            match, 2)) {
            if(matchfile(match) && xflag)
                continue;
        } else if(output == 0)
            nextfile(); /* at most once */
        Bwrite(output, line, Blinelen(b)-1);
        Bputc(output, '\n');
    }
}
```

⟨*function* badexp*(split.c)* 131d⟩≡ (192a)

```
void
badexp(void)
{
    fprint(STDERR, "split: bad regular expression\n");
    exits("bad regular expression");
}
```

⟨*function* fold*(split.c)* 131e⟩≡ (192a)

```
char *
fold(char *s, int n)
{
    static char *fline;
    static int linesize = 0;
    char *t;

    if(linesize < n+1){
        fline = realloc(fline,n+1);
```

```
        linesize = n+1;
    }
    for(t=fline; *t++ = tolower(*s++); )
        continue;
        /* we assume the 'A'-'Z' only appear as themselves
         * in a utf encoding.
         */
    return fline;
}
```

⟨*function* `matchfile` *(split.c)* 132⟩≡ (192a)

```
int
matchfile(Resub *match)
{
    if(match[1].s.sp) {
        int len = match[1].e.ep - match[1].s.sp;

        strncpy(name, match[1].s.sp, len);
        strcpy(name+len, suffix);
        openf();
        return 1;
    }
    return nextfile();
}
```

Chapter 13

Calculators

This chapter covers the calculator utilities: `hoc` for floating-point arithmetic, `dc` for arbitrary-precision computation with a stack-based (reverse Polish) interface, and `bc` which provides a C-like syntax that compiles down to `dc`. The `dc`/`bc` pair is one of the oldest language implementations in UNIX, predating even the C compiler.

13.1 hoc

13.2 dc

13.2.1 Introduction

`dc` ("desk calculator") is an arbitrary-precision calculator using reverse Polish notation: operands are pushed onto a stack, and operators pop their arguments and push the result. The implementation is essentially a stack machine interpreter with its own memory management, where numbers are stored as variable-length byte strings in `Blk` structures.

A worked example clarifies what RPN "feels like" for someone who has only used infix calculators. The expression `(3  4) * 5+` becomes the `dc` input `3 4  5 * p+`, where `p` prints the top of the stack:

```
input    stack (top on right)   action
------   --------------------   -------------------------
3        [3]                    push 3
4        [3, 4]                 push 4
+        [7]                    pop 4, pop 3, push 3+4
5        [7, 5]                 push 5
*        [35]                   pop 5, pop 7, push 7*5
p        [35]                   print top: 35
```

Two things fall out. First, there are *no parentheses ever*—the operator order encodes the parse tree directly, which is why the `commnds()` inner loop in `dc` can be a single switch with no recursive descent. Second, the same input syntax doubles as the bytecode for a stack machine: a `dc` script is, quite literally, the instructions of a virtual machine, and when `bc` (the algebraic frontend) compiles `(34)*5+` it emits exactly the byte sequence above. The `bc`/`dc` split is one of the earliest examples of the "compile to a stack VM" pattern that later showed up in the JVM, .NET CLR, Lua, Python, Forth, and PostScript.

`dc` sits in a long lineage of stack-based calculator languages, each occupying a slightly different niche:

```
year   tool             notes
----   --------------   ---------------------------------
1968   HP calculators   the original Reverse Polish hardware
```

		(HP-35, HP-41, HP-48, HP-50g)
1970	Forth	stack-based programming language
1971	dc	on PDP-11 \unix; predates the C compiler! original by Lorinda Cherry
1971	bc	algebraic preprocessor for dc (Robert Morris and Lorinda Cherry)
1982	PostScript	stack-based page description / printer language (heavily influenced by Forth)
1990s	Joy, Cat, Factor	‘‘concatenative’’ programming languages---direct descendants of Forth and PostScript
1996	Java JVM	operand stack instead of registers (same idea, much bigger VM)

The historical importance of `dc` is hard to overstate: it was the *first* program written for UNIX on the PDP-11, written even before `ed` and well before the C compiler existed (so the original was in assembler). The choice of RPN over algebraic notation was pragmatic—the parser fits in a switch statement—but it also turned `dc` into the direct ancestor of every stack-VM language that followed. `bc` is built *on top of* `dc` (it is just a parser that emits `dc` commands), making the `bc`/`dc` pair one of the earliest worked examples of the “compiler-targeting-a-bytecode-interpreter” architecture that now powers most scripting languages.

13.2.2 Data structures

A `Blk` is `dc`’s fundamental data type: a resizable byte buffer with separate read and write cursors. Numbers are stored in these buffers as sequences of base-100 digits (each byte holds a digit from 0 to 99), with the scale factor appended as the last byte. The extensive macro API (`sgetc`, `sputc`, `sbackc`, etc.) turns `Blk` into a bidirectional stream, allowing `dc` to process digits from either end. The `Wblk` is the same concept at a higher level: a resizable array of `Blk` pointers, used for the operand stack. `Sym` nodes form a free list for named variables (registers `a` through `z`).

⟨*struct* `Blk` *(dc.c)* 134a⟩≡ (457)
```
struct  Blk
{
    char    *rd;
    char    *wt;
    char    *beg;
    char    *last;
};
```

⟨*macros on* `Blk dc.c` 134b⟩≡ (457)
```
#define length(p)   ((p)->wt-(p)->beg)
#define rewind(p)   (p)->rd=(p)->beg
#define create_(p)   (p)->rd = (p)->wt = (p)->beg
#define fsfile(p)   (p)->rd = (p)->wt
#define truncate(p) (p)->wt = (p)->rd
#define sfeof(p)    (((p)->rd==(p)->wt)?1:0)
#define sfbeg(p)    (((p)->rd==(p)->beg)?1:0)
#define sungetc(p,c)    *(--(p)->rd)=c
#define sgetc(p)    (((p)->rd==(p)->wt)?-1:*(p)->rd++)
#define skipc(p)    {if((p)->rd<(p)->wt)(p)->rd++;}
#define slookc(p)   (((p)->rd==(p)->wt)?-1:*(p)->rd)
#define sbackc(p)   (((p)->rd==(p)->beg)?-1:*(--(p)->rd))
#define backc(p)    {if((p)->rd>(p)->beg) --(p)->rd;}
#define sputc(p,c)  {if((p)->wt==(p)->last)more(p);\
```

```
                *(p)->wt++ = c; }
#define salterc(p,c)    {if((p)->rd==(p)->last)more(p);\
                *(p)->rd++ = c;\
                if((p)->rd>(p)->wt)(p)->wt=(p)->rd;}
#define sunputc(p)  (*((p)->rd = --(p)->wt))
#define sclobber(p) ((p)->rd = --(p)->wt)
#define zero(p)     for(pp=(p)->beg;pp<(p)->last;)\
                *pp++='\0'
```

⟨*struct* Wblk*(dc.c)* 135a⟩≡ (457)

```
struct  Wblk
{
    Blk **rdw;
    Blk **wtw;
    Blk **begw;
    Blk **lastw;
};
```

⟨*struct* Sym*(dc.c)* 135b⟩≡ (457)

```
struct  Sym
{
    Sym *next;
    Blk *val;
};
```

13.2.3 main() and init()

⟨globals dc.c 135c⟩≡ (457) 135f▷

```
Biobuf  bin;
Biobuf  bout;
```

⟨*function* main*(dc.c)* 135d⟩≡ (457)

```
void
main(int argc, char *argv[])
{
    Binit(&bin, STDIN, OREAD);
    Binit(&bout, STDOUT, OWRITE);
    init(argc,argv);
    commnds();
    exits(nil);
}
```

⟨global flags dc.c 135e⟩≡ (457)

```
bool dbg;
```

⟨globals dc.c 135f⟩+≡ (457) ◁135c 456c▷

```
Biobuf *curfile;
```

⟨*function* init*(dc.c)* 135g⟩≡ (457)

```
void
init(int argc, char *argv[])
{
    Dir *d;
    Sym *sp;

    ARGBEGIN {
    default:
        dbg = true;
        break;
```

```
    } ARGEND
    ifile = 1;
    curfile = &bin;
    if(*argv){
        d = dirstat(*argv);
        if(d == nil) {
            fprint(STDERR, "dc: can't open file %s\n", *argv);
            exits("open");
        }
        if(d->mode & DMDIR) {
            fprint(STDERR, "dc: file %s is a directory\n", *argv);
            exits("open");
        }
        free(d);
        if((curfile = Bopen(*argv, OREAD)) == nil) {
            fprint(STDERR,"dc: can't open file %s\n", *argv);
            exits("open");
        }
    }
//  dummy = malloc(0);  /* prepare for garbage-collection */
    ⟨init() initialize globals(dc.c) 136⟩
}
```

⟨init() *initialize globals(dc.c)* 136⟩≡ (135g)

```
scalptr = salloc(1);
sputc(scalptr,0);
basptr = salloc(1);
sputc(basptr,10);
obase=10;
logten=log2_(10L);
ll=70;
fw=1;
fw1=0;
tenptr = salloc(1);
sputc(tenptr,10);
obase=10;
inbas = salloc(1);
sputc(inbas,10);
sqtemp = salloc(1);
sputc(sqtemp,2);
chptr = salloc(0);
strptr = salloc(0);
divxyz = salloc(0);
stkbeg = stkptr = &stack[0];
stkend = &stack[STKSZ];
stkerr = 0;
readptr = &readstk[0];
k=0;

sp = sptr = &symlst[0];
while(sptr < &symlst[TBLSZ-1]) {
    sptr->next = ++sp;
    sptr++;
}
sptr->next=0;
sfree = &symlst[0];
```

13.2.4 commnds()

The main command loop reads one character at a time: digits (including hex A-F) are accumulated into a number and pushed onto the stack, while operators are dispatched through a large switch statement. This is the heart of the reverse Polish interpreter—no parsing is needed because the notation is unambiguous.

⟨commnds() *locals (dc.c)* 137a⟩≡ (137c) 137b▷
```
int c;
Blk *p;
```

⟨commnds() *locals (dc.c)* 137b⟩+≡ (137c) ◁137a
```
Blk *q, **ptr, *s, *t;
long l;
Sym *sp;
int sk, sk1, sk2, sign, n, d;
```

⟨*function* commnds*(dc.c)* 137c⟩≡ (457)
```
void
commnds(void)
{
    ⟨commnds() locals (dc.c) 137a⟩

    while(true) {
        Bflush(&bout);
        if(((c = readc())>='0' && c <= '9') ||
            (c>='A' && c <='F') || c == '.') {
            unreadc(c);
            p = readin();
            pushp(p);
            continue;
        }
        switch(c) {
        ⟨commnds() switch c cases (dc.c) 141d⟩
        default:
            Bprint(&bout,"%o is unimplemented\n",c);
        }
    }
}
```

13.2.5 Reading characters

⟨*function* readc*(dc.c)* 137d⟩≡ (457)
```
int
readc(void)
{
loop:
    if((readptr != &readstk[0]) && (*readptr != 0)) {
        if(sfeof(*readptr) == 0)
            return(lastchar = sgetc(*readptr));
        release(*readptr);
        readptr--;
        goto loop;
    }
    lastchar = Bgetc(curfile);
    if(lastchar != -1)
        return(lastchar);
    // else
    if(readptr != &readptr[0]) {
        readptr--;
```

```
        if(*readptr == 0)
            curfile = &bin;
        goto loop;
    }
    if(curfile != &bin) {
        Bterm(curfile);
        curfile = &bin;
        goto loop;
    }
    exits(nil);
    return 0;   /* shut up ken */
}
```

⟨*function* `unreadc` *(dc.c)* 138a⟩≡ (457)

```
void
unreadc(char c)
{

    if((readptr != &readstk[0]) && (*readptr != 0)) {
        sungetc(*readptr,c);
    } else
        Bungetc(curfile);
    return;
}
```

⟨*function* `readin` *(dc.c)* 138b⟩≡ (457)

```
Blk*
readin(void)
{
    Blk *p, *q;
    int dp, dpct, c;

    dp = dpct=0;
    p = salloc(0);
    for(;;){
        c = readc();
        switch(c) {
        case '.':
            if(dp != 0)
                goto gotnum;
            dp++;
            continue;
        case '\\':
            readc();
            continue;
        default:
            if(c >= 'A' && c <= 'F')
                c = c - 'A' + 10;
            else
            if(c >= '0' && c <= '9')
                c -= '0';
            else
                goto gotnum;
            if(dp != 0) {
                if(dpct >= 99)
                    continue;
                dpct++;
            }
            create_(chptr);
            if(c != 0)
```

```
                sputc(chptr,c);
            q = mult(p,inbas);
            release(p);
            p = add(chptr,q);
            release(q);
        }
    }
gotnum:
    unreadc(c);
    if(dp == 0) {
        sputc(p,0);
        return(p);
    } else {
        q = scale(p,dpct);
        return(q);
    }
}
```

13.2.6 Stack operations

⟨*function* pushp*(dc.c)* 139a⟩≡ (457)

```
void
pushp(Blk *p)
{
    if(stkptr == stkend) {
        Bprint(&bout,"out of stack space\n");
        return;
    }
    stkerr=0;
    *++stkptr = p;
    return;
}
```

⟨*function* pop*(dc.c)* 139b⟩≡ (457)

```
Blk*
pop(void)
{
    if(stkptr == stack) {
        stkerr=1;
        return nil;
    }
    return(*stkptr--);
}
```

13.2.7 Memory management

⟨*function* salloc*(dc.c)* 139c⟩≡ (457)

```
Blk*
salloc(int size)
{
    Blk *hdr;
    char *ptr;

    all++;
    lall++;
    if(all - rel > active)
        active = all - rel;
    nbytes += size;
```

```
    lbytes += size;
    if(nbytes >maxsize)
        maxsize = nbytes;
    if(size > longest)
        longest = size;
    ptr = malloc((unsigned)size);
    if(ptr == 0){
        garbage("salloc");
        if((ptr = malloc((unsigned)size)) == 0)
            ospace("salloc");
    }
    if((hdr = hfree) == 0)
        hdr = morehd();
    hfree = (Blk *)hdr->rd;
    hdr->rd = hdr->wt = hdr->beg = ptr;
    hdr->last = ptr+size;
    return(hdr);
}
```

⟨*function* morehd*(dc.c)* 140a⟩≡ (457)

```
Blk*
morehd(void)
{
    Blk *h, *kk;

    headmor++;
    nbytes += HEADSZ;
    hfree = h = (Blk *)malloc(HEADSZ);
    if(hfree == 0) {
        garbage("morehd");
        if((hfree = h = (Blk*)malloc(HEADSZ)) == 0)
            ospace("headers");
    }
    kk = h;
    while(h<hfree+(HEADSZ/BLK))
        (h++)->rd = (char*)++kk;
    (h-1)->rd=0;
    return(hfree);
}
```

⟨*function* copy*(dc.c)* 140b⟩≡ (457)

```
Blk*
copy(Blk *hptr, int size)
{
    Blk *hdr;
    unsigned sz;
    char *ptr;

    all++;
    lall++;
    lcopy++;
    nbytes += size;
    lbytes += size;
    if(size > longest)
        longest = size;
    if(size > maxsize)
        maxsize = size;
    sz = length(hptr);
    ptr = malloc(size);
    if(ptr == 0) {
```

```
        Bprint(&bout,"copy size %d\n",size);
        ospace("copy");
    }
    memmove(ptr, hptr->beg, sz);
    if (size-sz > 0)
        memset(ptr+sz, 0, size-sz);
    if((hdr = hfree) == 0)
        hdr = morehd();
    hfree = (Blk *)hdr->rd;
    hdr->rd = hdr->beg = ptr;
    hdr->last = ptr+size;
    hdr->wt = ptr+sz;
    ptr = hdr->wt;
    while(ptr<hdr->last)
        *ptr++ = '\0';
    return(hdr);
}
```

⟨*function* ospace*(dc.c)* 141a⟩≡ (457)

```
void
ospace(char *s)
{
    Bprint(&bout,"out of space: %s\n",s);
    Bprint(&bout,"all %ld rel %ld headmor %ld\n",all,rel,headmor);
    Bprint(&bout,"nbytes %ld\n",nbytes);
    sdump("stk",*stkptr);
    abort();
}
```

⟨*function* garbage*(dc.c)* 141b⟩≡ (457)

```
void
garbage(char *s)
{
    USED(s);
}
```

⟨*function* release*(dc.c)* 141c⟩≡ (457)

```
void
release(Blk *p)
{
    rel++;
    lrel++;
    nbytes -= p->last - p->beg;
    p->rd = (char*)hfree;
    hfree = p;
    free(p->beg);
}
```

13.2.8 Basic commands

⟨commnds() *switch* c *cases (dc.c)* 141d⟩≡ (137c) 142a▷

```
case ' ':
case '\t':
case '\n':
case -1:
    continue;
```

Basic arithmetic

⟨commnds() *switch* c *cases (dc.c)* 142a⟩+≡ (137c) ◁141d 142c▷

```
case '+':
    if(eqk() != 0)
        continue;
    binop('+');
    continue;
```

⟨*function* binop*(dc.c)* 142b⟩≡ (457)

```
void
binop(char c)
{
    Blk *r;

    r = 0;
    switch(c) {
    case '+':
        r = add(arg1,arg2);
        break;
    case '*':
        r = mult(arg1,arg2);
        break;
    case '/':
        r = div_(arg1,arg2);
        break;
    }
    release(arg1);
    release(arg2);
    sputc(r,savk);
    pushp(r);
}
```

⟨commnds() *switch* c *cases (dc.c)* 142c⟩+≡ (137c) ◁142a 142e▷

```
case '-':
    subt();
    continue;
```

⟨*function* subt*(dc.c)* 142d⟩≡ (457)

```
int
subt(void)
{
    arg1=pop();
    EMPTYS;
    savk = sunputc(arg1);
    chsign(arg1);
    sputc(arg1,savk);
    pushp(arg1);
    if(eqk() != 0)
        return(1);
    binop('+');
    return(0);
}
```

⟨commnds() *switch* c *cases (dc.c)* 142e⟩+≡ (137c) ◁142c 143a▷

```
case '*':
    arg1 = pop();
    EMPTY;
    arg2 = pop();
    EMPTYR(arg1);
    sk1 = sunputc(arg1);
```

```
    sk2 = sunputc(arg2);
    savk = sk1+sk2;
    binop('*');
    p = pop();
    if(savk>k && savk>sk1 && savk>sk2) {
        sclobber(p);
        sk = sk1;
        if(sk<sk2)
            sk = sk2;
        if(sk<k)
            sk = k;
        p = removc(p,savk-sk);
        savk = sk;
        sputc(p,savk);
    }
    pushp(p);
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 143a⟩+≡ (137c) ◁142e 143b▷

```
  case '/':
  casediv:
    if(dscale() != 0)
        continue;
    binop('/');
    if(irem != 0)
        release(irem);
    release(rem);
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 143b⟩+≡ (137c) ◁143a 143c▷

```
  case '%':
    if(dscale() != 0)
        continue;
    binop('/');
    p = pop();
    release(p);
    if(irem == 0) {
        sputc(rem,skr+k);
        pushp(rem);
        continue;
    }
    p = add0(rem,skd-(skr+k));
    q = add(p,irem);
    release(p);
    release(irem);
    sputc(q,skd);
    pushp(q);
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 143c⟩+≡ (137c) ◁143b 144a▷

```
  case '_':
    p = readin();
    savk = sunputc(p);
    chsign(p);
    sputc(p,savk);
    pushp(p);
    continue;
```

Advanced arithmetic

⟨commnds() *switch* c *cases (dc.c)* 144a⟩+≡ (137c) ◁143c 144b▷

```
  case '^':
      neg = 0;
      arg1 = pop();
      EMPTY;
      if(sunputc(arg1) != 0)
          error("exp not an integer\n");
      arg2 = pop();
      EMPTYR(arg1);
      if(sfbeg(arg1) == 0 && sbackc(arg1)<0) {
          neg++;
          chsign(arg1);
      }
      if(length(arg1)>=3) {
          error("exp too big\n");
      }
      savk = sunputc(arg2);
      p = dcexp(arg2,arg1);
      release(arg2);
      rewind(arg1);
      c = sgetc(arg1);
      if(c == -1)
          c = 0;
      else
      if(sfeof(arg1) == 0)
          c = sgetc(arg1)*100 + c;
      d = c*savk;
      release(arg1);
  /*  if(neg == 0) {      removed to fix -exp bug*/
          if(k>=savk)
              n = k;
          else
              n = savk;
          if(n<d) {
              q = removc(p,d-n);
              sputc(q,n);
              pushp(q);
          } else {
              sputc(p,d);
              pushp(p);
          }
  /*  } else { this is disaster for exp <-127 */
  /*      sputc(p,d);     */
  /*      pushp(p);       */
  /*  }               */
      if(neg == 0)
          continue;
      p = pop();
      q = salloc(2);
      sputc(q,1);
      sputc(q,0);
      pushp(q);
      pushp(p);
      goto casediv;
```

Misc

⟨commnds() *switch* c *cases (dc.c)* 144b⟩+≡ (137c) ◁144a 145a▷

```
  case '<':
  case '>':
  case '=':
      if(cond(c) == 1)
          goto execute;
      continue;
```

⟨commnds() *switch* c *cases (dc.c)* 145a⟩+≡ (137c) ◁144b 145b▷

```
  case '[':
      n = 0;
      p = salloc(0);
      for(;;) {
          if((c = readc()) == ']') {
              if(n == 0)
                  break;
              n--;
          }
          sputc(p,c);
          if(c == '[')
              n++;
      }
      pushp(p);
      continue;
```

⟨commnds() *switch* c *cases (dc.c)* 145b⟩+≡ (137c) ◁145a 145c▷

```
  case 'q':
      if(readptr <= &readstk[1])
          exits(nil);
      // else
      if(*readptr != 0)
          release(*readptr);
      readptr--;
      if(*readptr != 0)
          release(*readptr);
      readptr--;
      continue;
```

⟨commnds() *switch* c *cases (dc.c)* 145c⟩+≡ (137c) ◁145b 145d▷

```
  case 'p':
      if(stkptr == &stack[0])
          Bprint(&bout,"empty stack\n");
      else {
          dcprint(*stkptr);
      }
      continue;
```

⟨commnds() *switch* c *cases (dc.c)* 145d⟩+≡ (137c) ◁145c 147a▷

```
  case 'P':
      p = pop();
      EMPTY;
      sputc(p,0);
      Bprint(&bout,"%s",p->beg);
      release(p);
      continue;
```

⟨*function* dcprint *(dc.c)* 145e⟩≡ (457)

```
  void
  dcprint(Blk *hptr)
  {
      Blk *p, *q, *dec;
```

```
int dig, dout, ct, sc;

rewind(hptr);
while(sfeof(hptr) == 0) {
    if(sgetc(hptr)>99) {
        rewind(hptr);
        while(sfeof(hptr) == 0) {
            Bprint(&bout,"%c",sgetc(hptr));
        }
        Bprint(&bout,"\n");
        return;
    }
}
fsfile(hptr);
sc = sbackc(hptr);
if(sfbeg(hptr) != 0) {
    Bprint(&bout,"0\n");
    return;
}
count = ll;
p = copy(hptr,length(hptr));
sclobber(p);
fsfile(p);
if(sbackc(p)<0) {
    chsign(p);
    OUTC('-');
}
if((obase == 0) || (obase == -1)) {
    oneot(p,sc,'d');
    return;
}
if(obase == 1) {
    oneot(p,sc,'1');
    return;
}
if(obase == 10) {
    tenot(p,sc);
    return;
}
/* sleazy hack to scale top of stack - divide by 1 */
pushp(p);
sputc(p, sc);
p=salloc(0);
create_(p);
sputc(p, 1);
sputc(p, 0);
pushp(p);
if(dscale() != 0)
    return;
p = div_(arg1, arg2);
release(arg1);
release(arg2);
sc = savk;

create_(strptr);
dig = logten*sc;
dout = ((dig/10) + dig) / logo;
dec = getdec(p,sc);
p = removc(p,sc);
while(length(p) != 0) {
```

```
        q = div_(p,basptr);
        release(p);
        p = q;
        (*outdit)(rem,0);
    }
    release(p);
    fsfile(strptr);
    while(sfbeg(strptr) == 0)
        OUTC(sbackc(strptr));
    if(sc == 0) {
        release(dec);
        Bprint(&bout,"\n");
        return;
    }
    create_(strptr);
    OUTC('.');
    ct=0;
    do {
        q = mult(basptr,dec);
        release(dec);
        dec = getdec(q,sc);
        p = removc(q,sc);
        (*outdit)(p,1);
    } while(++ct < dout);
    release(dec);
    rewind(strptr);
    while(sfeof(strptr) == 0)
        OUTC(sgetc(strptr));
    Bprint(&bout,"\n");
}
```

13.2.9 Advanced commands

⟨commnds() *switch* c *cases (dc.c)* 147a⟩+≡ (137c) ◁145d 148a▷

```
case 'Y':
    sdump("stk",*stkptr);
    Bprint(&bout, "all %ld rel %ld headmor %ld\n",all,rel,headmor);
    Bprint(&bout, "nbytes %ld\n",nbytes);
    Bprint(&bout, "longest %ld active %ld maxsize %ld\n", longest,
        active, maxsize);
    Bprint(&bout, "new all %d rel %d copy %d more %d lbytes %d\n",
        lall, lrel, lcopy, lmore, lbytes);
    lall = lrel = lcopy = lmore = lbytes = 0;
    continue;
```

⟨*function* sdump*(dc.c)* 147b⟩≡ (457)

```
void
sdump(char *s1, Blk *hptr)
{
    char *p;

    if(hptr == nil) {
        Bprint(&bout, "%s no block\n", s1);
        return;
    }
    Bprint(&bout,"%s %lx rd %lx wt %lx beg %lx last %lx\n",
        s1,hptr,hptr->rd,hptr->wt,hptr->beg,hptr->last);
    p = hptr->beg;
    while(p < hptr->wt)
```

```
        Bprint(&bout,"%d ",*p++);
    Bprint(&bout,"\n");
}
```

⟨commnds() *switch* c *cases (dc.c)* 148a⟩+≡ (137c) ◁147a 148b▷

```
case 'v':
    p = pop();
    EMPTY;
    savk = sunputc(p);
    if(length(p) == 0) {
        sputc(p,savk);
        pushp(p);
        continue;
    }
    if(sbackc(p)<0) {
        error("sqrt of neg number\n");
    }
    if(k<savk)
        n = savk;
    else {
        n = k*2-savk;
        savk = k;
    }
    arg1 = add0(p,n);
    arg2 = dcsqrt(arg1);
    sputc(arg2,savk);
    pushp(arg2);
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 148b⟩+≡ (137c) ◁148a 148c▷

```
case 'z':
    p = salloc(2);
    n = stkptr - stkbeg;
    if(n >= 100) {
        sputc(p,n/100);
        n %= 100;
    }
    sputc(p,n);
    sputc(p,0);
    pushp(p);
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 148c⟩+≡ (137c) ◁148b 149a▷

```
case 'Z':
    p = pop();
    EMPTY;
    n = (length(p)-1)<<1;
    fsfile(p);
    backc(p);
    if(sfbeg(p) == 0) {
        if((c = sbackc(p))<0) {
            n -= 2;
            if(sfbeg(p) == 1)
                n++;
            else {
                if((c = sbackc(p)) == 0)
                    n++;
                else
                if(c > 90)
```

```
                    n--;
                }
            } else
            if(c < 10)
                n--;
        }
        release(p);
        q = salloc(1);
        if(n >= 100) {
            sputc(q,n%100);
            n /= 100;
        }
        sputc(q,n);
        sputc(q,0);
        pushp(q);
        continue;
```

⟨commnds() *switch* c *cases (dc.c)* 149a⟩+≡ (137c) ◁148c 149b▷

```
  case 'i':
        p = pop();
        EMPTY;
        p = scalint(p);
        release(inbas);
        inbas = p;
        continue;
```

⟨commnds() *switch* c *cases (dc.c)* 149b⟩+≡ (137c) ◁149a 149c▷

```
  case 'I':
        p = copy(inbas,length(inbas)+1);
        sputc(p,0);
        pushp(p);
        continue;
```

⟨commnds() *switch* c *cases (dc.c)* 149c⟩+≡ (137c) ◁149b 150a▷

```
  case 'o':
        p = pop();
        EMPTY;
        p = scalint(p);
        sign = 0;
        n = length(p);
        q = copy(p,n);
        fsfile(q);
        l = c = sbackc(q);
        if(n != 1) {
            if(c<0) {
                sign = 1;
                chsign(q);
                n = length(q);
                fsfile(q);
                l = c = sbackc(q);
            }
            if(n != 1) {
                while(sfbeg(q) == 0)
                    l = l*100+sbackc(q);
            }
        }
        logo = log2_(l);
        obase = l;
        release(basptr);
        if(sign == 1)
```

```
        obase = -1;
    basptr = p;
    outdit = bigot;
    if(n == 1 && sign == 0) {
        if(c <= 16) {
            outdit = hexot;
            fw = 1;
            fw1 = 0;
            ll = 70;
            release(q);
            continue;
        }
    }
    n = 0;
    if(sign == 1)
        n++;
    p = salloc(1);
    sputc(p,-1);
    t = add(p,q);
    n += length(t)*2;
    fsfile(t);
    if(sbackc(t)>9)
        n++;
    release(t);
    release(q);
    release(p);
    fw = n;
    fw1 = n-1;
    ll = 70;
    if(fw>=ll)
        continue;
    ll = (70/fw)*fw;
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 150a⟩+≡ (137c) ◁149c 150b▷

```
  case 'O':
    p = copy(basptr,length(basptr)+1);
    sputc(p,0);
    pushp(p);
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 150b⟩+≡ (137c) ◁150a 150c▷

```
  case 'k':
    p = pop();
    EMPTY;
    p = scalint(p);
    if(length(p)>1) {
        error("scale too big\n");
    }
    rewind(p);
    k = 0;
    if(!sfeof(p))
        k = sgetc(p);
    release(scalptr);
    scalptr = p;
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 150c⟩+≡ (137c) ◁150b 151a▷

```
  case 'K':
    p = copy(scalptr,length(scalptr)+1);
```

```
        sputc(p,0);
        pushp(p);
        continue;
```

⟨commnds() *switch* c *cases (dc.c)* 151a⟩+≡ (137c) ◁150c 151b▷

```
    case 'X':
        p = pop();
        EMPTY;
        fsfile(p);
        n = sbackc(p);
        release(p);
        p = salloc(2);
        sputc(p,n);
        sputc(p,0);
        pushp(p);
        continue;
```

⟨commnds() *switch* c *cases (dc.c)* 151b⟩+≡ (137c) ◁151a 151c▷

```
    case 'Q':
        p = pop();
        EMPTY;
        if(length(p)>2) {
            error("Q?\n");
        }
        rewind(p);
        if((c =  sgetc(p))<0) {
            error("neg Q\n");
        }
        release(p);
        while(c-- > 0) {
            if(readptr == &readstk[0]) {
                error("readstk?\n");
            }
            if(*readptr != 0)
                release(*readptr);
            readptr--;
        }
        continue;
```

⟨commnds() *switch* c *cases (dc.c)* 151c⟩+≡ (137c) ◁151b 151d▷

```
    case 'f':
        if(stkptr == &stack[0])
            Bprint(&bout,"empty stack\n");
        else {
            for(ptr = stkptr; ptr > &stack[0];) {
                dcprint(*ptr--);
            }
        }
        continue;
```

⟨commnds() *switch* c *cases (dc.c)* 151d⟩+≡ (137c) ◁151c 152a▷

```
    case 'd':
        if(stkptr == &stack[0]) {
            Bprint(&bout,"empty stack\n");
            continue;
        }
        q = *stkptr;
        n = length(q);
        p = copy(*stkptr,n);
        pushp(p);
        continue;
```

⟨commnds() *switch* c *cases (dc.c)* 152a⟩+≡ (137c) ◁151d 152b▷
```
case 'c':
    while(stkerr == 0) {
        p = pop();
        if(stkerr == 0)
            release(p);
    }
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 152b⟩+≡ (137c) ◁152a 152c▷
```
case 'S':
    if(stkptr == &stack[0]) {
        error("save: args\n");
    }
    c = getstk() & 0377;
    sptr = stable[c];
    sp = stable[c] = sfree;
    sfree = sfree->next;
    if(sfree == 0)
        goto sempty;
    sp->next = sptr;
    p = pop();
    EMPTY;
    if(c >= ARRAYST) {
        q = copy(p,length(p)+PTRSZ);
        for(n = 0;n < PTRSZ;n++) {
            sputc(q,0);
        }
        release(p);
        p = q;
    }
    sp->val = p;
    continue;
sempty:
    error("symbol table overflow\n");
```

⟨commnds() *switch* c *cases (dc.c)* 152c⟩+≡ (137c) ◁152b 153a▷
```
case 's':
    if(stkptr == &stack[0]) {
        error("save:args\n");
    }
    c = getstk() & 0377;
    sptr = stable[c];
    if(sptr != 0) {
        p = sptr->val;
        if(c >= ARRAYST) {
            rewind(p);
            while(sfeof(p) == 0)
                release(dcgetwd(p));
        }
        release(p);
    } else {
        sptr = stable[c] = sfree;
        sfree = sfree->next;
        if(sfree == 0)
            goto sempty;
        sptr->next = 0;
    }
    p = pop();
    sptr->val = p;
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 153a⟩+≡ (137c) ◁152c 153b▷

```
  case 'l':
      load();
      continue;
```

⟨commnds() *switch* c *cases (dc.c)* 153b⟩+≡ (137c) ◁153a 153c▷

```
  case 'L':
      c = getstk() & 0377;
      sptr = stable[c];
      if(sptr == 0) {
          error("L?\n");
      }
      stable[c] = sptr->next;
      sptr->next = sfree;
      sfree = sptr;
      p = sptr->val;
      if(c >= ARRAYST) {
          rewind(p);
          while(sfeof(p) == 0) {
              q = dcgetwd(p);
              if(q != 0)
                  release(q);
          }
      }
      pushp(p);
      continue;
```

⟨commnds() *switch* c *cases (dc.c)* 153c⟩+≡ (137c) ◁153b 154a▷

```
  case ':':
      p = pop();
      EMPTY;
      q = scalint(p);
      fsfile(q);
      c = 0;
      if((sfbeg(q) == 0) && ((c = sbackc(q))<0)) {
          error("neg index\n");
      }
      if(length(q)>2) {
          error("index too big\n");
      }
      if(sfbeg(q) == 0)
          c = c*100+sbackc(q);
      if(c >= MAXIND) {
          error("index too big\n");
      }
      release(q);
      n = getstk() & 0377;
      sptr = stable[n];
      if(sptr == 0) {
          sptr = stable[n] = sfree;
          sfree = sfree->next;
          if(sfree == 0)
              goto sempty;
          sptr->next = 0;
          p = salloc((c+PTRSZ)*PTRSZ);
          zero(p);
      } else {
          p = sptr->val;
          if(length(p)-PTRSZ < c*PTRSZ) {
              q = copy(p,(c+PTRSZ)*PTRSZ);
```

```
            release(p);
            p = q;
        }
    }
    seekc(p,c*PTRSZ);
    q = lookwd(p);
    if(q!=0)
        release(q);
    s = pop();
    EMPTY;
    salterwd(p, s);
    sptr->val = p;
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 154a⟩+≡ (137c) ◁153c 154b▷

```
  case ';':
    p = pop();
    EMPTY;
    q = scalint(p);
    fsfile(q);
    c = 0;
    if((sfbeg(q) == 0) && ((c = sbackc(q))<0)) {
        error("neg index\n");
    }
    if(length(q)>2) {
        error("index too big\n");
    }
    if(sfbeg(q) == 0)
        c = c*100+sbackc(q);
    if(c >= MAXIND) {
        error("index too big\n");
    }
    release(q);
    n = getstk() & 0377;
    sptr = stable[n];
    if(sptr != 0){
        p = sptr->val;
        if(length(p)-PTRSZ >= c*PTRSZ) {
            seekc(p,c*PTRSZ);
            s = dcgetwd(p);
            if(s != 0) {
                q = copy(s,length(s));
                pushp(q);
                continue;
            }
        }
    }
    q = salloc(1);  /*so uninitialized array elt prints as 0*/
    sputc(q, 0);
    pushp(q);
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 154b⟩+≡ (137c) ◁154a 155a▷

```
  case 'x':
  execute:
    p = pop();
    EMPTY;
    if((readptr != &readstk[0]) && (*readptr != 0)) {
        if((*readptr)->rd == (*readptr)->wt)
            release(*readptr);
```

```
        else {
            if(readptr++ == &readstk[RDSKSZ]) {
                error("nesting depth\n");
            }
        }
    } else
        readptr++;
    *readptr = p;
    if(p != 0)
        rewind(p);
    else {
        if((c = readc()) != '\n')
            unreadc(c);
    }
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 155a⟩+≡ (137c) ◁154b 155b▷

```
  case '?':
    if(++readptr == &readstk[RDSKSZ]) {
        error("nesting depth\n");
    }
    *readptr = 0;
    fsave = curfile;
    curfile = &bin;
    while((c = readc()) == '!')
        command();
    p = salloc(0);
    sputc(p,c);
    while((c = readc()) != '\n') {
        sputc(p,c);
        if(c == '\\')
            sputc(p,readc());
    }
    curfile = fsave;
    *readptr = p;
    continue;
```

⟨commnds() *switch* c *cases (dc.c)* 155b⟩+≡ (137c) ◁155a

```
  case '!':
    if(command() == 1)
        goto execute;
    continue;
```

⟨*function* command*(dc.c)* 155c⟩≡ (457)

```
  int
  command(void)
  {
    char line[100], *sl;
    int pid, p, c;

    switch(c = readc()) {
    case '<':
        return(cond(NL));
    case '>':
        return(cond(NG));
    case '=':
        return(cond(NE));
    default:
        sl = line;
        *sl++ = c;
```

```
        while((c = readc()) != '\n')
            *sl++ = c;
        *sl = 0;
        if((pid = fork()) == 0) {
            execl("/bin/rc","rc","-c",line,nil);
            exits("shell");
        }
        for(;;) {
            if((p = waitpid()) < 0)
                break;
            if(p== pid)
                break;
        }
        Bprint(&bout,"!\n");
        return(0);
    }
}
```

13.3 bc

`bc` is not an independent calculator—it is a compiler that translates C-like infix expressions into `dc` commands. The yacc grammar parses assignments, control flow (`if`, `while`, `for`), and function definitions, then emits `dc` instructions as strings via `bundle`. The output is piped directly to `dc` for execution, making `bc` a textbook example of a source-to-source translator.

⟨*globals bc.y* 156⟩≡ (476a)

```
Biobuf  *in;
Biobuf  bstdin;
Biobuf  bstdout;

char    cary[1000];
char*   cp = { cary };
char    string[1000];
char*   str = { string };
int     crs = 128;
int     rcrs = 128;     /* reset crs */
int     bindx = 0;
int     lev = 0;
int     ln;
char*   ttp;
char*   ss = "";
int     bstack[10] = { 0 };

char*   numb[15] =
{
        " 0", " 1", " 2", " 3", " 4", " 5",
        " 6", " 7", " 8", " 9", " 10", " 11",
        " 12", " 13", " 14"
};
char*   pre;
char*   post;

long    peekc = -1;
int     sargc;
int     ifile;
char**  sargv;

char    *funtab[] =
```

```
{
        "<1>","<2>","<3>","<4>","<5>",
        "<6>","<7>","<8>","<9>","<10>",
        "<11>","<12>","<13>","<14>","<15>",
        "<16>","<17>","<18>","<19>","<20>",
        "<21>","<22>","<23>","<24>","<25>",
        "<26>"
};
char    *atab[] =
{
        "<221>","<222>","<223>","<224>","<225>",
        "<226>","<227>","<228>","<229>","<230>",
        "<231>","<232>","<233>","<234>","<235>",
        "<236>","<237>","<238>","<239>","<240>",
        "<241>","<242>","<243>","<244>","<245>",
        "<246>"
};
char*   letr[26] =
{
        "a","b","c","d","e","f","g","h","i","j",
        "k","l","m","n","o","p","q","r","s","t",
        "u","v","w","x","y","z"
};
char*   dot = { "." };
char*   bspace[bsp_max];
char**  bsp_nxt = bspace;
```

⟨*global flags bc.y* 157a⟩≡ (476a)

```
// -d for debugging
bool     bdebug;
// -l <lib> library loading (/sys/lib/bclib by default)
bool     lflag;
// ??
bool     cflag;
// ??
bool     sflag;
```

13.3.1 main()

⟨*function* `main`*(bc.y)* 157b⟩≡ (476b)

```
void
main(int argc, char **argv)
{
        fdt p[2];

        while(argc > 1 && *argv[1] == '-') {
                switch(argv[1][1]) {
                case 'd':
                        bdebug = true;
                        break;
                case 'c':
                        cflag = true;
                        break;
                case 'l':
                        lflag = true;
                        break;
                case 's':
                        sflag = true;
                        break;
```

```
                default:
                        fprint(STDERR, "Usage: bc [-cdls] [file ...]\n");
                        exits("usage");
                }
                argc--;
                argv++;
        }
        if(lflag) {
                argv--;
                argc++;
                argv[1] = "/sys/lib/bclib";
        }
        if(cflag) {
                yyinit(argc, argv);
                for(;;)
                        yyparse();
                /* not reached */
        }
        pipe(p);
        if(fork() == 0) {
                dup(p[1], STDOUT);
                close(p[0]);
                close(p[1]);
                yyinit(argc, argv);
                for(;;)
                        yyparse();
        }
        // else, parent
        dup(p[0], STDIN);
        close(p[0]);
        close(p[1]);
        execl("/bin/dc", "dc", nil);
}
```

⟨*function* yyerror*(bc.y)* 158a⟩≡ (476b)

```
void
yyerror(char *s, ...)
{
        if(ifile > sargc)
                ss = "stdin";
        Bprint(&bstdout, "c[%s:%d %s]pc\n", ss, ln+1, s);
        Bflush(&bstdout);
        cp = cary;
        crs = rcrs;
        bindx = 0;
        lev = 0;
        bsp_nxt = &bspace[0];
}
```

13.3.2 Lexer

⟨*union directive bc.y* 158b⟩≡ (476b)

```
%union
{
        char*   cptr;
        int     cc;
}
```

⟨*token directives bc.y* 159a⟩≡ (476b)

```
%token  <cptr>  LETTER EQOP _AUTO DOT
%token  <cc>    DIGIT SQRT LENGTH _IF FFF EQ
%token  <cc>    _PRINT _WHILE _FOR NE LE GE INCR DECR
%token  <cc>    _RETURN _BREAK _DEFINE BASE OBASE SCALE
%token  <cc>    QSTR ERROR
```

⟨*function* `yyinit` *(bc.y)* 159b⟩≡ (476b)

```
void
yyinit(int argc, char **argv)
{
        Binit(&bstdout, 1, OWRITE);
        sargv = argv;
        sargc = argc - 1;
        if(sargc == 0) {
                in = &bstdin;
                Binit(in, 0, OREAD);
        } else if((in = Bopen(sargv[1], OREAD)) == 0)
                yyerror("cannot open input file");
        ifile = 1;
        ln = 0;
        ss = sargv[1];
}
```

⟨*function* `yylex` *(bc.y)* 159c⟩≡ (476b)

```
int
yylex(void)
{
        int c, ch;

restart:
        c = getch();
        peekc = -1;
        while(c == ' ' || c == '\t')
                c = getch();
        if(c == '\\') {
                getch();
                goto restart;
        }
        if(c >= 'a' && c <= 'z') {
                /* look ahead to look for reserved words */
                peekc = getch();
                if(peekc >= 'a' && peekc <= 'z') { /* must be reserved word */
                        if(c=='p' && peekc=='r') {
                                c = _PRINT;
                                goto skip;
                        }
                        if(c=='i' && peekc=='f') {
                                c = _IF;
                                goto skip;
                        }
                        if(c=='w' && peekc=='h') {
                                c = _WHILE;
                                goto skip;
                        }
                        if(c=='f' && peekc=='o') {
                                c = _FOR;
                                goto skip;
                        }
                        if(c=='s' && peekc=='q') {
```

```
                c = SQRT;
                goto skip;
        }
        if(c=='r' && peekc=='e') {
                c = _RETURN;
                goto skip;
        }
        if(c=='b' && peekc=='r') {
                c = _BREAK;
                goto skip;
        }
        if(c=='d' && peekc=='e') {
                c = _DEFINE;
                goto skip;
        }
        if(c=='s' && peekc=='c') {
                c = SCALE;
                goto skip;
        }
        if(c=='b' && peekc=='a') {
                c = BASE;
                goto skip;
        }
        if(c=='i' && peekc=='b') {
                c = BASE;
                goto skip;
        }
        if(c=='o' && peekc=='b') {
                c = OBASE;
                goto skip;
        }
        if(c=='d' && peekc=='i') {
                c = FFF;
                goto skip;
        }
        if(c=='a' && peekc=='u') {
                c = _AUTO;
                goto skip;
        }
        if(c=='l' && peekc=='e') {
                c = LENGTH;
                goto skip;
        }
        if(c=='q' && peekc=='u')
                getout();
        /* could not be found */
        return ERROR;

skip:   /* skip over rest of word */
        peekc = -1;
        for(;;) {
                ch = getch();
                if(ch < 'a' || ch > 'z')
                        break;
        }
        peekc = ch;
        return c;
}

/* usual case; just one single letter */
```

```
        yylval.cptr = letr[c-'a'];
        return LETTER;
}
if((c >= '0' && c <= '9') || (c >= 'A' && c <= 'F')) {
        yylval.cc = c;
        return DIGIT;
}
switch(c) {
case '.':
        return DOT;
case '*':
        yylval.cptr = "*";
        return cpeek('=', EQOP, c);
case '%':
        yylval.cptr = "%%";
        return cpeek('=', EQOP, c);
case '^':
        yylval.cptr = "^";
        return cpeek('=', EQOP, c);
case '+':
        ch = cpeek('=', EQOP, c);
        if(ch == EQOP) {
                yylval.cptr = "+";
                return ch;
        }
        return cpeek('+', INCR, c);
case '-':
        ch = cpeek('=', EQOP, c);
        if(ch == EQOP) {
                yylval.cptr = "-";
                return ch;
        }
        return cpeek('-', DECR, c);
case '=':
        return cpeek('=', EQ, '=');
case '<':
        return cpeek('=', LE, '<');
case '>':
        return cpeek('=', GE, '>');
case '!':
        return cpeek('=', NE, '!');
case '/':
        ch = cpeek('=', EQOP, c);
        if(ch == EQOP) {
                yylval.cptr = "/";
                return ch;
        }
        if(peekc == '*') {
                peekc = -1;
                for(;;) {
                        ch = getch();
                        if(ch == '*') {
                                peekc = getch();
                                if(peekc == '/') {
                                        peekc = -1;
                                        goto restart;
                                }
                        }
                }
        }
```

```
                return c;
        case '"':
                yylval.cptr = str;
                while((c=getch()) != '"'){
                        *str++ = c;
                        if(str >= &string[999]){
                                yyerror("string space exceeded");
                                getout();
                        }
                }
                *str++ = 0;
                return QSTR;
        default:
                return c;
        }
}
```

13.3.3 Grammar

⟨*type directives bc.y* 162a⟩≡ (476b)

```
%type   <cptr>  pstat stat stat1 def slist dlets e ase nase
%type   <cptr>  re fprefix cargs eora cons constant lora
%type   <cptr>  crs
```

⟨*priority directives bc.y* 162b⟩≡ (476b)

```
%right  '=' EQOP
%left   '+' '-'
%left   '*' '/' '%'
%right  '^'
%left   UMINUS
```

⟨*grammar bc.y* 162c⟩≡ (476b)

```
start:
        start stuff
|       stuff

stuff:
        pstat tail
        {
                output($1);
        }
|       def dargs ')' '{' dlist slist '}'
        {
                ttp = bundle(6, pre, $6, post , "0", numb[lev], "Q");
                conout(ttp, (char*)$1);
                rcrs = crs;
                output("");
                lev = bindx = 0;
        }

dlist:
        tail
|       dlist _AUTO dlets tail

stat:
        stat1
|       nase
        {
                if(sflag)
```

```
                        $$ = bundle(2, $1, "s.");
        }

pstat:
        stat1
        {
                if(sflag)
                        $$ = bundle(2, $1, "0");
        }
|       nase
        {
                if(!sflag)
                        $$ = bundle(2, $1, "ps.");
        }

stat1:
        {
                $$ = bundle(1, "");
        }
|       ase
        {
                $$ = bundle(2, $1, "s.");
        }
|       SCALE '=' e
        {
                $$ = bundle(2, $3, "k");
        }
|       SCALE EQOP e
        {
                $$ = bundle(4, "K", $3, $2, "k");
        }
|       BASE '=' e
        {
                $$ = bundle(2, $3, "i");
        }
|       BASE EQOP e
        {
                $$ = bundle(4, "I", $3, $2, "i");
        }
|       OBASE '=' e
        {
                $$ = bundle(2, $3, "o");
        }
|       OBASE EQOP e
        {
                $$ = bundle(4, "O", $3, $2, "o");
        }
|       QSTR
        {
                $$ = bundle(3, "[", $1, "]P");
        }
|       _BREAK
        {
                $$ = bundle(2, numb[lev-bstack[bindx-1]], "Q");
        }
|       _PRINT e
        {
                $$ = bundle(2, $2, "ps.");
        }
|       _RETURN e
```

```
        {
                $$ = bundle(4, $2, post, numb[lev], "Q");
        }
|       _RETURN
        {
                $$ = bundle(4, "0", post, numb[lev], "Q");
        }
|       '{' slist '}'
        {
                $$ = $2;
        }
|       FFF
        {
                $$ = bundle(1, "fY");
        }
|       _IF crs BLEV '(' re ')' stat
        {
                conout($7, $2);
                $$ = bundle(3, $5, $2, " ");
        }
|       _WHILE crs '(' re ')' stat BLEV
        {
                $$ = bundle(3, $6, $4, $2);
                conout($$, $2);
                $$ = bundle(3, $4, $2, " ");
        }
|       fprefix crs re ';' e ')' stat BLEV
        {
                //XXX: pad: using $$ = strategy fails with
                // yacc error so commented for now
                //bundle(5, $7, $5, "s.", $3, $2);
                //conout($$, $2);
                //bundle(5, $1, "s.", $3, $2, " ");
        }
|       '~' LETTER '=' e
        {
                $$ = bundle(3, $4, "S", $2);
        }

fprefix:
        _FOR '(' e ';'
        {
                $$ = $3;
        }

BLEV:
        '='
        {
                --bindx;
        }

slist:
        stat
|       slist tail stat
        {
                $$ = bundle(2, $1, $3);
        }

tail:
        '\n'
```

```
        {
                ln++;
        }
|       ';'

re:
        e EQ e
        {
                $$ = bundle(3, $1, $3, "=");
        }
|       e '<' e
        {
                $$ = bundle(3, $1, $3, ">");
        }
|       e '>' e
        {
                $$ = bundle(3, $1, $3, "<");
        }
|       e NE e
        {
                $$ = bundle(3, $1, $3, "!=");
        }
|       e GE e
        {
                $$ = bundle(3, $1, $3, "!>");
        }
|       e LE e
        {
                $$ = bundle(3, $1, $3, "!<");
        }
|       e
        {
                $$ = bundle(2, $1, " 0!=");
        }

nase:
        '(' e ')'
        {
                $$ = $2;
        }
|       cons
        {
                $$ = bundle(3, " ", $1, " ");
        }
|       DOT cons
        {
                $$ = bundle(3, " .", $2, " ");
        }
|       cons DOT cons
        {
                $$ = bundle(5, " ", $1, ".", $3, " ");
        }
|       cons DOT
        {
                $$ = bundle(4, " ", $1, ".", " ");
        }
|       DOT
        {
                $<cptr>$ = "l.";
        }
```

```
|       LETTER '[' e ']'
        {
                $$ = bundle(3, $3, ";", geta($1));
        }
|       LETTER INCR
        {
                $$ = bundle(4, "l", $1, "d1+s", $1);
        }
|       INCR LETTER
        {
                $$ = bundle(4, "l", $2, "1+ds", $2);
        }
|       DECR LETTER
        {
                $$ = bundle(4, "l", $2, "1-ds", $2);
        }
|       LETTER DECR
        {
                $$ = bundle(4, "l", $1, "d1-s", $1);
        }
|       LETTER '[' e ']' INCR
        {
                $$ = bundle(7, $3, ";", geta($1), "d1+" ,$3, ":" ,geta($1));
        }
|       INCR LETTER '[' e ']'
        {
                $$ = bundle(7, $4, ";", geta($2), "1+d", $4, ":", geta($2));
        }
|       LETTER '[' e ']' DECR
        {
                $$ = bundle(7, $3, ";", geta($1), "d1-", $3, ":", geta($1));
        }
|       DECR LETTER '[' e ']'
        {
                $$ = bundle(7, $4, ";", geta($2), "1-d", $4, ":" ,geta($2));
        }
|       SCALE INCR
        {
                $$ = bundle(1, "Kd1+k");
        }
|       INCR SCALE
        {
                $$ = bundle(1, "K1+dk");
        }
|       SCALE DECR
        {
                $$ = bundle(1, "Kd1-k");
        }
|       DECR SCALE
        {
                $$ = bundle(1, "K1-dk");
        }
|       BASE INCR
        {
                $$ = bundle(1, "Id1+i");
        }
|       INCR BASE
        {
                $$ = bundle(1, "I1+di");
        }
```

```
|       BASE DECR
        {
                $$ = bundle(1, "Id1-i");
        }
|       DECR BASE
        {
                $$ = bundle(1, "I1-di");
        }
|       OBASE INCR
        {
                $$ = bundle(1, "Od1+o");
        }
|       INCR OBASE
        {
                $$ = bundle(1, "O1+do");
        }
|       OBASE DECR
        {
                $$ = bundle(1, "Od1-o");
        }
|       DECR OBASE
        {
                $$ = bundle(1, "O1-do");
        }
|       LETTER '(' cargs ')'
        {
                $$ = bundle(4, $3, "l", getf($1), "x");
        }
|       LETTER '(' ')'
        {
                $$ = bundle(3, "l", getf($1), "x");
        }
|       LETTER '=' {
                $$ = bundle(2, "l", $1);
        }
|       LENGTH '(' e ')'
        {
                $$ = bundle(2, $3, "Z");
        }
|       SCALE '(' e ')'
        {
                $$ = bundle(2, $3, "X");
        }
|       '?'
        {
                $$ = bundle(1, "?");
        }
|       SQRT '(' e ')'
        {
                $$ = bundle(2, $3, "v");
        }
|       '~' LETTER
        {
                $$ = bundle(2, "L", $2);
        }
|       SCALE
        {
                $$ = bundle(1, "K");
        }
|       BASE
```

```
        {
                $$ = bundle(1, "I");
        }
|       OBASE
        {
                $$ = bundle(1, "O");
        }
|       '-' e
        {
                $$ = bundle(3, " 0", $2, "-");
        }
|       e '+' e
        {
                $$ = bundle(3, $1, $3, "+");
        }
|       e '-' e
        {
                $$ = bundle(3, $1, $3, "-");
        }
|       e '*' e
        {
                $$ = bundle(3, $1, $3, "*");
        }
|       e '/' e
        {
                $$ = bundle(3, $1, $3, "/");
        }
|       e '%' e
        {
                $$ = bundle(3, $1, $3, "%%");
        }
|       e '^' e
        {
                $$ = bundle(3, $1, $3, "^");
        }

ase:
        LETTER '=' e
        {
                $$ = bundle(3, $3, "ds", $1);
        }
|       LETTER '[' e ']' '=' e
        {
                $$ = bundle(5, $6, "d", $3, ":", geta($1));
        }
|       LETTER EQOP e
        {
                $$ = bundle(6, "l", $1, $3, $2, "ds", $1);
        }
|       LETTER '[' e ']' EQOP e
        {
                $$ = bundle(9, $3, ";", geta($1), $6, $5, "d", $3, ":", geta($1));
        }

e:
        ase
|       nase

cargs:
        eora
```

```
|       cargs ',' eora
        {
                $$ = bundle(2, $1, $3);
        }

eora:
        e
|       LETTER '[' ']'
        {
                $$ = bundle(2, "l", geta($1));
        }

cons:
        constant
        {
                *cp++ = 0;
        }

constant:
        '_'
        {
                $<cptr>$ = cp;
                *cp++ = '_';
        }
|       DIGIT
        {
                $<cptr>$ = cp;
                *cp++ = $1;
        }
|       constant DIGIT
        {
                *cp++ = $2;
        }

crs:
        '='
        {
                $$ = cp;
                *cp++ = '<';
                *cp++ = crs/100+'0';
                *cp++ = (crs%100)/10+'0';
                *cp++ = crs%10+'0';
                *cp++ = '>';
                *cp++ = '\0';
                if(crs++ >= 220) {
                        yyerror("program too big");
                        getout();
                }
                bstack[bindx++] = lev++;
        }

def:
        _DEFINE LETTER '('
        {
                $$ = getf($2);
                pre = (char*)"";
                post = (char*)"";
                lev = 1;
                bindx = 0;
                bstack[bindx] = 0;
```

```
        }

dargs:
|       lora
        {
                pp((char*)$1);
        }
|       dargs ',' lora
        {
                pp((char*)$3);
        }

dlets:
        lora
        {
                tp((char*)$1);
        }
|       dlets ',' lora
        {
                tp((char*)$3);
        }

lora:
        LETTER
        {
                $<cptr>$=$1;
        }
|       LETTER '[' ']'
        {
                $$ = geta($1);
        }
```

13.3.4 Backend

Chapter 14

Misc

This chapter collects small utilities that don't fit neatly into the other categories.

14.1 basename

`basename` extracts the filename from a path by stripping the directory prefix and an optional suffix. With `-d`, it does the opposite: it strips the filename and returns the directory (like UNIX's `dirname`).

⟨*function* `main`*(basename.c)* 171⟩≡ (235c)

```
void
main(int argc, char *argv[])
{
    char *pr;
    int n;
    bool dflag = false;

    if(argc>1 && strcmp(argv[1], "-d") == ORD__EQ){
        --argc;
        ++argv;
        dflag = true;
    }
    if(argc < 2 || argc > 3){
        fprint(STDERR, "usage: basename [-d] string [suffix]\n");
        exits("usage");
    }
    pr = utfrrune(argv[1], '/');
    if(dflag){
        if(pr){
            *pr = '\0';
            print("%s\n", argv[1]);
            exits(nil);
        }
        // else
        print(".\n");
        exits(nil);
    }
    if(pr)
        pr++;
    else
        pr = argv[1];

    if(argc==3){
        n = strlen(pr)-strlen(argv[2]);
        if(n >= 0 && !strcmp(pr+n, argv[2]))
```

```
            pr[n] = '\0';
    }
    print("%s\n", pr);
    exits(nil);
}
```

14.2 file

14.3 iconv

14.4 strings

14.5 unicode

14.6 sleep

14.7 reboot

Chapter 15

Advanced topics

15.1 Optimizations

15.2 Internationalization

Chapter 16

Conclusion

You now know how the Plan 9 command-line utilities work, to the smallest details, and more generally how many UNIX utilities work.

This book has covered a wide range of tools: file and directory operations (`ls`, `cp`, `mkdir`), pattern matching and substitution (`grep`, `sed`), text processing with `awk`, file comparison (`diff`, `cmp`), process management, archiving (`tar`, `ar`), date and time utilities, pipe plumbing, byte-level tools, and calculators. Despite their diversity, these programs share a common architecture: parse command-line flags, read from standard input or named files, process line by line, and write to standard output. This is the UNIX philosophy in action—small, composable tools connected by pipes.

16.1 Patterns and techniques

These techniques apply far beyond command-line tools:

- *Filter pattern*: a function from one stream to another. The same pattern—composable transformations chained together—is the basis of Java 8 streams, Spark pipelines, and functional programming's `map`/`filter`. The power comes from the constraint: one thing in, one thing out, standard interface.
- *Compile-then-interpret*: `grep`, `sed`, and `awk` compile their patterns before processing input. The general rule—if you evaluate an expression many times, compile it first—drives SQL query planners and template engines alike.
- *Streaming with bounded memory*: processing input one line at a time, never loading the whole file. The same approach is used by Kafka consumers and SAX XML parsers—essential when processing data larger than available memory.
- *Diff as edit distance*: the longest common subsequence algorithm is one of the most widely reused in computer science. Git uses it for merges, bioinformatics uses it for DNA sequence alignment (Needleman-Wunsch is essentially `diff`), and spell checkers use a variant (Levenshtein distance).

16.2 Connections to other books

- Shell book [Pad18]: `rc` is the glue that connects these utilities. Pipes, redirections, and backquote substitution let you compose `grep`, `sed`, `awk`, `sort`, and friends into powerful one-liners.
- Kernel book [Pad14]: every utility relies on the kernel's system calls: `open()`/`read()`/`write()` for file I/O, `stat()` for file metadata, `rfork()`/`exec()`/`wait()` for process management, and `pipe()` for inter-process communication.

- LIBCORE book [Pad16]: the utilities use `libc` functions throughout—`Biobuf` for buffered I/O, `print()` for formatted output, the UTF-8 string functions for Unicode handling, and the argument parsing macros (`ARGBEGIN`/`ARGEND`).

16.3 Beyond the Plan 9 utilities

The Plan 9 utilities descend from the original UNIX tools and remain close to that tradition. The broader ecosystem has evolved in several directions:

- *GNU coreutils*: the GNU versions of these tools add many features—long options (`--recursive`), internationalization, and extensions like `grep -P` for Perl-compatible regular expressions. GNU `awk` (gawk) adds networking, loadable extensions, and persistent variables. These additions make the tools more capable but also much larger.

- *Modern replacements*: a wave of rewrites in Rust and Go offer improved ergonomics: `ripgrep` (faster, respects `.gitignore`), `fd` (friendlier `find`), `bat` (`cat` with syntax highlighting), `exa`/`eza` (`ls` with Git status and colors), `jq` and `yq` for structured data. These tools acknowledge that the traditional plain-text, line-oriented model does not always fit modern data formats like JSON and YAML.

- *BusyBox and Toybox*: at the other end of the spectrum, BusyBox packs hundreds of utilities into a single statically linked binary, aimed at embedded systems and containers. Toybox (BSD-licensed) serves the same role in Android. The Plan 9 utilities share this minimalist spirit.

- *Structured data pipelines*: PowerShell pipes objects instead of text. Nushell and `xsv`/`csvkit` operate on structured tabular data. These address a real limitation of the traditional UNIX model: parsing `ls -l` output is fragile, and tools like `awk` and `cut` depend on assumptions about whitespace and field separators.

- *POSIX standardization*: POSIX specifies the behavior of most of these utilities, providing a common baseline across systems. The Plan 9 versions often predate or intentionally diverge from POSIX, keeping a simpler interface at the cost of portability.

Ironically, the AI era has reinforced the relevance of these classic tools. AI coding assistants like Claude Code, GitHub Copilot, and Cursor operate by invoking `ls`, `grep`, `sed`, `awk`, and shell scripts to explore codebases, search for patterns, and apply transformations—the same tools described in this book. The UNIX philosophy of small, composable text-processing programs turns out to be an ideal interface for AI agents that need to understand and modify code programmatically.

The Plan 9 utilities show that a useful programmer's toolbox can be built from small, single-purpose programs—most under a few hundred lines of C. The UNIX philosophy they embody—text streams, pipes, composability—remains the foundation on which even the most modern tools are built.

Appendix A

Extra Code

A.1 files/

A.1.1 files/cat.c

⟨files/cat.c 176a⟩≡
⟨*plan9 includes* 14⟩

⟨*function* cat 16b⟩
⟨*function* main*(cat.c)* 16a⟩

A.1.2 files/ls.c

⟨files/ls.c 176b⟩≡
⟨*plan9 includes* 14⟩
```
#include <bio.h>
#ifdef Unix
#else
//DEAD include? remove the whole ifdef?
#include <fcall.h>
#endif

typedef struct NDir NDir;
```
⟨*struct* NDir 16d⟩

⟨*global flags ls.c* 21a⟩
⟨*globals ls.c* 17a⟩

```
// forward decls
error1  ls(char*, bool);
ord     compar(const NDir*, const NDir*);
char*   asciitime(long);
char*   darwx(long);
void    rwx(long, char*);
void    growto(long);
void    dowidths(Dir*);
void    format(Dir*, char*);
void    output(void);
char*   xcleanname(char*);
```

⟨*function* main*(ls.c)* 17d⟩

⟨*function* ls 17e⟩

⟨*function* `output` *(ls.c)* 19a⟩
⟨*function* `dowidths` *(ls.c)* 22b⟩
⟨*function* `fileflag` *(ls.c)* 22a⟩
⟨*function* `format` *(ls.c)* 20⟩

⟨*function* `growto` *(ls.c)* 18a⟩
⟨*function* `compar` *(ls.c)* 19b⟩
⟨*function* `asciitime` *(ls.c)* 23a⟩
⟨*function* `xcleanname` *(ls.c)* 18b⟩

A.1.3 `files/touch.c`

⟨`files/touch.c` 177a⟩≡
⟨*plan9 includes* 14⟩

```
// forward decls
error1 touch(bool, char *);
```

⟨*global* `now` *(touch.c)* 23c⟩

⟨*function* `usage` *(touch.c)* 23b⟩
⟨*function* `main` *(touch.c)* 23d⟩

⟨*function* `touch` 24⟩

A.1.4 `files/mkdir.c`

⟨`files/mkdir.c` 177b⟩≡
⟨*plan9 includes* 14⟩

⟨*global* `e` *(mkdir.c)* 25c⟩
⟨*global* `mode` *(mkdir.c)* 25b⟩

⟨*function* `usage` *(mkdir.c)* 25a⟩
⟨*function* `makedir` 25e⟩
⟨*function* `mkdirp` 26a⟩

⟨*function* `main` *(mkdir.c)* 25d⟩

A.1.5 `files/rm.c`

⟨`files/rm.c` 177c⟩≡
⟨*plan9 includes* 14⟩

⟨*global* `errbuf` *(rm.c)* 26c⟩
⟨*global* `ignerr` *(rm.c)* 26b⟩

⟨*function* `err` *(rm.c)* 28a⟩
⟨*function* `rmdir` 27⟩

⟨*function* `main` *(rm.c)* 26d⟩

A.1.6 files/cp.c

⟨files/cp.c 178a⟩≡
⟨*plan9 includes* 14⟩

⟨*constant* DEFB*(cp.c)* 30a⟩

⟨*global* failed*(cp.c)* 28b⟩
⟨*global flags(cp.c)* 28c⟩

```
// forward decls
void copy(char *from, char *to, bool todir);
errorneg1 copy1(fdt fdf, fdt fdt, char *from, char *to);
```

⟨*function* main*(cp.c)* 29a⟩

⟨*function* samefile*(cp.c)* 31⟩
⟨*function* copy 29b⟩
⟨*function* copy1 30b⟩

A.1.7 files/mv.c

⟨files/mv.c 178b⟩≡
⟨*plan9 includes* 14⟩

```
// forward decls
errorneg1 copy1(fdt fdf, fdt fdt, char *from, char *to);
void      hardremove(char *);
errorneg1 mv(char *from, char *todir, char *toelem);
errorneg1 mv1(char *from, Dir *dirb, char *todir, char *toelem);
bool      samefile(char *, char *);
void      split(char *, char **, char **);
```

⟨*function* main*(mv.c)* 32a⟩

⟨*function* mv 32b⟩
⟨*function* mv1 33⟩
⟨*function* copy1*(mv.c)* 34⟩
⟨*function* split*(mv.c)* 35a⟩
⟨*function* samefile*(mv.c)* 35b⟩
⟨*function* hardremove*(mv.c)* 35c⟩

A.1.8 files/chmod.c

⟨files/chmod.c 178c⟩≡
⟨*plan9 includes* 14⟩

⟨*macros chmod.c* 36a⟩

```
// forward decls
error0 parsemode(char *, ulong *, ulong *);
```

⟨*function* main*(chmod.c)* 36b⟩

⟨*function* parsemode*(chmod.c)* 36c⟩

A.1.9 files/chgrp.c

⟨files/chgrp.c 179a⟩≡
⟨*plan9 includes* 14⟩

⟨*global* uflag*(chgrp.c)* 38a⟩

⟨*function* main*(chgrp.c)* 38b⟩

A.1.10 files/mtime.c

⟨files/mtime.c 179b⟩≡
⟨*plan9 includes* 14⟩

⟨*function* usage*(mtime.c)* 38c⟩
⟨*function* main*(mtime.c)* 38d⟩

A.1.11 files/wc.c

⟨misc/wc.c 179c⟩≡
```
/*
 * Count bytes within runes, if it fits in a uvlong, and other things.
 */
```
⟨*plan9 includes* 14⟩
```
#include <bio.h>
```

⟨*globals wc.c* 39b⟩

⟨*enum wc.c* 40b⟩

⟨*function* wc 40c⟩
⟨*function* report*(wc.c)* 41⟩

⟨*function* main*(wc.c)* 40a⟩

A.1.12 files/du.c

⟨misc/du.c 179d⟩≡
```
/*
 * du - print disk usage
 */
```
⟨*plan9 includes* 14⟩
```
#include <str.h>

// forward decls
extern  vlong   du(char*, Dir*);
extern  void    err(char*);
extern  vlong   blkmultiple(vlong);
extern  int seen(Dir*);
extern  int warn(char*);

enum {
    Vkilo = 1024LL,
};

/* rounding up, how many units does amt occupy? */
#define HOWMANY(amt, unit)  (((amt)+(unit)-1) / (unit))
```

```
#define ROUNDUP(amt, unit)  (HOWMANY(amt, unit) * (unit))

int aflag;
int autoscale;
int fflag;
int fltflag;
int qflag;
int readflg;
int sflag;
int tflag;
int uflag;

char    *fmt = "%llud\t%q\n";
char    *readbuf;
vlong   blocksize = Vkilo;  /* actually more likely to be 4K or 8K */
vlong   unit;           /* scale factor for output */

static char *pfxes[] = {    /* SI prefixes for units > 1 */
    "",
    "k", "M", "G",
    "T", "P", "E",
    "Z", "Y",
    nil,
};

void
usage(void)
{
    fprint(2, "usage: du [-aefhnqstu] [-b size] [-p si-pfx] [file ...]\n");
    exits("usage");
}

void
printamt(vlong amt, char *name)
{
    if (readflg)
        return;
    if (autoscale) {
        int scale = 0;
        double val = (double)amt/unit;

        while (fabs(val) >= 1024 && scale < nelem(pfxes)-1) {
            scale++;
            val /= 1024;
        }
        print("%.6g%s\t%q\n", val, pfxes[scale], name);
    } else if (fltflag)
        print("%.6g\t%q\n", (double)amt/unit, name);
    else
        print(fmt, HOWMANY(amt, unit), name);
}

void
main(int argc, char *argv[])
{
    int i, scale;
    char *s, *ss, *name;

    //XXX: doquote = needsrcquote;
    quotefmtinstall();
```

```
ARGBEGIN {
case 'a':   /* all files */
    aflag = 1;
    break;
case 'b':   /* block size */
    s = ARGF();
    if(s) {
        blocksize = strtoul(s, &ss, 0);
        if(s == ss)
            blocksize = 1;
        while(*ss++ == 'k')
            blocksize *= 1024;
    }
    break;
case 'e':   /* print in %g notation */
    fltflag = 1;
    break;
case 'f':   /* don't print warnings */
    fflag = 1;
    break;
case 'h':   /* similar to -h in bsd but more precise */
    autoscale = 1;
    break;
case 'n':   /* all files, number of bytes */
    aflag = 1;
    blocksize = 1;
    unit = 1;
    break;
case 'p':
    s = ARGF();
    if(s) {
        for (scale = 0; pfxes[scale] != nil; scale++)
            if (cistrcmp(s, pfxes[scale]) == 0)
                break;
        if (pfxes[scale] == nil)
            sysfatal("unknown suffix %s", s);
        unit = 1;
        while (scale-- > 0)
            unit *= Vkilo;
    }
    break;
case 'q':   /* qid */
    fmt = "%.16llux\t%q\n";
    qflag = 1;
    break;
case 'r':
    /* undocumented: just read & ignore every block of every file */
    readflg = 1;
    break;
case 's':   /* only top level */
    sflag = 1;
    break;
case 't':   /* return modified/accessed time */
    tflag = 1;
    break;
case 'u':   /* accessed time */
    uflag = 1;
    break;
default:
```

```
        usage();
    } ARGEND

    if (unit == 0)
        if (qflag || tflag || uflag || autoscale)
            unit = 1;
        else
            unit = Vkilo;
    if (blocksize < 1)
        blocksize = 1;

    if (readflg) {
        readbuf = malloc(blocksize);
        if (readbuf == nil)
            sysfatal("out of memory");
    }
    if(argc==0)
        printamt(du(".", dirstat(".")), ".");
    else
        for(i=0; i<argc; i++) {
            name = argv[i];
            printamt(du(name, dirstat(name)), name);
        }
    exits(0);
}

vlong
dirval(Dir *d, vlong size)
{
    if(qflag)
        return d->qid.path;
    else if(tflag) {
        if(uflag)
            return d->atime;
        return d->mtime;
    } else
        return size;
}

void
readfile(char *name)
{
    int n, fd = open(name, OREAD);

    if(fd < 0) {
        warn(name);
        return;
    }
    while ((n = read(fd, readbuf, blocksize)) > 0)
        continue;
    if (n < 0)
        warn(name);
    close(fd);
}

vlong
dufile(char *name, Dir *d)
{
    vlong t = blkmultiple(d->length);
```

```
    if(aflag || readflg) {
        String *file = s_copy(name);

        s_append(file, "/");
        s_append(file, d->name);
        if (readflg)
            readfile(s_to_c(file));
        t = dirval(d, t);
        printamt(t, s_to_c(file));
        s_free(file);
    }
    return t;
}

vlong
du(char *name, Dir *dir)
{
    int fd, i, n;
    Dir *buf, *d;
    String *file;
    vlong nk, t;

    if(dir == nil)
        return warn(name);

    if((dir->qid.type&QTDIR) == 0)
        return dirval(dir, blkmultiple(dir->length));

    fd = open(name, OREAD);
    if(fd < 0)
        return warn(name);
    nk = 0;
    while((n=dirread(fd, &buf)) > 0) {
        d = buf;
        for(i = n; i > 0; i--, d++) {
            if((d->qid.type&QTDIR) == 0) {
                nk += dufile(name, d);
                continue;
            }

            if(strcmp(d->name, ".") == 0 ||
               strcmp(d->name, "..") == 0 ||
               /* !readflg && */ seen(d))
                continue;   /* don't get stuck */

            file = s_copy(name);
            s_append(file, "/");
            s_append(file, d->name);

            t = du(s_to_c(file), d);

            nk += t;
            t = dirval(d, t);
            if(!sflag)
                printamt(t, s_to_c(file));
            s_free(file);
        }
        free(buf);
    }
    if(n < 0)
```

```
        warn(name);
    close(fd);
    return dirval(dir, nk);
}

#define NCACHE  256 /* must be power of two */

typedef struct
{
    Dir*    cache;
    int n;
    int max;
} Cache;
Cache cache[NCACHE];

int
seen(Dir *dir)
{
    Dir *dp;
    int i;
    Cache *c;

    c = &cache[dir->qid.path&(NCACHE-1)];
    dp = c->cache;
    for(i=0; i<c->n; i++, dp++)
        if(dir->qid.path == dp->qid.path &&
           dir->type == dp->type &&
           dir->dev == dp->dev)
            return 1;
    if(c->n == c->max){
        if (c->max == 0)
            c->max = 8;
        else
            c->max += c->max/2;
        c->cache = realloc(c->cache, c->max*sizeof(Dir));
        if(c->cache == nil)
            err("malloc failure");
    }
    c->cache[c->n++] = *dir;
    return 0;
}

void
err(char *s)
{
    fprint(2, "du: %s: %r\n", s);
    exits(s);
}

int
warn(char *s)
{
    if(fflag == 0)
        fprint(2, "du: %s: %r\n", s);
    return 0;
}

/* round up n to nearest block */
vlong
blkmultiple(vlong n)
```

```
{
    if(blocksize == 1)      /* no quantization */
        return n;
    return ROUNDUP(n, blocksize);
}
```

A.2 byte/

A.2.1 byte/xd.c

⟨byte/xd.c 185⟩≡

```
⟨plan9 includes 14⟩
#include <bio.h>

uchar       odata[16];
uchar       data[32];
int     ndata;
int     nread;
ulong       addr;
int     repeats;
int     swizzle;
int     flush;
int     abase=2;
int     xd(char *, int);
void        xprint(char *, ...);
void        initarg(void), swizz(void);
enum{
    Narg=10,

    TNone=0,
    TAscii,
    TRune,
};
typedef struct Arg Arg;
typedef void fmtfn(char *);
struct Arg
{
    int chartype;       /* TNone, TAscii, TRunes */
    int loglen;     /* 0==1, 1==2, 2==4, 3==8 */
    int base;       /* 0==8, 1==10, 2==16 */
    fmtfn   *fn;        /* function to call with data */
    char    *afmt;      /* format to use to print address */
    char    *fmt;       /* format to use to print data */
}arg[Narg];
int narg;

fmtfn   fmt0, fmt1, fmt2, fmt3, fmtc, fmtr;
fmtfn *fmt[4] = {
    fmt0,
    fmt1,
    fmt2,
    fmt3
};

char *dfmt[4][3] = {
    " %.3uo",   " %.3ud",   " %.2ux",
    " %.6uo",   " %.5ud",   " %.4ux",
    " %.11luo", " %.10lud", " %.8lux",
```

```
    " %.22lluo",    " %.20llud",    " %.16llux",
};

char *cfmt[3][3] = {
    "   %c",    "   %c",    "  %c",
    " %.3s",    " %.3s",    " %.2s",
    " %.3uo",   " %.3ud",   " %.2ux",
};

char *rfmt[1][1] = {
    " %2.2C",
};

char *afmt[2][3] = {
    "%.7luo ",  "%.7lud ",  "%.7lux ",
    "%7luo ",   "%7lud ",   "%7lux ",
};

Biobuf  bin;
Biobuf  bout;

void
main(int argc, char *argv[])
{
    int i, err;
    Arg *ap;

    Binit(&bout, 1, OWRITE);
    err = 0;
    ap = 0;
    while(argc>1 && argv[1][0]=='-' && argv[1][1]){
        --argc;
        argv++;
        argv[0]++;
        if(argv[0][0] == 'r'){
            repeats = 1;
            if(argv[0][1])
                goto Usage;
            continue;
        }
        if(argv[0][0] == 's'){
            swizzle = 1;
            if(argv[0][1])
                goto Usage;
            continue;
        }
        if(argv[0][0] == 'u'){
            flush = 1;
            if(argv[0][1])
                goto Usage;
            continue;
        }
        if(argv[0][0] == 'a'){
            argv[0]++;
            switch(argv[0][0]){
            case 'o':
                abase = 0;
                break;
            case 'd':
                abase = 1;
```

```
            break;
        case 'x':
            abase = 2;
            break;
        default:
            goto Usage;
        }
        if(argv[0][1])
            goto Usage;
        continue;
    }
    ap = &arg[narg];
    initarg();
    while(argv[0][0]){
        switch(argv[0][0]){
        case 'c':
            ap->chartype = TAscii;
            ap->loglen = 0;
            if(argv[0][1] || argv[0][-1]!='-')
                goto Usage;
            break;
        case 'R':
            ap->chartype = TRune;
            ap->loglen = 0;
            if(argv[0][1] || argv[0][-1]!='-')
                goto Usage;
            break;
        case 'o':
            ap->base = 0;
            break;
        case 'd':
            ap->base = 1;
            break;
        case 'x':
            ap->base = 2;
            break;
        case 'b':
        case '1':
            ap->loglen = 0;
            break;
        case 'w':
        case '2':
            ap->loglen = 1;
            break;
        case 'l':
        case '4':
            ap->loglen = 2;
            break;
        case 'v':
        case '8':
            ap->loglen = 3;
            break;
        default:
        Usage:
fprint(2, "usage: xd [-u] [-r] [-s] [-a{odx}] [-c|{b1w2l4v8}{odx}] ... file ...\n");
            exits("usage");
        }
        argv[0]++;
    }
    if(ap->chartype == TRune)
```

```
            ap->fn = fmtr;
        else if(ap->chartype == TAscii)
            ap->fn = fmtc;
        else
            ap->fn = fmt[ap->loglen];
        ap->fmt = dfmt[ap->loglen][ap->base];
        ap->afmt = afmt[ap>arg][abase];
    }
    if(narg == 0)
        initarg();
    if(argc == 1)
        err = xd(0, 0);
    else if(argc == 2)
        err = xd(argv[1], 0);
    else for(i=1; i<argc; i++)
        err |= xd(argv[i], 1);
    exits(err? "error" : 0);
}

void
initarg(void)
{
    Arg *ap;

    ap = &arg[narg++];
    if(narg >= Narg){
        fprint(2, "xd: too many formats (max %d)\n", Narg);
        exits("usage");
    }
    ap->chartype = TNone;
    ap->loglen = 2;
    ap->base = 2;
    ap->fn = fmt2;
    ap->fmt = dfmt[ap->loglen][ap->base];
    ap->afmt = afmt[narg>1][abase];
}

int
xd(char *name, int title)
{
    int fd;
    int i, star, nsee, nleft;
    Arg *ap;
    Biobuf *bp;

    fd = 0;
    if(name){
        bp = Bopen(name, OREAD);
        if(bp == 0){
            fprint(2, "xd: can't open %s\n", name);
            return 1;
        }
    }else{
        bp = &bin;
        Binit(bp, fd, OREAD);
    }
    if(title)
        xprint("%s\n", name);
    addr = 0;
    star = 0;
```

```
    nsee = 16;
    nleft = 0;
    /* read 32 but see only 16 so that runes are happy */
    while((ndata=Bread(bp, data + nleft, 32 - nleft)) >= 0){
        ndata += nleft;
        nleft = 0;
        nread = ndata;
        if(ndata>nsee)
            ndata = nsee;
        else if(ndata<nsee)
            for(i=ndata; i<nsee; i++)
                data[i] = 0;
        if(swizzle)
            swizz();
        if(ndata==nsee && repeats){
            if(addr>0 && data[0]==odata[0]){
                for(i=1; i<nsee; i++)
                    if(data[i] != odata[i])
                        break;
                if(i == nsee){
                    addr += nsee;
                    if(star == 0){
                        star++;
                        xprint("*\n", 0);
                    }
                    continue;
                }
            }
            for(i=0; i<nsee; i++)
                odata[i] = data[i];
            star = 0;
        }
        for(ap=arg; ap<&arg[narg]; ap++){
            xprint(ap->afmt, addr);
            (*ap->fn)(ap->fmt);
            xprint("\n", 0);
            if(flush)
                Bflush(&bout);
        }
        addr += ndata;
        if(ndata<nsee){
            xprint(afmt[0][abase], addr);
            xprint("\n", 0);
            if(flush)
                Bflush(&bout);
            break;
        }
        if(nread>nsee){
            nleft = nread - nsee;
            memmove(data, data + nsee, nleft);
        }
    }
    Bterm(bp);
    return 0;
}

void
swizz(void)
{
    uchar *p, *q;
```

```
    int i;
    uchar swdata[16];

    p = data;
    q = swdata;
    for(i=0; i<16; i++)
        *q++ = *p++;
    p = data;
    q = swdata;
    for(i=0; i<4; i++){
        p[0] = q[3];
        p[1] = q[2];
        p[2] = q[1];
        p[3] = q[0];
        p += 4;
        q += 4;
    }
}

void
fmt0(char *f)
{
    int i;
    for(i=0; i<ndata; i++)
        xprint(f, data[i]);
}

void
fmt1(char *f)
{
    int i;
    for(i=0; i<ndata; i+=sizeof(ushort))
        xprint(f, (data[i]<<8)|data[i+1]);
}

void
fmt2(char *f)
{
    int i;
    for(i=0; i<ndata; i+=sizeof(ulong))
        xprint(f, (data[i]<<24)|(data[i+1]<<16)|(data[i+2]<<8)|data[i+3]);
}

void
fmt3(char *f)
{
    int i;
    uvlong v;

    for(i=0; i<ndata; i+=sizeof(uvlong)){
        v = (data[i]<<24)|(data[i+1]<<16)|(data[i+2]<<8)|data[i+3];
        v <<= 32;
        v |= (data[i+4]<<24)|(data[i+1+4]<<16)|(data[i+2+4]<<8)|data[i+3+4];
        if(Bprint(&bout, f, v)<0){
            fprint(2, "xd: i/o error\n");
            exits("i/o error");
        }
    }
}
```

```
void
onefmtc(uchar c)
{
    switch(c){
    case '\t':
        xprint(cfmt[1][2], "\\t");
        break;
    case '\r':
        xprint(cfmt[1][2], "\\r");
        break;
    case '\n':
        xprint(cfmt[1][2], "\\n");
        break;
    case '\b':
        xprint(cfmt[1][2], "\\b");
        break;
    default:
        if(c>=0x7F || ' '>c)
            xprint(cfmt[2][2], c);
        else
            xprint(cfmt[0][2], c);
        break;
    }
}

void
fmtc(char *f)
{
    int i;

    USED(f);
    for(i=0; i<ndata; i++)
        onefmtc(data[i]);
}

void
fmtr(char *f)
{
    int i, w, cw;
    Rune r;
    static int nstart;

    USED(f);
    if(nstart)
        xprint("%*c", 3*nstart, ' ');
    for(i=nstart; i<ndata; )
        if(data[i] < Runeself)
            onefmtc(data[i++]);
        else{
            w = chartorune(&r, (char *)data+i);
            if(w == 1 || i + w>nread)
                onefmtc(data[i++]);
            else{
                cw = w;
                if(i + w>ndata)
                    cw = ndata - i;
                xprint(rfmt[0][0], r);
                xprint("%*c", 3*cw-3, ' ');
                i += w;
            }
```

```
        }
    if(i > ndata)
        nstart = i - ndata;
    else
        nstart = 0;
}

void
xprint(char *fmt, ...)
{
    va_list arglist;

    va_start(arglist, fmt);
    if(Bvprint(&bout, fmt, arglist)<0){
        fprint(2, "xd: i/o error\n");
        exits("i/o error");
    }
    va_end(arglist);
}
```

A.2.2 byte/split.c

⟨byte/split.c 192a⟩≡

```
⟨plan9 includes 14⟩
#include <bio.h>
#include <regexp.h>

⟨globals split.c 128d⟩

// forward decls
extern bool nextfile(void);
extern int matchfile(Resub*);
extern void openf(void);
extern char *fold(char*,int);
extern void usage(void);
extern void badexp(void);

⟨function main(split.c) 129g⟩

⟨function nextfile(split.c) 130a⟩
⟨function matchfile(split.c) 132⟩
⟨function openf(split.c) 130b⟩
⟨function fold(split.c) 131e⟩
⟨function usage(split.c) 128a⟩
⟨function badexp(split.c) 131d⟩
```

A.2.3 byte/dd.c

⟨byte/dd.c 192b⟩≡

```
⟨plan9 includes 14⟩

#define BIG ((1UL<<31)-1)
#define VBIG    ((1ULL<<63)-1)
#define LCASE   (1<<0)
#define UCASE   (1<<1)
#define SWAB    (1<<2)
#define NERR    (1<<3)
#define SYNC    (1<<4)
```

```
int cflag;
int fflag;

char    *string;
char    *ifile;
char    *ofile;
char    *ibuf;
char    *obuf;

vlong   skip;
vlong   oseekn;
vlong   iseekn;
vlong   oseekb;
vlong   iseekb;
vlong   count;

long    files   = 1;
long    ibs = 512;
long    obs = 512;
long    bs;
long    cbs;
long    ibc;
long    obc;
long    cbc;
long    nifr;
long    nipr;
long    nofr;
long    nopr;
long    ntrunc;

int dotrunc = 1;
int ibf;
int obf;

char    *op;
int nspace;

uchar   etoa[256];
uchar   atoe[256];
uchar   atoibm[256];

int quiet;

void    flsh(void);
int match(char *s);
vlong   number(vlong big);
void    cnull(int cc);
void    null(int c);
void    ascii(int cc);
void    unblock(int cc);
void    ebcdic(int cc);
void    ibm(int cc);
void    block(int cc);
void    term(char*);
void    stats(void);

#define iskey(s)    ((key[0] == '-') && (strcmp(key+1, s) == 0))

int
```

```
main(int argc, char *argv[])
{
    void (*conv)(int);
    char *ip;
    char *key;
    int a, c;

    conv = null;
    for(c=1; c<argc; c++) {
        key = argv[c++];
        if(c >= argc){
            fprint(2, "dd: arg %s needs a value\n", key);
            exits("arg");
        }
        string = argv[c];
        if(iskey("ibs")) {
            ibs = number(BIG);
            continue;
        }
        if(iskey("obs")) {
            obs = number(BIG);
            continue;
        }
        if(iskey("cbs")) {
            cbs = number(BIG);
            continue;
        }
        if(iskey("bs")) {
            bs = number(BIG);
            continue;
        }
        if(iskey("if")) {
            ifile = string;
            continue;
        }
        if(iskey("of")) {
            ofile = string;
            continue;
        }
        if(iskey("trunc")) {
            dotrunc = number(BIG);
            continue;
        }
        if(iskey("quiet")) {
            quiet = number(BIG);
            continue;
        }
        if(iskey("skip")) {
            skip = number(VBIG);
            continue;
        }
        if(iskey("seek") || iskey("oseek")) {
            oseekn = number(VBIG);
            continue;
        }
        if(iskey("iseek")) {
            iseekn = number(VBIG);
            continue;
        }
        if(iskey("iseekb")) {
```

```
    iseekb = number(VBIG);
    continue;
}
if(iskey("oseekb")) {
    oseekb = number(VBIG);
    continue;
}
if(iskey("count")) {
    count = number(VBIG);
    continue;
}
if(iskey("files")) {
    files = number(BIG);
    continue;
}
if(iskey("conv")) {
cloop:
    if(match(","))
        goto cloop;
    if(*string == '\0')
        continue;
    if(match("ebcdic")) {
        conv = ebcdic;
        goto cloop;
    }
    if(match("ibm")) {
        conv = ibm;
        goto cloop;
    }
    if(match("ascii")) {
        conv = ascii;
        goto cloop;
    }
    if(match("block")) {
        conv = block;
        goto cloop;
    }
    if(match("unblock")) {
        conv = unblock;
        goto cloop;
    }
    if(match("lcase")) {
        cflag |= LCASE;
        goto cloop;
    }
    if(match("ucase")) {
        cflag |= UCASE;
        goto cloop;
    }
    if(match("swab")) {
        cflag |= SWAB;
        goto cloop;
    }
    if(match("noerror")) {
        cflag |= NERR;
        goto cloop;
    }
    if(match("sync")) {
        cflag |= SYNC;
        goto cloop;
```

```
            }
            fprint(2, "dd: bad conv %s\n", argv[c]);
            exits("arg");
        }
        fprint(2, "dd: bad arg: %s\n", key);
        exits("arg");
    }
    if(conv == null && cflag&(LCASE|UCASE))
        conv = cnull;
    if(ifile)
        ibf = open(ifile, 0);
    else
        ibf = dup(0, -1);
    if(ibf < 0) {
        fprint(2, "dd: open %s: %r\n", ifile);
        exits("open");
    }
    if(ofile){
        if(dotrunc)
            obf = create(ofile, 1, 0664);
        else
            obf = open(ofile, 1);
        if(obf < 0) {
            fprint(2, "dd: create %s: %r\n", ofile);
            exits("create");
        }
    }else{
        obf = dup(1, -1);
        if(obf < 0) {
            fprint(2, "dd: can't dup file descriptor: %s: %r\n", ofile);
            exits("dup");
        }
    }
    if(bs)
        ibs = obs = bs;
    if(ibs == obs && conv == null)
        fflag++;
    if(ibs == 0 || obs == 0) {
        fprint(2, "dd: counts: cannot be zero\n");
        exits("counts");
    }
    ibuf = sbrk(ibs);
    if(fflag)
        obuf = ibuf;
    else
        obuf = sbrk(obs);
    sbrk(64);   /* For good measure */
    if(ibuf == (char *)-1 || obuf == (char *)-1) {
        fprint(2, "dd: not enough memory: %r\n");
        exits("memory");
    }
    ibc = 0;
    obc = 0;
    cbc = 0;
    op = obuf;

/*
    if(signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, term);
*/
```

```
    seek(obf, obs*oseekn, 1);
    seek(ibf, ibs*iseekn, 1);
    if(iseekb)
        seek(ibf, iseekb, 0);
    if(oseekb)
        seek(obf, oseekb, 0);
    while(skip) {
        read(ibf, ibuf, ibs);
        skip--;
    }

    ip = 0;
loop:
    if(ibc-- == 0) {
        ibc = 0;
        if(count==0 || nifr+nipr!=count) {
            if(cflag&(NERR|SYNC))
            for(ip=ibuf+ibs; ip>ibuf;)
                *--ip = 0;
            ibc = read(ibf, ibuf, ibs);
        }
        if(ibc == -1) {
            perror("read");
            if((cflag&NERR) == 0) {
                flsh();
                term("errors");
            }
            ibc = 0;
            for(c=0; c<ibs; c++)
                if(ibuf[c] != 0)
                    ibc = c+1;
            seek(ibf, ibs, 1);
            stats();
        }else if(ibc == 0 && --files<=0) {
            flsh();
            term(nil);
        }
        if(ibc != ibs) {
            nipr++;
            if(cflag&SYNC)
                ibc = ibs;
        } else
            nifr++;
        ip = ibuf;
        c = (ibc>>1) & ~1;
        if(cflag&SWAB && c)
        do {
            a = *ip++;
            ip[-1] = *ip;
            *ip++ = a;
        } while(--c);
        ip = ibuf;
        if(fflag) {
            obc = ibc;
            flsh();
            ibc = 0;
        }
        goto loop;
    }
    c = 0;
```

```
    c |= *ip++;
    c &= 0377;
    (*conv)(c);
    goto loop;
}

void
flsh(void)
{
    int c;

    if(obc) {
        /* don't perror dregs of previous errors on a short write */
        werrstr("");
        c = write(obf, obuf, obc);
        if(c != obc) {
            if(c > 0)
                ++nopr;
            perror("write");
            term("errors");
        }
        if(obc == obs)
            nofr++;
        else
            nopr++;
        obc = 0;
    }
}

int
match(char *s)
{
    char *cs;

    cs = string;
    while(*cs++ == *s)
        if(*s++ == '\0')
            goto true;
    if(*s != '\0')
        return 0;

true:
    cs--;
    string = cs;
    return 1;
}

vlong
number(vlong big)
{
    char *cs;
    uvlong n;

    cs = string;
    n = 0;
    while(*cs >= '0' && *cs <= '9')
        n = n*10 + *cs++ - '0';
    for(;;)
    switch(*cs++) {
```

```
    case 'k':
        n *= 1024;
        continue;

    case 'b':
        n *= 512;
        continue;

/*  case '*':*/
    case 'x':
        string = cs;
        n *= number(VBIG);

    case '\0':
        if(n > big) {
            fprint(2, "dd: argument %llud out of range\n", n);
            exits("range");
        }
        return n;
    }
    /* never gets here */
}

void
cnull(int cc)
{
    int c;

    c = cc;
    if((cflag&UCASE) && c>='a' && c<='z')
        c += 'A'-'a';
    if((cflag&LCASE) && c>='A' && c<='Z')
        c += 'a'-'A';
    null(c);
}

void
null(int c)
{

    *op = c;
    op++;
    if(++obc >= obs) {
        flsh();
        op = obuf;
    }
}

void
ascii(int cc)
{
    int c;

    c = etoa[cc];
    if(cbs == 0) {
        cnull(c);
        return;
    }
    if(c == ' ') {
        nspace++;
```

```
        goto out;
    }
    while(nspace > 0) {
        null(' ');
        nspace--;
    }
    cnull(c);

out:
    if(++cbc >= cbs) {
        null('\n');
        cbc = 0;
        nspace = 0;
    }
}

void
unblock(int cc)
{
    int c;

    c = cc & 0377;
    if(cbs == 0) {
        cnull(c);
        return;
    }
    if(c == ' ') {
        nspace++;
        goto out;
    }
    while(nspace > 0) {
        null(' ');
        nspace--;
    }
    cnull(c);

out:
    if(++cbc >= cbs) {
        null('\n');
        cbc = 0;
        nspace = 0;
    }
}

void
ebcdic(int cc)
{
    int c;

    c = cc;
    if(cflag&UCASE && c>='a' && c<='z')
        c += 'A'-'a';
    if(cflag&LCASE && c>='A' && c<='Z')
        c += 'a'-'A';
    c = atoe[c];
    if(cbs == 0) {
        null(c);
        return;
    }
    if(cc == '\n') {
```

```
        while(cbc < cbs) {
            null(atoe[' ']);
            cbc++;
        }
        cbc = 0;
        return;
    }
    if(cbc == cbs)
        ntrunc++;
    cbc++;
    if(cbc <= cbs)
        null(c);
}

void
ibm(int cc)
{
    int c;

    c = cc;
    if(cflag&UCASE && c>='a' && c<='z')
        c += 'A'-'a';
    if(cflag&LCASE && c>='A' && c<='Z')
        c += 'a'-'A';
    c = atoibm[c] & 0377;
    if(cbs == 0) {
        null(c);
        return;
    }
    if(cc == '\n') {
        while(cbc < cbs) {
            null(atoibm[' ']);
            cbc++;
        }
        cbc = 0;
        return;
    }
    if(cbc == cbs)
        ntrunc++;
    cbc++;
    if(cbc <= cbs)
        null(c);
}

void
block(int cc)
{
    int c;

    c = cc;
    if(cflag&UCASE && c>='a' && c<='z')
        c += 'A'-'a';
    if(cflag&LCASE && c>='A' && c<='Z')
        c += 'a'-'A';
    c &= 0377;
    if(cbs == 0) {
        null(c);
        return;
    }
    if(cc == '\n') {
```

```
        while(cbc < cbs) {
            null(' ');
            cbc++;
        }
        cbc = 0;
        return;
    }
    if(cbc == cbs)
        ntrunc++;
    cbc++;
    if(cbc <= cbs)
        null(c);
}

void
term(char *status)
{
    stats();
    exits(status);
}

void
stats(void)
{
    if(quiet)
        return;
    fprint(2, "%lud+%lud records in\n", nifr, nipr);
    fprint(2, "%lud+%lud records out\n", nofr, nopr);
    if(ntrunc)
        fprint(2, "%lud truncated records\n", ntrunc);
}

uchar   etoa[] =
{
    0000,0001,0002,0003,0234,0011,0206,0177,
    0227,0215,0216,0013,0014,0015,0016,0017,
    0020,0021,0022,0023,0235,0205,0010,0207,
    0030,0031,0222,0217,0034,0035,0036,0037,
    0200,0201,0202,0203,0204,0012,0027,0033,
    0210,0211,0212,0213,0214,0005,0006,0007,
    0220,0221,0026,0223,0224,0225,0226,0004,
    0230,0231,0232,0233,0024,0025,0236,0032,
    0040,0240,0241,0242,0243,0244,0245,0246,
    0247,0250,0133,0056,0074,0050,0053,0041,
    0046,0251,0252,0253,0254,0255,0256,0257,
    0260,0261,0135,0044,0052,0051,0073,0136,
    0055,0057,0262,0263,0264,0265,0266,0267,
    0270,0271,0174,0054,0045,0137,0076,0077,
    0272,0273,0274,0275,0276,0277,0300,0301,
    0302,0140,0072,0043,0100,0047,0075,0042,
    0303,0141,0142,0143,0144,0145,0146,0147,
    0150,0151,0304,0305,0306,0307,0310,0311,
    0312,0152,0153,0154,0155,0156,0157,0160,
    0161,0162,0313,0314,0315,0316,0317,0320,
    0321,0176,0163,0164,0165,0166,0167,0170,
    0171,0172,0322,0323,0324,0325,0326,0327,
    0330,0331,0332,0333,0334,0335,0336,0337,
    0340,0341,0342,0343,0344,0345,0346,0347,
    0173,0101,0102,0103,0104,0105,0106,0107,
    0110,0111,0350,0351,0352,0353,0354,0355,
```

```
    0175,0112,0113,0114,0115,0116,0117,0120,
    0121,0122,0356,0357,0360,0361,0362,0363,
    0134,0237,0123,0124,0125,0126,0127,0130,
    0131,0132,0364,0365,0366,0367,0370,0371,
    0060,0061,0062,0063,0064,0065,0066,0067,
    0070,0071,0372,0373,0374,0375,0376,0377,
};
uchar   atoe[] =
{
    0000,0001,0002,0003,0067,0055,0056,0057,
    0026,0005,0045,0013,0014,0015,0016,0017,
    0020,0021,0022,0023,0074,0075,0062,0046,
    0030,0031,0077,0047,0034,0035,0036,0037,
    0100,0117,0177,0173,0133,0154,0120,0175,
    0115,0135,0134,0116,0153,0140,0113,0141,
    0360,0361,0362,0363,0364,0365,0366,0367,
    0370,0371,0172,0136,0114,0176,0156,0157,
    0174,0301,0302,0303,0304,0305,0306,0307,
    0310,0311,0321,0322,0323,0324,0325,0326,
    0327,0330,0331,0342,0343,0344,0345,0346,
    0347,0350,0351,0112,0340,0132,0137,0155,
    0171,0201,0202,0203,0204,0205,0206,0207,
    0210,0211,0221,0222,0223,0224,0225,0226,
    0227,0230,0231,0242,0243,0244,0245,0246,
    0247,0250,0251,0300,0152,0320,0241,0007,
    0040,0041,0042,0043,0044,0025,0006,0027,
    0050,0051,0052,0053,0054,0011,0012,0033,
    0060,0061,0032,0063,0064,0065,0066,0010,
    0070,0071,0072,0073,0004,0024,0076,0341,
    0101,0102,0103,0104,0105,0106,0107,0110,
    0111,0121,0122,0123,0124,0125,0126,0127,
    0130,0131,0142,0143,0144,0145,0146,0147,
    0150,0151,0160,0161,0162,0163,0164,0165,
    0166,0167,0170,0200,0212,0213,0214,0215,
    0216,0217,0220,0232,0233,0234,0235,0236,
    0237,0240,0252,0253,0254,0255,0256,0257,
    0260,0261,0262,0263,0264,0265,0266,0267,
    0270,0271,0272,0273,0274,0275,0276,0277,
    0312,0313,0314,0315,0316,0317,0332,0333,
    0334,0335,0336,0337,0352,0353,0354,0355,
    0356,0357,0372,0373,0374,0375,0376,0377,
};
uchar   atoibm[] =
{
    0000,0001,0002,0003,0067,0055,0056,0057,
    0026,0005,0045,0013,0014,0015,0016,0017,
    0020,0021,0022,0023,0074,0075,0062,0046,
    0030,0031,0077,0047,0034,0035,0036,0037,
    0100,0132,0177,0173,0133,0154,0120,0175,
    0115,0135,0134,0116,0153,0140,0113,0141,
    0360,0361,0362,0363,0364,0365,0366,0367,
    0370,0371,0172,0136,0114,0176,0156,0157,
    0174,0301,0302,0303,0304,0305,0306,0307,
    0310,0311,0321,0322,0323,0324,0325,0326,
    0327,0330,0331,0342,0343,0344,0345,0346,
    0347,0350,0351,0255,0340,0275,0137,0155,
    0171,0201,0202,0203,0204,0205,0206,0207,
    0210,0211,0221,0222,0223,0224,0225,0226,
    0227,0230,0231,0242,0243,0244,0245,0246,
    0247,0250,0251,0300,0117,0320,0241,0007,
```

```
    0040,0041,0042,0043,0044,0025,0006,0027,
    0050,0051,0052,0053,0054,0011,0012,0033,
    0060,0061,0032,0063,0064,0065,0066,0010,
    0070,0071,0072,0073,0004,0024,0076,0341,
    0101,0102,0103,0104,0105,0106,0107,0110,
    0111,0121,0122,0123,0124,0125,0126,0127,
    0130,0131,0142,0143,0144,0145,0146,0147,
    0150,0151,0160,0161,0162,0163,0164,0165,
    0166,0167,0170,0200,0212,0213,0214,0215,
    0216,0217,0220,0232,0233,0234,0235,0236,
    0237,0240,0252,0253,0254,0255,0256,0257,
    0260,0261,0262,0263,0264,0265,0266,0267,
    0270,0271,0272,0273,0274,0275,0276,0277,
    0312,0313,0314,0315,0316,0317,0332,0333,
    0334,0335,0336,0337,0352,0353,0354,0355,
    0356,0357,0372,0373,0374,0375,0376,0377,
};
```

A.3 compare/

A.3.1 compare/diff/

⟨diff/diff.h 204a⟩≡

```
char mode;          /* '\0', 'e', 'f', 'h' */
char bflag;         /* ignore multiple and trailing blanks */
char rflag;         /* recurse down directory trees */
char mflag;         /* pseudo flag: doing multiple files, one dir */
int anychange;
extern Biobuf   stdout;
extern int  binary;

#define MALLOC(t, n)        ((t *)emalloc((n)*sizeof(t)))
#define REALLOC(p, t, n)    ((t *)erealloc((void *)(p), (n)*sizeof(t)))
#define FREE(p)         free((void *)(p))

#define MAXPATHLEN  1024

int mkpathname(char *, char *, char *);
void *emalloc(unsigned);
void *erealloc(void *, unsigned);
void diff(char *, char *, int);
void diffdir(char *, char *, int);
void diffreg(char *, char *);
Biobuf *prepare(int, char *);
void panic(int, char *, ...);
void check(Biobuf *, Biobuf *);
void change(int, int, int, int);
void flushchanges(void);
```

⟨diff/main.c 204b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include "diff.h"

#define DIRECTORY(s)        ((s)->qid.type&QTDIR)
#define REGULAR_FILE(s)     ((s)->type == 'M' && !DIRECTORY(s))
```

```
Biobuf  stdout;

static char *tmp[] = {"/tmp/diff1XXXXXXXXXX", "/tmp/diff2XXXXXXXXXX"};
static int whichtmp;
static char *progname;
static char usage[] = "diff [-abcefmnrw] file1 ... file2\n";

static void
rmtmpfiles(void)
{
    while (whichtmp > 0) {
        whichtmp--;
        remove(tmp[whichtmp]);
    }
}

void
done(int status)
{
    rmtmpfiles();
    switch(status)
    {
    case 0:
        exits("");
    case 1:
        exits("some");
    default:
        exits("error");
    }
    /*NOTREACHED*/
}

void
panic(int status, char *fmt, ...)
{
    va_list arg;

    Bflush(&stdout);

    fprint(2, "%s: ", progname);
    va_start(arg, fmt);
    vfprint(2, fmt, arg);
    va_end(arg);
    if (status)
        done(status);
        /*NOTREACHED*/
}

static int
catch(void *a, char *msg)
{
    USED(a);
    panic(2, msg);
    return 1;
}

int
mkpathname(char *pathname, char *path, char *name)
{
```

```
    if (strlen(path) + strlen(name) > MAXPATHLEN) {
        panic(0, "pathname %s/%s too long\n", path, name);
        return 1;
    }
    sprint(pathname, "%s/%s", path, name);
    return 0;
}

static char *
mktmpfile(int input, Dir **sb)
{
    int fd, i;
    char *p;
    char buf[8192];

    atnotify(catch, 1);
    p = mktemp(tmp[whichtmp++]);
    fd = create(p, OWRITE, 0600);
    if (fd < 0) {
        panic(mflag ? 0: 2, "cannot create %s: %r\n", p);
        return 0;
    }
    while ((i = read(input, buf, sizeof(buf))) > 0) {
        if ((i = write(fd, buf, i)) < 0)
            break;
    }
    *sb = dirfstat(fd);
    close(fd);
    if (i < 0) {
        panic(mflag ? 0: 2, "cannot read/write %s: %r\n", p);
        return 0;
    }
    return p;
}

static char *
statfile(char *file, Dir **sb)
{
    Dir *dir;
    int input;

    dir = dirstat(file);
    if(dir == nil) {
        if (strcmp(file, "-") || (dir = dirfstat(0)) == nil) {
            panic(mflag ? 0: 2, "cannot stat %s: %r\n", file);
            return 0;
        }
        free(dir);
        return mktmpfile(0, sb);
    }
    else if (!REGULAR_FILE(dir) && !DIRECTORY(dir)) {
        free(dir);
        if ((input = open(file, OREAD)) == -1) {
            panic(mflag ? 0: 2, "cannot open %s: %r\n", file);
            return 0;
        }
        file = mktmpfile(input, sb);
        close(input);
    }
    else
```

```
        *sb = dir;
    return file;
}

void
diff(char *f, char *t, int level)
{
    char *fp, *tp, *p, fb[MAXPATHLEN+1], tb[MAXPATHLEN+1];
    Dir *fsb, *tsb;

    if ((fp = statfile(f, &fsb)) == 0)
        goto Return;
    if ((tp = statfile(t, &tsb)) == 0){
        free(fsb);
        goto Return;
    }
    if (DIRECTORY(fsb) && DIRECTORY(tsb)) {
        if (rflag || level == 0)
            diffdir(fp, tp, level);
        else
            Bprint(&stdout, "Common subdirectories: %s and %s\n",
                fp, tp);
    }
    else if (REGULAR_FILE(fsb) && REGULAR_FILE(tsb))
        diffreg(fp, tp);
    else {
        if (REGULAR_FILE(fsb)) {
            if ((p = utfrrune(f, '/')) == 0)
                p = f;
            else
                p++;
            if (mkpathname(tb, tp, p) == 0)
                diffreg(fp, tb);
        }
        else {
            if ((p = utfrrune(t, '/')) == 0)
                p = t;
            else
                p++;
            if (mkpathname(fb, fp, p) == 0)
                diffreg(fb, tp);
        }
    }
    free(fsb);
    free(tsb);
Return:
    rmtmpfiles();
}

void
main(int argc, char *argv[])
{
    char *p;
    int i;
    Dir *fsb, *tsb;

    Binit(&stdout, 1, OWRITE);
    progname = argv0 = *argv;
    while (--argc && (*++argv)[0] == '-' && (*argv)[1]) {
        for (p = *argv+1; *p; p++) {
```

```
            switch (*p) {

            case 'e':
            case 'f':
            case 'n':
            case 'c':
            case 'a':
                mode = *p;
                break;

            case 'w':
                bflag = 2;
                break;

            case 'b':
                bflag = 1;
                break;

            case 'r':
                rflag = 1;
                break;

            case 'm':
                mflag = 1;
                break;

            case 'h':
            default:
                progname = "Usage";
                panic(2, usage);
            }
        }
    }
    if (argc < 2)
        panic(2, usage, progname);
    if ((tsb = dirstat(argv[argc-1])) == nil)
        panic(2, "can't stat %s\n", argv[argc-1]);
    if (argc > 2) {
        if (!DIRECTORY(tsb))
            panic(2, usage, progname);
        mflag = 1;
    }
    else {
        if ((fsb = dirstat(argv[0])) == nil)
            panic(2, "can't stat %s\n", argv[0]);
        if (DIRECTORY(fsb) && DIRECTORY(tsb))
            mflag = 1;
        free(fsb);
    }
    free(tsb);
    for (i = 0; i < argc-1; i++)
        diff(argv[i], argv[argc-1], 0);
    done(anychange);
    /*NOTREACHED*/
}

static char noroom[] = "out of memory - try diff -h\n";

void *
emalloc(unsigned n)
```

```
{
    register void *p;

    if ((p = malloc(n)) == 0)
        panic(2, noroom);
    return p;
}

void *
erealloc(void *p, unsigned n)
{
    register void *rp;

    if ((rp = realloc(p, n)) == 0)
        panic(2, noroom);
    return rp;
}
```

⟨diff/diffdir.c 209⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include "diff.h"

static int
itemcmp(void *v1, void *v2)
{
    char **d1 = v1, **d2 = v2;

    return strcmp(*d1, *d2);
}

static char **
scandir(char *name)
{
    char **cp;
    Dir *db;
    int nitems;
    int fd, n;

    if ((fd = open(name, OREAD)) < 0) {
        fprint(2, "%s: can't open %s: %r\n", argv0, name);
        /* fake an empty directory */
        cp = MALLOC(char*, 1);
        cp[0] = 0;
        return cp;
    }
    cp = 0;
    nitems = 0;
    if((n = dirreadall(fd, &db)) > 0){
        while (n--) {
            cp = REALLOC(cp, char *, (nitems+1));
            cp[nitems] = MALLOC(char, strlen((db+n)->name)+1);
            strcpy(cp[nitems], (db+n)->name);
            nitems++;
        }
        free(db);
    }
    cp = REALLOC(cp, char*, (nitems+1));
    cp[nitems] = 0;
```

```
    close(fd);
    qsort((char *)cp, nitems, sizeof(char*), itemcmp);
    return cp;
}

static int
isdotordotdot(char *p)
{
    if (*p == '.') {
        if (!p[1])
            return 1;
        if (p[1] == '.' && !p[2])
            return 1;
    }
    return 0;
}

void
diffdir(char *f, char *t, int level)
{
    char  **df, **dt, **dirf, **dirt;
    char *from, *to;
    int res;
    char fb[MAXPATHLEN+1], tb[MAXPATHLEN+1];

    df = scandir(f);
    dt = scandir(t);
    dirf = df;
    dirt = dt;
    while (*df || *dt) {
        from = *df;
        to = *dt;
        if (from && isdotordotdot(from)) {
            df++;
            continue;
        }
        if (to && isdotordotdot(to)) {
            dt++;
            continue;
        }
        if (!from)
            res = 1;
        else if (!to)
            res = -1;
        else
            res = strcmp(from, to);
        if (res < 0) {
            if (mode == 0 || mode == 'n')
                Bprint(&stdout, "Only in %s: %s\n", f, from);
            df++;
            continue;
        }
        if (res > 0) {
            if (mode == 0 || mode == 'n')
                Bprint(&stdout, "Only in %s: %s\n", t, to);
            dt++;
            continue;
        }
        if (mkpathname(fb, f, from))
            continue;
```

```
        if (mkpathname(tb, t, to))
            continue;
        diff(fb, tb, level+1);
        df++; dt++;
    }
    for (df = dirf; *df; df++)
        FREE(*df);
    for (dt = dirt; *dt; dt++)
        FREE(*dt);
    FREE(dirf);
    FREE(dirt);
}
```

⟨diff/diffio.c 211⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include "diff.h"

struct line {
    int serial;
    int value;
};
extern struct line *file[2];
extern int len[2];
extern long *ixold, *ixnew;
extern int *J;

static Biobuf *input[2];
static char *file1, *file2;
static int firstchange;

#define MAXLINELEN  4096
#define MIN(x, y)   ((x) < (y) ? (x): (y))

static int
readline(Biobuf *bp, char *buf)
{
    int c;
    char *p, *e;

    p = buf;
    e = p + MAXLINELEN-1;
    do {
        c = Bgetc(bp);
        if (c < 0) {
            if (p == buf)
                return -1;
            break;
        }
        if (c == '\n')
            break;
        *p++ = c;
    } while (p < e);
    *p = 0;
    if (c != '\n' && c >= 0) {
        do c = Bgetc(bp);
        while (c >= 0 && c != '\n');
    }
```

```
    return p - buf;
}

#define HALFLONG 16
#define low(x)  (x&((1L<<HALFLONG)-1))
#define high(x) (x>>HALFLONG)

/*
 * hashing has the effect of
 * arranging line in 7-bit bytes and then
 * summing 1-s complement in 16-bit hunks
 */
static int
readhash(Biobuf *bp, char *buf)
{
    long sum;
    unsigned shift;
    char *p;
    int len, space;

    sum = 1;
    shift = 0;
    if ((len = readline(bp, buf)) == -1)
        return 0;
    p = buf;
    switch(bflag)   /* various types of white space handling */
    {
    case 0:
        while (len--) {
            sum += (long)*p++ << (shift &= (HALFLONG-1));
            shift += 7;
        }
        break;
    case 1:
        /*
         * coalesce multiple white-space
         */
        for (space = 0; len--; p++) {
            if (isspace(*p)) {
                space++;
                continue;
            }
            if (space) {
                shift += 7;
                space = 0;
            }
            sum += (long)*p << (shift &= (HALFLONG-1));
            shift += 7;
        }
        break;
    default:
        /*
         * strip all white-space
         */
        while (len--) {
            if (isspace(*p)) {
                p++;
                continue;
            }
            sum += (long)*p++ << (shift &= (HALFLONG-1));
```

```
            shift += 7;
        }
        break;
    }
    sum = low(sum) + high(sum);
    return ((short)low(sum) + (short)high(sum));
}

Biobuf *
prepare(int i, char *arg)
{
    struct line *p;
    int j, h;
    Biobuf *bp;
    char *cp, buf[MAXLINELEN];
    int nbytes;
    Rune r;

    bp = Bopen(arg, OREAD);
    if (!bp) {
        panic(mflag ? 0: 2, "cannot open %s: %r\n", arg);
        return 0;
    }
    if (binary)
        return bp;
    nbytes = Bread(bp, buf, MIN(1024, MAXLINELEN));
    if (nbytes > 0) {
        cp = buf;
        while (cp < buf+nbytes-UTFmax) {
            /*
             * heuristic for a binary file in the
             * brave new UNICODE world
             */
            cp += chartorune(&r, cp);
            if (r == 0 || (r > 0x7f && r <= 0xa0)) {
                binary++;
                return bp;
            }
        }
        Bseek(bp, 0, 0);
    }
    p = MALLOC(struct line, 3);
    for (j = 0; h = readhash(bp, buf); p[j].value = h)
        p = REALLOC(p, struct line, (++j+3));
    len[i] = j;
    file[i] = p;
    input[i] = bp;          /*fix*/
    if (i == 0) {           /*fix*/
        file1 = arg;
        firstchange = 0;
    }
    else
        file2 = arg;
    return bp;
}

static int
squishspace(char *buf)
{
    char *p, *q;
```

```
    int space;

    for (space = 0, q = p = buf; *q; q++) {
        if (isspace(*q)) {
            space++;
            continue;
        }
        if (space && bflag == 1) {
            *p++ = ' ';
            space = 0;
        }
        *p++ = *q;
    }
    *p = 0;
    return p - buf;
}

/*
 * need to fix up for unexpected EOF's
 */
void
check(Biobuf *bf, Biobuf *bt)
{
    int f, t, flen, tlen;
    char fbuf[MAXLINELEN], tbuf[MAXLINELEN];

    ixold[0] = ixnew[0] = 0;
    for (f = t = 1; f < len[0]; f++) {
        flen = readline(bf, fbuf);
        ixold[f] = ixold[f-1] + flen + 1;       /* ftell(bf) */
        if (J[f] == 0)
            continue;
        do {
            tlen = readline(bt, tbuf);
            ixnew[t] = ixnew[t-1] + tlen + 1;   /* ftell(bt) */
        } while (t++ < J[f]);
        if (bflag) {
            flen = squishspace(fbuf);
            tlen = squishspace(tbuf);
        }
        if (flen != tlen || strcmp(fbuf, tbuf))
            J[f] = 0;
    }
    while (t < len[1]) {
        tlen = readline(bt, tbuf);
        ixnew[t] = ixnew[t-1] + tlen + 1;   /* fseek(bt) */
        t++;
    }
}

static void
range(int a, int b, char *separator)
{
    Bprint(&stdout, "%d", a > b ? b: a);
    if (a < b)
        Bprint(&stdout, "%s%d", separator, b);
}

static void
fetch(long *f, int a, int b, Biobuf *bp, char *s)
```

```
{
    char buf[MAXLINELEN];
    int maxb;

    if(a <= 1)
        a = 1;
    if(bp == input[0])
        maxb = len[0];
    else
        maxb = len[1];
    if(b > maxb)
        b = maxb;
    if(a > maxb)
        return;
    Bseek(bp, f[a-1], 0);
    while (a++ <= b) {
        readline(bp, buf);
        Bprint(&stdout, "%s%s\n", s, buf);
    }
}

typedef struct Change Change;
struct Change
{
    int a;
    int b;
    int c;
    int d;
};

Change *changes;
int nchanges;

void
change(int a, int b, int c, int d)
{
    char verb;
    char buf[4];
    Change *ch;

    if (a > b && c > d)
        return;
    anychange = 1;
    if (mflag && firstchange == 0) {
        if(mode) {
            buf[0] = '-';
            buf[1] = mode;
            buf[2] = ' ';
            buf[3] = '\0';
        } else {
            buf[0] = '\0';
        }
        Bprint(&stdout, "diff %s%s %s\n", buf, file1, file2);
        firstchange = 1;
    }
    verb = a > b ? 'a': c > d ? 'd': 'c';
    switch(mode) {
    case 'e':
        range(a, b, ",");
        Bputc(&stdout, verb);
```

```
        break;
    case 0:
        range(a, b, ",");
        Bputc(&stdout, verb);
        range(c, d, ",");
        break;
    case 'n':
        Bprint(&stdout, "%s:", file1);
        range(a, b, ",");
        Bprint(&stdout, " %c ", verb);
        Bprint(&stdout, "%s:", file2);
        range(c, d, ",");
        break;
    case 'f':
        Bputc(&stdout, verb);
        range(a, b, " ");
        break;
    case 'c':
    case 'a':
        if(nchanges%1024 == 0)
            changes = erealloc(changes, (nchanges+1024)*sizeof(changes[0]));
        ch = &changes[nchanges++];
        ch->a = a;
        ch->b = b;
        ch->c = c;
        ch->d = d;
        return;
    }
    Bputc(&stdout, '\n');
    if (mode == 0 || mode == 'n') {
        fetch(ixold, a, b, input[0], "< ");
        if (a <= b && c <= d)
            Bprint(&stdout, "---\n");
    }
    fetch(ixnew, c, d, input[1], mode == 0 || mode == 'n' ? "> ": "");
    if (mode != 0 && mode != 'n' && c <= d)
        Bprint(&stdout, ".\n");
}

enum
{
    Lines = 3,  /* number of lines of context shown */
};

int
changeset(int i)
{
    while(i<nchanges && changes[i].b+1+2*Lines > changes[i+1].a)
        i++;
    if(i<nchanges)
        return i+1;
    return nchanges;
}

void
flushchanges(void)
{
    int a, b, c, d, at;
    int i, j;
```

```
    if(nchanges == 0)
        return;

    for(i=0; i<nchanges; ){
        j = changeset(i);
        a = changes[i].a-Lines;
        b = changes[j-1].b+Lines;
        c = changes[i].c-Lines;
        d = changes[j-1].d+Lines;
        if(a < 1)
            a = 1;
        if(c < 1)
            c = 1;
        if(b > len[0])
            b = len[0];
        if(d > len[1])
            d = len[1];
        if(mode == 'a'){
            a = 1;
            b = len[0];
            c = 1;
            d = len[1];
            j = nchanges;
        }
        Bprint(&stdout, "%s:", file1);
        range(a, b, ",");
        Bprint(&stdout, " - ");
        Bprint(&stdout, "%s:", file2);
        range(c, d, ",");
        Bputc(&stdout, '\n');
        at = a;
        for(; i<j; i++){
            fetch(ixold, at, changes[i].a-1, input[0], "  ");
            fetch(ixold, changes[i].a, changes[i].b, input[0], "- ");
            fetch(ixnew, changes[i].c, changes[i].d, input[1], "+ ");
            at = changes[i].b+1;
        }
        fetch(ixold, at, b, input[0], "  ");
    }
    nchanges = 0;
}
```

⟨diff/diffreg.c 217⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include "diff.h"

/*  diff - differential file comparison
*
*   Uses an algorithm due to Harold Stone, which finds
*   a pair of longest identical subsequences in the two
*   files.
*
*   The major goal is to generate the match vector J.
*   J[i] is the index of the line in file1 corresponding
*   to line i file0. J[i] = 0 if there is no
*   such line in file1.
*
*   Lines are hashed so as to work in core. All potential
```

```
*   matches are located by sorting the lines of each file
*   on the hash (called value). In particular, this
*   collects the equivalence classes in file1 together.
*   Subroutine equiv replaces the value of each line in
*   file0 by the index of the first element of its
*   matching equivalence in (the reordered) file1.
*   To save space equiv squeezes file1 into a single
*   array member in which the equivalence classes
*   are simply concatenated, except that their first
*   members are flagged by changing sign.
*
*   Next the indices that point into member are unsorted into
*   array class according to the original order of file0.
*
*   The cleverness lies in routine stone. This marches
*   through the lines of file0, developing a vector klist
*   of "k-candidates". At step i a k-candidate is a matched
*   pair of lines x,y (x in file0 y in file1) such that
*   there is a common subsequence of lenght k
*   between the first i lines of file0 and the first y
*   lines of file1, but there is no such subsequence for
*   any smaller y. x is the earliest possible mate to y
*   that occurs in such a subsequence.
*
*   Whenever any of the members of the equivalence class of
*   lines in file1 matable to a line in file0 has serial number
*   less than the y of some k-candidate, that k-candidate
*   with the smallest such y is replaced. The new
*   k-candidate is chained (via pred) to the current
*   k-1 candidate so that the actual subsequence can
*   be recovered. When a member has serial number greater
*   that the y of all k-candidates, the klist is extended.
*   At the end, the longest subsequence is pulled out
*   and placed in the array J by unravel.
*
*   With J in hand, the matches there recorded are
*   check'ed against reality to assure that no spurious
*   matches have crept in due to hashing. If they have,
*   they are broken, and "jackpot " is recorded--a harmless
*   matter except that a true match for a spuriously
*   mated line may now be unnecessarily reported as a change.
*
*   Much of the complexity of the program comes simply
*   from trying to minimize core utilization and
*   maximize the range of doable problems by dynamically
*   allocating what is needed and reusing what is not.
*   The core requirements for problems larger than somewhat
*   are (in words) 2*length(file0) + length(file1) +
*   3*(number of k-candidates installed),  typically about
*   6n words for files of length n.
*/
/* TIDY THIS UP */
struct cand {
    int x;
    int y;
    int pred;
} cand;
struct line {
    int serial;
    int value;
```

```
} *file[2], line;
int len[2];
int binary;
struct line *sfile[2];  /*shortened by pruning common prefix and suffix*/
int slen[2];
int pref, suff; /*length of prefix and suffix*/
int *class; /*will be overlaid on file[0]*/
int *member;    /*will be overlaid on file[1]*/
int *klist;     /*will be overlaid on file[0] after class*/
struct cand *clist; /* merely a free storage pot for candidates */
int clen;
int *J;     /*will be overlaid on class*/
long *ixold;    /*will be overlaid on klist*/
long *ixnew;    /*will be overlaid on file[1]*/
/* END OF SOME TIDYING */

static void
sort(struct line *a, int n) /*shellsort CACM #201*/
{
    int m;
    struct line *ai, *aim, *j, *k;
    struct line w;
    int i;

    m = 0;
    for (i = 1; i <= n; i *= 2)
        m = 2*i - 1;
    for (m /= 2; m != 0; m /= 2) {
        k = a+(n-m);
        for (j = a+1; j <= k; j++) {
            ai = j;
            aim = ai+m;
            do {
                if (aim->value > ai->value ||
                   aim->value == ai->value &&
                   aim->serial > ai->serial)
                    break;
                w = *ai;
                *ai = *aim;
                *aim = w;

                aim = ai;
                ai -= m;
            } while (ai > a && aim >= ai);
        }
    }
}

static void
unsort(struct line *f, int l, int *b)
{
    int *a;
    int i;

    a = MALLOC(int, (l+1));
    for(i=1;i<=l;i++)
        a[f[i].serial] = f[i].value;
    for(i=1;i<=l;i++)
        b[i] = a[i];
    FREE(a);
```

```
}

static void
prune(void)
{
    int i,j;

    for(pref=0;pref<len[0]&&pref<len[1]&&
        file[0][pref+1].value==file[1][pref+1].value;
        pref++ ) ;
    for(suff=0;suff<len[0]-pref&&suff<len[1]-pref&&
        file[0][len[0]-suff].value==file[1][len[1]-suff].value;
        suff++) ;
    for(j=0;j<2;j++) {
        sfile[j] = file[j]+pref;
        slen[j] = len[j]-pref-suff;
        for(i=0;i<=slen[j];i++)
            sfile[j][i].serial = i;
    }
}

static void
equiv(struct line *a, int n, struct line *b, int m, int *c)
{
    int i, j;

    i = j = 1;
    while(i<=n && j<=m) {
        if(a[i].value < b[j].value)
            a[i++].value = 0;
        else if(a[i].value == b[j].value)
            a[i++].value = j;
        else
            j++;
    }
    while(i <= n)
        a[i++].value = 0;
    b[m+1].value = 0;
    j = 0;
    while(++j <= m) {
        c[j] = -b[j].serial;
        while(b[j+1].value == b[j].value) {
            j++;
            c[j] = b[j].serial;
        }
    }
    c[j] = -1;
}

static int
newcand(int x, int  y, int pred)
{
    struct cand *q;

    clist = REALLOC(clist, struct cand, (clen+1));
    q = clist + clen;
    q->x = x;
    q->y = y;
    q->pred = pred;
    return clen++;
```

```
}

static int
search(int *c, int k, int y)
{
    int i, j, l;
    int t;

    if(clist[c[k]].y < y)   /*quick look for typical case*/
        return k+1;
    i = 0;
    j = k+1;
    while((l=(i+j)/2) > i) {
        t = clist[c[l]].y;
        if(t > y)
            j = l;
        else if(t < y)
            i = l;
        else
            return l;
    }
    return l+1;
}

static int
stone(int *a, int n, int *b, int *c)
{
    int i, k,y;
    int j, l;
    int oldc, tc;
    int oldl;

    k = 0;
    c[0] = newcand(0,0,0);
    for(i=1; i<=n; i++) {
        j = a[i];
        if(j==0)
            continue;
        y = -b[j];
        oldl = 0;
        oldc = c[0];
        do {
            if(y <= clist[oldc].y)
                continue;
            l = search(c, k, y);
            if(l!=oldl+1)
                oldc = c[l-1];
            if(l<=k) {
                if(clist[c[l]].y <= y)
                    continue;
                tc = c[l];
                c[l] = newcand(i,y,oldc);
                oldc = tc;
                oldl = l;
            } else {
                c[l] = newcand(i,y,oldc);
                k++;
                break;
            }
        } while((y=b[++j]) > 0);
```

```
    }
    return k;
}

static void
unravel(int p)
{
    int i;
    struct cand *q;

    for(i=0; i<=len[0]; i++) {
        if (i <= pref)
            J[i] = i;
        else if (i > len[0]-suff)
            J[i] = i+len[1]-len[0];
        else
            J[i] = 0;
    }
    for(q=clist+p;q->y!=0;q=clist+q->pred)
        J[q->x+pref] = q->y+pref;
}

static void
output(void)
{
    int m, i0, i1, j0, j1;

    m = len[0];
    J[0] = 0;
    J[m+1] = len[1]+1;
    if (mode != 'e') {
        for (i0 = 1; i0 <= m; i0 = i1+1) {
            while (i0 <= m && J[i0] == J[i0-1]+1)
                i0++;
            j0 = J[i0-1]+1;
            i1 = i0-1;
            while (i1 < m && J[i1+1] == 0)
                i1++;
            j1 = J[i1+1]-1;
            J[i1] = j1;
            change(i0, i1, j0, j1);
        }
    }
    else {
        for (i0 = m; i0 >= 1; i0 = i1-1) {
            while (i0 >= 1 && J[i0] == J[i0+1]-1 && J[i0])
                i0--;
            j0 = J[i0+1]-1;
            i1 = i0+1;
            while (i1 > 1 && J[i1-1] == 0)
                i1--;
            j1 = J[i1-1]+1;
            J[i1] = j1;
            change(i1 , i0, j1, j0);
        }
    }
    if (m == 0)
        change(1, 0, 1, len[1]);
    flushchanges();
}
```

```
#define BUF 4096
static int
cmp(Biobuf* b1, Biobuf* b2)
{
    int n;
    uchar buf1[BUF], buf2[BUF];
    int f1, f2;
    vlong nc = 1;
    uchar *b1s, *b1e, *b2s, *b2e;

    f1 = Bfildes(b1);
    f2 = Bfildes(b2);
    seek(f1, 0, 0);
    seek(f2, 0, 0);
    b1s = b1e = buf1;
    b2s = b2e = buf2;
    for(;;){
        if(b1s >= b1e){
            if(b1s >= &buf1[BUF])
                b1s = buf1;
            n = read(f1, b1s,  &buf1[BUF] - b1s);
            b1e = b1s + n;
        }
        if(b2s >= b2e){
            if(b2s >= &buf2[BUF])
                b2s = buf2;
            n = read(f2, b2s,  &buf2[BUF] - b2s);
            b2e = b2s + n;
        }
        n = b2e - b2s;
        if(n > b1e - b1s)
            n = b1e - b1s;
        if(n <= 0)
            break;
        if(memcmp((void *)b1s, (void *)b2s, n) != 0){
            return 1;
        }
        nc += n;
        b1s += n;
        b2s += n;
    }
    if(b1e - b1s == b2e - b2s)
        return 0;
    return 1;
}

void
diffreg(char *f, char *t)
{
    Biobuf *b0, *b1;
    int k;

    binary = 0;
    b0 = prepare(0, f);
    if (!b0)
        return;
    b1 = prepare(1, t);
    if (!b1) {
        Bterm(b0);
```

```
        return;
    }
    if (binary){
        // could use b0 and b1 but this is simpler.
        if (cmp(b0, b1))
            print("binary files %s %s differ\n", f, t);
        Bterm(b0);
        Bterm(b1);
        return;
    }
    clen = 0;
    prune();
    sort(sfile[0], slen[0]);
    sort(sfile[1], slen[1]);

    member = (int *)file[1];
    equiv(sfile[0], slen[0], sfile[1], slen[1], member);
    member = REALLOC(member, int, slen[1]+2);

    class = (int *)file[0];
    unsort(sfile[0], slen[0], class);
    class = REALLOC(class, int, slen[0]+2);

    klist = MALLOC(int, slen[0]+2);
    clist = MALLOC(struct cand, 1);
    k = stone(class, slen[0], member, klist);
    FREE(member);
    FREE(class);

    J = MALLOC(int, len[0]+2);
    unravel(klist[k]);
    FREE(clist);
    FREE(klist);

    ixold = MALLOC(long, len[0]+2);
    ixnew = MALLOC(long, len[1]+2);
    Bseek(b0, 0, 0); Bseek(b1, 0, 0);
    check(b0, b1);
    output();
    FREE(J); FREE(ixold); FREE(ixnew);
    Bterm(b0); Bterm(b1);           /* ++++ */
}
```

A.3.2 compare/cmp.c

⟨compare/cmp.c 224⟩≡

```
⟨plan9 includes 14⟩
#include <ctype.h>

⟨constant BUF(cmp.c) 77c⟩

⟨global flags (cmp.c) 77b⟩

// forward decl
static void usage(void);

⟨function seekoff(cmp.c) 80a⟩

⟨function main(cmp.c) 78b⟩
```

⟨*function* usage*(cmp.c)* 77a⟩

A.3.3 compare/comm.c

⟨compare/comm.c 225a⟩≡
```
⟨plan9 includes 14⟩
#include <bio.h>

⟨constant LB(comm.c) 81a⟩
⟨global flags comm.c 80c⟩
⟨globals comm.c 80d⟩

// forward decls
Biobuf* openfil(char*);
int     rd(Biobuf*, char*);
void    wr(char*, int);
void    copy(Biobuf*, char*, int);
int     compare(char*, char*);

⟨function main(comm.c) 81b⟩

⟨function rd(comm.c) 83a⟩
⟨function wr(comm.c) 83c⟩
⟨function copy(comm.c) 83d⟩
⟨function compare(comm.c) 83b⟩
⟨function openfil(comm.c) 82⟩
```

A.4 pipe/

A.4.1 pipe/mc.c

⟨pipe/mc.c 225b⟩≡
```
/*
 * mc - columnate
 *
 * mc[-][-LINEWIDTH][-t][file...]
 *  - causes break on colon
 *  -LINEWIDTH sets width of line in which to columnate(default 80)
 *  -t suppresses expanding multiple blanks into tabs
 *
 */
⟨plan9 includes 14⟩
#include    <bio.h>

#ifdef Unix
char* font;
#else
#include    <draw.h>
Font *font;
#endif

⟨constants mc.c 116a⟩
⟨global flags mc.c 116c⟩
⟨globals mc.c 116b⟩

// forward decls
```

```
void getwidth(void), readbuf(int), error(char *);
void scanwords(void), columnate(void), morechars(void);
int wordwidth(Rune*, int);
int nexttab(int);

⟨function main(mc.c) 117⟩

⟨function error(mc.c) 118a⟩
⟨function readbuf(mc.c) 118c⟩
⟨function scanwords(mc.c) 120b⟩
⟨function columnate(mc.c) 119e⟩
⟨function nexttab(mc.c) 121b⟩
⟨function morechars(mc.c) 119a⟩

#ifdef Unix
⟨function getwidth(mc.c)(unix) 119b⟩
⟨function wordwidth(mc.c)(unix) 119c⟩
#else
int
wordwidth(Rune *w, int nw)
{
    if(font)
        return runestringnwidth(font, w, nw);
    return nw;
}

/*
 * These routines discover the width of the display.
 * It takes some work.  If we do the easy calls to the
 * draw library, the screen flashes due to repainting
 * when mc exits.
 */

jmp_buf drawjmp;

void
terror(Display*, char*)
{
    longjmp(drawjmp, 1);
}

void
getwidth(void)
{
    int n, fd;
    char buf[128], *f[10], *p;

    if(access("/dev/acme", OREAD) >= 0){
        if((fd = open("/dev/acme/ctl", OREAD)) < 0)
            return;
        n = read(fd, buf, sizeof buf-1);
        close(fd);
        if(n <= 0)
            return;
        buf[n] = 0;
        n = tokenize(buf, f, nelem(f));
        if(n < 7)
            return;
        if((font = openfont(nil, f[6])) == nil)
            return;
```

```
        if(n >= 8)
            tabwidth = atoi(f[7]);
        else
            tabwidth = 4*stringwidth(font, "0");
        mintab = stringwidth(font, "0");
        linewidth = atoi(f[5]);
        tabflag = 1;
        return;
    }

    if((p = getenv("font")) == nil)
        return;
    if((font = openfont(nil, p)) == nil)
        return;
    if((fd = open("/dev/window", OREAD)) < 0){
        font = nil;
        return;
    }
    n = read(fd, buf, 5*12);
    close(fd);
    if(n < 5*12){
        font = nil;
        return;
    }
    buf[n] = 0;

    /* window stucture:
        4 bit left edge
        1 bit gap
        12 bit scrollbar
        4 bit gap
        text
        4 bit right edge
    */
    linewidth = atoi(buf+3*12) - atoi(buf+1*12) - (4+1+12+4+4);
    mintab = stringwidth(font, "0");
    if((p = getenv("tabstop")) != nil)
        tabwidth = atoi(p)*stringwidth(font, "0");
    if(tabwidth == 0)
        tabwidth = 4*stringwidth(font, "0");
    tabflag = 1;
}
#endif
```

A.4.2 pipe/p.c

⟨pipe/p.c 227⟩≡

```
⟨plan9 includes 14⟩
#include <bio.h>

⟨constant DEF(p.c) 114b⟩
⟨globals p.c 114c⟩

// forward decls
void printfile(fdt);

⟨function main(p.c) 115a⟩

⟨function printfile(p.c) 115b⟩
```

A.4.3 pipe/tail.c

⟨pipe/tail.c 228⟩≡
⟨*plan9 includes* 14⟩

```
#include    <ctype.h>
#include    <bio.h>

/*
 * tail command, posix plus v10 option -r.
 * the simple command tail -c, legal in v10, is illegal
 */

long    count;
int anycount;
static int  follow;
int file    = 0;

char*   umsg    = "usage: tail [-n N] [-c N] [-f] [-r] [+-N[bc][fr]] [file]";

Biobuf  bout;
enum
{
    BEG,
    END
} origin = END;
enum
{
    CHARS,
    LINES
} units = LINES;
enum
{
    FWD,
    REV
} dir = FWD;

extern  void    copy(void);
extern  void    fatal(char*);
extern  int getnumber(char*);
extern  void    keep(void);
extern  void    reverse(void);
extern  void    skip(void);
extern  void    suffix(char*);
extern  long    tread(char*, long);
extern  void    trunc_(Dir*, Dir**);
extern  vlong   tseek(vlong, int);
extern  void    twrite(char*, long);
extern  void    usage(void);
static  int isseekable(int fd);

#define JUMP(o,p) tseek(o,p), copy()

void
main(int argc, char **argv)
{
    int seekable, c;

    Binit(&bout, 1, OWRITE);
    for(; argc > 1 && ((c=*argv[1])=='-'||c=='+'); argc--,argv++ ) {
        if(getnumber(argv[1])) {
```

```
        suffix(argv[1]);
        continue;
    } else
    if(c == '-')
        switch(argv[1][1]) {
        case 'c':
            units = CHARS;
        case 'n':
            if(getnumber(argv[1]+2))
                continue;
            else
            if(argc > 2 && getnumber(argv[2])) {
                argc--, argv++;
                continue;
            } else
                usage();
        case 'r':
            dir = REV;
            continue;
        case 'f':
            follow++;
            continue;
        case '-':
            argc--, argv++;
        }
    break;
}
if(dir==REV && (units==CHARS || follow || origin==BEG))
    fatal("incompatible options");
if(!anycount)
    count = dir==REV? ~0UL>>1: 10;
if(origin==BEG && units==LINES && count>0)
    count--;
if(argc > 2)
    usage();
if(argc > 1 && (file=open(argv[1],0)) < 0)
    fatal(argv[1]);
seekable = isseekable(file);

if(!seekable && origin==END)
    keep();
else
if(!seekable && origin==BEG)
    skip();
else
if(units==CHARS && origin==END)
    JUMP(-count, 2);
else
if(units==CHARS && origin==BEG)
    JUMP(count, 0);
else
if(units==LINES && origin==END)
    reverse();
else
if(units==LINES && origin==BEG)
    skip();
if(follow && seekable)
    for(;;) {
        static Dir *sb0, *sb1;
        trunc_(sb1, &sb0);
```

```
            copy();
            trunc_(sb0, &sb1);
            sleep(5000);
        }
    exits(0);
}

void
trunc_(Dir *old, Dir **new)
{
    Dir *d;
    vlong olength;

    d = dirfstat(file);
    if(d == nil)
        return;
    olength = 0;
    if(old)
        olength = old->length;
    if(d->length < olength)
        d->length = tseek(0LL, 0);
    free(*new);
    *new = d;
}

void
suffix(char *s)
{
    while(*s && strchr("0123456789+-", *s))
        s++;
    switch(*s) {
    case 'b':
        if((count *= 1024) < 0)
            fatal("too big");
    case 'c':
        units = CHARS;
    case 'l':
        s++;
    }
    switch(*s) {
    case 'r':
        dir = REV;
        return;
    case 'f':
        follow++;
        return;
    case 0:
        return;
    }
    usage();
}

/*
 * read past head of the file to find tail
 */
void
skip(void)
{
    int i;
    long n;
```

```
    char buf[Bsize];
    if(units == CHARS) {
        for( ; count>0; count -=n) {
            n = count<Bsize? count: Bsize;
            if(!(n = tread(buf, n)))
                return;
        }
    } else /*units == LINES*/ {
        n = i = 0;
        while(count > 0) {
            if(!(n = tread(buf, Bsize)))
                return;
            for(i=0; i<n && count>0; i++)
                if(buf[i]=='\n')
                    count--;
        }
        twrite(buf+i, n-i);
    }
    copy();
}

void
copy(void)
{
    long n;
    char buf[Bsize];
    while((n=tread(buf, Bsize)) > 0) {
        twrite(buf, n);
        Bflush(&bout);  /* for FWD on pipe; else harmless */
    }
}

/*
 * read whole file, keeping the tail
 *  complexity is length(file)*length(tail).
 *  could be linear.
 */
void
keep(void)
{
    int len = 0;
    long bufsiz = 0;
    char *buf = 0;
    int j, k, n;

    for(n=1; n;) {
        if(len+Bsize > bufsiz) {
            bufsiz += 2*Bsize;
            if(!(buf = realloc(buf, bufsiz+1)))
                fatal("out of space");
        }
        for(; n && len<bufsiz; len+=n)
            n = tread(buf+len, bufsiz-len);
        if(count >= len)
            continue;
        if(units == CHARS)
            j = len - count;
        else {
            /* units == LINES */
            j = buf[len-1]=='\n'? len-1: len;
```

```
            for(k=0; j>0; j--)
                if(buf[j-1] == '\n')
                    if(++k >= count)
                        break;
        }
        memmove(buf, buf+j, len-=j);
    }
    if(dir == REV) {
        if(len>0 && buf[len-1]!='\n')
            buf[len++] = '\n';
        for(j=len-1 ; j>0; j--)
            if(buf[j-1] == '\n') {
                twrite(buf+j, len-j);
                if(--count <= 0)
                    return;
                len = j;
            }
    }
    if(count > 0)
        twrite(buf, len);
}

/*
 * count backward and print tail of file
 */
void
reverse(void)
{
    int first;
    long len = 0;
    long n = 0;
    long bufsiz = 0;
    char *buf = 0;
    vlong pos = tseek(0LL, 2);

    for(first=1; pos>0 && count>0; first=0) {
        n = pos>Bsize? Bsize: (long)pos;
        pos -= n;
        if(len+n > bufsiz) {
            bufsiz += 2*Bsize;
            if(!(buf = realloc(buf, bufsiz+1)))
                fatal("out of space");
        }
        memmove(buf+n, buf, len);
        len += n;
        tseek(pos, 0);
        if(tread(buf, n) != n)
            fatal("length error");
        if(first && buf[len-1]!='\n')
            buf[len++] = '\n';
        for(n=len-1 ; n>0 && count>0; n--)
            if(buf[n-1] == '\n') {
                count--;
                if(dir == REV)
                    twrite(buf+n, len-n);
                len = n;
            }
    }
    if(dir == FWD) {
        if(n)
```

```
            tseek(pos+n+1, 0);
        else
            tseek(0, 0);
        copy();
    } else
    if(count > 0)
        twrite(buf, len);
}

vlong
tseek(vlong o, int p)
{
    o = seek(file, o, p);
    if(o == -1)
        fatal("");
    return o;
}

long
tread(char *buf, long n)
{
    int r = read(file, buf, n);
    if(r == -1)
        fatal("");
    return r;
}

void
twrite(char *s, long n)
{
    if(Bwrite(&bout, s, n) != n)
        fatal("");
}

int
getnumber(char *s)
{
    if(*s=='-' || *s=='+')
        s++;
    if(!isdigit(*s))
        return 0;
    if(s[-1] == '+')
        origin = BEG;
    if(anycount++)
        fatal("excess option");
    count = atol(s);

    /* check range of count */
    if(count < 0 || (int)count != count)
        fatal("too big");
    return 1;
}

void
fatal(char *s)
{
    char buf[ERRMAX];

    errstr(buf, sizeof buf);
    fprint(2, "tail: %s: %s\n", s, buf);
```

```
    exits(s);
}

void
usage(void)
{
    fprint(2, "%s\n", umsg);
    exits("usage");
}

/* return true if seeks work and if the file is > 0 length.
 * this will eventually bite me in the ass if seeking a file
 * is not conservative. - presotto
 */
static int
isseekable(int fd)
{
    vlong m;

    m = seek(fd, 0, 1);
    if(m < 0)
        return 0;
    return 1;
}
```

A.4.4 pipe/tee.c

⟨pipe/tee.c 234a⟩≡

```
/*
 * tee-- pipe fitting
 */
```

⟨*plan9 includes* 14⟩

⟨*global flags(tee.c)* 113a⟩
⟨*global* openf *(tee.c)* 113b⟩
⟨*global* in*(tee.c)* 113c⟩

```
// forward decls
bool intignore(void*, char*);
```

⟨*function* main*(tee.c)* 113d⟩

⟨*function* intignore*(tee.c)* 114a⟩

A.4.5 pipe/xargs.c

⟨pipe/xargs.c 234b⟩≡

⟨*plan9 includes* 14⟩

```
#include <bio.h>
```

⟨*function* usage*(xargs.c)* 125a⟩
⟨*function* dowait*(xargs.c)* 126⟩

⟨*function* main*(xargs.c)* 125b⟩

A.5 time/

A.5.1 time/date.c

⟨time/date.c 235a⟩≡
⟨*plan9 includes* 14⟩

⟨*global flags(date.c)* 106a⟩

⟨*function* main*(date.c)* 106b⟩

A.5.2 time/cal.c

⟨time/cal.c 235b⟩≡
⟨*plan9 includes* 14⟩
```
#include <bio.h>
```

⟨*constants cal.c* 107⟩
⟨*globals cal.c* 108a⟩

```
// forward decls
int number(char *str);
void pstr(char *str, int n);
void cal(int m, int y, char *p, int w);
int jan1(int yr);
int curmo(void);
int curyr(void);
```

⟨*function* main*(cal.c)* 108b⟩

⟨*struct* dict*(cal.c)* 111c⟩
⟨*function* number*(cal.c)* 112a⟩
⟨*function* pstr*(cal.c)* 112b⟩

⟨*function* cal 109⟩

⟨*function* jan1*(cal.c)* 110⟩

⟨*function* curmo*(cal.c)* 111a⟩
⟨*function* curyr*(cal.c)* 111b⟩

A.6 misc/

A.6.1 misc/basename.c

⟨misc/basename.c 235c⟩≡
⟨*plan9 includes* 14⟩

⟨*function* main*(basename.c)* 171⟩

A.6.2 misc/file.c

⟨misc/file.c 235d⟩≡
⟨*plan9 includes* 14⟩
```
#include <bio.h>
#include <ctype.h>
```

```
#include <mach.h>

/*
 * file - determine type of file
 */
#define LENDIAN(p)  ((p)[0] | ((p)[1]<<8) | ((p)[2]<<16) | ((p)[3]<<24))

uchar   buf[6001];
short   cfreq[140];
short   wfreq[50];
int nbuf;
Dir*    mbuf;
int fd;
char    *fname;
char    *slash;

enum
{
    Cword,
    Fword,
    Aword,
    Alword,
    Lword,
    I1,
    I2,
    I3,
    Clatin  = 128,
    Cbinary,
    Cnull,
    Ceascii,
    Cutf,
};
struct
{
    char*   word;
    int class;
} dict[] =
{
    "PATH",     Lword,
    "TEXT",     Aword,
    "adt",      Alword,
    "aggr",     Alword,
    "alef",     Alword,
    "array",    Lword,
    "block",    Fword,
    "char",     Cword,
    "common",   Fword,
    "con",      Lword,
    "data",     Fword,
    "dimension",    Fword,
    "double",   Cword,
    "extern",   Cword,
    "bio",      I2,
    "float",    Cword,
    "fn",       Lword,
    "function", Fword,
    "h",        I3,
    "implement",    Lword,
    "import",   Lword,
    "include",  I1,
```

```
    "int",      Cword,
    "integer",  Fword,
    "iota",     Lword,
    "libc",     I2,
    "long",     Cword,
    "module",   Lword,
    "real",     Fword,
    "ref",      Lword,
    "register", Cword,
    "self",     Lword,
    "short",    Cword,
    "static",   Cword,
    "stdio",    I2,
    "struct",   Cword,
    "subroutine",   Fword,
    "u",        I2,
    "void",     Cword,
};

/* codes for 'mode' field in language structure */
enum    {
        Normal  = 0,
        First,      /* first entry for language spanning several ranges */
        Multi,      /* later entries "   "       "  ... */
        Shared,     /* codes used in several languages */
    };

struct
{
    int mode;       /* see enum above */
    int     count;
    int low;
    int high;
    char    *name;

} language[] =
{
    Normal, 0,  0x0100, 0x01FF, "Extended Latin",
    Normal, 0,  0x0370, 0x03FF, "Greek",
    Normal, 0,  0x0400, 0x04FF, "Cyrillic",
    Normal, 0,  0x0530, 0x058F, "Armenian",
    Normal, 0,  0x0590, 0x05FF, "Hebrew",
    Normal, 0,  0x0600, 0x06FF, "Arabic",
    Normal, 0,  0x0900, 0x097F, "Devanagari",
    Normal, 0,  0x0980, 0x09FF, "Bengali",
    Normal, 0,  0x0A00, 0x0A7F, "Gurmukhi",
    Normal, 0,  0x0A80, 0x0AFF, "Gujarati",
    Normal, 0,  0x0B00, 0x0B7F, "Oriya",
    Normal, 0,  0x0B80, 0x0BFF, "Tamil",
    Normal, 0,  0x0C00, 0x0C7F, "Telugu",
    Normal, 0,  0x0C80, 0x0CFF, "Kannada",
    Normal, 0,  0x0D00, 0x0D7F, "Malayalam",
    Normal, 0,  0x0E00, 0x0E7F, "Thai",
    Normal, 0,  0x0E80, 0x0EFF, "Lao",
    Normal, 0,  0x1000, 0x105F, "Tibetan",
    Normal, 0,  0x10A0, 0x10FF, "Georgian",
    Normal, 0,  0x3040, 0x30FF, "Japanese",
    Normal, 0,  0x3100, 0x312F, "Chinese",
    First,  0,  0x3130, 0x318F, "Korean",
    Multi,  0,  0x3400, 0x3D2F, "Korean",
```

```
    Shared, 0,  0x4e00, 0x9fff, "CJK",
    Normal, 0,  0,  0,  0,      /* terminal entry */
};


enum
{
    Fascii,     /* printable ascii */
    Flatin,     /* latin 1*/
    Futf,       /* UTF character set */
    Fbinary,    /* binary */
    Feascii,    /* ASCII with control chars */
    Fnull,      /* NULL in file */
} guess;

void    bump_utf_count(Rune);
int cistrncmp(char*, char*, int);
void    filetype(int);
int getfontnum(uchar*, uchar**);
int isas(void);
int isc(void);
int iscint(void);
int isenglish(void);
int ishp(void);
int ishtml(void);
int isrfc822(void);
int ismbox(void);
int islimbo(void);
int ismung(void);
int isp9bit(void);
int isp9font(void);
int isrtf(void);
int ismsdos(void);
int iself(void);
int istring(void);
int isoffstr(void);
int iff(void);
int long0(void);
int longoff(void);
int istar(void);
int isface(void);
int isexec(void);
int p9bitnum(uchar*);
int p9subfont(uchar*);
void    print_utf(void);
void    type(char*, int);
int utf_count(void);
void    wordfreq(void);

int (*call[])(void) =
{
    long0,      /* recognizable by first 4 bytes */
    istring,    /* recognizable by first string */
    iself,      /* ELF (foreign) executable */
    isexec,     /* native executables */
    iff,        /* interchange file format (strings) */
    longoff,    /* recognizable by 4 bytes at some offset */
    isoffstr,   /* recognizable by string at some offset */
    isrfc822,   /* email file */
    ismbox,     /* mail box */
```

```
    istar,      /* recognizable by tar checksum */
    ishtml,     /* html keywords */
    iscint,     /* compiler/assembler intermediate */
    islimbo,    /* limbo source */
    isc,        /* c & alef compiler key words */
    isas,       /* assembler key words */
    isp9font,   /* plan 9 font */
    isp9bit,    /* plan 9 image (as from /dev/window) */
    isrtf,      /* rich text format */
    ismsdos,    /* msdos exe (virus file attachement) */
    isface,     /* ascii face file */

    /* last resorts */
    ismung,     /* entropy compressed/encrypted */
    isenglish,  /* char frequency English */
    0
};

int mime;

char OCTET[] =  "application/octet-stream\n";
char PLAIN[] =  "text/plain\n";

void
main(int argc, char *argv[])
{
    int i, j, maxlen;
    char *cp;
    Rune r;

    ARGBEGIN{
    case 'm':
        mime = 1;
        break;
    default:
        fprint(2, "usage: file [-m] [file...]\n");
        exits("usage");
    }ARGEND;

    maxlen = 0;
    if(mime == 0 || argc > 1){
        for(i = 0; i < argc; i++) {
            for (j = 0, cp = argv[i]; *cp; j++, cp += chartorune(&r, cp))
                    ;
            if(j > maxlen)
                maxlen = j;
        }
    }
    if (argc <= 0) {
        if(!mime)
            print ("stdin: ");
        filetype(0);
    }
    else {
        for(i = 0; i < argc; i++)
            type(argv[i], maxlen);
    }
    exits(0);
}
```

```
void
type(char *file, int nlen)
{
    Rune r;
    int i;
    char *p;

    if(nlen > 0){
        slash = 0;
        for (i = 0, p = file; *p; i++) {
            if (*p == '/')          /* find rightmost slash */
                slash = p;
            p += chartorune(&r, p);     /* count runes */
        }
        print("%s:%*s",file, nlen-i+1, "");
    }
    fname = file;
    if ((fd = open(file, OREAD)) < 0) {
        print("cannot open: %r\n");
        return;
    }
    filetype(fd);
    close(fd);
}

void
filetype(int fd)
{
    Rune r;
    int i, f, n;
    char *p, *eob;

    free(mbuf);
    mbuf = dirfstat(fd);
    if(mbuf == nil){
        print("cannot stat: %r\n");
        return;
    }
    if(mbuf->mode & DMDIR) {
        print(mime ? OCTET : "directory\n");
        return;
    }
    if(mbuf->type != 'M' && mbuf->type != '|') {
        print(mime ? OCTET : "special file #%C/%s\n",
            mbuf->type, mbuf->name);
        return;
    }
    /* may be reading a pipe on standard input */
    nbuf = readn(fd, buf, sizeof(buf)-1);
    if(nbuf < 0) {
        print("cannot read: %r\n");
        return;
    }
    if(nbuf == 0) {
        print(mime ? PLAIN : "empty file\n");
        return;
    }
    buf[nbuf] = 0;

    /*
```

```
 * build histogram table
 */
memset(cfreq, 0, sizeof(cfreq));
for (i = 0; language[i].name; i++)
    language[i].count = 0;
eob = (char *)buf+nbuf;
for(n = 0, p = (char *)buf; p < eob; n++) {
    if (!fullrune(p, eob-p) && eob-p < UTFmax)
        break;
    p += chartorune(&r, p);
    if (r == 0)
        f = Cnull;
    else if (r <= 0x7f) {
        if (!isprint(r) && !isspace(r))
            f = Ceascii;    /* ASCII control char */
        else f = r;
    } else if (r == 0x80) {
        bump_utf_count(r);
        f = Cutf;
    } else if (r < 0xA0)
        f = Cbinary;    /* Invalid Runes */
    else if (r <= 0xff)
        f = Clatin; /* Latin 1 */
    else {
        bump_utf_count(r);
        f = Cutf;       /* UTF extension */
    }
    cfreq[f]++;         /* ASCII chars peg directly */
}
/*
 * gross classify
 */
if (cfreq[Cbinary])
    guess = Fbinary;
else if (cfreq[Cutf])
    guess = Futf;
else if (cfreq[Clatin])
    guess = Flatin;
else if (cfreq[Ceascii])
    guess = Feascii;
else if (cfreq[Cnull])
    guess = Fbinary;
else
    guess = Fascii;
/*
 * lookup dictionary words
 */
memset(wfreq, 0, sizeof(wfreq));
if(guess == Fascii || guess == Flatin || guess == Futf)
    wordfreq();
/*
 * call individual classify routines
 */
for(i=0; call[i]; i++)
    if((*call[i])())
        return;

/*
 * if all else fails,
 * print out gross classification
```

```
     */
    if (nbuf < 100 && !mime)
        print(mime ? PLAIN : "short ");
    if (guess == Fascii)
        print(mime ? PLAIN : "Ascii\n");
    else if (guess == Feascii)
        print(mime ? PLAIN : "extended ascii\n");
    else if (guess == Flatin)
        print(mime ? PLAIN : "latin ascii\n");
    else if (guess == Futf && utf_count() < 4)
        print_utf();
    else print(mime ? OCTET : "binary\n");
}

void
bump_utf_count(Rune r)
{
    int low, high, mid;

    high = sizeof(language)/sizeof(language[0])-1;
    for (low = 0; low < high;) {
        mid = (low+high)/2;
        if (r >= language[mid].low) {
            if (r <= language[mid].high) {
                language[mid].count++;
                break;
            } else low = mid+1;
        } else high = mid;
    }
}

int
utf_count(void)
{
    int i, count;

    count = 0;
    for (i = 0; language[i].name; i++)
        if (language[i].count > 0)
            switch (language[i].mode) {
            case Normal:
            case First:
                count++;
                break;
            default:
                break;
            }
    return count;
}

int
chkascii(void)
{
    int i;

    for (i = 'a'; i < 'z'; i++)
        if (cfreq[i])
            return 1;
    for (i = 'A'; i < 'Z'; i++)
        if (cfreq[i])
```

```
            return 1;
    return 0;
}

int
find_first(char *name)
{
    int i;

    for (i = 0; language[i].name != 0; i++)
        if (language[i].mode == First
            && strcmp(language[i].name, name) == 0)
            return i;
    return -1;
}

void
print_utf(void)
{
    int i, printed, j;

    if(mime){
        print(PLAIN);
        return;
    }
    if (chkascii()) {
        printed = 1;
        print("Ascii");
    } else
        printed = 0;
    for (i = 0; language[i].name; i++)
        if (language[i].count) {
            switch(language[i].mode) {
            case Multi:
                j = find_first(language[i].name);
                if (j < 0)
                    break;
                if (language[j].count > 0)
                    break;
                /* Fall through */
            case Normal:
            case First:
                if (printed)
                    print(" & ");
                else printed = 1;
                print("%s", language[i].name);
                break;
            case Shared:
            default:
                break;
            }
        }
    if(!printed)
        print("UTF");
    print(" text\n");
}

void
wordfreq(void)
{
```

```
    int low, high, mid, r;
    uchar *p, *p2, c;

    p = buf;
    for(;;) {
        while (p < buf+nbuf && !isalpha(*p))
            p++;
        if (p >= buf+nbuf)
            return;
        p2 = p;
        while(p < buf+nbuf && isalpha(*p))
            p++;
        c = *p;
        *p = 0;
        high = sizeof(dict)/sizeof(dict[0]);
        for(low = 0;low < high;) {
            mid = (low+high)/2;
            r = strcmp(dict[mid].word, (char*)p2);
            if(r == 0) {
                wfreq[dict[mid].class]++;
                break;
            }
            if(r < 0)
                low = mid+1;
            else
                high = mid;
        }
        *p++ = c;
    }
}

typedef struct Filemagic Filemagic;
struct Filemagic {
    ulong x;
    ulong mask;
    char *desc;
    char *mime;
};

/*
 * integers in this table must be as seen on a little-endian machine
 * when read from a file.
 */
Filemagic long0tab[] = {
    0xF16DF16D, 0xFFFFFFFF, "pac1 audio file\n",    OCTET,
    /* "pac1" */
    0x31636170, 0xFFFFFFFF, "pac3 audio file\n",    OCTET,
    /* "pXc2 */
    0x32630070, 0xFFFF00FF, "pac4 audio file\n",    OCTET,
    0xBA010000, 0xFFFFFFFF, "mpeg system stream\n", OCTET,
    0x43614c66, 0xFFFFFFFF, "FLAC audio file\n",    OCTET,
    0x30800CC0, 0xFFFFFFFF, "inferno .dis executable\n", OCTET,
    0x04034B50, 0xFFFFFFFF, "zip archive\n", "application/zip",
    070707,     0xFFFF,     "cpio archive\n", OCTET,
    0x2F7,      0xFFFF,     "tex dvi\n", "application/dvi",
    0xfaff,     0xfeff,     "mp3 audio\n",  "audio/mpeg",
    0xf0ff,     0xf6ff,     "aac audio\n",  "audio/mpeg",
    0xfeff0000, 0xffffffff, "utf-32be\n",   "text/plain charset=utf-32be",
    0xfffe,     0xffffffff, "utf-32le\n",   "text/plain charset=utf-32le",
    0xfeff,     0xffff,     "utf-16be\n",   "text/plain charset=utf-16be",
```

```
    0xfffe,     0xffff,     "utf-16le\n",   "text/plain charset=utf-16le",
    /* 0xfeedface: this could alternately be a Next Plan 9 boot image */
    0xcefaedfe, 0xFFFFFFFF, "32-bit power Mach-O executable\n", OCTET,
    /* 0xfeedfacf */
    0xcffaedfe, 0xFFFFFFFF, "64-bit power Mach-O executable\n", OCTET,
    /* 0xcefaedfe */
    0xfeedface, 0xFFFFFFFF, "386 Mach-O executable\n", OCTET,
    /* 0xcffaedfe */
    0xfeedfacf, 0xFFFFFFFF, "amd64 Mach-O executable\n", OCTET,
    /* 0xcafebabe */
    0xbebafeca, 0xFFFFFFFF, "Mach-O universal executable\n", OCTET,
    /*
     * these magic numbers are stored big-endian on disk,
     * thus the numbers appear reversed in this table.
     */
    0xad4e5cd1, 0xFFFFFFFF, "venti arena\n", OCTET,
    0x2bb19a52, 0xFFFFFFFF, "paq archive\n", OCTET,
};

int
filemagic(Filemagic *tab, int ntab, ulong x)
{
    int i;

    for(i=0; i<ntab; i++)
        if((x&tab[i].mask) == tab[i].x){
            print(mime ? tab[i].mime : tab[i].desc);
            return 1;
        }
    return 0;
}

int
long0(void)
{
    return filemagic(long0tab, nelem(long0tab), LENDIAN(buf));
}

typedef struct Fileoffmag Fileoffmag;
struct Fileoffmag {
    ulong   off;
    Filemagic;
};

/*
 * integers in this table must be as seen on a little-endian machine
 * when read from a file.
 */
Fileoffmag longofftab[] = {
    /*
     * these magic numbers are stored big-endian on disk,
     * thus the numbers appear reversed in this table.
     */
    256*1024, 0xe7a5e4a9, 0xFFFFFFFF, "venti arenas partition\n", OCTET,
    256*1024, 0xc75e5cd1, 0xFFFFFFFF, "venti index section\n", OCTET,
    128*1024, 0x89ae7637, 0xFFFFFFFF, "fossil write buffer\n", OCTET,
    4,    0x31647542, 0xFFFFFFFF, "OS X finder properties\n", OCTET,
};

int
```

```
fileoffmagic(Fileoffmag *tab, int ntab)
{
    int i;
    ulong x;
    Fileoffmag *tp;
    uchar buf[sizeof(long)];

    for(i=0; i<ntab; i++) {
        tp = tab + i;
        seek(fd, tp->off, 0);
        if (readn(fd, buf, sizeof buf) != sizeof buf)
            continue;
        x = LENDIAN(buf);
        if((x&tp->mask) == tp->x){
            print(mime? tp->mime: tp->desc);
            return 1;
        }
    }
    return 0;
}

int
longoff(void)
{
    return fileoffmagic(longofftab, nelem(longofftab));
}

int
isexec(void)
{
    Fhdr f;

    seek(fd, 0, 0);     /* reposition to start of file */
    if(crackhdr(fd, &f)) {
        print(mime ? OCTET : "%s\n", f.name);
        return 1;
    }
    return 0;
}


/* from tar.c */
enum { NAMSIZ = 100, TBLOCK = 512 };

union   hblock
{
    char    dummy[TBLOCK];
    struct  header
    {
        char    name[NAMSIZ];
        char    mode[8];
        char    uid[8];
        char    gid[8];
        char    size[12];
        char    mtime[12];
        char    chksum[8];
        char    linkflag;
        char    linkname[NAMSIZ];
        /* rest are defined by POSIX's ustar format; see p1003.2b */
        char    magic[6];   /* "ustar" */
```

```
        char    version[2];
        char    uname[32];
        char    gname[32];
        char    devmajor[8];
        char    devminor[8];
        char    prefix[155];  /* if non-null, path = prefix "/" name */
    } dbuf;
};

int
checksum(union hblock *hp)
{
    int i;
    char *cp;
    struct header *hdr = &hp->dbuf;

    for (cp = hdr->chksum; cp < &hdr->chksum[sizeof hdr->chksum]; cp++)
        *cp = ' ';
    i = 0;
    for (cp = hp->dummy; cp < &hp->dummy[TBLOCK]; cp++)
        i += *cp & 0xff;
    return i;
}

int
istar(void)
{
    int chksum;
    char tblock[TBLOCK];
    union hblock *hp = (union hblock *)tblock;
    struct header *hdr = &hp->dbuf;

    seek(fd, 0, 0);     /* reposition to start of file */
    if (readn(fd, tblock, sizeof tblock) != sizeof tblock)
        return 0;
    chksum = strtol(hdr->chksum, nil, 8);
    if (hdr->name[0] != '\0' && checksum(hp) == chksum) {
        if (strcmp(hdr->magic, "ustar") == 0)
            print(mime? "application/x-ustar\n":
                "posix tar archive\n");
        else
            print(mime? "application/x-tar\n": "tar archive\n");
        return 1;
    }
    return 0;
}

/*
 * initial words to classify file
 */
struct  FILE_STRING
{
    char    *key;
    char    *filetype;
    int length;
    char    *mime;
} file_string[] =
{
    "!<arch>\n__.SYMDEF",   "archive random library",   16, "application/octet-stream",
    "!<arch>\n",        "archive",          8,  "application/octet-stream",
```

```
    "070707",       "cpio archive - ascii header",  6,  "application/octet-stream",
    "#!/bin/rc",        "rc executable file",       9,  "text/plain",
    "#!/bin/sh",        "sh executable file",       9,  "text/plain",
    "%!",           "postscript",           2,  "application/postscript",
    "\004%!",       "postscript",           3,  "application/postscript",
    "x T post",     "troff output for post",    8,  "application/troff",
    "x T Latin1",       "troff output for Latin1",  10, "application/troff",
    "x T utf",      "troff output for UTF",     7,  "application/troff",
    "x T 202",      "troff output for 202",     7,  "application/troff",
    "x T aps",      "troff output for aps",     7,  "application/troff",
    "x T ",         "troff output",         4,  "application/troff",
    "GIF",          "GIF image",            3,  "image/gif",
    "\0PC Research, Inc\0", "ghostscript fax file",     18, "application/ghostscript",
    "%PDF",         "PDF",              4,  "application/pdf",
    "<html>\n",     "HTML file",            7,  "text/html",
    "<HTML>\n",     "HTML file",            7,  "text/html",
    "\111\111\052\000", "tiff",             4,  "image/tiff",
    "\115\115\000\052", "tiff",             4,  "image/tiff",
    "\377\330\377\340", "jpeg",             4,  "image/jpeg",
    "\377\330\377\341", "jpeg",             4,  "image/jpeg",
    "\377\330\377\333", "jpeg",             4,  "image/jpeg",
    "BM",           "bmp",              2,  "image/bmp",
    "\xD0\xCF\x11\xE0\xA1\xB1\x1A\xE1", "microsoft office document",    8,  "application/octet-stream",
    "<MakerFile ",      "FrameMaker file",      11, "application/framemaker",
    "\033E\033",    "HP PCL printer data",      3,  OCTET,
    "\033&",    "HP PCL printer data",      2,  OCTET,
    "\033%-12345X", "HPJCL file",       9,  "application/hpjcl",
    "\033Lua",      "Lua bytecode",     4,  OCTET,
    "ID3",          "mp3 audio with id3",   3,  "audio/mpeg",
    "\211PNG",      "PNG image",        4,  "image/png",
    "P3\n",         "ppm",              3,  "image/ppm",
    "P6\n",         "ppm",              3,  "image/ppm",
    "/* XPM */\n",  "xbm",              10, "image/xbm",
    ".HTML ",       "troff -ms input",  6,  "text/troff",
    ".LP",          "troff -ms input",  3,  "text/troff",
    ".ND",          "troff -ms input",  3,  "text/troff",
    ".PP",          "troff -ms input",  3,  "text/troff",
    ".TL",          "troff -ms input",  3,  "text/troff",
    ".TR",          "troff -ms input",  3,  "text/troff",
    ".TH",          "manual page",      3,  "text/troff",
    ".\\\"",        "troff input",      3,  "text/troff",
    ".de",          "troff input",      3,  "text/troff",
    ".if",          "troff input",      3,  "text/troff",
    ".nr",          "troff input",      3,  "text/troff",
    ".tr",          "troff input",      3,  "text/troff",
    "vac:",         "venti score",      4,  "text/plain",
    "-----BEGIN CERTIFICATE-----\n",
                "pem certificate",  -1, "text/plain",
    "-----BEGIN TRUSTED CERTIFICATE-----\n",
                "pem trusted certificate", -1,  "text/plain",
    "-----BEGIN X509 CERTIFICATE-----\n",
                "pem x.509 certificate", -1,    "text/plain",
    "subject=/C=",      "pem certificate with header", -1, "text/plain",
    "process snapshot ",    "process snapshot", -1, "application/snapfs",
    "BEGIN:VCARD\r\n",  "vCard",        13, "text/directory;profile=vcard",
    "BEGIN:VCARD\n",    "vCard",        12, "text/directory;profile=vcard",
    0,0,0,0
};

int
```

```
istring(void)
{
    int i, l;
    struct FILE_STRING *p;

    for(p = file_string; p->key; p++) {
        l = p->length;
        if(l == -1)
            l = strlen(p->key);
        if(nbuf >= l && memcmp(buf, p->key, l) == 0) {
            if(mime)
                print("%s\n", p->mime);
            else
                print("%s\n", p->filetype);
            return 1;
        }
    }
    if(strncmp((char*)buf, "TYPE=", 5) == 0) {  /* td */
        for(i = 5; i < nbuf; i++)
            if(buf[i] == '\n')
                break;
        if(mime)
            print(OCTET);
        else
            print("%.*s picture\n", utfnlen((char*)buf+5, i-5), (char*)buf+5);
        return 1;
    }
    return 0;
}

struct offstr
{
    ulong   off;
    struct FILE_STRING;
} offstrs[] = {
    32*1024, "\001CD001\001",   "ISO9660 CD image", 7,  OCTET,
    0, 0, 0, 0, 0
};

int
isoffstr(void)
{
    int n;
    char buf[256];
    struct offstr *p;

    for(p = offstrs; p->key; p++) {
        seek(fd, p->off, 0);
        n = p->length;
        if (n > sizeof buf)
            n = sizeof buf;
        if (readn(fd, buf, n) != n)
            continue;
        if(memcmp(buf, p->key, n) == 0) {
            if(mime)
                print("%s\n", p->mime);
            else
                print("%s\n", p->filetype);
            return 1;
        }
```

```
    }
    return 0;
}

int
iff(void)
{
    if (strncmp((char*)buf, "FORM", 4) == 0 &&
        strncmp((char*)buf+8, "AIFF", 4) == 0) {
        print("%s\n", mime? "audio/x-aiff": "aiff audio");
        return 1;
    }
    if (strncmp((char*)buf, "RIFF", 4) == 0) {
        if (strncmp((char*)buf+8, "WAVE", 4) == 0)
            print("%s\n", mime? "audio/wave": "wave audio");
        else if (strncmp((char*)buf+8, "AVI ", 4) == 0)
            print("%s\n", mime? "video/avi": "avi video");
        else
            print("%s\n", mime? "application/octet-stream":
                "riff file");
        return 1;
    }
    return 0;
}

char*   html_string[] =
{
    "title",
    "body",
    "head",
    "strong",
    "h1",
    "h2",
    "h3",
    "h4",
    "h5",
    "h6",
    "ul",
    "li",
    "dl",
    "br",
    "em",
    0,
};

int
ishtml(void)
{
    uchar *p, *q;
    int i, count;

        /* compare strings between '<' and '>' to html table */
    count = 0;
    p = buf;
    for(;;) {
        while (p < buf+nbuf && *p != '<')
            p++;
        p++;
        if (p >= buf+nbuf)
            break;
```

```
        if(*p == '/')
            p++;
        q = p;
        while(p < buf+nbuf && *p != '>')
            p++;
        if (p >= buf+nbuf)
            break;
        for(i = 0; html_string[i]; i++) {
            if(cistrncmp(html_string[i], (char*)q, p-q) == 0) {
                if(count++ > 4) {
                    print(mime ? "text/html\n" : "HTML file\n");
                    return 1;
                }
                break;
            }
        }
        p++;
    }
    return 0;
}

char*   rfc822_string[] =
{
    "from:",
    "date:",
    "to:",
    "subject:",
    "received:",
    "reply to:",
    "sender:",
    0,
};

int
isrfc822(void)
{

    char *p, *q, *r;
    int i, count;

    count = 0;
    p = (char*)buf;
    for(;;) {
        q = strchr(p, '\n');
        if(q == nil)
            break;
        *q = 0;
        if(p == (char*)buf && strncmp(p, "From ", 5) == 0 && strstr(p, " remote from ")){
            count++;
            *q = '\n';
            p = q+1;
            continue;
        }
        *q = '\n';
        if(*p != '\t' && *p != ' '){
            r = strchr(p, ':');
            if(r == 0 || r > q)
                break;
            for(i = 0; rfc822_string[i]; i++) {
                if(cistrncmp(p, rfc822_string[i], strlen(rfc822_string[i])) == 0){
```

```
                    count++;
                    break;
                }
            }
        }
        p = q+1;
    }
    if(count >= 3){
        print(mime ? "message/rfc822\n" : "email file\n");
        return 1;
    }
    return 0;
}

int
ismbox(void)
{
    char *p, *q;

    p = (char*)buf;
    q = strchr(p, '\n');
    if(q == nil)
        return 0;
    *q = 0;
    if(strncmp(p, "From ", 5) == 0 && strstr(p, " remote from ") == nil){
        print(mime ? "text/plain\n" : "mail box\n");
        return 1;
    }
    *q = '\n';
    return 0;
}

int
iscint(void)
{
    int type;
    char *name;
    Biobuf b;

    if(Binit(&b, fd, OREAD) == Beof)
        return 0;
    seek(fd, 0, 0);
    type = objtype(&b, &name);
    if(type < 0)
        return 0;
    if(mime)
        print(OCTET);
    else
        print("%s intermediate\n", name);
    return 1;
}

int
isc(void)
{
    int n;

    n = wfreq[I1];
    /*
     * includes
```

```
     */
    if(n >= 2 && wfreq[I2] >= n && wfreq[I3] >= n && cfreq['.'] >= n)
        goto yes;
    if(n >= 1 && wfreq[Alword] >= n && wfreq[I3] >= n && cfreq['.'] >= n)
        goto yes;
    /*
     * declarations
     */
    if(wfreq[Cword] >= 5 && cfreq[';'] >= 5)
        goto yes;
    /*
     * assignments
     */
    if(cfreq[';'] >= 10 && cfreq['='] >= 10 && wfreq[Cword] >= 1)
        goto yes;
    return 0;

yes:
    if(mime){
        print(PLAIN);
        return 1;
    }
    if(wfreq[Alword] > 0)
        print("alef program\n");
    else
        print("c program\n");
    return 1;
}

int
islimbo(void)
{

    /*
     * includes
     */
    if(wfreq[Lword] < 4)
        return 0;
    print(mime ? PLAIN : "limbo program\n");
    return 1;
}

int
isas(void)
{

    /*
     * includes
     */
    if(wfreq[Aword] < 2)
        return 0;
    print(mime ? PLAIN : "as program\n");
    return 1;
}

/*
 * low entropy means encrypted
 */
int
ismung(void)
```

```
{
    int i, bucket[8];
    float cs;

    if(nbuf < 64)
        return 0;
    memset(bucket, 0, sizeof(bucket));
    for(i=nbuf-64; i<nbuf; i++)
        bucket[(buf[i]>>5)&07] += 1;

    cs = 0.;
    for(i=0; i<8; i++)
        cs += (bucket[i]-8)*(bucket[i]-8);
    cs /= 8.;
    if(cs <= 24.322) {
        if(buf[0]==0x1f && buf[1]==0x9d)
            print(mime ? OCTET : "compressed\n");
        else
        if(buf[0]==0x1f && buf[1]==0x8b)
            print(mime ? OCTET : "gzip compressed\n");
        else
        if(buf[0]=='B' && buf[1]=='Z' && buf[2]=='h')
            print(mime ? OCTET : "bzip2 compressed\n");
        else
            print(mime ? OCTET : "encrypted\n");
        return 1;
    }
    return 0;
}

/*
 * english by punctuation and frequencies
 */
int
isenglish(void)
{
    int vow, comm, rare, badpun, punct;
    char *p;

    if(guess != Fascii && guess != Feascii)
        return 0;
    badpun = 0;
    punct = 0;
    for(p = (char *)buf; p < (char *)buf+nbuf-1; p++)
        switch(*p) {
        case '.':
        case ',':
        case ')':
        case '%':
        case ';':
        case ':':
        case '?':
            punct++;
            if(p[1] != ' ' && p[1] != '\n')
                badpun++;
        }
    if(badpun*5 > punct)
        return 0;
    if(cfreq['>']+cfreq['<']+cfreq['/'] > cfreq['e'])   /* shell file test */
        return 0;
```

```
    if(2*cfreq[';'] > cfreq['e'])
        return 0;

    vow = 0;
    for(p="AEIOU"; *p; p++) {
        vow += cfreq[*p];
        vow += cfreq[tolower(*p)];
    }
    comm = 0;
    for(p="ETAION"; *p; p++) {
        comm += cfreq[*p];
        comm += cfreq[tolower(*p)];
    }
    rare = 0;
    for(p="VJKQXZ"; *p; p++) {
        rare += cfreq[*p];
        rare += cfreq[tolower(*p)];
    }
    if(vow*5 >= nbuf-cfreq[' '] && comm >= 10*rare) {
        print(mime ? PLAIN : "English text\n");
        return 1;
    }
    return 0;
}

/*
 * pick up a number with
 * syntax _*[0-9]+_
 */
#define P9BITLEN    12
int
p9bitnum(uchar *bp)
{
    int n, c, len;

    len = P9BITLEN;
    while(*bp == ' ') {
        bp++;
        len--;
        if(len <= 0)
            return -1;
    }
    n = 0;
    while(len > 1) {
        c = *bp++;
        if(!isdigit(c))
            return -1;
        n = n*10 + c-'0';
        len--;
    }
    if(*bp != ' ')
        return -1;
    return n;
}

int
depthof(char *s, int *newp)
{
    char *es;
    int d;
```

```
    *newp = 0;
    es = s+12;
    while(s<es && *s==' ')
        s++;
    if(s == es)
        return -1;
    if('0'<=*s && *s<='9')
        return 1<<strtol(s, nil, 0);

    *newp = 1;
    d = 0;
    while(s<es && *s!=' '){
        s++;            /* skip letter */
        d += strtoul(s, &s, 10);
    }

    if(d % 8 == 0 || 8 % d == 0)
        return d;
    else
        return -1;
}

int
isp9bit(void)
{
    int dep, lox, loy, hix, hiy, px, new, cmpr;
    ulong t;
    long len;
    char *newlabel;
    uchar *cp;

    cp = buf;
    cmpr = 0;
    newlabel = "old ";

    if(memcmp(cp, "compressed\n", 11) == 0) {
        cmpr = 1;
        cp = buf + 11;
    }

    dep = depthof((char*)cp + 0*P9BITLEN, &new);
    if(new)
        newlabel = "";
    lox = p9bitnum(cp + 1*P9BITLEN);
    loy = p9bitnum(cp + 2*P9BITLEN);
    hix = p9bitnum(cp + 3*P9BITLEN);
    hiy = p9bitnum(cp + 4*P9BITLEN);
    if(dep < 0 || lox < 0 || loy < 0 || hix < 0 || hiy < 0)
        return 0;

    if(dep < 8){
        px = 8/dep;     /* pixels per byte */
        /* set l to number of bytes of data per scan line */
        if(lox >= 0)
            len = (hix+px-1)/px - lox/px;
        else{           /* make positive before divide */
            t = (-lox)+px-1;
            t = (t/px)*px;
            len = (t+hix+px-1)/px;
```

```
        }
    }else
        len = (hix-lox)*dep/8;
    len *= hiy - loy;       /* col length */
    len += 5 * P9BITLEN;        /* size of initial ascii */

    /*
     * for compressed images, don't look any further. otherwise:
     * for image file, length is non-zero and must match calculation above.
     * for /dev/window and /dev/screen the length is always zero.
     * for subfont, the subfont header should follow immediately.
     */
    if (cmpr) {
        print(mime ? OCTET : "Compressed %splan 9 image or subfont, depth %d\n",
            newlabel, dep);
        return 1;
    }
    /*
     * mbuf->length == 0 probably indicates reading a pipe.
     * Ghostscript sometimes produces a little extra on the end.
     */
    if (len != 0 && (mbuf->length == 0 || mbuf->length == len ||
        mbuf->length > len && mbuf->length < len+P9BITLEN)) {
        print(mime ? OCTET : "%splan 9 image, depth %d\n", newlabel, dep);
        return 1;
    }
    if (p9subfont(buf+len)) {
        print(mime ? OCTET : "%ssubfont file, depth %d\n", newlabel, dep);
        return 1;
    }
    return 0;
}

int
p9subfont(uchar *p)
{
    int n, h, a;

    /* if image too big, assume it's a subfont */
    if (p+3*P9BITLEN > buf+sizeof(buf))
        return 1;

    n = p9bitnum(p + 0*P9BITLEN);   /* char count */
    if (n < 0)
        return 0;
    h = p9bitnum(p + 1*P9BITLEN);   /* height */
    if (h < 0)
        return 0;
    a = p9bitnum(p + 2*P9BITLEN);   /* ascent */
    if (a < 0)
        return 0;
    return 1;
}

#define WHITESPACE(c)       ((c) == ' ' || (c) == '\t' || (c) == '\n')

int
isp9font(void)
{
    uchar *cp, *p;
```

```
    int i, n;
    char pathname[1024];

    cp = buf;
    if (!getfontnum(cp, &cp))   /* height */
        return 0;
    if (!getfontnum(cp, &cp))   /* ascent */
        return 0;
    for (i = 0; cp=(uchar*)strchr((char*)cp, '\n'); i++) {
        if (!getfontnum(cp, &cp))   /* min */
            break;
        if (!getfontnum(cp, &cp))   /* max */
            return 0;
        getfontnum(cp, &cp);    /* optional offset */
        while (WHITESPACE(*cp))
            cp++;
        for (p = cp; *cp && !WHITESPACE(*cp); cp++)
                ;
            /* construct a path name, if needed */
        n = 0;
        if (*p != '/' && slash) {
            n = slash-fname+1;
            if (n < sizeof(pathname))
                memcpy(pathname, fname, n);
            else n = 0;
        }
        if (n+cp-p+4 < sizeof(pathname)) {
            memcpy(pathname+n, p, cp-p);
            n += cp-p;
            pathname[n] = 0;
            if (access(pathname, AEXIST) < 0) {
                strcpy(pathname+n, ".0");
                if (access(pathname, AEXIST) < 0)
                    return 0;
            }
        }
    }
    if (i) {
        print(mime ? "text/plain\n" : "font file\n");
        return 1;
    }
    return 0;
}

int
getfontnum(uchar *cp, uchar **rp)
{
    while (WHITESPACE(*cp))     /* extract ulong delimited by whitespace */
        cp++;
    if (*cp < '0' || *cp > '9')
        return 0;
    strtoul((char *)cp, (char **)rp, 0);
    if (!WHITESPACE(**rp)) {
        *rp = cp;
        return 0;
    }
    return 1;
}

int
```

```
isrtf(void)
{
    if(strstr((char *)buf, "\\rtf1")){
        print(mime ? "application/rtf\n" : "rich text format\n");
        return 1;
    }
    return 0;
}

int
ismsdos(void)
{
    if (buf[0] == 0x4d && buf[1] == 0x5a){
        print(mime ? "application/x-msdownload\n" : "MSDOS executable\n");
        return 1;
    }
    return 0;
}

int
iself(void)
{
    static char *cpu[] = {      /* NB: incomplete and arbitary list */
    [1] "WE32100",
    [2] "SPARC",
    [3] "i386",
    [4] "M68000",
    [5] "M88000",
    [6] "i486",
    [7] "i860",
    [8] "R3000",
    [9] "S370",
    [10]    "R4000",
    [15]    "HP-PA",
    [18]    "sparc v8+",
    [19]    "i960",
    [20]    "PPC-32",
    [21]    "PPC-64",
    [40]    "ARM",
    [41]    "Alpha",
    [43]    "sparc v9",
    [50]    "IA-64",
    [62]    "AMD64",
    [75]    "VAX",
    };
    static char *type[] = {
    [1] "relocatable object",
    [2] "executable",
    [3] "shared library",
    [4] "core dump",
    };

    if (memcmp(buf, "\x7fELF", 4) == 0){
        if (!mime){
            int isdifend = 0;
            int n = (buf[19] << 8) | buf[18];
            char *p = "unknown";
            char *t = "unknown";

            if (n > 0 && n < nelem(cpu) && cpu[n])
```

```
                p = cpu[n];
            else {
                /* try the other byte order */
                isdifend = 1;
                n = (buf[18] << 8) | buf[19];
                if (n > 0 && n < nelem(cpu) && cpu[n])
                    p = cpu[n];
            }
            if(isdifend)
                n = (buf[16]<< 8) | buf[17];
            else
                n = (buf[17]<< 8) | buf[16];

            if(n>0 && n < nelem(type) && type[n])
                t = type[n];
            print("%s ELF%s %s\n", p, (buf[4] == 2? "64": "32"), t);
        }
        else
            print("application/x-elf-executable");
        return 1;
    }

    return 0;
}

int
isface(void)
{
    int i, j, ldepth, l;
    char *p;

    ldepth = -1;
    for(j = 0; j < 3; j++){
        for(p = (char*)buf, i=0; i<3; i++){
            if(p[0] != '0' || p[1] != 'x')
                return 0;
            if(buf[2+8] == ',')
                l = 2;
            else if(buf[2+4] == ',')
                l = 1;
            else
                return 0;
            if(ldepth == -1)
                ldepth = l;
            if(l != ldepth)
                return 0;
            strtoul(p, &p, 16);
            if(*p++ != ',')
                return 0;
            while(*p == ' ' || *p == '\t')
                p++;
        }
        if (*p++ != '\n')
            return 0;
    }

    if(mime)
        print("application/x-face\n");
    else
        print("face image depth %d\n", ldepth);
```

```
    return 1;
}
```

A.6.3 misc/iconv.c

⟨misc/iconv.c 261⟩≡

```
⟨plan9 includes 14⟩
#include <draw.h>
#include <memdraw.h>

void
usage(void)
{
    fprint(2, "usage: iconv [-u] [-c chanstr] [file]\n");
    exits("usage");
}

void
writeuncompressed(int fd, Memimage *m)
{
    char chanstr[32];
    int bpl, y, j;
    uchar *buf;

    if(chantostr(chanstr, m->chan) == nil)
        sysfatal("can't convert channel descriptor: %r");
    fprint(fd, "%11s %11d %11d %11d %11d ",
        chanstr, m->r.min.x, m->r.min.y, m->r.max.x, m->r.max.y);

    bpl = bytesperline(m->r, m->depth);
    buf = malloc(bpl);
    if(buf == nil)
        sysfatal("malloc failed: %r");
    for(y=m->r.min.y; y<m->r.max.y; y++){
        j = unloadmemimage(m, Rect(m->r.min.x, y, m->r.max.x, y+1), buf, bpl);
        if(j != bpl)
            sysfatal("image unload failed: %r");
        if(write(fd, buf, bpl) != bpl)
            sysfatal("write failed: %r");
    }
    free(buf);
}

void
main(int argc, char *argv[])
{
    char *tostr, *file;
    int fd, uncompressed;
    ulong tochan;
    Memimage *m, *n;

    tostr = nil;
    uncompressed = 0;
    ARGBEGIN{
    case 'c':
        tostr = EARGF(usage());
        break;
    case 'u':
```

```
        uncompressed = 1;
        break;
    default:
        usage();
    }ARGEND

    memimageinit();

    file = "<stdin>";
    m = nil;

    switch(argc){
    case 0:
        m = readmemimage(0);
        break;
    case 1:
        file = argv[0];
        fd = open(file, OREAD);
        if(fd < 0)
            sysfatal("can't open %s: %r", file);
        m = readmemimage(fd);
        close(fd);
        break;
    default:
        usage();
    }

    if(m == nil)
        sysfatal("can't read %s: %r", file);

    if(tostr == nil)
        tochan = m->chan;
    else{
        tochan = strtochan(tostr);
        if(tochan == 0)
            sysfatal("bad channel descriptor '%s'", tostr);
    }

    n = allocmemimage(m->r, tochan);
    if(n == nil)
        sysfatal("can't allocate new image: %r");

    memimagedraw(n, n->r, m, m->r.min, nil, ZP, S);
    if(uncompressed)
        writeuncompressed(1, n);
    else
        writememimage(1, n);
    exits(nil);
}
```

A.6.4 misc/strings.c

⟨misc/strings.c 262⟩≡

```
⟨plan9 includes 14⟩
#include    <bio.h>

Biobuf  *fin;
Biobuf  fout;
```

```
#define MINSPAN     6       /* Min characters in string (default) */
#define BUFSIZE     70

void stringit(char *);
int isprint(Rune);

static int minspan = MINSPAN;

static void
usage(void)
{
    fprint(2, "usage: %s [-m min] [file...]\n", argv0);
    exits("usage");
}

void
main(int argc, char **argv)
{
    int i;

    ARGBEGIN{
    case 'm':
        minspan = atoi(EARGF(usage()));
        break;
    default:
        usage();
        break;
    }ARGEND
    Binit(&fout, 1, OWRITE);
    if(argc < 1) {
        stringit("/fd/0");
        exits(0);
    }

    for(i = 0; i < argc; i++) {
        if(argc > 2)
            print("%s:\n", argv[i]);

        stringit(argv[i]);
    }

    exits(0);
}

void
stringit(char *str)
{
    long posn, start;
    int cnt = 0;
    long c;

    Rune buf[BUFSIZE];

    if ((fin = Bopen(str, OREAD)) == 0) {
        perror("open");
        return;
    }

    start = 0;
    posn = Boffset(fin);
```

```
    while((c = Bgetrune(fin)) >= 0) {
        if(isprint(c)) {
            if(start == 0)
                start = posn;
            buf[cnt++] = c;
            if(cnt == BUFSIZE-1) {
                buf[cnt] = 0;
                Bprint(&fout, "%8ld: %S ...\n", start, buf);
                start = 0;
                cnt = 0;
            }
        } else {
             if(cnt >= minspan) {
                buf[cnt] = 0;
                Bprint(&fout, "%8ld: %S\n", start, buf);
            }
            start = 0;
            cnt = 0;
        }
        posn = Boffset(fin);
    }

    if(cnt >= minspan){
        buf[cnt] = 0;
        Bprint(&fout, "%8ld: %S\n", start, buf);
    }
    Bterm(fin);
}

int
isprint(Rune r)
{
    if (r != Runeerror)
    if ((r >= ' ' && r < 0x7F) || r > 0xA0)
        return 1;
    return 0;
}
```

A.6.5 misc/unicode.c

⟨misc/unicode.c 264⟩≡

```
⟨plan9 includes 14⟩
#include <bio.h>

char    usage[] = "unicode { [-t] hex hex ... | hexmin-hexmax ... | [-n] char ... }";
char    hex[] = "0123456789abcdefABCDEF";
int numout = 0;
int text = 0;
char    *err;
Biobuf  bout;

char    *range(char*[]);
char    *nums(char*[]);
char    *chars(char*[]);

void
main(int argc, char *argv[])
{
    ARGBEGIN{
```

```
    case 'n':
        numout = 1;
        break;
    case 't':
        text = 1;
        break;
    }ARGEND
    Binit(&bout, 1, OWRITE);
    if(argc == 0){
        fprint(2, "usage: %s\n", usage);
        exits("usage");
    }
    if(!numout && utfrune(argv[0], '-'))
        exits(range(argv));
    if(numout || strchr(hex, argv[0][0])==0)
        exits(nums(argv));
    exits(chars(argv));
}

char*
range(char *argv[])
{
    char *q;
    int min, max;
    int i;

    while(*argv){
        q = *argv;
        if(strchr(hex, q[0]) == 0){
    err:
            fprint(2, "unicode: bad range %s\n", *argv);
            return "bad range";
        }
        min = strtoul(q, &q, 16);
        if(min<0 || min>Runemax || *q!='-')
            goto err;
        q++;
        if(strchr(hex, *q) == 0)
            goto err;
        max = strtoul(q, &q, 16);
        if(max<0 || max>Runemax || max<min || *q!=0)
            goto err;
        i = 0;
        do{
            Bprint(&bout, "%.6x %C", min, min);
            i++;
            if(min==max || (i&7)==0)
                Bprint(&bout, "\n");
            else
                Bprint(&bout, "\t");
            min++;
        }while(min<=max);
        argv++;
    }
    return 0;
}

char*
nums(char *argv[])
{
```

```
    char *q;
    Rune r;
    int w, rsz;
    char utferr[UTFmax];

    r = Runeerror;
    rsz = runetochar(utferr, &r);
    while(*argv){
        q = *argv;
        while(*q){
            w = chartorune(&r, q);
            if(r==Runeerror){
                if(strlen(q) != rsz || memcmp(q, utferr, rsz) != 0){
                    fprint(2, "unicode: invalid utf string %s\n", *argv);
                    return "bad utf";
                }
            }
            Bprint(&bout, "%.6x\n", r);
            q += w;
        }
        argv++;
    }
    return 0;
}

char*
chars(char *argv[])
{
    char *q;
    int m;

    while(*argv){
        q = *argv;
        if(strchr(hex, q[0]) == 0){
    err:
            fprint(2, "unicode: bad unicode value %s\n", *argv);
            return "bad char";
        }
        m = strtoul(q, &q, 16);
        if(m<0 || m>Runemax || *q!=0)
            goto err;
        Bprint(&bout, "%C", m);
        if(!text)
            Bprint(&bout, "\n");
        argv++;
    }
    return 0;
}
```

A.7 text/misc/

A.7.1 pipe/uniq.c

⟨pipe/uniq.c 266⟩≡

```
/*
 * Deal with duplicated lines in a file
 */
⟨plan9 includes 14⟩
```

```
#include <bio.h>
#include <ctype.h>

⟨constant SIZE(uniq.c) 122f⟩

⟨globals uniq.c 122a⟩

// forward decls
bool   gline(char *buf);
void  pline(char *buf);
bool   equal(char *b1, char *b2);
char* skip(char *s);

⟨function main(uniq.c) 122g⟩

⟨function gline(uniq.c) 123⟩
⟨function pline(uniq.c) 124a⟩
⟨function equal(uniq.c) 124b⟩
⟨function skip(uniq.c) 124c⟩
```

A.7.2 pipe/sort.c

```
⟨pipe/sort.c 267⟩≡
  ⟨plan9 includes 14⟩
  #include    <bio.h>

  /*
  bugs:
      00/ff for end of file can conflict with 00/ff characters
  */

  enum
  {
      Nline   = 100000,       /* default max number of lines saved in memory */
      Nmerge  = 10,           /* max number of temporary files merged */
      Nfield  = 20,           /* max number of argument fields */

      Bflag   = 1<<0,         /* flags per field */
      B1flag  = 1<<1,

      Dflag   = 1<<2,
      Fflag   = 1<<3,
      Gflag   = 1<<4,
      Iflag   = 1<<5,
      Mflag   = 1<<6,
      Nflag   = 1<<7,
      Rflag   = 1<<8,
      Wflag   = 1<<9,

      NSstart = 0,            /* states for number to key decoding */
      NSsign,
      NSzero,
      NSdigit,
      NSpoint,
      NSfract,
      NSzerofract,
      NSexp,
      NSexpsign,
      NSexpdigit,
```

```
};

typedef struct  Line    Line;
typedef struct  Key Key;
typedef struct  Merge   Merge;
typedef struct  Field   Field;

struct  Line
{
    Key*    key;
    int llen;       /* always >= 1 */
    uchar   line[1];    /* always ends in '\n' */
};

struct  Merge
{
    Key*    key;        /* copy of line->key so (Line*) looks like (Merge*) */
    Line*   line;       /* line at the head of a merged temp file */
    int fd;     /* file descriptor */
    Biobuf  b;      /* iobuf for reading a temp file */
};

struct  Key
{
    int klen;
    uchar   key[1];
};

struct  Field
{
    int beg1;
    int beg2;
    int end1;
    int end2;

    long    flags;
    uchar   mapto[1+255];

    void    (*dokey)(Key*, uchar*, uchar*, Field*);
};

struct args
{
    char*   ofile;
    char*   tname;
    Rune    tabchar;
    char    cflag;
    char    uflag;
    char    vflag;
    int nfield;
    int nfile;
    Field   field[Nfield];

    Line**  linep;
    long    nline;          /* number of lines in this temp file */
    long    lineno;         /* overall ordinal for -s option */
    int ntemp;
    long    mline;          /* max lines per file */
} args;
```

```
extern  Rune*   month[12];

void    buildkey(Line*);
void    doargs(int, char*[]);
void    dofield(char*, int*, int*, int, int);
void    dofile(Biobuf*);
void    dokey_(Key*, uchar*, uchar*, Field*);
void    dokey_dfi(Key*, uchar*, uchar*, Field*);
void    dokey_gn(Key*, uchar*, uchar*, Field*);
void    dokey_m(Key*, uchar*, uchar*, Field*);
void    dokey_r(Key*, uchar*, uchar*, Field*);
void    done(char*);
int kcmp(Key*, Key*);
void    makemapd(Field*);
void    makemapm(Field*);
void    mergefiles(int, int, Biobuf*);
void    mergeout(Biobuf*);
void    newfield(void);
Line*   newline(Biobuf*);
void    nomem(void);
void    notifyf(void*, char*);
void    printargs(void);
void    printout(Biobuf*);
void    setfield(int, int);
uchar*  skip(uchar*, int, int, int, int);
void    sort4(void*, ulong);
char*   tempfile(int);
void    tempout(void);
void    lineout(Biobuf*, Line*);

void
main(int argc, char *argv[])
{
    int i, f;
    char *s;
    Biobuf bbuf;

    notify(notifyf);    /**/
    doargs(argc, argv);
    if(args.vflag)
        printargs();

    for(i=1; i<argc; i++) {
        s = argv[i];
        if(s == 0)
            continue;
        if(strcmp(s, "-") == 0) {
            Binit(&bbuf, 0, OREAD);
            dofile(&bbuf);
            Bterm(&bbuf);
            continue;
        }
        f = open(s, OREAD);
        if(f < 0) {
            fprint(2, "sort: open %s: %r\n", s);
            done("open");
        }
        Binit(&bbuf, f, OREAD);
        dofile(&bbuf);
        Bterm(&bbuf);
```

```
        close(f);
    }
    if(args.nfile == 0) {
        Binit(&bbuf, 0, OREAD);
        dofile(&bbuf);
        Bterm(&bbuf);
    }
    if(args.cflag)
        done(0);
    if(args.vflag)
        fprint(2, "=========\n");

    f = 1;
    if(args.ofile) {
        f = create(args.ofile, OWRITE, 0666);
        if(f < 0) {
            fprint(2, "sort: create %s: %r\n", args.ofile);
            done("create");
        }
    }

    Binit(&bbuf, f, OWRITE);
    if(args.ntemp) {
        tempout();
        mergeout(&bbuf);
    } else {
        printout(&bbuf);
    }
    Bterm(&bbuf);
    done(0);
}

void
dofile(Biobuf *b)
{
    Line *l, *ol;
    int n;

    if(args.cflag) {
        ol = newline(b);
        if(ol == 0)
            return;
        for(;;) {
            l = newline(b);
            if(l == 0)
                break;
            n = kcmp(ol->key, l->key);
            if(n > 0 || (n == 0 && args.uflag)) {
                fprint(2, "sort: -c file not in sort\n"); /**/
                done("order");
            }
            free(ol->key);
            free(ol);
            ol = l;
        }
        return;
    }

    if(args.linep == 0) {
        args.linep = malloc(args.mline * sizeof(args.linep));
```

```
        if(args.linep == 0)
            nomem();
    }
    for(;;) {
        l = newline(b);
        if(l == 0)
            break;
        if(args.nline >= args.mline)
            tempout();
        args.linep[args.nline] = l;
        args.nline++;
        args.lineno++;
    }
}

void
notifyf(void*, char *s)
{

    if(strcmp(s, "interrupt") == 0)
        done(0);
    if(strcmp(s, "hangup") == 0)
        done(0);
    if(strcmp(s, "kill") == 0)
        done(0);
    if(strncmp(s, "sys: write on closed pipe", 25) == 0)
        done(0);
    fprint(2, "sort: note: %s\n", s);
    abort();
}

Line*
newline(Biobuf *b)
{
    Line *l;
    char *p;
    int n, c;

    p = Brdline(b, '\n');
    n = Blinelen(b);
    if(p == 0) {
        if(n == 0)
            return 0;
        l = 0;
        for(n=0;;) {
            if((n & 31) == 0) {
                l = realloc(l, sizeof(Line) +
                    (n+31)*sizeof(l->line[0]));
                if(l == 0)
                    nomem();
            }
            c = Bgetc(b);
            if(c < 0) {
                fprint(2, "sort: newline added\n");
                c = '\n';
            }
            l->line[n++] = c;
            if(c == '\n')
                break;
        }
```

```
        l->llen = n;
        buildkey(l);
        return l;
    }
    l = malloc(sizeof(Line) +
        (n-1)*sizeof(l->line[0]));
    if(l == 0)
        nomem();
    l->llen = n;
    memmove(l->line, p, n);
    buildkey(l);
    return l;
}

void
lineout(Biobuf *b, Line *l)
{
    int n, m;

    n = l->llen;
    m = Bwrite(b, l->line, n);
    if(n != m)
        exits("write");
}

void
tempout(void)
{
    long n;
    Line **lp, *l;
    char *tf;
    int f;
    Biobuf tb;

    sort4(args.linep, args.nline);
    tf = tempfile(args.ntemp);
    args.ntemp++;
    f = create(tf, OWRITE, 0666);
    if(f < 0) {
        fprint(2, "sort: create %s: %r\n", tf);
        done("create");
    }

    Binit(&tb, f, OWRITE);
    lp = args.linep;
    for(n=args.nline; n>0; n--) {
        l = *lp++;
        lineout(&tb, l);
        free(l->key);
        free(l);
    }
    args.nline = 0;
    Bterm(&tb);
    close(f);
}

void
done(char *xs)
{
    int i;
```

```
    for(i=0; i<args.ntemp; i++)
        remove(tempfile(i));
    exits(xs);
}

void
nomem(void)
{
    fprint(2, "sort: out of memory\n");
    done("mem");
}

char*
tempfile(int n)
{
    static char file[100];
    static uint pid;
    char *dir;

    dir = "/tmp";
    if(args.tname)
        dir = args.tname;
    if(strlen(dir) >= nelem(file)-20) {
        fprint(2, "temp file directory name is too long: %s\n", dir);
        done("tdir");
    }

    if(pid == 0) {
        pid = getpid();
        if(pid == 0) {
            pid = time(0);
            if(pid == 0)
                pid = 1;
        }
    }

    sprint(file, "%s/sort.%.4d.%.4d", dir, pid%10000, n);
    return file;
}

void
mergeout(Biobuf *b)
{
    int n, i, f;
    char *tf;
    Biobuf tb;

    for(i=0; i<args.ntemp; i+=n) {
        n = args.ntemp - i;
        if(n > Nmerge) {
            tf = tempfile(args.ntemp);
            args.ntemp++;
            f = create(tf, OWRITE, 0666);
            if(f < 0) {
                fprint(2, "sort: create %s: %r\n", tf);
                done("create");
            }
            Binit(&tb, f, OWRITE);
```

```
            n = Nmerge;
            mergefiles(i, n, &tb);

            Bterm(&tb);
            close(f);
        } else
            mergefiles(i, n, b);
    }
}

void
mergefiles(int t, int n, Biobuf *b)
{
    Merge *m, *mp, **mmp;
    Key *ok;
    Line *l;
    char *tf;
    int i, f, nn;

    mmp = malloc(n*sizeof(*mmp));
    mp = malloc(n*sizeof(*mp));
    if(mmp == 0 || mp == 0)
        nomem();

    nn = 0;
    m = mp;
    for(i=0; i<n; i++,m++) {
        tf = tempfile(t+i);
        f = open(tf, OREAD);
        if(f < 0) {
            fprint(2, "sort: reopen %s: %r\n", tf);
            done("open");
        }
        m->fd = f;
        Binit(&m->b, f, OREAD);
        mmp[nn] = m;

        l = newline(&m->b);
        if(l == 0)
            continue;
        nn++;
        m->line = l;
        m->key = l->key;
    }

    ok = 0;
    for(;;) {
        sort4(mmp, nn);
        m = *mmp;
        if(nn == 0)
            break;
        for(;;) {
            l = m->line;
            if(args.uflag && ok && kcmp(ok, l->key) == 0) {
                free(l->key);
                free(l);
            } else {
                lineout(b, l);
                if(ok)
                    free(ok);
```

```
                ok = l->key;
                free(l);
            }

            l = newline(&m->b);
            if(l == 0) {
                nn--;
                mmp[0] = mmp[nn];
                break;
            }
            m->line = l;
            m->key = l->key;
            if(nn > 1 && kcmp(mmp[0]->key, mmp[1]->key) > 0)
                break;
        }
    }
    if(ok)
        free(ok);

    m = mp;
    for(i=0; i<n; i++,m++) {
        Bterm(&m->b);
        close(m->fd);
    }

    free(mp);
    free(mmp);
}

int
kcmp(Key *ka, Key *kb)
{
    int n, m;

    /*
     * set n to length of smaller key
     */
    n = ka->klen;
    m = kb->klen;
    if(n > m)
        n = m;
    return memcmp(ka->key, kb->key, n);
}

void
printout(Biobuf *b)
{
    long n;
    Line **lp, *l;
    Key *ok;

    sort4(args.linep, args.nline);
    lp = args.linep;
    ok = 0;
    for(n=args.nline; n>0; n--) {
        l = *lp++;
        if(args.uflag && ok && kcmp(ok, l->key) == 0)
            continue;
        lineout(b, l);
        ok = l->key;
```

```
    }
}

void
setfield(int n, int c)
{
    Field *f;

    f = &args.field[n];
    switch(c) {
    default:
        fprint(2, "sort: unknown option: field.%C\n", c);
        done("option");
    case 'b':   /* skip blanks */
        f->flags |= Bflag;
        break;
    case 'd':   /* directory order */
        f->flags |= Dflag;
        break;
    case 'f':   /* fold case */
        f->flags |= Fflag;
        break;
    case 'g':   /* floating point -n case */
        f->flags |= Gflag;
        break;
    case 'i':   /* ignore non-ascii */
        f->flags |= Iflag;
        break;
    case 'M':   /* month */
        f->flags |= Mflag;
        break;
    case 'n':   /* numbers */
        f->flags |= Nflag;
        break;
    case 'r':   /* reverse */
        f->flags |= Rflag;
        break;
    case 'w':   /* ignore white */
        f->flags |= Wflag;
        break;
    }
}

void
dofield(char *s, int *n1, int *n2, int off1, int off2)
{
    int c, n;

    c = *s++;
    if(c >= '0' && c <= '9') {
        n = 0;
        while(c >= '0' && c <= '9') {
            n = n*10 + (c-'0');
            c = *s++;
        }
        n -= off1;  /* posix committee: rot in hell */
        if(n < 0) {
            fprint(2, "sort: field offset must be positive\n");
            done("option");
        }
```

```
        *n1 = n;
    }
    if(c == '.') {
        c = *s++;
        if(c >= '0' && c <= '9') {
            n = 0;
            while(c >= '0' && c <= '9') {
                n = n*10 + (c-'0');
                c = *s++;
            }
            n -= off2;
            if(n < 0) {
                fprint(2, "sort: character offset must be positive\n");
                done("option");
            }
            *n2 = n;
        }
    }
    while(c != 0) {
        setfield(args.nfield, c);
        c = *s++;
    }
}

void
printargs(void)
{
    int i, n;
    Field *f;
    char *prefix;

    fprint(2, "sort");
    for(i=0; i<=args.nfield; i++) {
        f = &args.field[i];
        prefix = " -";
        if(i) {
            n = f->beg1;
            if(n >= 0)
                fprint(2, " +%d", n);
            else
                fprint(2, " +*");
            n = f->beg2;
            if(n >= 0)
                fprint(2, ".%d", n);
            else
                fprint(2, ".*");

            if(f->flags & B1flag)
                fprint(2, "b");

            n = f->end1;
            if(n >= 0)
                fprint(2, " -%d", n);
            else
                fprint(2, " -*");
            n = f->end2;
            if(n >= 0)
                fprint(2, ".%d", n);
            else
                fprint(2, ".*");
```

```
            prefix = "";
        }
        if(f->flags & Bflag)
            fprint(2, "%sb", prefix);
        if(f->flags & Dflag)
            fprint(2, "%sd", prefix);
        if(f->flags & Fflag)
            fprint(2, "%sf", prefix);
        if(f->flags & Gflag)
            fprint(2, "%sg", prefix);
        if(f->flags & Iflag)
            fprint(2, "%si", prefix);
        if(f->flags & Mflag)
            fprint(2, "%sM", prefix);
        if(f->flags & Nflag)
            fprint(2, "%sn", prefix);
        if(f->flags & Rflag)
            fprint(2, "%sr", prefix);
        if(f->flags & Wflag)
            fprint(2, "%sw", prefix);
    }
    if(args.cflag)
        fprint(2, " -c");
    if(args.uflag)
        fprint(2, " -u");
    if(args.ofile)
        fprint(2, " -o %s", args.ofile);
    if(args.mline != Nline)
        fprint(2, " -l %ld", args.mline);
    fprint(2, "\n");
}

void
newfield(void)
{
    int n;
    Field *f;

    n = args.nfield + 1;
    if(n >= Nfield) {
        fprint(2, "sort: too many fields specified\n");
        done("option");
    }
    args.nfield = n;
    f = &args.field[n];
    f->beg1 = -1;
    f->beg2 = -1;
    f->end1 = -1;
    f->end2 = -1;
}

void
doargs(int argc, char *argv[])
{
    int i, c, hadplus;
    char *s, *p, *q;
    Field *f;

    hadplus = 0;
    args.mline = Nline;
```

```
for(i=1; i<argc; i++) {
    s = argv[i];
    c = *s++;
    if(c == '-') {
        c = *s;
        if(c == 0)      /* forced end of arg marker */
            break;
        argv[i] = 0;        /* clobber args processed */
        if(c == '.' || (c >= '0' && c <= '9')) {
            if(!hadplus)
                newfield();
            f = &args.field[args.nfield];
            dofield(s, &f->end1, &f->end2, 0, 0);
            hadplus = 0;
            continue;
        }

        while(c = *s++)
        switch(c) {
        case '-':   /* end of options */
            i = argc;
            continue;
        case 'T':   /* temp directory */
            if(*s == 0) {
                i++;
                if(i < argc) {
                    args.tname = argv[i];
                    argv[i] = 0;
                }
            } else
                args.tname = s;
            s = strchr(s, 0);
            break;
        case 'o':   /* output file */
            if(*s == 0) {
                i++;
                if(i < argc) {
                    args.ofile = argv[i];
                    argv[i] = 0;
                }
            } else
                args.ofile = s;
            s = strchr(s, 0);
            break;
        case 'k':   /* posix key (what were they thinking?) */
            p = 0;
            if(*s == 0) {
                i++;
                if(i < argc) {
                    p = argv[i];
                    argv[i] = 0;
                }
            } else
                p = s;
            s = strchr(s, 0);
            if(p == 0)
                break;

            newfield();
            q = strchr(p, ',');
```

```
    if(q)
        *q++ = 0;
    f = &args.field[args.nfield];
    dofield(p, &f->beg1, &f->beg2, 1, 1);
    if(f->flags & Bflag) {
        f->flags |= B1flag;
        f->flags &= ~Bflag;
    }
    if(q) {
        dofield(q, &f->end1, &f->end2, 1, 0);
        if(f->end2 <= 0)
            f->end1++;
    }
    hadplus = 0;
    break;
case 't':   /* tab character */
    if(*s == 0) {
        i++;
        if(i < argc) {
            chartorune(&args.tabchar, argv[i]);
            argv[i] = 0;
        }
    } else
        s += chartorune(&args.tabchar, s);
    if(args.tabchar == '\n') {
        fprint(2, "aw come on, rob\n");
        done("rob");
    }
    break;
case 'c':   /* check order */
    args.cflag = 1;
    break;
case 'u':   /* unique */
    args.uflag = 1;
    break;
case 'v':   /* debugging noise */
    args.vflag = 1;
    break;
case 'l':
    if(*s == 0) {
        i++;
        if(i < argc) {
            args.mline = atol(argv[i]);
            argv[i] = 0;
        }
    } else
        args.mline = atol(s);
    s = strchr(s, 0);
    break;

case 'M':   /* month */
case 'b':   /* skip blanks */
case 'd':   /* directory order */
case 'f':   /* fold case */
case 'g':   /* floating numbers */
case 'i':   /* ignore non-ascii */
case 'n':   /* numbers */
case 'r':   /* reverse */
case 'w':   /* ignore white */
    if(args.nfield > 0)
```

```
                fprint(2, "sort: global field set after -k\n");
            setfield(0, c);
            break;
        case 'm':
            /* option m silently ignored but required by posix */
            break;
        default:
            fprint(2, "sort: unknown option: -%C\n", c);
            done("option");
        }
        continue;
    }
    if(c == '+') {
        argv[i] = 0;        /* clobber args processed */
        c = *s;
        if(c == '.' || (c >= '0' && c <= '9')) {
            newfield();
            f = &args.field[args.nfield];
            dofield(s, &f->beg1, &f->beg2, 0, 0);
            if(f->flags & Bflag) {
                f->flags |= B1flag;
                f->flags &= ~Bflag;
            }
            hadplus = 1;
            continue;
        }
        fprint(2, "sort: unknown option: +%C\n", c);
        done("option");
    }
    args.nfile++;
}

for(i=0; i<=args.nfield; i++) {
    f = &args.field[i];

    /*
     * global options apply to fields that
     * specify no options
     */
    if(f->flags == 0) {
        f->flags = args.field[0].flags;
        if(args.field[0].flags & Bflag)
            f->flags |= B1flag;
    }


    /*
     * build buildkey specification
     */
    switch(f->flags & ~(Bflag|B1flag)) {
    default:
        fprint(2, "sort: illegal combination of flags: %lx\n", f->flags);
        done("option");
    case 0:
        f->dokey = dokey_;
        break;
    case Rflag:
        f->dokey = dokey_r;
        break;
    case Gflag:
```

```
        case Nflag:
        case Gflag|Nflag:
        case Gflag|Rflag:
        case Nflag|Rflag:
        case Gflag|Nflag|Rflag:
            f->dokey = dokey_gn;
            break;
        case Mflag:
        case Mflag|Rflag:
            f->dokey = dokey_m;
            makemapm(f);
            break;
        case Dflag:
        case Dflag|Fflag:
        case Dflag|Fflag|Iflag:
        case Dflag|Fflag|Iflag|Rflag:
        case Dflag|Fflag|Iflag|Rflag|Wflag:
        case Dflag|Fflag|Iflag|Wflag:
        case Dflag|Fflag|Rflag:
        case Dflag|Fflag|Rflag|Wflag:
        case Dflag|Fflag|Wflag:
        case Dflag|Iflag:
        case Dflag|Iflag|Rflag:
        case Dflag|Iflag|Rflag|Wflag:
        case Dflag|Iflag|Wflag:
        case Dflag|Rflag:
        case Dflag|Rflag|Wflag:
        case Dflag|Wflag:
        case Fflag:
        case Fflag|Iflag:
        case Fflag|Iflag|Rflag:
        case Fflag|Iflag|Rflag|Wflag:
        case Fflag|Iflag|Wflag:
        case Fflag|Rflag:
        case Fflag|Rflag|Wflag:
        case Fflag|Wflag:
        case Iflag:
        case Iflag|Rflag:
        case Iflag|Rflag|Wflag:
        case Iflag|Wflag:
        case Wflag:
            f->dokey = dokey_dfi;
            makemapd(f);
            break;
        }
    }

    /*
     * random spot checks
     */
    if(args.nfile > 1 && args.cflag) {
        fprint(2, "sort: -c can have at most one input file\n");
        done("option");
    }
    return;
}

uchar*
skip(uchar *l, int n1, int n2, int bflag, int endfield)
{
```

```
int i, c, tc;
Rune r;

if(endfield && n1 < 0)
    return 0;

c = *l++;
tc = args.tabchar;
if(tc) {
    if(tc < Runeself) {
        for(i=n1; i>0; i--) {
            while(c != tc) {
                if(c == '\n')
                    return 0;
                c = *l++;
            }
            if(!(endfield && i == 1))
                c = *l++;
        }
    } else {
        l--;
        l += chartorune(&r, (char*)l);
        for(i=n1; i>0; i--) {
            while(r != tc) {
                if(r == '\n')
                    return 0;
                l += chartorune(&r, (char*)l);
            }
            if(!(endfield && i == 1))
                l += chartorune(&r, (char*)l);
        }
        c = r;
    }
} else {
    for(i=n1; i>0; i--) {
        while(c == ' ' || c == '\t')
            c = *l++;
        while(c != ' ' && c != '\t') {
            if(c == '\n')
                return 0;
            c = *l++;
        }
    }
}

if(bflag)
    while(c == ' ' || c == '\t')
        c = *l++;

l--;
for(i=n2; i>0; i--) {
    c = *l;
    if(c < Runeself) {
        if(c == '\n')
            return 0;
        l++;
        continue;
    }
    l += chartorune(&r, (char*)l);
}
```

```
    return l;
}

void
dokey_gn(Key *k, uchar *lp, uchar *lpe, Field *f)
{
    uchar *kp;
    int c, cl, dp;
    int state, nzero, exp, expsign, rflag;

    cl = k->klen + 3;
    kp = k->key + cl;   /* skip place for sign, exponent[2] */

    nzero = 0;      /* number of trailing zeros */
    exp = 0;        /* value of the exponent */
    expsign = 0;        /* sign of the exponent */
    dp = 0x4040;        /* location of decimal point */
    rflag = f->flags&Rflag; /* xor of rflag and - sign */
    state = NSstart;

    for(;; lp++) {
        if(lp >= lpe)
            break;
        c = *lp;

        if(c == ' ' || c == '\t') {
            switch(state) {
            case NSstart:
            case NSsign:
                continue;
            }
            break;
        }
        if(c == '+' || c == '-') {
            switch(state) {
            case NSstart:
                state = NSsign;
                if(c == '-')
                    rflag = !rflag;
                continue;
            case NSexp:
                state = NSexpsign;
                if(c == '-')
                    expsign = 1;
                continue;
            }
            break;
        }
        if(c == '0') {
            switch(state) {
            case NSdigit:
                if(rflag)
                    c = ~c;
                *kp++ = c;
                cl++;
                nzero++;
                dp++;
                state = NSdigit;
                continue;
            case NSfract:
```

```
            if(rflag)
                c = ~c;
            *kp++ = c;
            cl++;
            nzero++;
            state = NSfract;
            continue;
        case NSstart:
        case NSsign:
        case NSzero:
            state = NSzero;
            continue;
        case NSzerofract:
        case NSpoint:
            dp--;
            state = NSzerofract;
            continue;
        case NSexpsign:
        case NSexp:
        case NSexpdigit:
            exp = exp*10 + (c - '0');
            state = NSexpdigit;
            continue;
        }
        break;
    }
    if(c >= '1' && c <= '9') {
        switch(state) {
        case NSzero:
        case NSstart:
        case NSsign:
        case NSdigit:
            if(rflag)
                c = ~c;
            *kp++ = c;
            cl++;
            nzero = 0;
            dp++;
            state = NSdigit;
            continue;
        case NSzerofract:
        case NSpoint:
        case NSfract:
            if(rflag)
                c = ~c;
            *kp++ = c;
            cl++;
            nzero = 0;
            state = NSfract;
            continue;
        case NSexpsign:
        case NSexp:
        case NSexpdigit:
            exp = exp*10 + (c - '0');
            state = NSexpdigit;
            continue;
        }
        break;
    }
    if(c == '.') {
```

```
        switch(state) {
        case NSstart:
        case NSsign:
            state = NSpoint;
            continue;
        case NSzero:
            state = NSzerofract;
            continue;
        case NSdigit:
            state = NSfract;
            continue;
        }
        break;
    }
    if((f->flags & Gflag) && (c == 'e' || c == 'E')) {
        switch(state) {
        case NSdigit:
        case NSfract:
            state = NSexp;
            continue;
        }
        break;
    }
    break;
}

switch(state) {
/*
 * result is zero
 */
case NSstart:
case NSsign:
case NSzero:
case NSzerofract:
case NSpoint:
    kp = k->key + k->klen;
    k->klen += 2;
    kp[0] = 0x20;   /* between + and - */
    kp[1] = 0;
    return;
/*
 * result has exponent
 */
case NSexpsign:
case NSexp:
case NSexpdigit:
    if(expsign)
        exp = -exp;
    dp += exp;

/*
 * result is fixed point number
 */
case NSdigit:
case NSfract:
    kp -= nzero;
    cl -= nzero;
    break;
}
```

```
    /*
     * end of number
     */
    c = 0;
    if(rflag)
        c = ~c;
    *kp = c;

    /*
     * sign and exponent
     */
    c = 0x30;
    if(rflag) {
        c = 0x10;
        dp = ~dp;
    }
    kp = k->key + k->klen;
    kp[0] = c;
    kp[1] = (dp >> 8);
    kp[2] = dp;
    k->klen = cl+1;
}

void
dokey_m(Key *k, uchar *lp, uchar *lpe, Field *f)
{
    uchar *kp;
    Rune r, place[3];
    int c, cl, pc;
    int rflag;

    rflag = f->flags&Rflag;
    pc = 0;

    cl = k->klen;
    kp = k->key + cl;

    for(;;) {
        /*
         * get the character
         */
        if(lp >= lpe)
            break;
        c = *lp;
        if(c >= Runeself) {
            lp += chartorune(&r, (char*)lp);
            c = r;
        } else
            lp++;

        if(c < nelem(f->mapto)) {
            c = f->mapto[c];
            if(c == 0)
                continue;
        }
        place[pc++] = c;
        if(pc < 3)
            continue;
        for(c=11; c>=0; c--)
            if(memcmp(month[c], place, sizeof(place)) == 0)
```

```
                break;
        c += 10;
        if(rflag)
            c = ~c;
        *kp++ = c;
        cl++;
        break;
    }

    c = 0;
    if(rflag)
        c = ~c;
    *kp = c;
    k->klen = cl+1;
}

void
dokey_dfi(Key *k, uchar *lp, uchar *lpe, Field *f)
{
    uchar *kp;
    Rune r;
    int c, cl, n, rflag;

    cl = k->klen;
    kp = k->key + cl;
    rflag = f->flags & Rflag;

    for(;;) {
        /*
         * get the character
         */
        if(lp >= lpe)
            break;
        c = *lp;
        if(c >= Runeself) {
            lp += chartorune(&r, (char*)lp);
            c = r;
        } else
            lp++;

        /*
         * do the various mappings.
         * the common case is handled
         * completely by the table.
         */
        if(c != 0 && c < Runeself) {
            c = f->mapto[c];
            if(c) {
                *kp++ = c;
                cl++;
            }
            continue;
        }

        /*
         * for characters out of range,
         * the table does not do Rflag.
         * ignore is based on mapto[nelem(f->mapto)-1]
         */
        if(c != 0 && c < nelem(f->mapto)) {
```

```
            c = f->mapto[c];
            if(c == 0)
                continue;
        } else {
            if(f->mapto[nelem(f->mapto)-1] == 0)
                continue;
            /*
             * consider building maps as necessary
             */
            if(f->flags & Fflag)
                c = tolowerrune(tobaserune(c));
            if(f->flags & Dflag && !isalpharune(c) &&
                !isdigitrune(c) && !isspacerune(c))
                continue;
            if((f->flags & Wflag) && isspacerune(c))
                continue;
        }

        /*
         * put it in the key
         */
        r = c;
        n = runetochar((char*)kp, &r);
        kp += n;
        cl += n;
        if(rflag)
            while(n > 0) {
                kp[-n] = ~kp[-n];
                n--;
            }
    }

    /*
     * end of key
     */
    k->klen = cl+1;
    if(rflag) {
        *kp = ~0;
        return;
    }
    *kp = 0;
}

void
dokey_r(Key *k, uchar *lp, uchar *lpe, Field*)
{
    int cl, n;
    uchar *kp;

    n = lpe - lp;
    if(n < 0)
        n = 0;
    cl = k->klen;
    kp = k->key + cl;
    k->klen = cl+n+1;

    lpe -= 3;
    while(lp < lpe) {
        kp[0] = ~lp[0];
        kp[1] = ~lp[1];
```

```
        kp[2] = ~lp[2];
        kp[3] = ~lp[3];
        kp += 4;
        lp += 4;
    }

    lpe += 3;
    while(lp < lpe)
        *kp++ = ~*lp++;
    *kp = ~0;
}

void
dokey_(Key *k, uchar *lp, uchar *lpe, Field*)
{
    int n, cl;
    uchar *kp;

    n = lpe - lp;
    if(n < 0)
        n = 0;
    cl = k->klen;
    kp = k->key + cl;
    k->klen = cl+n+1;
    memmove(kp, lp, n);
    kp[n] = 0;
}

void
buildkey(Line *l)
{
    Key *k;
    uchar *lp, *lpe;
    int ll, kl, cl, i, n;
    Field *f;

    ll = l->llen - 1;
    kl = 0;         /* allocated length */
    cl = 0;         /* current length */
    k = 0;

    for(i=1; i<=args.nfield; i++) {
        f = &args.field[i];
        lp = skip(l->line, f->beg1, f->beg2, f->flags&B1flag, 0);
        if(lp == 0)
            lp = l->line + ll;
        lpe = skip(l->line, f->end1, f->end2, f->flags&Bflag, 1);
        if(lpe == 0)
            lpe = l->line + ll;
        n = (lpe - lp) + 1;
        if(n <= 0)
            n = 1;
        if(cl+(n+4) > kl) {
            kl = cl+(n+4);
            k = realloc(k, sizeof(Key) +
                (kl-1)*sizeof(k->key[0]));
            if(k == 0)
                nomem();
        }
        k->klen = cl;
```

```
        (*f->dokey)(k, lp, lpe, f);
        cl = k->klen;
    }

    /*
     * global comparisons
     */
    if(!(args.uflag && cl > 0)) {
        f = &args.field[0];
        if(cl+(ll+4) > kl) {
            kl = cl+(ll+4);
            k = realloc(k, sizeof(Key) +
                (kl-1)*sizeof(k->key[0]));
            if(k == 0)
                nomem();
        }
        k->klen = cl;
        (*f->dokey)(k, l->line, l->line+ll, f);
        cl = k->klen;
    }

    l->key = k;
    k->klen = cl;

    if(args.vflag) {
        if(write(2, l->line, l->llen) != l->llen)
            exits("write");
        for(i=0; i<k->klen; i++) {
            fprint(2, " %.2x", k->key[i]);
            if(k->key[i] == 0x00 || k->key[i] == 0xff)
                fprint(2, "\n");
        }
    }
}

void
makemapm(Field *f)
{
    int i, c;

    for(i=0; i<nelem(f->mapto); i++) {
        c = 1;
        if(i == ' ' || i == '\t')
            c = 0;
        if(i >= 'a' && i <= 'z')
            c = i + ('A' - 'a');
        if(i >= 'A' && i <= 'Z')
            c = i;
        f->mapto[i] = c;
        if(args.vflag) {
            if((i & 15) == 0)
                fprint(2, " ");
            fprint(2, " %.2x", c);
            if((i & 15) == 15)
                fprint(2, "\n");
        }
    }
}

void
```

```
makemapd(Field *f)
{
    int i, c;

    for(i=0; i<nelem(f->mapto); i++) {
        c = i;
        if(f->flags & Iflag)
            if(c < 040 || c > 0176)
                c = -1;
        if((f->flags & Wflag) && c >= 0)
            if(c == ' ' || c == '\t')
                c = -1;
        if((f->flags & Dflag) && c >= 0)
            if(!(c == ' ' || c == '\t' ||
                (c >= 'a' && c <= 'z') ||
                (c >= 'A' && c <= 'Z') ||
                (c >= '0' && c <= '9'))){
                if(!isupperrune(c = toupperrune(c)))
                    c = -1;
            }
        if((f->flags & Fflag) && c >= 0)
            c = toupperrune(tobaserune(c));
        if((f->flags & Rflag) && c >= 0 && i > 0 && i < Runeself)
            c = ~c & 0xff;
        if(c < 0)
            c = 0;
        f->mapto[i] = c;
        if(args.vflag) {
            if((i & 15) == 0)
                fprint(2, " ");
            fprint(2, " %.2x", c);
            if((i & 15) == 15)
                fprint(2, "\n");
        }
    }
}

Rune*   month[12] =
{
    L"JAN",
    L"FEB",
    L"MAR",
    L"APR",
    L"MAY",
    L"JUN",
    L"JUL",
    L"AUG",
    L"SEP",
    L"OCT",
    L"NOV",
    L"DEC",
};

/************** radix sort ***********/

enum
{
    Threshold   = 14,
};
```

```
void    rsort4(Key***, ulong, int);
void    bsort4(Key***, ulong, int);

void
sort4(void *a, ulong n)
{
    if(n > Threshold)
        rsort4((Key***)a, n, 0);
    else
        bsort4((Key***)a, n, 0);
}

void
rsort4(Key ***a, ulong n, int b)
{
    Key ***ea, ***t, ***u, **t1, **u1, *k;
    Key ***part[257];
    static long count[257];
    long clist[257+257], *cp, *cp1;
    int c, lowc, higc;

    /*
     * pass 1 over all keys,
     * count the number of each key[b].
     * find low count and high count.
     */
    lowc = 256;
    higc = 0;
    ea = a+n;
    for(t=a; t<ea; t++) {
        k = **t;
        n = k->klen;
        if(b >= n) {
            count[256]++;
            continue;
        }
        c = k->key[b];
        n = count[c]++;
        if(n == 0) {
            if(c < lowc)
                lowc = c;
            if(c > higc)
                higc = c;
        }
    }

    /*
     * pass 2 over all counts,
     * put partition pointers in part[c].
     * save compacted indexes and counts
     * in clist[].
     */
    t = a;
    n = count[256];
    clist[0] = n;
    part[256] = t;
    t += n;

    cp1 = clist+1;
    cp = count+lowc;
```

```
    for(c=lowc; c<=higc; c++,cp++) {
        n = *cp;
        if(n) {
            cp1[0] = n;
            cp1[1] = c;
            cp1 += 2;
            part[c] = t;
            t += n;
        }
    }
    *cp1 = 0;

    /*
     * pass 3 over all counts.
     * chase lowest pointer in each partition
     * around a permutation until it comes
     * back and is stored where it started.
     * static array, count[], should be
     * reduced to zero entries except maybe
     * count[256].
     */
    for(cp1=clist+1; cp1[0]; cp1+=2) {
        c = cp1[1];
        cp = count+c;
        while(*cp) {
            t1 = *part[c];
            for(;;) {
                k = *t1;
                n = 256;
                if(b < k->klen)
                    n = k->key[b];
                u = part[n]++;
                count[n]--;
                u1 = *u;
                *u = t1;
                if(n == c)
                    break;
                t1 = u1;
            }
        }
    }

    /*
     * pass 4 over all partitions.
     * call recursively.
     */
    b++;
    t = a + clist[0];
    count[256] = 0;
    for(cp1=clist+1; n=cp1[0]; cp1+=2) {
        if(n > Threshold)
            rsort4(t, n, b);
        else
        if(n > 1)
            bsort4(t, n, b);
        t += n;
    }
}

/*
```

```
 * bubble sort to pick up
 * the pieces.
 */
void
bsort4(Key ***a, ulong n, int b)
{
    Key ***i, ***j, ***k, ***l, **t;
    Key *ka, *kb;
    int n1, n2;

    l = a+n;
    j = a;

loop:
    i = j;
    j++;
    if(j >= l)
        return;

    ka = **i;
    kb = **j;
    n1 = ka->klen - b;
    n2 = kb->klen - b;
    if(n1 > n2)
        n1 = n2;
    if(n1 <= 0)
        goto loop;
    n2 = ka->key[b] - kb->key[b];
    if(n2 == 0)
        n2 = memcmp(ka->key+b, kb->key+b, n1);
    if(n2 <= 0)
        goto loop;

    for(;;) {
        k = i+1;

        t = *k;
        *k = *i;
        *i = t;

        if(i <= a)
            goto loop;

        i--;
        ka = **i;
        kb = *t;
        n1 = ka->klen - b;
        n2 = kb->klen - b;
        if(n1 > n2)
            n1 = n2;
        if(n1 <= 0)
            goto loop;
        n2 = ka->key[b] - kb->key[b];
        if(n2 == 0)
            n2 = memcmp(ka->key+b, kb->key+b, n1);
        if(n2 <= 0)
            goto loop;
    }
}
```

A.7.3 text/sed.c

⟨text/sed.c 296⟩≡

```
/*
 * sed -- stream editor
 */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <regexp.h>

enum {
    DEPTH       = 20,       /* max nesting depth of {} */
    MAXCMDS     = 512,      /* max sed commands */
    ADDSIZE     = 10000,    /* size of add & read buffer */
    MAXADDS     = 20,       /* max pending adds and reads */
    LBSIZE      = 8192,     /* input line size */
    LABSIZE     = 50,       /* max number of labels */
    MAXSUB      = 10,       /* max number of sub reg exp */
    MAXFILES    = 120,      /* max output files */
};

/*
 * An address is a line #, a R.E., "$", a reference to the last
 * R.E., or nothing.
 */
typedef struct {
    enum {
        A_NONE,
        A_DOL,
        A_LINE,
        A_RE,
        A_LAST,
    }type;
    union {
        long    line;       /* Line # */
        Reprog  *rp;        /* Compiled R.E. */
    };
} Addr;

typedef struct  SEDCOM {
    Addr    ad1;            /* optional start address */
    Addr    ad2;            /* optional end address */
    union {
        Reprog  *re1;       /* compiled R.E. */
        Rune    *text;      /* added text or file name */
        struct  SEDCOM  *lb1;   /* destination command of branch */
    };
    Rune    *rhs;           /* Right-hand side of substitution */
    Biobuf* fcode;          /* File ID for read and write */
    char    command;        /* command code -see below */
    char    gfl;            /* 'Global' flag for substitutions */
    char    pfl;            /* 'print' flag for substitutions */
    char    active;         /* 1 => data between start and end */
    char    negfl;          /* negation flag */
} SedCom;

/* Command Codes for field SedCom.command */
#define ACOM    01
#define BCOM    020
```

```
#define CCOM    02
#define CDCOM   025
#define CNCOM   022
#define COCOM   017
#define CPCOM   023
#define DCOM    03
#define ECOM    015
#define EQCOM   013
#define FCOM    016
#define GCOM    027
#define CGCOM   030
#define HCOM    031
#define CHCOM   032
#define ICOM    04
#define LCOM    05
#define NCOM    012
#define PCOM    010
#define QCOM    011
#define RCOM    06
#define SCOM    07
#define TCOM    021
#define WCOM    014
#define CWCOM   024
#define YCOM    026
#define XCOM    033

typedef struct label {          /* Label symbol table */
    Rune    uninm[9];       /* Label name */
    SedCom  *chain;
    SedCom  *address;       /* Command associated with label */
} Label;

typedef struct  FILE_CACHE {        /* Data file control block */
    struct FILE_CACHE *next;    /* Forward Link */
    char    *name;          /* Name of file */
} FileCache;

SedCom pspace[MAXCMDS];         /* Command storage */
SedCom *pend = pspace+MAXCMDS;      /* End of command storage */
SedCom *rep = pspace;           /* Current fill point */

Reprog  *lastre = 0;            /* Last regular expression */
Resub   subexp[MAXSUB];         /* sub-patterns of pattern match*/

Rune    addspace[ADDSIZE];      /* Buffer for a, c, & i commands */
Rune    *addend = addspace+ADDSIZE;

SedCom  *abuf[MAXADDS];         /* Queue of pending adds & reads */
SedCom  **aptr = abuf;

struct {                /* Sed program input control block */
    enum PTYPE {            /* Either on command line or in file */
        P_ARG,
        P_FILE,
    } type;
    union PCTL {            /* Pointer to data */
        Biobuf  *bp;
        char    *curr;
    };
} prog;
```

```
Rune    genbuf[LBSIZE];         /* Miscellaneous buffer */

FileCache   *fhead = 0;     /* Head of File Cache Chain */
FileCache   *ftail = 0;     /* Tail of File Cache Chain */

Rune    *loc1;              /* Start of pattern match */
Rune    *loc2;              /* End of pattern match */
Rune    seof;               /* Pattern delimiter char */

Rune    linebuf[LBSIZE+1];      /* Input data buffer */
Rune    *lbend = linebuf+LBSIZE;    /* End of buffer */
Rune    *spend = linebuf;       /* End of input data */
Rune    *cp;                /* Current scan point in linebuf */

Rune    holdsp[LBSIZE+1];       /* Hold buffer */
Rune    *hend = holdsp+LBSIZE;      /* End of hold buffer */
Rune    *hspend = holdsp;       /* End of hold data */

int nflag;              /* Command line flags */
int gflag;

int dolflag;            /* Set when at true EOF */
int sflag;              /* Set when substitution done */
int jflag;              /* Set when jump required */
int delflag;            /* Delete current line when set */

long    lnum = 0;           /* Input line count */

char    fname[MAXFILES][40];        /* File name cache */
Biobuf  *fcode[MAXFILES];       /* File ID cache */
int nfiles = 0;         /* Cache fill point */

Biobuf  fout;               /* Output stream */
Biobuf  stdin;              /* Default input */
Biobuf* f = 0;              /* Input data */

Label   ltab[LABSIZE];          /* Label name symbol table */
Label   *labend = ltab+LABSIZE;     /* End of label table */
Label   *lab = ltab+1;          /* Current Fill point */

int depth = 0;          /* {} stack pointer */

Rune    bad;                /* Dummy err ptr reference */
Rune    *badp = &bad;


char    CGMES[]  =  "%S command garbled: %S";
char    TMMES[]  =  "Too much text: %S";
char    LTL[]    =  "Label too long: %S";
char    AD0MES[] =  "No addresses allowed: %S";
char    AD1MES[] =  "Only one address allowed: %S";

void    address(Addr *);
void    arout(void);
int cmp(char *, char *);
int rcmp(Rune *, Rune *);
void    command(SedCom *);
Reprog  *compile(void);
Rune    *compsub(Rune *, Rune *);
```

```
void    dechain(void);
void    dosub(Rune *);
int ecmp(Rune *, Rune *, int);
void    enroll(char *);
void    errexit(void);
int executable(SedCom *);
void    execute(void);
void    fcomp(void);
long    getrune(void);
Rune    *gline(Rune *);
int match(Reprog *, Rune *);
void    newfile(enum PTYPE, char *);
int     opendata(void);
Biobuf  *open_file(char *);
Rune    *place(Rune *, Rune *, Rune *);
void    quit(char *, ...);
int rline(Rune *, Rune *);
Label   *search(Label *);
int substitute(SedCom *);
char    *text(char *);
Rune    *stext(Rune *, Rune *);
int ycomp(SedCom *);
char *  trans(int c);
void    putline(Biobuf *bp, Rune *buf, int n);

void
main(int argc, char **argv)
{
    int compfl;

    lnum = 0;
    Binit(&fout, 1, OWRITE);
    fcode[nfiles++] = &fout;
    compfl = 0;

    if(argc == 1)
        exits(0);
    ARGBEGIN{
    case 'e':
        if (argc <= 1)
            quit("missing pattern");
        newfile(P_ARG, ARGF());
        fcomp();
        compfl = 1;
        continue;
    case 'f':
        if(argc <= 1)
            quit("no pattern-file");
        newfile(P_FILE, ARGF());
        fcomp();
        compfl = 1;
        continue;
    case 'g':
        gflag++;
        continue;
    case 'n':
        nflag++;
        continue;
    default:
        fprint(2, "sed: Unknown flag: %c\n", ARGC());
```

```
        continue;
    } ARGEND

    if(compfl == 0) {
        if (--argc < 0)
            quit("missing pattern");
        newfile(P_ARG, *argv++);
        fcomp();
    }

    if(depth)
        quit("Too many {'s");

    ltab[0].address = rep;

    dechain();

    if(argc <= 0)
        enroll(0);      /* Add stdin to cache */
    else
        while(--argc >= 0)
            enroll(*argv++);
    execute();
    exits(0);
}

void
fcomp(void)
{
    int i;
    Label   *lpt;
    Rune    *tp;
    SedCom  *pt, *pt1;
    static Rune *p = addspace;
    static SedCom   **cmpend[DEPTH];    /* stack of {} operations */

    while (rline(linebuf, lbend) >= 0) {
        cp = linebuf;
comploop:
        while(*cp == L' ' || *cp == L'\t')
            cp++;
        if(*cp == L'\0' || *cp == L'#')
            continue;
        if(*cp == L';') {
            cp++;
            goto comploop;
        }

        address(&rep->ad1);
        if (rep->ad1.type != A_NONE) {
            if (rep->ad1.type == A_LAST) {
                if (!lastre)
                    quit("First RE may not be null");
                rep->ad1.type = A_RE;
                rep->ad1.rp = lastre;
            }
            if(*cp == L',' || *cp == L';') {
                cp++;
                address(&rep->ad2);
                if (rep->ad2.type == A_LAST) {
```

```
                rep->ad2.type = A_RE;
                rep->ad2.rp = lastre;
            }
        } else
            rep->ad2.type = A_NONE;
    }
    while(*cp == L' ' || *cp == L'\t')
        cp++;

swit:
    switch(*cp++) {
    default:
        quit("Unrecognized command: %S", linebuf);

    case '!':
        rep->negfl = 1;
        goto swit;

    case '{':
        rep->command = BCOM;
        rep->negfl = !rep->negfl;
        cmpend[depth++] = &rep->lb1;
        if(++rep >= pend)
            quit("Too many commands: %S", linebuf);
        if(*cp == '\0')
            continue;
        goto comploop;

    case '}':
        if(rep->ad1.type != A_NONE)
            quit(ADOMES, linebuf);
        if(--depth < 0)
            quit("Too many }'s");
        *cmpend[depth] = rep;
        if(*cp == 0)
            continue;
        goto comploop;

    case '=':
        rep->command = EQCOM;
        if(rep->ad2.type != A_NONE)
            quit(AD1MES, linebuf);
        break;

    case ':':
        if(rep->ad1.type != A_NONE)
            quit(ADOMES, linebuf);

        while(*cp == L' ')
            cp++;
        tp = lab->uninm;
        while (*cp && *cp != L';' && *cp != L' ' &&
            *cp != L'\t' && *cp != L'#') {
            *tp++ = *cp++;
            if(tp >= &lab->uninm[8])
                quit(LTL, linebuf);
        }
        *tp = L'\0';

        if (*lab->uninm == L'\0')       /* no label? */
```

```
            quit(CGMES, L":", linebuf);
        if(lpt = search(lab)) {
            if(lpt->address)
                quit("Duplicate labels: %S", linebuf);
        } else {
            lab->chain = 0;
            lpt = lab;
            if(++lab >= labend)
                quit("Too many labels: %S", linebuf);
        }
        lpt->address = rep;
        if (*cp == L'#')
            continue;
        rep--;          /* reuse this slot */
        break;

    case 'a':
        rep->command = ACOM;
        if(rep->ad2.type != A_NONE)
            quit(AD1MES, linebuf);
        if(*cp == L'\\')
            cp++;
        if(*cp++ != L'\n')
            quit(CGMES, L"a", linebuf);
        rep->text = p;
        p = stext(p, addend);
        break;
    case 'c':
        rep->command = CCOM;
        if(*cp == L'\\')
            cp++;
        if(*cp++ != L'\n')
            quit(CGMES, L"c", linebuf);
        rep->text = p;
        p = stext(p, addend);
        break;
    case 'i':
        rep->command = ICOM;
        if(rep->ad2.type != A_NONE)
            quit(AD1MES, linebuf);
        if(*cp == L'\\')
            cp++;
        if(*cp++ != L'\n')
            quit(CGMES, L"i", linebuf);
        rep->text = p;
        p = stext(p, addend);
        break;

    case 'g':
        rep->command = GCOM;
        break;

    case 'G':
        rep->command = CGCOM;
        break;

    case 'h':
        rep->command = HCOM;
        break;
```

```
        case 'H':
            rep->command = CHCOM;
            break;

        case 't':
            rep->command = TCOM;
            goto jtcommon;

        case 'b':
            rep->command = BCOM;
jtcommon:
            while(*cp == L' ')
                cp++;
            if(*cp == L'\0' || *cp == L';') {
                /* no label; jump to end */
                if(pt = ltab[0].chain) {
                    while((pt1 = pt->lb1) != nil)
                        pt = pt1;
                    pt->lb1 = rep;
                } else
                    ltab[0].chain = rep;
                break;
            }

            /* copy label into lab->uninm */
            tp = lab->uninm;
            while((*tp = *cp++) != L'\0' && *tp != L';')
                if(++tp >= &lab->uninm[8])
                    quit(LTL, linebuf);
            cp--;
            *tp = L'\0';

            if (*lab->uninm == L'\0')
                /* shouldn't get here */
                quit(CGMES, L"b or t", linebuf);
            if((lpt = search(lab)) != nil) {
                if(lpt->address)
                    rep->lb1 = lpt->address;
                else {
                    for(pt = lpt->chain; pt != nil &&
                        (pt1 = pt->lb1) != nil; pt = pt1)
                        ;
                    if (pt)
                        pt->lb1 = rep;
                }
            } else {            /* add new label */
                lab->chain = rep;
                lab->address = 0;
                if(++lab >= labend)
                    quit("Too many labels: %S", linebuf);
            }
            break;

        case 'n':
            rep->command = NCOM;
            break;

        case 'N':
            rep->command = CNCOM;
            break;
```

```
case 'p':
    rep->command = PCOM;
    break;

case 'P':
    rep->command = CPCOM;
    break;

case 'r':
    rep->command = RCOM;
    if(rep->ad2.type != A_NONE)
        quit(AD1MES, linebuf);
    if(*cp++ != L' ')
        quit(CGMES, L"r", linebuf);
    rep->text = p;
    p = stext(p, addend);
    break;

case 'd':
    rep->command = DCOM;
    break;

case 'D':
    rep->command = CDCOM;
    rep->lb1 = pspace;
    break;

case 'q':
    rep->command = QCOM;
    if(rep->ad2.type != A_NONE)
        quit(AD1MES, linebuf);
    break;

case 'l':
    rep->command = LCOM;
    break;

case 's':
    rep->command = SCOM;
    seof = *cp++;
    if ((rep->re1 = compile()) == 0) {
        if(!lastre)
            quit("First RE may not be null.");
        rep->re1 = lastre;
    }
    rep->rhs = p;
    if((p = compsub(p, addend)) == 0)
        quit(CGMES, L"s", linebuf);
    if(*cp == L'g') {
        cp++;
        rep->gfl++;
    } else if(gflag)
        rep->gfl++;

    if(*cp == L'p') {
        cp++;
        rep->pfl = 1;
    }
```

```
            if(*cp == L'P') {
                cp++;
                rep->pfl = 2;
            }

            if(*cp == L'w') {
                cp++;
                if(*cp++ !=  L' ')
                    quit(CGMES, L"s", linebuf);
                text(fname[nfiles]);
                for(i = nfiles - 1; i >= 0; i--)
                    if(cmp(fname[nfiles], fname[i]) == 0) {
                        rep->fcode = fcode[i];
                        goto done;
                    }
                if(nfiles >= MAXFILES)
                    quit("Too many files in w commands 1");
                rep->fcode = open_file(fname[nfiles]);
            }
            break;

        case 'w':
            rep->command = WCOM;
            if(*cp++ != L' ')
                quit(CGMES, L"w", linebuf);
            text(fname[nfiles]);
            for(i = nfiles - 1; i >= 0; i--)
                if(cmp(fname[nfiles], fname[i]) == 0) {
                    rep->fcode = fcode[i];
                    goto done;
                }
            if(nfiles >= MAXFILES){
                fprint(2, "sed: Too many files in w commands 2 \n");
                fprint(2, "nfiles = %d; MAXF = %d\n",
                    nfiles, MAXFILES);
                errexit();
            }
            rep->fcode = open_file(fname[nfiles]);
            break;

        case 'x':
            rep->command = XCOM;
            break;

        case 'y':
            rep->command = YCOM;
            seof = *cp++;
            if (ycomp(rep) == 0)
                quit(CGMES, L"y", linebuf);
            break;

        }
done:
        if(++rep >= pend)
            quit("Too many commands, last: %S", linebuf);
        if(*cp++ != L'\0') {
            if(cp[-1] == L';')
                goto comploop;
            quit(CGMES, cp - 1, linebuf);
        }
```

```
    }
}

Biobuf *
open_file(char *name)
{
    int fd;
    Biobuf *bp;

    if ((bp = malloc(sizeof(Biobuf))) == 0)
        quit("Out of memory");
    if ((fd = open(name, OWRITE)) < 0 &&
        (fd = create(name, OWRITE, 0666)) < 0)
        quit("Cannot create %s", name);
    Binit(bp, fd, OWRITE);
    Bseek(bp, 0, 2);
    fcode[nfiles++] = bp;
    return bp;
}

Rune *
compsub(Rune *rhs, Rune *end)
{
    Rune r;

    while ((r = *cp++) != '\0') {
        if(r == '\\') {
            if (rhs < end)
                *rhs++ = Runemax;
            else
                return 0;
            r = *cp++;
            if(r == 'n')
                r = '\n';
        } else {
            if(r == seof) {
                if (rhs < end)
                    *rhs++ = '\0';
                else
                    return 0;
                return rhs;
            }
        }
        if (rhs < end)
            *rhs++ = r;
        else
            return 0;
    }
    return 0;
}

Reprog *
compile(void)
{
    Rune c;
    char *ep;
    char expbuf[512];

    if((c = *cp++) == seof)     /* L'//' */
        return 0;
```

```
    ep = expbuf;
    do {
        if (c == L'\0' || c == L'\n')
            quit(TMMES, linebuf);
        if (c == L'\\') {
            if (ep >= expbuf+sizeof(expbuf))
                quit(TMMES, linebuf);
            ep += runetochar(ep, &c);
            if ((c = *cp++) == L'n')
                c = L'\n';
        }
        if (ep >= expbuf + sizeof(expbuf))
            quit(TMMES, linebuf);
        ep += runetochar(ep, &c);
    } while ((c = *cp++) != seof);
    *ep = 0;
    return lastre = regcomp(expbuf);
}

void
regerror(char *s)
{
    USED(s);
    quit(CGMES, L"r.e.-using", linebuf);
}

void
newfile(enum PTYPE type, char *name)
{
    if (type == P_ARG)
        prog.curr = name;
    else if ((prog.bp = Bopen(name, OREAD)) == 0)
        quit("Cannot open pattern-file: %s\n", name);
    prog.type = type;
}

int
rline(Rune *buf, Rune *end)
{
    long c;
    Rune r;

    while ((c = getrune()) >= 0) {
        r = c;
        if (r == '\\') {
            if (buf <= end)
                *buf++ = r;
            if ((c = getrune()) < 0)
                break;
            r = c;
        } else if (r == '\n') {
            *buf = '\0';
            return 1;
        }
        if (buf <= end)
            *buf++ = r;
    }
    *buf = '\0';
    return -1;
}
```

```
long
getrune(void)
{
    long c;
    Rune r;
    char *p;

    if (prog.type == P_ARG) {
        if ((p = prog.curr) != 0) {
            if (*p) {
                prog.curr += chartorune(&r, p);
                c = r;
            } else {
                c = '\n';   /* fake an end-of-line */
                prog.curr = 0;
            }
        } else
            c = -1;
    } else if ((c = Bgetrune(prog.bp)) < 0)
        Bterm(prog.bp);
    return c;
}

void
address(Addr *ap)
{
    int c;
    long lno;

    if((c = *cp++) == '$')
        ap->type = A_DOL;
    else if(c == '/') {
        seof = c;
        if (ap->rp = compile())
            ap->type = A_RE;
        else
            ap->type = A_LAST;
    }
    else if (c >= '0' && c <= '9') {
        lno = c - '0';
        while ((c = *cp) >= '0' && c <= '9')
            lno = lno*10 + *cp++ - '0';
        if(!lno)
            quit("line number 0 is illegal",0);
        ap->type = A_LINE;
        ap->line = lno;
    }
    else {
        cp--;
        ap->type = A_NONE;
    }
}

int
cmp(char *a, char *b)       /* compare characters */
{
    while(*a == *b++)
        if (*a == '\0')
            return 0;
```

```
        else
            a++;
    return 1;
}

int
rcmp(Rune *a, Rune *b)      /* compare runes */
{
    while(*a == *b++)
        if (*a == '\0')
            return 0;
        else
            a++;
    return 1;
}

char *
text(char *p)       /* extract character string */
{
    Rune r;

    while(*cp == ' ' || *cp == '\t')
        cp++;
    while (*cp) {
        if ((r = *cp++) == '\\' && (r = *cp++) == '\0')
            break;
        if (r == '\n')
            while (*cp == ' ' || *cp == '\t')
                cp++;
        p += runetochar(p, &r);
    }
    *p++ = '\0';
    return p;
}

Rune *
stext(Rune *p, Rune *end)       /* extract rune string */
{
    while(*cp == L' ' || *cp == L'\t')
        cp++;
    while (*cp) {
        if (*cp == L'\\' && *++cp == L'\0')
            break;
        if (p >= end-1)
            quit(TMMES, linebuf);
        if ((*p++ = *cp++) == L'\n')
            while(*cp == L' ' || *cp == L'\t')
                cp++;
    }
    *p++ = 0;
    return p;
}


Label *
search(Label *ptr)
{
    Label   *rp;

    for (rp = ltab; rp < ptr; rp++)
```

```
        if(rcmp(rp->uninm, ptr->uninm) == 0)
            return(rp);
    return(0);
}

void
dechain(void)
{
    Label   *lptr;
    SedCom  *rptr, *trptr;

    for(lptr = ltab; lptr < lab; lptr++) {
        if(lptr->address == 0)
            quit("Undefined label: %S", lptr->uninm);
        if(lptr->chain) {
            rptr = lptr->chain;
            while((trptr = rptr->lb1) != nil) {
                rptr->lb1 = lptr->address;
                rptr = trptr;
            }
            rptr->lb1 = lptr->address;
        }
    }
}

int
ycomp(SedCom *r)
{
    int i;
    Rune *rp, *sp, *tsp;
    Rune c, highc;

    highc = 0;
    for(tsp = cp; *tsp != seof; tsp++) {
        if(*tsp == L'\\')
            tsp++;
        if(*tsp == L'\n' || *tsp == L'\0')
            return 0;
        if (*tsp > highc)
            highc = *tsp;
    }
    tsp++;
    if ((rp = r->text = (Rune *)malloc(sizeof(Rune) * (highc+2))) == nil)
        quit("Out of memory");
    *rp++ = highc;              /* save upper bound */
    for (i = 0; i <= highc; i++)
        rp[i] = i;
    sp = cp;
    while((c = *sp++) != seof) {
        if(c == L'\\' && *sp == L'n') {
            sp++;
            c = L'\n';
        }
        if((rp[c] = *tsp++) == L'\\' && *tsp == L'n') {
            rp[c] = L'\n';
            tsp++;
        }
        if(rp[c] == seof || rp[c] == L'\0') {
            free(r->re1);
            r->re1 = nil;
```

```
            return 0;
        }
    }
    if(*tsp != seof) {
        free(r->re1);
        r->re1 = nil;
        return 0;
    }
    cp = tsp+1;
    return 1;
}

void
execute(void)
{
    SedCom  *ipc;

    while (spend = gline(linebuf)){
        for(ipc = pspace; ipc->command; ) {
            if (!executable(ipc)) {
                ipc++;
                continue;
            }
            command(ipc);

            if(delflag)
                break;
            if(jflag) {
                jflag = 0;
                if((ipc = ipc->lb1) == 0)
                    break;
            } else
                ipc++;
        }
        if(!nflag && !delflag)
            putline(&fout, linebuf, spend - linebuf);
        if(aptr > abuf)
            arout();
        delflag = 0;
    }
}

/* determine if a statement should be applied to an input line */
int
executable(SedCom *ipc)
{
    if (ipc->active) {  /* Addr1 satisfied - accept until Addr2 */
        if (ipc->active == 1)       /* Second line */
            ipc->active = 2;
        switch(ipc->ad2.type) {
        case A_NONE:        /* No second addr; use first */
            ipc->active = 0;
            break;
        case A_DOL:     /* Accept everything */
            return !ipc->negfl;
        case A_LINE:        /* Line at end of range? */
            if (lnum <= ipc->ad2.line) {
                if (ipc->ad2.line == lnum)
                    ipc->active = 0;
                return !ipc->negfl;
```

```
            }
            ipc->active = 0;    /* out of range */
            return ipc->negfl;
        case A_RE:      /* Check for matching R.E. */
            if (match(ipc->ad2.rp, linebuf))
                ipc->active = 0;
            return !ipc->negfl;
        default:
            quit("Internal error");
        }
    }
    switch (ipc->ad1.type) {    /* Check first address */
    case A_NONE:            /* Everything matches */
        return !ipc->negfl;
    case A_DOL:         /* Only last line */
        if (dolflag)
            return !ipc->negfl;
        break;
    case A_LINE:            /* Check line number */
        if (ipc->ad1.line == lnum) {
            ipc->active = 1;    /* In range */
            return !ipc->negfl;
        }
        break;
    case A_RE:          /* Check R.E. */
        if (match(ipc->ad1.rp, linebuf)) {
            ipc->active = 1;    /* In range */
            return !ipc->negfl;
        }
        break;
    default:
        quit("Internal error");
    }
    return ipc->negfl;
}

int
match(Reprog *pattern, Rune *buf)
{
    if (!pattern)
        return 0;
    subexp[0].s.rsp = buf;
    subexp[0].e.ep = 0;
    if (rregexec(pattern, linebuf, subexp, MAXSUB) > 0) {
        loc1 = subexp[0].s.rsp;
        loc2 = subexp[0].e.rep;
        return 1;
    }
    loc1 = loc2 = 0;
    return 0;
}

int
substitute(SedCom *ipc)
{
    int len;

    if(!match(ipc->re1, linebuf))
        return 0;
```

```
    /*
     * we have at least one match.  some patterns, e.g. '$' or '^', can
     * produce 0-length matches, so during a global substitute we must
     * bump to the character after a 0-length match to keep from looping.
     */
    sflag = 1;
    if(ipc->gfl == 0)           /* single substitution */
        dosub(ipc->rhs);
    else
        do{             /* global substitution */
            len = loc2 - loc1;  /* length of match */
            dosub(ipc->rhs);    /* dosub moves loc2 */
            if(*loc2 == 0)      /* end of string */
                break;
            if(len == 0)        /* zero-length R.E. match */
                loc2++;     /* bump over 0-length match */
            if(*loc2 == 0)      /* end of string */
                break;
        } while(match(ipc->re1, loc2));
    return 1;
}

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

    lp = linebuf;
    sp = genbuf;
    rp = rhsbuf;
    while (lp < loc1)
        *sp++ = *lp++;
    while(c = *rp++) {
        if (c == '&') {
            sp = place(sp, loc1, loc2);
            continue;
        }
        if (c == Runemax && (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;
            }
            else {
                fprint(2, "sed: Invalid back reference \\%d\n",n);
                errexit();
            }
        }
        *sp++ = c;
        if (sp >= &genbuf[LBSIZE])
            fprint(2, "sed: Output line too long.\n");
    }
    lp = loc2;
    loc2 = sp - genbuf + linebuf;
    while (*sp++ = *lp++)
        if (sp >= &genbuf[LBSIZE])
            fprint(2, "sed: Output line too long.\n");
    lp = linebuf;
    sp = genbuf;
```

```
    while (*lp++ = *sp++)
        ;
    spend = lp - 1;
}

Rune *
place(Rune *sp, Rune *l1, Rune *l2)
{
    while (l1 < l2) {
        *sp++ = *l1++;
        if (sp >= &genbuf[LBSIZE])
            fprint(2, "sed: Output line too long.\n");
    }
    return sp;
}

char *
trans(int c)
{
    static char buf[] = "\\x0000";
    static char hex[] = "0123456789abcdef";

    switch(c) {
    case '\b':
        return "\\b";
    case '\n':
        return "\\n";
    case '\r':
        return "\\r";
    case '\t':
        return "\\t";
    case '\\':
        return "\\\\";
    }
    buf[2] = hex[(c>>12)&0xF];
    buf[3] = hex[(c>>8)&0xF];
    buf[4] = hex[(c>>4)&0xF];
    buf[5] = hex[c&0xF];
    return buf;
}

void
command(SedCom *ipc)
{
    int i, c;
    char *ucp;
    Rune *execp, *p1, *p2, *rp;

    switch(ipc->command) {
    case ACOM:
        *aptr++ = ipc;
        if(aptr >= abuf+MAXADDS)
            quit("sed: Too many appends after line %ld\n",
                (char *)lnum);
        *aptr = 0;
        break;
    case CCOM:
        delflag = 1;
        if(ipc->active == 1) {
            for(rp = ipc->text; *rp; rp++)
```

```
            Bputrune(&fout, *rp);
        Bputc(&fout, '\n');
    }
    break;
case DCOM:
    delflag++;
    break;
case CDCOM:
    p1 = p2 = linebuf;
    while(*p1 != '\n') {
        if(*p1++ == 0) {
            delflag++;
            return;
        }
    }
    p1++;
    while(*p2++ = *p1++)
        ;
    spend = p2 - 1;
    jflag++;
    break;
case EQCOM:
    Bprint(&fout, "%ld\n", lnum);
    break;
case GCOM:
    p1 = linebuf;
    p2 = holdsp;
    while(*p1++ = *p2++)
        ;
    spend = p1 - 1;
    break;
case CGCOM:
    *spend++ = '\n';
    p1 = spend;
    p2 = holdsp;
    while(*p1++ = *p2++)
        if(p1 >= lbend)
            break;
    spend = p1 - 1;
    break;
case HCOM:
    p1 = holdsp;
    p2 = linebuf;
    while(*p1++ = *p2++);
    hspend = p1 - 1;
    break;
case CHCOM:
    *hspend++ = '\n';
    p1 = hspend;
    p2 = linebuf;
    while(*p1++ = *p2++)
        if(p1 >= hend)
            break;
    hspend = p1 - 1;
    break;
case ICOM:
    for(rp = ipc->text; *rp; rp++)
        Bputrune(&fout, *rp);
    Bputc(&fout, '\n');
    break;
```

```
    case BCOM:
        jflag = 1;
        break;
    case LCOM:
        c = 0;
        for (i = 0, rp = linebuf; *rp; rp++) {
            c = *rp;
            if(c >= 0x20 && c < 0x7F && c != '\\') {
                Bputc(&fout, c);
                if(i++ > 71) {
                    Bprint(&fout, "\\\n");
                    i = 0;
                }
            } else {
                for (ucp = trans(*rp); *ucp; ucp++){
                    c = *ucp;
                    Bputc(&fout, c);
                    if(i++ > 71) {
                        Bprint(&fout, "\\\n");
                        i = 0;
                    }
                }
            }
        }
        if(c == ' ')
            Bprint(&fout, "\\n");
        Bputc(&fout, '\n');
        break;
    case NCOM:
        if(!nflag)
            putline(&fout, linebuf, spend-linebuf);

        if(aptr > abuf)
            arout();
        if((execp = gline(linebuf)) == 0) {
            delflag = 1;
            break;
        }
        spend = execp;
        break;
    case CNCOM:
        if(aptr > abuf)
            arout();
        *spend++ = '\n';
        if((execp = gline(spend)) == 0) {
            delflag = 1;
            break;
        }
        spend = execp;
        break;
    case PCOM:
        putline(&fout, linebuf, spend-linebuf);
        break;
    case CPCOM:
cpcom:
        for(rp = linebuf; *rp && *rp != '\n'; rp++)
            Bputc(&fout, *rp);
        Bputc(&fout, '\n');
        break;
    case QCOM:
```

```
        if(!nflag)
            putline(&fout, linebuf, spend-linebuf);
        if(aptr > abuf)
            arout();
        exits(0);
    case RCOM:
        *aptr++ = ipc;
        if(aptr >= &abuf[MAXADDS])
            quit("sed: Too many reads after line %ld\n",
                (char *)lnum);
        *aptr = 0;
        break;
    case SCOM:
        i = substitute(ipc);
        if(i && ipc->pfl)
            if(ipc->pfl == 1)
                putline(&fout, linebuf, spend-linebuf);
            else
                goto cpcom;
        if(i && ipc->fcode)
            goto wcom;
        break;

    case TCOM:
        if(sflag) {
            sflag = 0;
            jflag = 1;
        }
        break;

    case WCOM:
wcom:
        putline(ipc->fcode,linebuf, spend - linebuf);
        break;
    case XCOM:
        p1 = linebuf;
        p2 = genbuf;
        while(*p2++ = *p1++)
            ;
        p1 = holdsp;
        p2 = linebuf;
        while(*p2++ = *p1++)
            ;
        spend = p2 - 1;
        p1 = genbuf;
        p2 = holdsp;
        while(*p2++ = *p1++)
            ;
        hspend = p2 - 1;
        break;
    case YCOM:
        p1 = linebuf;
        p2 = ipc->text;
        for (i = *p2++; *p1; p1++)
            if (*p1 <= i)
                *p1 = p2[*p1];
        break;
    }
}
```

```
void
putline(Biobuf *bp, Rune *buf, int n)
{
    while (n--)
        Bputrune(bp, *buf++);
    Bputc(bp, '\n');
}

int
ecmp(Rune *a, Rune *b, int count)
{
    while(count--)
        if(*a++ != *b++)
            return 0;
    return 1;
}

void
arout(void)
{
    int c;
    char    *s;
    char    buf[128];
    Rune    *p1;
    Biobuf  *fi;

    for (aptr = abuf; *aptr; aptr++) {
        if((*aptr)->command == ACOM) {
            for(p1 = (*aptr)->text; *p1; p1++ )
                Bputrune(&fout, *p1);
            Bputc(&fout, '\n');
        } else {
            for(s = buf, p1 = (*aptr)->text; *p1; p1++)
                s += runetochar(s, p1);
            *s = '\0';
            if((fi = Bopen(buf, OREAD)) == 0)
                continue;
            while((c = Bgetc(fi)) >= 0)
                Bputc(&fout, c);
            Bterm(fi);
        }
    }
    aptr = abuf;
    *aptr = 0;
}

void
errexit(void)
{
    exits("error");
}

void
quit(char *fmt, ...)
{
    char *p, *ep;
    char msg[256];
    va_list arg;

    ep = msg + sizeof msg;
```

```
    p = seprint(msg, ep, "sed: ");
    va_start(arg, fmt);
    p = vseprint(p, ep, fmt, arg);
    va_end(arg);
    p = seprint(p, ep, "\n");
    write(2, msg, p - msg);
    errexit();
}

Rune *
gline(Rune *addr)
{
    long c;
    Rune *p;
    static long peekc = 0;

    if (f == 0 && opendata() < 0)
        return 0;
    sflag = 0;
    lnum++;
/*  Bflush(&fout);********* dumped 4/30/92 - bobf****/
    do {
        p = addr;
        for (c = (peekc? peekc: Bgetrune(f)); c >= 0; c = Bgetrune(f)) {
            if (c == '\n') {
                if ((peekc = Bgetrune(f)) < 0 && fhead == 0)
                    dolflag = 1;
                *p = '\0';
                return p;
            }
            if (c && p < lbend)
                *p++ = c;
        }
        /* return partial final line, adding implicit newline */
        if(p != addr) {
            *p = '\0';
            peekc = -1;
            if (fhead == 0)
                dolflag = 1;
            return p;
        }
        peekc = 0;
        Bterm(f);
    } while (opendata() > 0);       /* Switch to next stream */
    f = 0;
    return 0;
}

/*
 * Data file input section - the intent is to transparently
 *  catenate all data input streams.
 */
void
enroll(char *filename)      /* Add a file to the input file cache */
{
    FileCache *fp;

    if ((fp = (FileCache *)malloc(sizeof (FileCache))) == nil)
        quit("Out of memory");
    if (ftail == nil)
```

```
        fhead = fp;
    else
        ftail->next = fp;
    ftail = fp;
    fp->next = nil;
    fp->name = filename;        /* 0 => stdin */
}

int
opendata(void)
{
    if (fhead == nil)
        return -1;
    if (fhead->name) {
        if ((f = Bopen(fhead->name, OREAD)) == nil)
            quit("Can't open %s", fhead->name);
    } else {
        Binit(&stdin, 0, OREAD);
        f = &stdin;
    }
    fhead = fhead->next;
    return 1;
}
```

A.7.4 text/join.c

⟨text/join.c 320⟩≡

```
/*  join F1 F2 on stuff */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>

enum {
    F1,
    F2,
    NIN,
    F0,
};

#define NFLD    100 /* max field per line */
#define comp() runestrcmp(ppi[F1][j1], ppi[F2][j2])

Biobuf *f[NIN];
Rune buf[NIN][Bsize];   /* input lines */
Rune *ppi[NIN][NFLD+1]; /* pointers to fields in lines */
Rune    sep1    = ' ';  /* default field separator */
Rune    sep2    = '\t';
int j1  = 1;    /* join of this field of file 1 */
int j2  = 1;    /* join of this field of file 2 */
int a1;
int     a2;

int olist[NIN*NFLD];  /* output these fields */
int olistf[NIN*NFLD]; /* from these files */
int no;     /* number of entries in olist */
char *sepstr    = " ";
int discard;    /* count of truncated lines */
Rune    null[Bsize] = L"";
```

```
Biobuf binbuf, boutbuf;
Biobuf *bin, *bout;

char    *getoptarg(int*, char***);
int input(int);
void    join(int);
void    oparse(char*);
void    output(int, int);
Rune    *strtorune(Rune *, char *);

void
main(int argc, char **argv)
{
    int i;
    vlong off1, off2;

    bin = &binbuf;
    bout = &boutbuf;
    Binit(bin, 0, OREAD);
    Binit(bout, 1, OWRITE);

    argv0 = argv[0];
    while (argc > 1 && argv[1][0] == '-') {
        if (argv[1][1] == '\0')
            break;
        switch (argv[1][1]) {
        case '-':
            argc--;
            argv++;
            goto proceed;
        case 'a':
            switch(*getoptarg(&argc, &argv)) {
            case '1':
                a1++;
                break;
            case '2':
                a2++;
                break;
            default:
                sysfatal("incomplete option -a");
            }
            break;
        case 'e':
            strtorune(null, getoptarg(&argc, &argv));
            break;
        case 't':
            sepstr=getoptarg(&argc, &argv);
            chartorune(&sep1, sepstr);
            sep2 = sep1;
            break;
        case 'o':
            if(argv[1][2]!=0 ||
               argc>2 && strchr(argv[2],',')!=0)
                oparse(getoptarg(&argc, &argv));
            else for (no = 0; no<2*NFLD && argc>2; no++){
                if (argv[2][0] == '1' && argv[2][1] == '.') {
                    olistf[no] = F1;
                    olist[no] = atoi(&argv[2][2]);
                } else if (argv[2][0] == '2' && argv[2][1] == '.') {
                    olist[no] = atoi(&argv[2][2]);
```

```
                    olistf[no] = F2;
                } else if (argv[2][0] == '0')
                    olistf[no] = F0;
                else
                    break;
                argc--;
                argv++;
            }
            break;
        case 'j':
            if(argc <= 2)
                break;
            if (argv[1][2] == '1')
                j1 = atoi(argv[2]);
            else if (argv[1][2] == '2')
                j2 = atoi(argv[2]);
            else
                j1 = j2 = atoi(argv[2]);
            argc--;
            argv++;
            break;
        case '1':
            j1 = atoi(getoptarg(&argc, &argv));
            break;
        case '2':
            j2 = atoi(getoptarg(&argc, &argv));
            break;
        }
        argc--;
        argv++;
    }
proceed:
    for (i = 0; i < no; i++)
        if (olist[i]-- > NFLD)  /* 0 origin */
            sysfatal("field number too big in -o");
    if (argc != 3) {
        fprint(2, "usage: join [-1 x -2 y] [-o list] file1 file2\n");
        exits("usage");
    }
    if (j1 < 1  || j2 < 1)
        sysfatal("invalid field indices");
    j1--;
    j2--;   /* everyone else believes in 0 origin */

    if (strcmp(argv[1], "-") == 0)
        f[F1] = bin;
    else if ((f[F1] = Bopen(argv[1], OREAD)) == 0)
        sysfatal("can't open %s: %r", argv[1]);
    if(strcmp(argv[2], "-") == 0)
        f[F2] = bin;
    else if ((f[F2] = Bopen(argv[2], OREAD)) == 0)
        sysfatal("can't open %s: %r", argv[2]);

    off1 = Boffset(f[F1]);
    off2 = Boffset(f[F2]);
    if(Bseek(f[F2], 0, 2) >= 0){
        Bseek(f[F2], off2, 0);
        join(F2);
    }else if(Bseek(f[F1], 0, 2) >= 0){
        Bseek(f[F1], off1, 0);
```

```
        Bseek(f[F2], off2, 0);
        join(F1);
    }else
        sysfatal("neither file is randomly accessible");
    if (discard)
        sysfatal("some input line was truncated");
    exits("");
}

char *
runetostr(char *buf, Rune *r)
{
    char *s;

    for(s = buf; *r; r++)
        s += runetochar(s, r);
    *s = '\0';
    return buf;
}

Rune *
strtorune(Rune *buf, char *s)
{
    Rune *r;

    for (r = buf; *s; r++)
        s += chartorune(r, s);
    *r = '\0';
    return buf;
}

void
readboth(int n[])
{
    n[F1] = input(F1);
    n[F2] = input(F2);
}

void
seekbotreadboth(int seekf, vlong bot, int n[])
{
    Bseek(f[seekf], bot, 0);
    readboth(n);
}

void
join(int seekf)
{
    int cmp, less;
    int n[NIN];
    vlong top, bot;

    less = seekf == F2;
    top = 0;
    bot = Boffset(f[seekf]);
    readboth(n);
    while(n[F1]>0 && n[F2]>0 || (a1||a2) && n[F1]+n[F2]>0) {
        cmp = comp();
        if(n[F1]>0 && n[F2]>0 && cmp>0 || n[F1]==0) {
            if(a2)
```

```
                output(0, n[F2]);
            if (seekf == F2)
                bot = Boffset(f[seekf]);
            n[F2] = input(F2);
        } else if(n[F1]>0 && n[F2]>0 && cmp<0 || n[F2]==0) {
            if(a1)
                output(n[F1], 0);
            if (seekf == F1)
                bot = Boffset(f[seekf]);
            n[F1] = input(F1);
        } else {
            /* n[F1]>0 && n[F2]>0 && cmp==0 */
            while(n[F2]>0 && cmp==0) {
                output(n[F1], n[F2]);
                top = Boffset(f[seekf]);
                n[seekf] = input(seekf);
                cmp = comp();
            }
            seekbotreadboth(seekf, bot, n);
            for(;;) {
                cmp = comp();
                if(n[F1]>0 && n[F2]>0 && cmp==0) {
                    output(n[F1], n[F2]);
                    n[seekf] = input(seekf);
                } else if(n[F1]>0 && n[F2]>0 &&
                    (less? cmp<0 :cmp>0) || n[seekf]==0)
                    seekbotreadboth(seekf, bot, n);
                else {
                    /*
                     * n[F1]>0 && n[F2]>0 &&
                     * (less? cmp>0 :cmp<0) ||
                     * n[seekf==F1? F2: F1]==0
                     */
                    Bseek(f[seekf], top, 0);
                    bot = top;
                    n[seekf] = input(seekf);
                    break;
                }
            }
        }
    }
}

int
input(int n)        /* get input line and split into fields */
{
    int c, i, len;
    char *line;
    Rune *bp;
    Rune **pp;

    bp = buf[n];
    pp = ppi[n];
    line = Brdline(f[n], '\n');
    if (line == nil)
        return(0);
    len = Blinelen(f[n]) - 1;
    c = line[len];
    line[len] = '\0';
    strtorune(bp, line);
```

```
    line[len] = c;          /* restore delimiter */
    if (c != '\n')
        discard++;

    i = 0;
    do {
        i++;
        if (sep1 == ' ')    /* strip multiples */
            while ((c = *bp) == sep1 || c == sep2)
                bp++;   /* skip blanks */
        *pp++ = bp;     /* record beginning */
        while ((c = *bp) != sep1 && c != sep2 && c != '\0')
            bp++;
        *bp++ = '\0';       /* mark end by overwriting blank */
    } while (c != '\0' && i < NFLD-1);

    *pp = 0;
    return(i);
}

void
prfields(int f, int on, int jn)
{
    int i;
    char buf[Bsize];

    for (i = 0; i < on; i++)
        if (i != jn)
            Bprint(bout, "%s%s", sepstr, runetostr(buf, ppi[f][i]));
}

void
output(int on1, int on2)    /* print items from olist */
{
    int i;
    Rune *temp;
    char buf[Bsize];

    if (no <= 0) {  /* default case */
        Bprint(bout, "%s", runetostr(buf, on1? ppi[F1][j1]: ppi[F2][j2]));
        prfields(F1, on1, j1);
        prfields(F2, on2, j2);
        Bputc(bout, '\n');
    } else {
        for (i = 0; i < no; i++) {
            if (olistf[i]==F0 && on1>j1)
                temp = ppi[F1][j1];
            else if (olistf[i]==F0 && on2>j2)
                temp = ppi[F2][j2];
            else {
                temp = ppi[olistf[i]][olist[i]];
                if(olistf[i]==F1 && on1<=olist[i] ||
                   olistf[i]==F2 && on2<=olist[i] ||
                   *temp==0)
                    temp = null;
            }
            Bprint(bout, "%s", runetostr(buf, temp));
            if (i == no - 1)
                Bputc(bout, '\n');
            else
```

```
                Bprint(bout, "%s", sepstr);
        }
    }
}

char *
getoptarg(int *argcp, char ***argvp)
{
    int argc = *argcp;
    char **argv = *argvp;
    if(argv[1][2] != 0)
        return &argv[1][2];
    if(argc<=2 || argv[2][0]=='-')
        sysfatal("incomplete option %s", argv[1]);
    *argcp = argc-1;
    *argvp = ++argv;
    return argv[1];
}

void
oparse(char *s)
{
    for (no = 0; no<2*NFLD && *s; no++, s++) {
        switch(*s) {
        case 0:
            return;
        case '0':
            olistf[no] = F0;
            break;
        case '1':
        case '2':
            if(s[1] == '.' && isdigit(s[2])) {
                olistf[no] = *s=='1'? F1: F2;
                olist[no] = atoi(s += 2);
                break;
            }
            /* fall thru */
        default:
            sysfatal("invalid -o list");
        }
        if(s[1] == ',')
            s++;
    }
}
```

A.7.5 text/tr.c

⟨text/tr.c 326⟩≡

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

typedef struct PCB  /* Control block controlling specification parse */
{
    char    *base;      /* start of specification */
    char    *current;   /* current parse point */
    long    last;       /* last Rune returned */
    long    final;      /* final Rune in a span */
} Pcb;
```

```
uchar   bits[] = { 1, 2, 4, 8, 16, 32, 64, 128 };

#define SETBIT(a, c)        ((a)[(c)/8] |= bits[(c)&07])
#define CLEARBIT(a,c)       ((a)[(c)/8] &= ~bits[(c)&07])
#define BITSET(a,c)     ((a)[(c)/8] & bits[(c)&07])

uchar   f[(Runemax+1)/8];
uchar   t[(Runemax+1)/8];
char    wbuf[4096];
char    *wptr;

Pcb pfrom, pto;

int cflag;
int dflag;
int sflag;

void    complement(void);
void    delete(void);
void    squeeze(void);
void    translit(void);
long    canon(Pcb*);
char    *getrune(char*, Rune*);
void    Pinit(Pcb*, char*);
void    Prewind(Pcb *p);
int readrune(int, long*);
void    wflush(int);
void    writerune(int, Rune);

static void
usage(void)
{
    fprint(2, "usage: %s [-cds] [string1 [string2]]\n", argv0);
    exits("usage");
}

void
main(int argc, char **argv)
{
    ARGBEGIN{
    case 's':   sflag++; break;
    case 'd':   dflag++; break;
    case 'c':   cflag++; break;
    default:    usage();
    }ARGEND
    if(argc>0)
        Pinit(&pfrom, argv[0]);
    if(argc>1)
        Pinit(&pto, argv[1]);
    if(argc>2)
        usage();
    if(dflag) {
        if ((sflag && argc != 2) || (!sflag && argc != 1))
            usage();
        delete();
    } else {
        if (argc != 2)
            usage();
        if (cflag)
            complement();
```

```
        else translit();
    }
    exits(0);
}

void
delete(void)
{
    long c, last;

    if (cflag) {
        memset((char *) f, 0xff, sizeof f);
        while ((c = canon(&pfrom)) >= 0)
            CLEARBIT(f, c);
    } else {
        while ((c = canon(&pfrom)) >= 0)
            SETBIT(f, c);
    }
    if (sflag) {
        while ((c = canon(&pto)) >= 0)
            SETBIT(t, c);
    }

    last = 0x10000;
    while (readrune(0, &c) > 0) {
        if(!BITSET(f, c) && (c != last || !BITSET(t,c))) {
            last = c;
            writerune(1, (Rune) c);
        }
    }
    wflush(1);
}

void
complement(void)
{
    Rune *p;
    int i;
    long from, to, lastc, high;

    lastc = 0;
    high = 0;
    while ((from = canon(&pfrom)) >= 0) {
        if (from > high) high = from;
        SETBIT(f, from);
    }
    while ((to = canon(&pto)) > 0) {
        if (to > high) high = to;
        SETBIT(t,to);
    }
    Prewind(&pto);
    if ((p = (Rune *) malloc((high+1)*sizeof(Rune))) == 0)
        sysfatal("no memory");
    for (i = 0; i <= high; i++){
        if (!BITSET(f,i)) {
            if ((to = canon(&pto)) < 0)
                to = lastc;
            else lastc = to;
            p[i] = to;
        }
```

```
        else p[i] = i;
    }
    if (sflag){
        lastc = 0x10000;
        while (readrune(0, &from) > 0) {
            if (from > high)
                from = to;
            else
                from = p[from];
            if (from != lastc || !BITSET(t,from)) {
                lastc = from;
                writerune(1, (Rune) from);
            }
        }

    } else {
        while (readrune(0, &from) > 0){
            if (from > high)
                from = to;
            else
                from = p[from];
            writerune(1, (Rune) from);
        }
    }
    wflush(1);
}

void
translit(void)
{
    Rune *p;
    int i;
    long from, to, lastc, high;

    lastc = 0;
    high = 0;
    while ((from = canon(&pfrom)) >= 0)
        if (from > high) high = from;
    Prewind(&pfrom);
    if ((p = (Rune *) malloc((high+1)*sizeof(Rune))) == 0)
        sysfatal("no memory");
    for (i = 0; i <= high; i++)
        p[i] = i;
    while ((from = canon(&pfrom)) >= 0) {
        if ((to = canon(&pto)) < 0)
            to = lastc;
        else lastc = to;
        if (BITSET(f,from) && p[from] != to)
            sysfatal("ambiguous translation");
        SETBIT(f,from);
        p[from] = to;
        SETBIT(t,to);
    }
    while ((to = canon(&pto)) >= 0) {
        SETBIT(t,to);
    }
    if (sflag){
        lastc = 0x10000;
        while (readrune(0, &from) > 0) {
            if (from <= high)
```

```
                from = p[from];
            if (from != lastc || !BITSET(t,from)) {
                lastc = from;
                writerune(1, (Rune) from);
            }
        }

    } else {
        while (readrune(0, &from) > 0) {
            if (from <= high)
                from = p[from];
            writerune(1, (Rune) from);
        }
    }
    wflush(1);
}

int
readrune(int fd, long *rp)
{
    Rune r;
    int j;
    static int i, n;
    static char buf[4096];

    j = i;
    for (;;) {
        if (i >= n) {
            wflush(1);
            if (j != i)
                memcpy(buf, buf+j, n-j);
            i = n-j;
            n = read(fd, &buf[i], sizeof(buf)-i);
            if (n < 0)
                sysfatal("read error: %r");
            if (n == 0)
                return 0;
            j = 0;
            n += i;
        }
        i++;
        if (fullrune(&buf[j], i-j))
            break;
    }
    chartorune(&r, &buf[j]);
    *rp = r;
    return 1;
}

void
writerune(int fd, Rune r)
{
    char buf[UTFmax];
    int n;

    if (!wptr)
        wptr = wbuf;
    n = runetochar(buf, (Rune*)&r);
    if (wptr+n >= wbuf+sizeof(wbuf))
        wflush(fd);
```

```
    memcpy(wptr, buf, n);
    wptr += n;
}

void
wflush(int fd)
{
    if (wptr && wptr > wbuf)
        if (write(fd, wbuf, wptr-wbuf) != wptr-wbuf)
            sysfatal("write error: %r");
    wptr = wbuf;
}

char *
getrune(char *s, Rune *rp)
{
    Rune r;
    char *save;
    int i, n;

    s += chartorune(rp, s);
    if((r = *rp) == '\\' && *s){
        n = 0;
        if (*s == 'x') {
            s++;
            for (i = 0; i < 4; i++) {
                save = s;
                s += chartorune(&r, s);
                if ('0' <= r && r <= '9')
                    n = 16*n + r - '0';
                else if ('a' <= r && r <= 'f')
                    n = 16*n + r - 'a' + 10;
                else if ('A' <= r && r <= 'F')
                    n = 16*n + r - 'A' + 10;
                else {
                    if (i == 0)
                        *rp = 'x';
                    else *rp = n;
                    return save;
                }
            }
        } else {
            for(i = 0; i < 3; i++) {
                save = s;
                s += chartorune(&r, s);
                if('0' <= r && r <= '7')
                    n = 8*n + r - '0';
                else {
                    if (i == 0)
                    {
                        *rp = r;
                        return s;
                    }
                    *rp = n;
                    return save;
                }
            }
            if(n > 0377)
                sysfatal("character > 0377");
        }
```

```
            *rp = n;
        }
        return s;
    }

    long
    canon(Pcb *p)
    {
        Rune r;

        if (p->final >= 0) {
            if (p->last < p->final)
                return ++p->last;
            p->final = -1;
        }
        if (*p->current == '\0')
            return -1;
        if(*p->current == '-' && p->last >= 0 && p->current[1]){
            p->current = getrune(p->current+1, &r);
            if (r < p->last)
                sysfatal("invalid range specification");
            if (r > p->last) {
                p->final = r;
                return ++p->last;
            }
        }
        p->current = getrune(p->current, &r);
        p->last = r;
        return p->last;
    }

    void
    Pinit(Pcb *p, char *cp)
    {
        p->current = p->base = cp;
        p->last = p->final = -1;
    }
    void
    Prewind(Pcb *p)
    {
        p->current = p->base;
        p->last = p->final = -1;
    }
```

A.8 text/grep/

A.8.1 grep/main.c

⟨grep/main.c 332⟩≡

```
#include    "grep.h"

⟨constant validflags(grep) 43b⟩
⟨function usage(grep) 43a⟩

⟨function main(grep) 46⟩

⟨function search(grep) 56⟩
⟨function initstate(grep) 54c⟩
```

A.8.2 grep/grep.h

⟨grep/grep.h 333⟩≡

```
⟨plan9 includes 14⟩
#include    <bio.h>

typedef struct  Re  Re;
typedef struct  Re2 Re2;
typedef struct  State   State;

⟨struct State(grep) 44c⟩
⟨struct Re2(grep) 45a⟩
⟨struct Re(grep) 44a⟩

⟨enum Re_type 44b⟩

enum
{
    Caselim     = 7,
    Nhunk       = 1<<16,
    Cbegin      = Runemax+1,
    Flshcnt     = (1<<9)-1,
};
⟨enum grep_flags 43c⟩

⟨union U(grep) 45c⟩
extern union U u;

// defined in globals.c
extern  char    *filename;
extern  char    *pattern;
extern  Biobuf  bout;
extern  char    flags[256];
extern  Re**    follow;
extern  ushort  gen;
extern  char*   input;
extern  long    lineno;
extern  int literal;
extern  int matched;
extern  long    maxfollow;
extern  long    nfollow;
extern  int peekc;
extern  Biobuf* rein;
extern  State*  state0;
extern  Re2 topre;

// forward decls
extern  Re* addcase(Re*);
extern  void    appendnext(Re*, Re*);
extern  void    error(char*);
extern  int fcmp(void*, void*);     /* (Re**, Re**) */
extern  void    fol1(Re*, int);
extern  int getrec(void);
extern  void    increment(State*, int);
extern  State*  initstate_(Re*);
extern  void*   mal(int);
extern  void    patchnext(Re*, Re*);
extern  Re* ral(int);
extern  Re2 re2cat(Re2, Re2);
extern  Re2 re2class(char*);
```

```
extern  Re2 re2or(Re2, Re2);
extern  Re2 re2char(int, int);
extern  Re2 re2star(Re2);
extern  State*  sal(int);
extern  int search(char*, int);
extern  void    str2top(char*);
extern  int yyparse(void);
extern  void    reprint(char*, Re*);
extern  void    yyerror(char*, ...);
```

A.8.3 grep/grep.y

A.8.4 grep/globals.c

⟨*globals grep* 334a⟩+≡ (334b) ◁49e

```
Re**    follow;
int matched;
long    maxfollow;
long    nfollow;
```

⟨grep/globals.c 334b⟩≡

```
#include "grep.h"

⟨globals grep 45b⟩
```

A.8.5 grep/sub.c

⟨grep/sub.c 334c⟩≡

```
#include    "grep.h"

⟨function mal(grep) 48a⟩
⟨function sal(grep) 48b⟩
⟨function ral(grep) 48c⟩

⟨function error(grep) 47c⟩

// addcase() helpers
⟨function countor(grep) 59a⟩
⟨function oralloc(grep) 59b⟩
⟨function case1(grep) 60a⟩

⟨function addcase(grep) 54d⟩

⟨function str2top(grep) 47b⟩

⟨function appendnext(grep) 54a⟩
⟨function patchnext(grep) 53b⟩

⟨function getrec(grep) 51a⟩

⟨function re2cat(grep) 53a⟩
⟨function re2star(grep) 53c⟩
⟨function re2or(grep) 53d⟩
⟨function re2char(grep) 54b⟩

⟨function reprint1(grep) 60c⟩
⟨function reprint(grep) 60b⟩
```

A.8.6 grep/comp.c

⟨grep/comp.c 335⟩≡

```
#include    "grep.h"

/*
 * incremental compiler.
 * add the branch c to the
 * state s.
 */
void
increment(State *s, int c)
{
    int i;
    State *t, **tt;
    Re *re1, *re2;

    nfollow = 0;
    gen++;
    matched = 0;
    for(i=0; i<s->count; i++)
        fol1(s->re[i], c);
    qsort(follow, nfollow, sizeof(*follow), fcmp);
    for(tt=&state0; t = *tt;) {
        if(t->count > nfollow) {
            tt = &t->linkleft;
            goto cont;
        }
        if(t->count < nfollow) {
            tt = &t->linkright;
            goto cont;
        }
        for(i=0; i<nfollow; i++) {
            re1 = t->re[i];
            re2 = follow[i];
            if(re1 > re2) {
                tt = &t->linkleft;
                goto cont;
            }
            if(re1 < re2) {
                tt = &t->linkright;
                goto cont;
            }
        }
        if(!!matched && !t->match) {
            tt = &t->linkleft;
            goto cont;
        }
        if(!matched && !!t->match) {
            tt = &t->linkright;
            goto cont;
        }
        s->next[c] = t;
        return;
    cont:;
    }

    t = sal(nfollow);
    *tt = t;
    for(i=0; i<nfollow; i++) {
```

```
        re1 = follow[i];
        t->re[i] = re1;
    }
    s->next[c] = t;
    t->match = matched;
}

int
fcmp(void *va, void *vb)
{
    Re **aa, **bb;
    Re *a, *b;

    aa = va;
    bb = vb;
    a = *aa;
    b = *bb;
    if(a > b)
        return 1;
    if(a < b)
        return -1;
    return 0;
}

void
fol1(Re *r, int c)
{
    Re *r1;

loop:
    if(r->gen == gen)
        return;
    if(nfollow >= maxfollow)
        error("nfollow");
    r->gen = gen;
    switch(r->type) {
    default:
        error("fol1");

    case Tcase:
        if(c >= 0 && c < 256)
        if(r1 = r->cases[c])
            follow[nfollow++] = r1;
        if(r = r->next)
            goto loop;
        break;

    case Talt:
    case Tor:
        fol1(r->alt, c);
        r = r->next;
        goto loop;

    case Tbegin:
        if(c == '\n' || c == Cbegin)
            follow[nfollow++] = r->next;
        break;

    case Tend:
        if(c == '\n') {
```

```
            if(r->next == 0) {
                matched = 1;
                break;
            }
            r = r->next;
            goto loop;
        }
        break;

    case Tclass:
        if(c >= r->lo && c <= r->hi)
            follow[nfollow++] = r->next;
        break;
    }
}

Rune    tab1[] =
{
    0x007f,
    0x07ff,
};
Rune    tab2[] =
{
    0x003f,
    0x0fff,
};

Re2
rclass(Rune p0, Rune p1)
{
    char xc0[6], xc1[6];
    int i, n, m;
    Re2 x;

    if(p0 > p1)
        return re2char(0xff, 0xff); // no match

    /*
     * bust range into same length
     * character sequences
     */
    for(i=0; i<nelem(tab1); i++) {
        m = tab1[i];
        if(p0 <= m && p1 > m)
            return re2or(rclass(p0, m), rclass(m+1, p1));
    }

    /*
     * bust range into part of a single page
     * or into full pages
     */
    for(i=0; i<nelem(tab2); i++) {
        m = tab2[i];
        if((p0 & ~m) != (p1 & ~m)) {
            if((p0 & m) != 0)
                return re2or(rclass(p0, p0|m), rclass((p0|m)+1, p1));
            if((p1 & m) != m)
                return re2or(rclass(p0, (p1&~m)-1), rclass(p1&~m, p1));
        }
    }
```

```
    n = runetochar(xc0, &p0);
    i = runetochar(xc1, &p1);
    if(i != n)
        error("length");

    x = re2char(xc0[0], xc1[0]);
    for(i=1; i<n; i++)
        x = re2cat(x, re2char(xc0[i], xc1[i]));
    return x;
}

int
pcmp(void *va, void *vb)
{
    int n;
    Rune *a, *b;

    a = va;
    b = vb;

    n = a[0] - b[0];
    if(n)
        return n;
    return a[1] - b[1];
}

/*
 * convert character chass into
 * run-pair ranges of matches.
 * this is 10646/utf specific and
 * needs to be changed for some
 * other input character set.
 * this is the key to a fast
 * regular search of characters
 * by looking at sequential bytes.
 */
Re2
re2class(char *s)
{
    Rune pairs[200+2], *p, *q, ov;
    int nc;
    Re2 x;

    nc = 0;
    if(*s == '^') {
        nc = 1;
        s++;
    }

    p = pairs;
    s += chartorune(p, s);
    for(;;) {
        if(*p == '\\')
            s += chartorune(p, s);
        if(*p == 0)
            break;
        p[1] = *p;
        p += 2;
        if(p >= pairs + nelem(pairs) - 2)
```

```
            error("class too big");
        s += chartorune(p, s);
        if(*p != '-')
            continue;
        s += chartorune(p, s);
        if(*p == '\\')
            s += chartorune(p, s);
        if(*p == 0)
            break;
        p[-1] = *p;
        s += chartorune(p, s);
    }
    *p = 0;
    qsort(pairs, (p-pairs)/2, 2*sizeof(*pairs), pcmp);

    q = pairs;
    for(p=pairs+2; *p; p+=2) {
        if(p[0] > p[1])
            continue;
        if(p[0] > q[1] || p[1] < q[0]) {
            q[2] = p[0];
            q[3] = p[1];
            q += 2;
            continue;
        }
        if(p[0] < q[0])
            q[0] = p[0];
        if(p[1] > q[1])
            q[1] = p[1];
    }
    q[2] = 0;

    p = pairs;
    if(nc) {
        x = rclass(0, p[0]-1);
        ov = p[1]+1;
        for(p+=2; *p; p+=2) {
            x = re2or(x, rclass(ov, p[0]-1));
            ov = p[1]+1;
        }
        x = re2or(x, rclass(ov, Runemask));
    } else {
        x = rclass(p[0], p[1]);
        for(p+=2; *p; p+=2)
            x = re2or(x, rclass(p[0], p[1]));
    }
    return x;
}
```

A.9 text/awk/

A.9.1 awk/awk.h

⟨awk/awk.h 339⟩≡

```
/*
Copyright (c) Lucent Technologies 1997
    All Rights Reserved
```

```
*/

typedef double  Awkfloat;

/* unsigned char is more trouble than it's worth */

typedef unsigned char uschar;

#define xfree(a)    { if ((a) != NULL) { free((char *) a); a = NULL; } }

#define DEBUG
#ifdef  DEBUG
            /* uses have to be doubly parenthesized */
#   define  dprintf(x)  if (dbg) printf x
#else
#   define  dprintf(x)
#endif

extern  char    errbuf[];

extern int  compile_time;   /* 1 if compiling, 0 if running */
extern int  safe;       /* 0 => unsafe, 1 => safe */

#define RECSIZE (8 * 1024)  /* sets limit on records, fields, etc., etc. */
extern int  recsize;    /* size of current record, orig RECSIZE */

extern char **FS;
extern char **RS;
extern char **ORS;
extern char **OFS;
extern char **OFMT;
extern Awkfloat *NR;
extern Awkfloat *FNR;
extern Awkfloat *NF;
extern char **FILENAME;
extern char **SUBSEP;
extern Awkfloat *RSTART;
extern Awkfloat *RLENGTH;

extern char *record;    /* points to $0 */
extern int  lineno;     /* line number in awk program */
extern int  errorflag;  /* 1 if error has occurred */
extern int  donefld;    /* 1 if record broken into fields */
extern int  donerec;    /* 1 if record is valid (no fld has changed */
extern char inputFS[];  /* FS at time of input, for field splitting */

extern int  dbg;

extern  char    *patbeg;    /* beginning of pattern matched */
extern  int patlen;     /* length of pattern matched.  set in b.c */

/* Cell:  all information about a variable or constant */

typedef struct Cell {
    uschar  ctype;      /* OCELL, OBOOL, OJUMP, etc. */
    uschar  csub;       /* CCON, CTEMP, CFLD, etc. */
    char    *nval;      /* name, for variables only */
    char    *sval;      /* string value */
    Awkfloat fval;      /* value as number */
    int  tval;      /* type info: STR|NUM|ARR|FCN|FLD|CON|DONTFREE */
```

```
    struct Cell *cnext; /* ptr to next if chained */
} Cell;

typedef struct Array {      /* symbol table array */
    int nelem;      /* elements in table right now */
    int size;       /* size of tab */
    Cell    **tab;      /* hash table pointers */
} Array;

#define NSYMTAB 50  /* initial size of a symbol table */
extern Array    *symtab;

extern Cell *nrloc;     /* NR */
extern Cell *fnrloc;    /* FNR */
extern Cell *nfloc;     /* NF */
extern Cell *rstartloc; /* RSTART */
extern Cell *rlengthloc;    /* RLENGTH */

/* Cell.tval values: */
#define NUM 01  /* number value is valid */
#define STR 02  /* string value is valid */
#define DONTFREE 04 /* string space is not freeable */
#define CON 010 /* this is a constant */
#define ARR 020 /* this is an array */
#define FCN 040 /* this is a function name */
#define FLD 0100    /* this is a field $1, $2, ... */
#define REC 0200    /* this is $0 */


/* function types */
#define FLENGTH 1
#define FSQRT   2
#define FEXP    3
#define FLOG    4
#define FINT    5
#define FSYSTEM 6
#define FRAND   7
#define FSRAND  8
#define FSIN    9
#define FCOS    10
#define FATAN   11
#define FTOUPPER 12
#define FTOLOWER 13
#define FFLUSH  14
#define FUTF    15

/* Node:  parse tree is made of nodes, with Cell's at bottom */

typedef struct Node {
    int ntype;
    struct  Node *nnext;
    int lineno;
    int nobj;
    struct  Node *narg[1];  /* variable: actual size set by calling malloc */
} Node;

#define NIL ((Node *) 0)

extern Node *winner;
extern Node *nullstat;
```

```
extern Node *nullnode;

/* ctypes */
#define OCELL   1
#define OBOOL   2
#define OJUMP   3

/* Cell subtypes: csub */
#define CFREE   7
#define CCOPY   6
#define CCON    5
#define CTEMP   4
#define CNAME   3
#define CVAR    2
#define CFLD    1
#define CUNK    0

/* bool subtypes */
#define BTRUE   11
#define BFALSE  12

/* jump subtypes */
#define JEXIT   21
#define JNEXT   22
#define JBREAK  23
#define JCONT   24
#define JRET    25
#define JNEXTFILE   26

/* node types */
#define NVALUE  1
#define NSTAT   2
#define NEXPR   3


extern  int pairstack[], paircnt;

#define notlegal(n) (n <= FIRSTTOKEN || n >= LASTTOKEN || proctab[n-FIRSTTOKEN] == nullproc)
#define isvalue(n)  ((n)->ntype == NVALUE)
#define isexpr(n)   ((n)->ntype == NEXPR)
#define isjump(n)   ((n)->ctype == OJUMP)
#define isexit(n)   ((n)->csub == JEXIT)
#define isbreak(n)  ((n)->csub == JBREAK)
#define iscont(n)   ((n)->csub == JCONT)
#define isnext(n)   ((n)->csub == JNEXT)
#define isnextfile(n)   ((n)->csub == JNEXTFILE)
#define isret(n)    ((n)->csub == JRET)
#define isrec(n)    ((n)->tval & REC)
#define isfld(n)    ((n)->tval & FLD)
#define isstr(n)    ((n)->tval & STR)
#define isnum(n)    ((n)->tval & NUM)
#define isarr(n)    ((n)->tval & ARR)
#define isfcn(n)    ((n)->tval & FCN)
#define istrue(n)   ((n)->csub == BTRUE)
#define istemp(n)   ((n)->csub == CTEMP)
#define isargument(n)   ((n)->nobj == ARG)
/* #define freeable(p)  (!((p)->tval & DONTFREE)) */
#define freeable(p) ( ((p)->tval & (STR|DONTFREE)) == STR )

#include "proto.h"
```

A.9.2 awk/awkgram.y

⟨*awk copyright lucent* 343a⟩≡ (415 408 376 371 368 364 362 350 344 343b)

```
/****************************************************************
Copyright (C) Lucent Technologies 1997
All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that the copyright notice and this
permission notice and warranty disclaimer appear in supporting
documentation, and that the name Lucent Technologies or any of
its entities not be used in advertising or publicity pertaining
to distribution of the software without specific, written prior
permission.

LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
THIS SOFTWARE.
****************************************************************/
```

⟨*awk/awkgram.y* 343b⟩≡

```
⟨awk copyright lucent 343a⟩
%{
⟨awkgram.y prelude 66a⟩
%}

⟨union directive awkgram.y 66b⟩
⟨token directives awkgram.y 66c⟩
⟨type directives awkgram.y 70a⟩
⟨priority directives awkgram.y 70b⟩

%%
⟨grammar awkgram.y 70c⟩
%%

void setfname(Cell *p)
{
    if (isarr(p))
        SYNTAX("%s is an array, not a function", p->nval);
    else if (isfcn(p))
        SYNTAX("you can't define function %s more than once", p->nval);
    curfname = p->nval;
}

int constnode(Node *p)
{
    return isvalue(p) && ((Cell *) (p->narg[0]))->csub == CCON;
}

char *strnode(Node *p)
{
    return ((Cell *)(p->narg[0]))->sval;
}
```

```
Node *notnull(Node *n)
{
    switch (n->nobj) {
    case LE: case LT: case EQ: case NE: case GT: case GE:
    case BOR: case AND: case NOT:
        return n;
    default:
        return op2(NE, n, nullnode);
    }
}

void checkdup(Node *vl, Cell *cp)   /* check if name already in list */
{
    char *s = cp->nval;
    for ( ; vl; vl = vl->nnext) {
        if (strcmp(s, ((Cell *)(vl->narg[0]))->nval) == 0) {
            SYNTAX("duplicate argument %s", s);
            break;
        }
    }
}
```

A.9.3 awk/lex.c

⟨awk/lex.c 344⟩≡

```
⟨awk copyright lucent 343a⟩
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "awk.h"
#include "y.tab.h"

extern YYSTYPE  yylval;
extern int  infunc;

int lineno  = 1;
int bracecnt = 0;
int brackcnt  = 0;
int parencnt = 0;

typedef struct Keyword {
    char    *word;
    int sub;
    int type;
} Keyword;

Keyword keywords[] ={   /* keep sorted: binary searched */
    { "BEGIN",  XBEGIN,     XBEGIN },
    { "END",    XEND,       XEND },
    { "NF",     VARNF,      VARNF },
    { "atan2",  FATAN,      BLTIN },
    { "break",  BREAK,      BREAK },
    { "close",  CLOSE,      CLOSE },
    { "continue",   CONTINUE,   CONTINUE },
    { "cos",    FCOS,       BLTIN },
    { "delete", DELETE,     DELETE },
    { "do",     DO,     DO },
```

```
    { "else",   ELSE,       ELSE },
    { "exit",   EXIT,       EXIT },
    { "exp",    FEXP,       BLTIN },
    { "fflush", FFLUSH,     BLTIN },
    { "for",    FOR,        FOR },
    { "func",   FUNC,       FUNC },
    { "function",   FUNC,       FUNC },
    { "getline",    GETLINE,    GETLINE },
    { "gsub",   GSUB,       GSUB },
    { "if",     IF,     IF },
    { "in",     IN,     IN },
    { "index",  INDEX,      INDEX },
    { "int",    FINT,       BLTIN },
    { "length", FLENGTH,    BLTIN },
    { "log",    FLOG,       BLTIN },
    { "match",  MATCHFCN,   MATCHFCN },
    { "next",   NEXT,       NEXT },
    { "nextfile",   NEXTFILE,   NEXTFILE },
    { "print",  PRINT,      PRINT },
    { "printf", PRINTF,     PRINTF },
    { "rand",   FRAND,      BLTIN },
    { "return", RETURN,     RETURN },
    { "sin",    FSIN,       BLTIN },
    { "split",  SPLIT,      SPLIT },
    { "sprintf",    SPRINTF,    SPRINTF },
    { "sqrt",   FSQRT,      BLTIN },
    { "srand",  FSRAND,     BLTIN },
    { "sub",    SUB,        SUB },
    { "substr", SUBSTR,     SUBSTR },
    { "system", FSYSTEM,    BLTIN },
    { "tolower",    FTOLOWER,   BLTIN },
    { "toupper",    FTOUPPER,   BLTIN },
    { "utf",    FUTF,       BLTIN },
    { "while",  WHILE,      WHILE },
};

#define DEBUG
#ifdef  DEBUG
#define RET(x)  { if(dbg)printf("lex %s\n", tokname(x)); return(x); }
#else
#define RET(x)  return(x)
#endif

int peek(void)
{
    int c = input();
    unput(c);
    return c;
}

int gettok(char **pbuf, int *psz)   /* get next input token */
{
    int c;
    char *buf = *pbuf;
    int sz = *psz;
    char *bp = buf;

    c = input();
    if (c == 0)
        return 0;
```

```
    buf[0] = c;
    buf[1] = 0;
    if (!isalnum(c) && c != '.' && c != '_')
        return c;

    *bp++ = c;
    if (isalpha(c) || c == '_') {   /* it's a varname */
        for ( ; (c = input()) != 0; ) {
            if (bp-buf >= sz)
                if (!adjbuf(&buf, &sz, bp-buf+2, 100, &bp, 0))
                    FATAL( "out of space for name %.10s...", buf );
            if (isalnum(c) || c == '_')
                *bp++ = c;
            else {
                *bp = 0;
                unput(c);
                break;
            }
        }
    } else {    /* it's a number */
        char *rem;
        /* read input until can't be a number */
        for ( ; (c = input()) != 0; ) {
            if (bp-buf >= sz)
                if (!adjbuf(&buf, &sz, bp-buf+2, 100, &bp, 0))
                    FATAL( "out of space for number %.10s...", buf );
            if (isdigit(c) || c == 'e' || c == 'E'
              || c == '.' || c == '+' || c == '-')
                *bp++ = c;
            else {
                unput(c);
                break;
            }
        }
        *bp = 0;
        strtod(buf, &rem);  /* parse the number */
        unputstr(rem);      /* put rest back for later */
        rem[0] = 0;
    }
    *pbuf = buf;
    *psz = sz;
    return buf[0];
}

int word(char *);
int string(void);
int regexpr(void);
int sc  = 0;    /* 1 => return a } right now */
int reg = 0;    /* 1 => return a REGEXPR now */
```

⟨*function* yylex*(awk)* 66d⟩

```
int string(void)
{
    int c, n;
    char *s, *bp;
    static char *buf = 0;
    static int bufsz = 500;

    if (buf == 0 && (buf = (char *) malloc(bufsz)) == NULL)
```

```
    FATAL("out of space for strings");
for (bp = buf; (c = input()) != '"'; ) {
    if (!adjbuf(&buf, &bufsz, bp-buf+2, 500, &bp, 0))
        FATAL("out of space for string %.10s...", buf);
    switch (c) {
    case '\n':
    case '\r':
    case 0:
        SYNTAX( "non-terminated string %.10s...", buf );
        lineno++;
        break;
    case '\\':
        c = input();
        switch (c) {
        case '"': *bp++ = '"'; break;
        case 'n': *bp++ = '\n'; break;
        case 't': *bp++ = '\t'; break;
        case 'f': *bp++ = '\f'; break;
        case 'r': *bp++ = '\r'; break;
        case 'b': *bp++ = '\b'; break;
        case 'v': *bp++ = '\v'; break;
        case 'a': *bp++ = '\007'; break;
        case '\\': *bp++ = '\\'; break;

        case '0': case '1': case '2': /* octal: \d \dd \ddd */
        case '3': case '4': case '5': case '6': case '7':
            n = c - '0';
            if ((c = peek()) >= '0' && c < '8') {
                n = 8 * n + input() - '0';
                if ((c = peek()) >= '0' && c < '8')
                    n = 8 * n + input() - '0';
            }
            *bp++ = n;
            break;

        case 'x':   /* hex  \x0-9a-fA-F + */
            {   char xbuf[100], *px;
            for (px = xbuf; (c = input()) != 0 && px-xbuf < 100-2; ) {
                if (isdigit(c)
                 || (c >= 'a' && c <= 'f')
                 || (c >= 'A' && c <= 'F'))
                    *px++ = c;
                else
                    break;
            }
            *px = 0;
            unput(c);
            sscanf(xbuf, "%x", &n);
            *bp++ = n;
            break;
            }

        default:
            *bp++ = c;
            break;
        }
        break;
    default:
        *bp++ = c;
        break;
```

```
        }
    }
    *bp = 0;
    s = tostring(buf);
    *bp++ = ' '; *bp++ = 0;
    yylval.cp = setsymtab(buf, s, 0.0, CON|STR|DONTFREE, symtab);
    RET(STRING);
}


int binsearch(char *w, Keyword *kp, int n)
{
    int cond, low, mid, high;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if ((cond = strcmp(w, kp[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}

int word(char *w)
{
    Keyword *kp;
    int c, n;

    n = binsearch(w, keywords, sizeof(keywords)/sizeof(keywords[0]));
    kp = keywords + n;
    if (n != -1) {  /* found in table */
        yylval.i = kp->sub;
        switch (kp->type) { /* special handling */
        case FSYSTEM:
            if (safe)
                SYNTAX( "system is unsafe" );
            RET(kp->type);
        case FUNC:
            if (infunc)
                SYNTAX( "illegal nested function" );
            RET(kp->type);
        case RETURN:
            if (!infunc)
                SYNTAX( "return not in function" );
            RET(kp->type);
        case VARNF:
            yylval.cp = setsymtab("NF", "", 0.0, NUM, symtab);
            RET(VARNF);
        default:
            RET(kp->type);
        }
    }
    c = peek(); /* look for '(' */
    if (c != '(' && infunc && (n=isarg(w)) >= 0) {
        yylval.i = n;
```

```
        RET(ARG);
    } else {
        yylval.cp = setsymtab(w, "", 0.0, STR|NUM|DONTFREE, symtab);
        if (c == '(') {
            RET(CALL);
        } else {
            RET(VAR);
        }
    }
}

void startreg(void) /* next call to yyles will return a regular expression */
{
    reg = 1;
}

int regexpr(void)
{
    int c;
    static char *buf = 0;
    static int bufsz = 500;
    char *bp;

    if (buf == 0 && (buf = (char *) malloc(bufsz)) == NULL)
        FATAL("out of space for rex expr");
    bp = buf;
    for ( ; (c = input()) != '/' && c != 0; ) {
        if (!adjbuf(&buf, &bufsz, bp-buf+3, 500, &bp, 0))
            FATAL("out of space for reg expr %.10s...", buf);
        if (c == '\n') {
            SYNTAX( "newline in regular expression %.10s...", buf );
            unput('\n');
            break;
        } else if (c == '\\') {
            *bp++ = '\\';
            *bp++ = input();
        } else {
            *bp++ = c;
        }
    }
    *bp = 0;
    yylval.s = tostring(buf);
    unput('/');
    RET(REGEXPR);
}

/* low-level lexical stuff, sort of inherited from lex */

char    ebuf[300];
char    *ep = ebuf;
char    yysbuf[100];    /* pushback buffer */
char    *yysptr = yysbuf;
FILE    *yyin = 0;

int input(void) /* get next lexical input character */
{
    int c;
    extern char *lexprog;

    if (yysptr > yysbuf)
```

```
        c = *--yysptr;
    else if (lexprog != NULL) { /* awk '...' */
        if ((c = *lexprog) != 0)
            lexprog++;
    } else              /* awk -f ... */
        c = pgetc();
    if (c == '\n')
        lineno++;
    else if (c == EOF)
        c = 0;
    if (ep >= ebuf + sizeof ebuf)
        ep = ebuf;
    return *ep++ = c;
}

void unput(int c)   /* put lexical character back on input */
{
    if (c == '\n')
        lineno--;
    if (yysptr >= yysbuf + sizeof(yysbuf))
        FATAL("pushed back too much: %.20s...", yysbuf);
    *yysptr++ = c;
    if (--ep < ebuf)
        ep = ebuf + sizeof(ebuf) - 1;
}

void unputstr(char *s)  /* put a string back on input */
{
    int i;

    for (i = strlen(s)-1; i >= 0; i--)
        unput(s[i]);
}
```

A.9.4 awk/lib.c

⟨awk/lib.c 350⟩≡

```
⟨awk copyright lucent 343a⟩
#define DEBUG
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <stdlib.h>
#include <stdarg.h>
#include "awk.h"
#include "y.tab.h"

FILE    *infile = NULL;
char    *file   = "";
char    *record;
int recsize = RECSIZE;
char    *fields;
int fieldssize = RECSIZE;

Cell    **fldtab;   /* pointers to Cells */
char    inputFS[100] = " ";

#define MAXFLD  200
```

```
int nfields = MAXFLD;   /* last allocated slot for $i */

int donefld;    /* 1 = implies rec broken into fields */
int donerec;    /* 1 = record is valid (no flds have changed) */

int lastfld = 0;    /* last used field */
int argno   = 1;    /* current input argument number */
extern  Awkfloat *ARGC;

static Cell dollar0 = { OCELL, CFLD, NULL, "", 0.0, REC|STR|DONTFREE };
static Cell dollar1 = { OCELL, CFLD, NULL, "", 0.0, FLD|STR|DONTFREE };

void recinit(unsigned int n)
{
    record = (char *) malloc(n);
    fields = (char *) malloc(n);
    fldtab = (Cell **) malloc((nfields+1) * sizeof(Cell *));
    if (record == NULL || fields == NULL || fldtab == NULL)
        FATAL("out of space for $0 and fields");
    fldtab[0] = (Cell *) malloc(sizeof (Cell));
    *fldtab[0] = dollar0;
    fldtab[0]->sval = record;
    fldtab[0]->nval = tostring("0");
    makefields(1, nfields);
}

void makefields(int n1, int n2)     /* create $n1..$n2 inclusive */
{
    char temp[50];
    int i;

    for (i = n1; i <= n2; i++) {
        fldtab[i] = (Cell *) malloc(sizeof (struct Cell));
        if (fldtab[i] == NULL)
            FATAL("out of space in makefields %d", i);
        *fldtab[i] = dollar1;
        sprintf(temp, "%d", i);
        fldtab[i]->nval = tostring(temp);
    }
}

void initgetrec(void)
{
    int i;
    char *p;

    for (i = 1; i < *ARGC; i++) {
        if (!isclvar(p = getargv(i))) { /* find 1st real filename */
            setsval(lookup("FILENAME", symtab), getargv(i));
            return;
        }
        setclvar(p);    /* a commandline assignment before filename */
        argno++;
    }
    infile = stdin;     /* no filenames, so use stdin */
}

int getrec(char **pbuf, int *pbufsize, int isrecord)    /* get next input record */
{           /* note: cares whether buf == record */
    int c;
```

```
static int firsttime = 1;
char *buf = *pbuf;
int bufsize = *pbufsize;

if (firsttime) {
    firsttime = 0;
    initgetrec();
}
   dprintf( ("RS=<%s>, FS=<%s>, ARGC=%g, FILENAME=%s\n",
    *RS, *FS, *ARGC, *FILENAME) );
if (isrecord) {
    donefld = 0;
    donerec = 1;
}
buf[0] = 0;
while (argno < *ARGC || infile == stdin) {
       dprintf( ("argno=%d, file=|%s|\n", argno, file) );
    if (infile == NULL) {   /* have to open a new file */
        file = getargv(argno);
        if (*file == '\0') {    /* it's been zapped */
            argno++;
            continue;
        }
        if (isclvar(file)) {    /* a var=value arg */
            setclvar(file);
            argno++;
            continue;
        }
        *FILENAME = file;
           dprintf( ("opening file %s\n", file) );
        if (*file == '-' && *(file+1) == '\0')
            infile = stdin;
        else if ((infile = fopen(file, "r")) == NULL)
            FATAL("can't open file %s", file);
        setfval(fnrloc, 0.0);
    }
    c = readrec(&buf, &bufsize, infile);
    if (c != 0 || buf[0] != '\0') { /* normal record */
        if (isrecord) {
            if (freeable(fldtab[0]))
                xfree(fldtab[0]->sval);
            fldtab[0]->sval = buf;  /* buf == record */
            fldtab[0]->tval = REC | STR | DONTFREE;
            if (is_number(fldtab[0]->sval)) {
                fldtab[0]->fval = atof(fldtab[0]->sval);
                fldtab[0]->tval |= NUM;
            }
        }
        setfval(nrloc, nrloc->fval+1);
        setfval(fnrloc, fnrloc->fval+1);
        *pbuf = buf;
        *pbufsize = bufsize;
        return 1;
    }
    /* EOF arrived on this file; set up next */
    if (infile != stdin)
        fclose(infile);
    infile = NULL;
    argno++;
}
```

```
    *pbuf = buf;
    *pbufsize = bufsize;
    return 0;   /* true end of file */
}

void nextfile(void)
{
    if (infile != stdin)
        fclose(infile);
    infile = NULL;
    argno++;
}

int readrec(char **pbuf, int *pbufsize, FILE *inf)  /* read one record into buf */
{
    int sep, c;
    char *rr, *buf = *pbuf;
    int bufsize = *pbufsize;

    if (strlen(*FS) >= sizeof(inputFS))
        FATAL("field separator %.10s... is too long", *FS);
    strcpy(inputFS, *FS);   /* for subsequent field splitting */
    if ((sep = **RS) == 0) {
        sep = '\n';
        while ((c=getc(inf)) == '\n' && c != EOF)   /* skip leading \n's */
            ;
        if (c != EOF)
            ungetc(c, inf);
    }
    for (rr = buf; ; ) {
        for (; (c=getc(inf)) != sep && c != EOF; ) {
            if (rr-buf+1 > bufsize)
                if (!adjbuf(&buf, &bufsize, 1+rr-buf, recsize, &rr, "readrec 1"))
                    FATAL("input record `%.30s...' too long", buf);
            *rr++ = c;
        }
        if (**RS == sep || c == EOF)
            break;
        if ((c = getc(inf)) == '\n' || c == EOF) /* 2 in a row */
            break;
        if (!adjbuf(&buf, &bufsize, 2+rr-buf, recsize, &rr, "readrec 2"))
            FATAL("input record `%.30s...' too long", buf);
        *rr++ = '\n';
        *rr++ = c;
    }
    if (!adjbuf(&buf, &bufsize, 1+rr-buf, recsize, &rr, "readrec 3"))
        FATAL("input record `%.30s...' too long", buf);
    *rr = 0;
       dprintf( ("readrec saw <%s>, returns %d\n", buf, c == EOF && rr == buf ? 0 : 1) );
    *pbuf = buf;
    *pbufsize = bufsize;
    return c == EOF && rr == buf ? 0 : 1;
}

char *getargv(int n)    /* get ARGV[n] */
{
    Cell *x;
    char *s, temp[50];
    extern Array *ARGVtab;
```

```
    sprintf(temp, "%d", n);
    x = setsymtab(temp, "", 0.0, STR, ARGVtab);
    s = getsval(x);
       dprintf( ("getargv(%d) returns |%s|\n", n, s) );
    return s;
}

void setclvar(char *s)  /* set var=value from s */
{
    char *p;
    Cell *q;

    for (p=s; *p != '='; p++)
        ;
    *p++ = 0;
    p = qstring(p, '\0');
    q = setsymtab(s, p, 0.0, STR, symtab);
    setsval(q, p);
    if (is_number(q->sval)) {
        q->fval = atof(q->sval);
        q->tval |= NUM;
    }
       dprintf( ("command line set %s to |%s|\n", s, p) );
}


void fldbld(void)   /* create fields from current record */
{
    /* this relies on having fields[] the same length as $0 */
    /* the fields are all stored in this one array with \0's */
    char *r, *fr, sep;
    Cell *p;
    int i, j, n;

    if (donefld)
        return;
    if (!isstr(fldtab[0]))
        getsval(fldtab[0]);
    r = fldtab[0]->sval;
    n = strlen(r);
    if (n > fieldssize) {
        xfree(fields);
        if ((fields = (char *) malloc(n+1)) == NULL)
            FATAL("out of space for fields in fldbld %d", n);
        fieldssize = n;
    }
    fr = fields;
    i = 0;  /* number of fields accumulated here */
    if (strlen(inputFS) > 1) {  /* it's a regular expression */
        i = refldbld(r, inputFS);
    } else if ((sep = *inputFS) == ' ') {   /* default whitespace */
        for (i = 0; ; ) {
            while (*r == ' ' || *r == '\t' || *r == '\n')
                r++;
            if (*r == 0)
                break;
            i++;
            if (i > nfields)
                growfldtab(i);
            if (freeable(fldtab[i]))
```

```
            xfree(fldtab[i]->sval);
        fldtab[i]->sval = fr;
        fldtab[i]->tval = FLD | STR | DONTFREE;
        do
            *fr++ = *r++;
        while (*r != ' ' && *r != '\t' && *r != '\n' && *r != '\0');
        *fr++ = 0;
    }
    *fr = 0;
} else if ((sep = *inputFS) == 0) {     /* new: FS="" => 1 char/field */
    for (i = 0; *r != 0; r++) {
        char buf[2];
        i++;
        if (i > nfields)
            growfldtab(i);
        if (freeable(fldtab[i]))
            xfree(fldtab[i]->sval);
        buf[0] = *r;
        buf[1] = 0;
        fldtab[i]->sval = tostring(buf);
        fldtab[i]->tval = FLD | STR;
    }
    *fr = 0;
} else if (*r != 0) {   /* if 0, it's a null field */
    for (;;) {
        i++;
        if (i > nfields)
            growfldtab(i);
        if (freeable(fldtab[i]))
            xfree(fldtab[i]->sval);
        fldtab[i]->sval = fr;
        fldtab[i]->tval = FLD | STR | DONTFREE;
        while (*r != sep && *r != '\n' && *r != '\0')   /* \n is always a separator */
            *fr++ = *r++;
        *fr++ = 0;
        if (*r++ == 0)
            break;
    }
    *fr = 0;
}
if (i > nfields)
    FATAL("record `%.30s...' has too many fields; can't happen", r);
cleanfld(i+1, lastfld); /* clean out junk from previous record */
lastfld = i;
donefld = 1;
for (j = 1; j <= lastfld; j++) {
    p = fldtab[j];
    if(is_number(p->sval)) {
        p->fval = atof(p->sval);
        p->tval |= NUM;
    }
}
setfval(nfloc, (Awkfloat) lastfld);
if (dbg) {
    for (j = 0; j <= lastfld; j++) {
        p = fldtab[j];
        printf("field %d (%s): |%s|\n", j, p->nval, p->sval);
    }
}
}
```

```
void cleanfld(int n1, int n2)   /* clean out fields n1 .. n2 inclusive */
{               /* nvals remain intact */
    Cell *p;
    int i;

    for (i = n1; i <= n2; i++) {
        p = fldtab[i];
        if (freeable(p))
            xfree(p->sval);
        p->sval = "";
        p->tval = FLD | STR | DONTFREE;
    }
}

void newfld(int n)  /* add field n after end of existing lastfld */
{
    if (n > nfields)
        growfldtab(n);
    cleanfld(lastfld+1, n);
    lastfld = n;
    setfval(nfloc, (Awkfloat) n);
}

Cell *fieldadr(int n)   /* get nth field */
{
    if (n < 0)
        FATAL("trying to access field %d", n);
    if (n > nfields)    /* fields after NF are empty */
        growfldtab(n);  /* but does not increase NF */
    return(fldtab[n]);
}

void growfldtab(int n)  /* make new fields up to at least $n */
{
    int nf = 2 * nfields;

    if (n > nf)
        nf = n;
    fldtab = (Cell **) realloc(fldtab, (nf+1) * (sizeof (struct Cell *)));
    if (fldtab == NULL)
        FATAL("out of space creating %d fields", nf);
    makefields(nfields+1, nf);
    nfields = nf;
}

int refldbld(char *rec, char *fs)   /* build fields from reg expr in FS */
{
    /* this relies on having fields[] the same length as $0 */
    /* the fields are all stored in this one array with \0's */
    char *fr;
    void *p;
    int i, tempstat, n;

    n = strlen(rec);
    if (n > fieldssize) {
        xfree(fields);
        if ((fields = (char *) malloc(n+1)) == NULL)
            FATAL("out of space for fields in refldbld %d", n);
        fieldssize = n;
```

```
    }
    fr = fields;
    *fr = '\0';
    if (*rec == '\0')
        return 0;
    p = compre(fs);
       dprintf( ("into refldbld, rec = <%s>, pat = <%s>\n", rec, fs) );
    for (i = 1; ; i++) {
        if (i > nfields)
            growfldtab(i);
        if (freeable(fldtab[i]))
            xfree(fldtab[i]->sval);
        fldtab[i]->tval = FLD | STR | DONTFREE;
        fldtab[i]->sval = fr;
           dprintf( ("refldbld: i=%d\n", i) );
        if (nematch(p, rec, rec)) {
               dprintf( ("match %s (%d chars)\n", patbeg, patlen) );
            strncpy(fr, rec, patbeg-rec);
            fr += patbeg - rec + 1;
            *(fr-1) = '\0';
            rec = patbeg + patlen;
        } else {
               dprintf( ("no match %s\n", rec) );
            strcpy(fr, rec);
            break;
        }
    }
    return i;
}

void recbld(void)   /* create $0 from $1..$NF if necessary */
{
    int i;
    char *r, *p;

    if (donerec == 1)
        return;
    r = record;
    for (i = 1; i <= *NF; i++) {
        p = getsval(fldtab[i]);
        if (!adjbuf(&record, &recsize, 1+strlen(p)+r-record, recsize, &r, "recbld 1"))
            FATAL("created $0 `%.30s...' too long", record);
        while ((*r = *p++) != 0)
            r++;
        if (i < *NF) {
            if (!adjbuf(&record, &recsize, 2+strlen(*OFS)+r-record, recsize, &r, "recbld 2"))
                FATAL("created $0 `%.30s...' too long", record);
            for (p = *OFS; (*r = *p++) != 0; )
                r++;
        }
    }
    if (!adjbuf(&record, &recsize, 2+r-record, recsize, &r, "recbld 3"))
        FATAL("built giant record `%.30s...'", record);
    *r = '\0';
       dprintf( ("in recbld inputFS=%s, fldtab[0]=%p\n", inputFS, fldtab[0]) );

    if (freeable(fldtab[0]))
        xfree(fldtab[0]->sval);
    fldtab[0]->tval = REC | STR | DONTFREE;
    fldtab[0]->sval = record;
```

```
       dprintf( ("in recbld inputFS=%s, fldtab[0]=%p\n", inputFS, fldtab[0]) );
       dprintf( ("recbld = |%s|\n", record) );
    donerec = 1;
}

int errorflag   = 0;

void yyerror(char *s)
{
    SYNTAX(s);
}

void SYNTAX(char *fmt, ...)
{
    extern char *cmdname, *curfname;
    static int been_here = 0;
    va_list varg;

    if (been_here++ > 2)
        return;
    fprintf(stderr, "%s: ", cmdname);
    va_start(varg, fmt);
    vfprintf(stderr, fmt, varg);
    va_end(varg);
    if(compile_time == 1 && cursource() != NULL)
        fprintf(stderr, " at %s:%d", cursource(), lineno);
    else
        fprintf(stderr, " at line %d", lineno);
    if (curfname != NULL)
        fprintf(stderr, " in function %s", curfname);
    fprintf(stderr, "\n");
    errorflag = 2;
    eprint();
}

void fpecatch(int n)
{
    FATAL("floating point exception %d", n);
}

extern int bracecnt, brackcnt, parencnt;

void bracecheck(void)
{
    int c;
    static int beenhere = 0;

    if (beenhere++)
        return;
    while ((c = input()) != EOF && c != '\0')
        bclass(c);
    bcheck2(bracecnt, '{', '}');
    bcheck2(brackcnt, '[', ']');
    bcheck2(parencnt, '(', ')');
}

void bcheck2(int n, int c1, int c2)
{
    if (n == 1)
```

```
        fprintf(stderr, "\tmissing %c\n", c2);
    else if (n > 1)
        fprintf(stderr, "\t%d missing %c's\n", n, c2);
    else if (n == -1)
        fprintf(stderr, "\textra %c\n", c2);
    else if (n < -1)
        fprintf(stderr, "\t%d extra %c's\n", -n, c2);
}

void FATAL(char *fmt, ...)
{
    extern char *cmdname;
    va_list varg;

    fflush(stdout);
    fprintf(stderr, "%s: ", cmdname);
    va_start(varg, fmt);
    vfprintf(stderr, fmt, varg);
    va_end(varg);
    error();
    if (dbg > 1)        /* core dump if serious debugging on */
        abort();
    exit(2);
}

void WARNING(char *fmt, ...)
{
    extern char *cmdname;
    va_list varg;

    fflush(stdout);
    fprintf(stderr, "%s: ", cmdname);
    va_start(varg, fmt);
    vfprintf(stderr, fmt, varg);
    va_end(varg);
    error();
}

void error()
{
    extern Node *curnode;
    int line;

    fprintf(stderr, "\n");
    if (compile_time != 2 && NR && *NR > 0) {
        if (strcmp(*FILENAME, "-") != 0)
            fprintf(stderr, " input record %s:%d", *FILENAME, (int) (*FNR));
        else
            fprintf(stderr, " input record number %d", (int) (*FNR));
        fprintf(stderr, "\n");
    }
    if (compile_time != 2 && curnode)
        line = curnode->lineno;
    else if (compile_time != 2 && lineno)
        line = lineno;
    else
        line = -1;
    if (compile_time == 1 && cursource() != NULL){
        if(line >= 0)
            fprintf(stderr, " source %s:%d", cursource(), line);
```

```
        else
            fprintf(stderr, " source file %s", cursource());
    }else if(line >= 0)
        fprintf(stderr, " source line %d", line);
    fprintf(stderr, "\n");
    eprint();
}

void eprint(void)   /* try to print context around error */
{
    char *p, *q;
    int c;
    static int been_here = 0;
    extern char ebuf[], *ep;

    if (compile_time == 2 || compile_time == 0 || been_here++ > 0)
        return;
    p = ep - 1;
    if (p > ebuf && *p == '\n')
        p--;
    for ( ; p > ebuf && *p != '\n' && *p != '\0'; p--)
        ;
    while (*p == '\n')
        p++;
    fprintf(stderr, " context is\n\t");
    for (q=ep-1; q>=p && *q!=' ' && *q!='\t' && *q!='\n'; q--)
        ;
    for ( ; p < q; p++)
        if (*p)
            putc(*p, stderr);
    fprintf(stderr, " >>> ");
    for ( ; p < ep; p++)
        if (*p)
            putc(*p, stderr);
    fprintf(stderr, " <<< ");
    if (*ep)
        while ((c = input()) != '\n' && c != '\0' && c != EOF) {
            putc(c, stderr);
            bclass(c);
        }
    putc('\n', stderr);
    ep = ebuf;
}

void bclass(int c)
{
    switch (c) {
    case '{': bracecnt++; break;
    case '}': bracecnt--; break;
    case '[': brackcnt++; break;
    case ']': brackcnt--; break;
    case '(': parencnt++; break;
    case ')': parencnt--; break;
    }
}

double errcheck(double x, char *s)
{

    if (errno == EDOM) {
```

```
        errno = 0;
        WARNING("%s argument out of domain", s);
        x = 1;
    } else if (errno == ERANGE) {
        errno = 0;
        WARNING("%s result out of range", s);
        x = 1;
    }
    return x;
}

int isclvar(char *s)    /* is s of form var=something ? */
{
    char *os = s;

    if (!isalpha(*s) && *s != '_')
        return 0;
    for ( ; *s; s++)
        if (!(isalnum(*s) || *s == '_'))
            break;
    return *s == '=' && s > os && *(s+1) != '=';
}

/* strtod is supposed to be a proper test of what's a valid number */

#include <math.h>
int is_number(char *s)
{
    double r;
    char *ep;

    /*
     * fast could-it-be-a-number check before calling strtod,
     * which takes a surprisingly long time to reject non-numbers.
     */
    switch (*s) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    case '\t':
    case '\n':
    case '\v':
    case '\f':
    case '\r':
    case ' ':
    case '-':
    case '+':
    case '.':
    case 'n':       /* nans */
    case 'N':
    case 'i':       /* infs */
    case 'I':
        break;
    default:
        return 0;   /* can't be a number */
    }

    errno = 0;
    r = strtod(s, &ep);
    if (ep == s || r == HUGE_VAL || errno == ERANGE)
        return 0;
```

```
    while (*ep == ' ' || *ep == '\t' || *ep == '\n')
        ep++;
    if (*ep == '\0')
        return 1;
    else
        return 0;
}
```

A.9.5 awk/maketab.c

⟨awk/maketab.c 362⟩≡

```
⟨awk copyright lucent 343a⟩
/*
 * this program makes the table to link function names
 * and type indices that is used by execute() in run.c.
 * it finds the indices in y.tab.h, produced by yacc.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "awk.h"
#include "y.tab.h"

struct xx
{   int token;
    char *name;
    char *pname;
} proc[] = {
    { PROGRAM, "program", NULL },
    { BOR, "boolop", " || " },
    { AND, "boolop", " && " },
    { NOT, "boolop", " !" },
    { NE, "relop", " != " },
    { EQ, "relop", " == " },
    { LE, "relop", " <= " },
    { LT, "relop", " < " },
    { GE, "relop", " >= " },
    { GT, "relop", " > " },
    { ARRAY, "array", NULL },
    { INDIRECT, "indirect", "$(" },
    { SUBSTR, "substr", "substr" },
    { SUB, "sub", "sub" },
    { GSUB, "gsub", "gsub" },
    { INDEX, "sindex", "sindex" },
    { SPRINTF, "awksprintf", "sprintf " },
    { ADD, "arith", " + " },
    { MINUS, "arith", " - " },
    { MULT, "arith", " * " },
    { DIVIDE, "arith", " / " },
    { MOD, "arith", " % " },
    { UMINUS, "arith", " -" },
    { POWER, "arith", " **" },
    { PREINCR, "incrdecr", "++" },
    { POSTINCR, "incrdecr", "++" },
    { PREDECR, "incrdecr", "--" },
    { POSTDECR, "incrdecr", "--" },
    { CAT, "cat", " " },
    { PASTAT, "pastat", NULL },
```

```
    { PASTAT2, "dopa2", NULL },
    { MATCH, "matchop", " ~ " },
    { NOTMATCH, "matchop", " !~ " },
    { MATCHFCN, "matchop", "matchop" },
    { INTEST, "intest", "intest" },
    { PRINTF, "awkprintf", "printf" },
    { PRINT, "printstat", "print" },
    { CLOSE, "closefile", "closefile" },
    { DELETE, "awkdelete", "awkdelete" },
    { SPLIT, "split", "split" },
    { ASSIGN, "assign", " = " },
    { ADDEQ, "assign", " += " },
    { SUBEQ, "assign", " -= " },
    { MULTEQ, "assign", " *= " },
    { DIVEQ, "assign", " /= " },
    { MODEQ, "assign", " %= " },
    { POWEQ, "assign", " ^= " },
    { CONDEXPR, "condexpr", " ?: " },
    { IF, "ifstat", "if(" },
    { WHILE, "whilestat", "while(" },
    { FOR, "forstat", "for(" },
    { DO, "dostat", "do" },
    { IN, "instat", "instat" },
    { NEXT, "jump", "next" },
    { NEXTFILE, "jump", "nextfile" },
    { EXIT, "jump", "exit" },
    { BREAK, "jump", "break" },
    { CONTINUE, "jump", "continue" },
    { RETURN, "jump", "ret" },
    { BLTIN, "bltin", "bltin" },
    { CALL, "call", "call" },
    { ARG, "arg", "arg" },
    { VARNF, "getnf", "NF" },
    { GETLINE, "getline", "getline" },
    { 0, "", "" },
};

#define SIZE    (LASTTOKEN - FIRSTTOKEN + 1)
char *table[SIZE];
char *names[SIZE];

int main(int argc, char *argv[])
{
    struct xx *p;
    int i, n, tok;
    char c;
    FILE *fp;
    char buf[200], name[200], def[200];

    printf("#include <stdio.h>\n");
    printf("#include \"awk.h\"\n");
    printf("#include \"y.tab.h\"\n\n");
    for (i = SIZE; --i >= 0; )
        names[i] = "";

    if ((fp = fopen("y.tab.h", "r")) == NULL) {
        fprintf(stderr, "maketab can't open y.tab.h!\n");
        exit(1);
    }
    printf("static char *printname[%d] = {\n", SIZE);
```

```
    i = 0;
    while (fgets(buf, sizeof buf, fp) != NULL) {
        n = sscanf(buf, "%1c %s %s %d", &c, def, name, &tok);
        if (c != '#' || (n != 4 && strcmp(def,"define") != 0))  /* not a valid #define */
            continue;
        if (tok < FIRSTTOKEN || tok > LASTTOKEN) {
            fprintf(stderr, "maketab funny token %d %s\n", tok, buf);
            // switched to 'continue' because bison generates mant YYxx
            // macros that we just want to ignore
            //exit(1);
            continue;
        }
        names[tok-FIRSTTOKEN] = (char *) malloc(strlen(name)+1);
        strcpy(names[tok-FIRSTTOKEN], name);
        printf("\t(char *) \"%s\",\t/* %d */\n", name, tok);
        i++;
    }
    printf("};\n\n");

    for (p=proc; p->token!=0; p++)
        table[p->token-FIRSTTOKEN] = p->name;
    printf("\nCell *(*proctab[%d])(Node **, int) = {\n", SIZE);
    for (i=0; i<SIZE; i++)
        if (table[i]==0)
            printf("\tnullproc,\t/* %s */\n", names[i]);
        else
            printf("\t%s,\t/* %s */\n", table[i], names[i]);
    printf("};\n\n");

    printf("char *tokname(int n)\n");   /* print a tokname() function */
    printf("{\n");
    printf("    static char buf[100];\n\n");
    printf("    if (n < FIRSTTOKEN || n > LASTTOKEN) {\n");
    printf("        sprintf(buf, \"token %%d\", n);\n");
    printf("        return buf;\n");
    printf("    }\n");
    printf("    return printname[n-FIRSTTOKEN];\n");
    printf("}\n");
    return 0;
}
```

A.9.6 awk/parse.c

⟨awk/parse.c 364⟩≡

```
⟨awk copyright lucent 343a⟩
#define DEBUG
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "awk.h"
#include "y.tab.h"

Node *nodealloc(int n)
{
    Node *x;

    x = (Node *) malloc(sizeof(Node) + (n-1)*sizeof(Node *));
    if (x == NULL)
        FATAL("out of space in nodealloc");
```

```
    x->nnext = NULL;
    x->lineno = lineno;
    return(x);
}

Node *exptostat(Node *a)
{
    a->ntype = NSTAT;
    return(a);
}

Node *node1(int a, Node *b)
{
    Node *x;

    x = nodealloc(1);
    x->nobj = a;
    x->narg[0]=b;
    return(x);
}

Node *node2(int a, Node *b, Node *c)
{
    Node *x;

    x = nodealloc(2);
    x->nobj = a;
    x->narg[0] = b;
    x->narg[1] = c;
    return(x);
}

Node *node3(int a, Node *b, Node *c, Node *d)
{
    Node *x;

    x = nodealloc(3);
    x->nobj = a;
    x->narg[0] = b;
    x->narg[1] = c;
    x->narg[2] = d;
    return(x);
}

Node *node4(int a, Node *b, Node *c, Node *d, Node *e)
{
    Node *x;

    x = nodealloc(4);
    x->nobj = a;
    x->narg[0] = b;
    x->narg[1] = c;
    x->narg[2] = d;
    x->narg[3] = e;
    return(x);
}

Node *stat1(int a, Node *b)
{
    Node *x;
```

```
    x = node1(a,b);
    x->ntype = NSTAT;
    return(x);
}

Node *stat2(int a, Node *b, Node *c)
{
    Node *x;

    x = node2(a,b,c);
    x->ntype = NSTAT;
    return(x);
}

Node *stat3(int a, Node *b, Node *c, Node *d)
{
    Node *x;

    x = node3(a,b,c,d);
    x->ntype = NSTAT;
    return(x);
}

Node *stat4(int a, Node *b, Node *c, Node *d, Node *e)
{
    Node *x;

    x = node4(a,b,c,d,e);
    x->ntype = NSTAT;
    return(x);
}

Node *op1(int a, Node *b)
{
    Node *x;

    x = node1(a,b);
    x->ntype = NEXPR;
    return(x);
}

Node *op2(int a, Node *b, Node *c)
{
    Node *x;

    x = node2(a,b,c);
    x->ntype = NEXPR;
    return(x);
}

Node *op3(int a, Node *b, Node *c, Node *d)
{
    Node *x;

    x = node3(a,b,c,d);
    x->ntype = NEXPR;
    return(x);
}
```

```
Node *op4(int a, Node *b, Node *c, Node *d, Node *e)
{
    Node *x;

    x = node4(a,b,c,d,e);
    x->ntype = NEXPR;
    return(x);
}

Node *celltonode(Cell *a, int b)
{
    Node *x;

    a->ctype = OCELL;
    a->csub = b;
    x = node1(0, (Node *) a);
    x->ntype = NVALUE;
    return(x);
}

Node *rectonode(void)   /* make $0 into a Node */
{
    extern Cell *literal0;
    return op1(INDIRECT, celltonode(literal0, CUNK));
}

Node *makearr(Node *p)
{
    Cell *cp;

    if (isvalue(p)) {
        cp = (Cell *) (p->narg[0]);
        if (isfcn(cp))
            SYNTAX( "%s is a function, not an array", cp->nval );
        else if (!isarr(cp)) {
            xfree(cp->sval);
            cp->sval = (char *) makesymtab(NSYMTAB);
            cp->tval = ARR;
        }
    }
    return p;
}

#define PA2NUM  50  /* max number of pat,pat patterns allowed */
int paircnt;        /* number of them in use */
int pairstack[PA2NUM];  /* state of each pat,pat */

Node *pa2stat(Node *a, Node *b, Node *c)    /* pat, pat {...} */
{
    Node *x;

    x = node4(PASTAT2, a, b, c, itonp(paircnt));
    if (paircnt++ >= PA2NUM)
        SYNTAX( "limited to %d pat,pat statements", PA2NUM );
    x->ntype = NSTAT;
    return(x);
}

Node *linkum(Node *a, Node *b)
{
```

```
    Node *c;

    if (errorflag)  /* don't link things that are wrong */
        return a;
    if (a == NULL)
        return(b);
    else if (b == NULL)
        return(a);
    for (c = a; c->nnext != NULL; c = c->nnext)
        ;
    c->nnext = b;
    return(a);
}

void defn(Cell *v, Node *vl, Node *st)  /* turn on FCN bit in definition, */
{                   /*   body of function, arglist */
    Node *p;
    int n;

    if (isarr(v)) {
        SYNTAX( "`%s' is an array name and a function name", v->nval );
        return;
    }
    v->tval = FCN;
    v->sval = (char *) st;
    n = 0;  /* count arguments */
    for (p = vl; p; p = p->nnext)
        n++;
    v->fval = n;
    dprintf( ("defining func %s (%d args)\n", v->nval, n) );
}

int isarg(char *s)      /* is s in argument list for current function? */
{           /* return -1 if not, otherwise arg # */
    extern Node *arglist;
    Node *p = arglist;
    int n;

    for (n = 0; p != 0; p = p->nnext, n++)
        if (strcmp(((Cell *)(p->narg[0]))->nval, s) == 0)
            return n;
    return -1;
}

int ptoi(void *p)   /* convert pointer to integer */
{
    return (int) (long) p;  /* swearing that p fits, of course */
}

Node *itonp(int i)  /* and vice versa */
{
    return (Node *) (long) i;
}
```

A.9.7 awk/proto.h

⟨awk/proto.h 368⟩≡
  ⟨*awk copyright lucent* 343a⟩
  // for goken

```
#define getline p9getline

// forward decls
extern  int yywrap(void);
extern  void    setfname(Cell *);
extern  int     constnode(Node *);
extern  char    *strnode(Node *);
extern  Node    *notnull(Node *);
extern  int     yyparse(void);

extern  int     yylex(void);
extern  void    startreg(void);
extern  int     input(void);
extern  void    unput(int);
extern  void    unputstr(char *);
extern  int     yylook(void);
extern  int     yyback(int *, int);
extern  int     yyinput(void);

extern  void    *compre(char *);
extern  int     hexstr(char **);
extern  void    quoted(char **, char **, char *);
extern  int     match(void *, char *, char *);
extern  int     pmatch(void *, char *, char *);
extern  int     nematch(void *, char *, char *);
extern  int     countposn(char *, int);
extern  void    overflow(void);

extern  int     pgetc(void);
extern  char    *cursource(void);

extern  Node    *nodealloc(int);
extern  Node    *exptostat(Node *);
extern  Node    *node1(int, Node *);
extern  Node    *node2(int, Node *, Node *);
extern  Node    *node3(int, Node *, Node *, Node *);
extern  Node    *node4(int, Node *, Node *, Node *, Node *);
extern  Node    *stat3(int, Node *, Node *, Node *);
extern  Node    *op2(int, Node *, Node *);
extern  Node    *op1(int, Node *);
extern  Node    *stat1(int, Node *);
extern  Node    *op3(int, Node *, Node *, Node *);
extern  Node    *op4(int, Node *, Node *, Node *, Node *);
extern  Node    *stat2(int, Node *, Node *);
extern  Node    *stat4(int, Node *, Node *, Node *, Node *);
extern  Node    *celltonode(Cell *, int);
extern  Node    *rectonode(void);
extern  Node    *makearr(Node *);
extern  Node    *pa2stat(Node *, Node *, Node *);
extern  Node    *linkum(Node *, Node *);
extern  void    defn(Cell *, Node *, Node *);
extern  int     isarg(char *);
extern  char    *tokname(int);
extern  Cell    *(*proctab[])(Node **, int);
extern  int     ptoi(void *);
extern  Node    *itonp(int);

extern  void    syminit(void);
extern  void    arginit(int, char **);
extern  void    envinit(char **);
```

```
extern  Array   *makesymtab(int);
extern  void    freesymtab(Cell *);
extern  void    freeelem(Cell *, char *);
extern  Cell    *setsymtab(char *, char *, double, unsigned int, Array *);
extern  int     hash(char *, int);
extern  void    rehash(Array *);
extern  Cell    *lookup(char *, Array *);
extern  double  setfval(Cell *, double);
extern  void    funnyvar(Cell *, char *);
extern  char    *setsval(Cell *, char *);
extern  double  getfval(Cell *);
extern  char    *getsval(Cell *);
extern  char    *tostring(char *);
extern  char    *qstring(char *, int);

extern  void    recinit(unsigned int);
extern  void    initgetrec(void);
extern  void    makefields(int, int);
extern  void    growfldtab(int n);
extern  int     getrec(char **, int *, int);
extern  void    nextfile(void);
extern  int     readrec(char **buf, int *bufsize, FILE *inf);
extern  char    *getargv(int);
extern  void    setclvar(char *);
extern  void    fldbld(void);
extern  void    cleanfld(int, int);
extern  void    newfld(int);
extern  int     refldbld(char *, char *);
extern  void    recbld(void);
extern  Cell    *fieldadr(int);
extern  void    yyerror(char *);
extern  void    fpecatch(int);
extern  void    bracecheck(void);
extern  void    bcheck2(int, int, int);
extern  void    SYNTAX(char *, ...);
extern  void    FATAL(char *, ...);
extern  void    WARNING(char *, ...);
extern  void    error(void);
extern  void    eprint(void);
extern  void    bclass(int);
extern  double  errcheck(double, char *);
extern  int     isclvar(char *);
extern  int     is_number(char *);

extern  int     adjbuf(char **pb, int *sz, int min, int q, char **pbp, char *what);
extern  void    run(Node *);
extern  Cell    *execute(Node *);
extern  Cell    *program(Node **, int);
extern  Cell    *call(Node **, int);
extern  Cell    *copycell(Cell *);
extern  Cell    *arg(Node **, int);
extern  Cell    *jump(Node **, int);
extern  Cell    *getline(Node **, int);
extern  Cell    *getnf(Node **, int);
extern  Cell    *array(Node **, int);
extern  Cell    *awkdelete(Node **, int);
extern  Cell    *intest(Node **, int);
extern  Cell    *matchop(Node **, int);
extern  Cell    *boolop(Node **, int);
extern  Cell    *relop(Node **, int);
```

```
extern  void    tfree(Cell *);
extern  Cell    *gettemp(void);
extern  Cell    *field(Node **, int);
extern  Cell    *indirect(Node **, int);
extern  Cell    *substr(Node **, int);
extern  Cell    *sindex(Node **, int);
extern  int     format(char **, int *, char *, Node *);
extern  Cell    *awksprintf(Node **, int);
extern  Cell    *awkprintf(Node **, int);
extern  Cell    *arith(Node **, int);
extern  double  ipow(double, int);
extern  Cell    *incrdecr(Node **, int);
extern  Cell    *assign(Node **, int);
extern  Cell    *cat(Node **, int);
extern  Cell    *pastat(Node **, int);
extern  Cell    *dopa2(Node **, int);
extern  Cell    *split(Node **, int);
extern  Cell    *condexpr(Node **, int);
extern  Cell    *ifstat(Node **, int);
extern  Cell    *whilestat(Node **, int);
extern  Cell    *dostat(Node **, int);
extern  Cell    *forstat(Node **, int);
extern  Cell    *instat(Node **, int);
extern  Cell    *bltin(Node **, int);
extern  Cell    *printstat(Node **, int);
extern  Cell    *nullproc(Node **, int);
extern  FILE    *redirect(int, Node *);
extern  FILE    *openfile(int, char *);
extern  char    *filename(FILE *);
extern  Cell    *closefile(Node **, int);
extern  void    closeall(void);
extern  Cell    *sub(Node **, int);
extern  Cell    *gsub(Node **, int);

extern  FILE    *popen(const char *, const char *);
extern  int pclose(FILE *);
```

A.9.8 awk/re.c

⟨awk/re.c 371⟩≡

```
⟨awk copyright lucent 343a⟩
#define DEBUG
#include <stdio.h>

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

#include <ctype.h>

//#include <setjmp.h>
//#include <math.h>
//#include <string.h>
//#include <stdlib.h>
//#include <time.h>
#include <bio.h>
#include <regexp.h>

#include "awk.h"
#include "y.tab.h"
```

```
//#include "regexp.h"

    /* This file provides the interface between the main body of
     * awk and the pattern matching package.  It preprocesses
     * patterns prior to compilation to provide awk-like semantics
     * to character sequences not supported by the pattern package.
     * The following conversions are performed:
     *
     *  "()"        ->  "[]"
     *  "[-"        ->  "[\-"
     *  "[^-"       ->  "[^\-"
     *  "-]"        ->  "\-]"
     *  "[]"        ->  "[]*"
     *  "\xdddd"    ->  "\z" where 'z' is the UTF sequence
     *                  for the hex value
     *  "\ddd"      ->  "\o" where 'o' is a char octal value
     *  "\b"        ->  "\B"    where 'B' is backspace
     *  "\t"        ->  "\T"    where 'T' is tab
     *  "\f"        ->  "\F"    where 'F' is form feed
     *  "\n"        ->  "\N"    where 'N' is newline
     *  "\r"        ->  "\r"    where 'C' is cr
     */

#define MAXRE   512

static char re[MAXRE];  /* copy buffer */

char    *patbeg;
int patlen;         /* number of chars in pattern */

#define NPATS   20      /* number of slots in pattern cache */

static struct pat_list      /* dynamic pattern cache */
{
    char    *re;
    int use;
    Reprog  *program;
} pattern[NPATS];

static int npats;       /* cache fill level */

    /* Compile a pattern */
void
*compre(char *pat)
{
    int i, j, inclass;
    char c, *p, *s;
    Reprog *program;

    if (!compile_time) {    /* search cache for dynamic pattern */
        for (i = 0; i < npats; i++)
            if (!strcmp(pat, pattern[i].re)) {
                pattern[i].use++;
                return((void *) pattern[i].program);
            }
    }
        /* Preprocess Pattern for compilation */
    p = re;
    s = pat;
    inclass = 0;
```

```
while (c = *s++) {
    if (c == '\\') {
        quoted(&s, &p, re+MAXRE);
        continue;
    }
    else if (!inclass && c == '(' && *s == ')') {
        if (p < re+MAXRE-2) {   /* '()' -> '[]*' */
            *p++ = '[';
            *p++ = ']';
            c = '*';
            s++;
        }
        else overflow();
    }
    else if (c == '['){         /* '[-' -> '[\-' */
        inclass = 1;
        if (*s == '-') {
            if (p < re+MAXRE-2) {
                *p++ = '[';
                *p++ = '\\';
                c = *s++;
            }
            else overflow();
        }               /* '[^-' -> '[^\-'*/
        else if (*s == '^' && s[1] == '-'){
            if (p < re+MAXRE-3) {
                *p++ = '[';
                *p++ = *s++;
                *p++ = '\\';
                c = *s++;
            }
            else overflow();
        }
        else if (*s == '['){        /* skip '[[' */
            if (p < re+MAXRE-1)
                *p++ = c;
            else overflow();
            c = *s++;
        }
        else if (*s == '^' && s[1] == '[') {    /* skip '[^['*/
            if (p < re+MAXRE-2) {
                *p++ = c;
                *p++ = *s++;
                c = *s++;
            }
            else overflow();
        }
        else if (*s == ']') {       /* '[]' -> '[]*' */
            if (p < re+MAXRE-2) {
                *p++ = c;
                *p++ = *s++;
                c = '*';
                inclass = 0;
            }
            else overflow();
        }
    }
    else if (c == '-' && *s == ']') {   /* '-]' -> '\-]' */
        if (p < re+MAXRE-1)
            *p++ = '\\';
```

```
            else overflow();
        }
        else if (c == ']')
            inclass = 0;
        if (p < re+MAXRE-1)
            *p++ = c;
        else overflow();
    }
    *p = 0;
    program = regcomp(re);      /* compile pattern */
    if (!compile_time) {
        if (npats < NPATS)  /* Room in cache */
            i = npats++;
        else {          /* Throw out least used */
            int use = pattern[0].use;
            i = 0;
            for (j = 1; j < NPATS; j++) {
                if (pattern[j].use < use) {
                    use = pattern[j].use;
                    i = j;
                }
            }
            xfree(pattern[i].program);
            xfree(pattern[i].re);
        }
        pattern[i].re = tostring(pat);
        pattern[i].program = program;
        pattern[i].use = 1;
    }
    return((void *) program);
}

    /* T/F match indication - matched string not exported */
int
match(void *p, char *s, char *start)
{
    return regexec((Reprog *) p, (char *) s, 0, 0);
}

    /* match and delimit the matched string */
int
pmatch(void *p, char *s, char *start)
{
    Resub m;

    m.s.sp = start;
    m.e.ep = 0;
    if (regexec((Reprog *) p, (char *) s, &m, 1)) {
        patbeg = m.s.sp;
        patlen = m.e.ep-m.s.sp;
        return 1;
    }
    patlen = -1;
    patbeg = start;
    return 0;
}

    /* perform a non-empty match */
int
nematch(void *p, char *s, char *start)
```

```
{
    if (pmatch(p, s, start) == 1 && patlen > 0)
        return 1;
    patlen = -1;
    patbeg = start;
    return 0;
}
/* in the parsing of regular expressions, metacharacters like . have */
/* to be seen literally;  \056 is not a metacharacter. */
int
hexstr(char **pp)   /* find and eval hex string at pp, return new p */
{
    char c;
    int n = 0;
    int i;

    for (i = 0, c = (*pp)[i]; i < 4 && isxdigit(c); i++, c = (*pp)[i]) {
        if (isdigit(c))
            n = 16 * n + c - '0';
        else if ('a' <= c && c <= 'f')
            n = 16 * n + c - 'a' + 10;
        else if ('A' <= c && c <= 'F')
            n = 16 * n + c - 'A' + 10;
    }
    *pp += i;
    return n;
}

    /* look for awk-specific escape sequences */

#define isoctdigit(c) ((c) >= '0' && (c) <= '7') /* multiple use of arg */

void
quoted(char **s, char **to, char *end)  /* handle escaped sequence */
{
    char *p = *s;
    char *t = *to;
    wchar_t c;

    switch(c = *p++) {
    case 't':
        c = '\t';
        break;
    case 'n':
        c = '\n';
        break;
    case 'f':
        c = '\f';
        break;
    case 'r':
        c = '\r';
        break;
    case 'b':
        c = '\b';
        break;
    default:
        if (t < end-1)      /* all else must be escaped */
            *t++ = '\\';
        if (c == 'x') {     /* hexadecimal goo follows */
            c = hexstr(&p);
```

```
            if (t < end-MB_CUR_MAX)
                t += wctomb(t, c);
            else overflow();
            *to = t;
            *s = p;
            return;
        } else if (isoctdigit(c)) { /* \d \dd \ddd */
            c -= '0';
            if (isoctdigit(*p)) {
                c = 8 * c + *p++ - '0';
                if (isoctdigit(*p))
                    c = 8 * c + *p++ - '0';
            }
        }
        break;
    }
    if (t < end-1)
        *t++ = c;
    *s = p;
    *to = t;
}
    /* count rune positions */
int
countposn(char *s, int n)
{
    int i, j;
    char *end;

    for (i = 0, end = s+n; *s && s < end; i++){
        j = mblen(s, n);
        if(j <= 0)
            j = 1;
        s += j;
    }
    return(i);
}

    /* pattern package error handler */

void
regerror(char *s)
{
    FATAL("%s", s);
}

void
overflow(void)
{
    FATAL("%s", "regular expression too big");
}
```

A.9.9 awk/run.c

⟨awk/run.c 376⟩≡

```
⟨awk copyright lucent 343a⟩
#define DEBUG
#include <stdio.h>
#include <ctype.h>
#include <setjmp.h>
```

```
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <utf.h>
#include "awk.h"
#include "y.tab.h"

#define tempfree(x) if (istemp(x)) tfree(x); else

/*
#undef tempfree

void tempfree(Cell *p) {
    if (p->ctype == OCELL && (p->csub < CUNK || p->csub > CFREE)) {
        WARNING("bad csub %d in Cell %d %s",
            p->csub, p->ctype, p->sval);
    }
    if (istemp(p))
        tfree(p);
}
*/

#ifdef _NFILE
#ifndef FOPEN_MAX
#define FOPEN_MAX _NFILE
#endif
#endif

#ifndef FOPEN_MAX
#define FOPEN_MAX   40  /* max number of open files */
#endif

#ifndef RAND_MAX
#define RAND_MAX    32767   /* all that ansi guarantees */
#endif

jmp_buf env;
extern  int pairstack[];

Node    *winner = NULL; /* root of parse tree */
Cell    *tmps;      /* free temporary cells for execution */

static Cell truecell    ={ OBOOL, BTRUE, 0, 0, 1.0, NUM };
Cell    *True   = &truecell;
static Cell falsecell   ={ OBOOL, BFALSE, 0, 0, 0.0, NUM };
Cell    *False  = &falsecell;
static Cell breakcell   ={ OJUMP, JBREAK, 0, 0, 0.0, NUM };
Cell    *jbreak = &breakcell;
static Cell contcell    ={ OJUMP, JCONT, 0, 0, 0.0, NUM };
Cell    *jcont  = &contcell;
static Cell nextcell    ={ OJUMP, JNEXT, 0, 0, 0.0, NUM };
Cell    *jnext  = &nextcell;
static Cell nextfilecell    ={ OJUMP, JNEXTFILE, 0, 0, 0.0, NUM };
Cell    *jnextfile  = &nextfilecell;
static Cell exitcell    ={ OJUMP, JEXIT, 0, 0, 0.0, NUM };
Cell    *jexit  = &exitcell;
static Cell retcell     ={ OJUMP, JRET, 0, 0, 0.0, NUM };
Cell    *jret   = &retcell;
static Cell tempcell    ={ OCELL, CTEMP, 0, "", 0.0, NUM|STR|DONTFREE };
```

```
Node    *curnode = NULL;    /* the node being executed, for debugging */

/* buffer memory management */
int adjbuf(char **pbuf, int *psiz, int minlen, int quantum, char **pbptr,
    char *whatrtn)
/* pbuf:    address of pointer to buffer being managed
 * psiz:    address of buffer size variable
 * minlen:  minimum length of buffer needed
 * quantum: buffer size quantum
 * pbptr:   address of movable pointer into buffer, or 0 if none
 * whatrtn: name of the calling routine if failure should cause fatal error
 *
 * return   0 for realloc failure, !=0 for success
 */
{
    if (minlen > *psiz) {
        char *tbuf;
        int rminlen = quantum ? minlen % quantum : 0;
        int boff = pbptr ? *pbptr - *pbuf : 0;
        /* round up to next multiple of quantum */
        if (rminlen)
            minlen += quantum - rminlen;
        tbuf = (char *) realloc(*pbuf, minlen);
        if (tbuf == NULL) {
            if (whatrtn)
                FATAL("out of memory in %s", whatrtn);
            return 0;
        }
        *pbuf = tbuf;
        *psiz = minlen;
        if (pbptr)
            *pbptr = tbuf + boff;
    }
    return 1;
}

void run(Node *a)   /* execution of parse tree starts here */
{
    extern void stdinit(void);

    stdinit();
    execute(a);
    closeall();
}

Cell *execute(Node *u)  /* execute a node of the parse tree */
{
    int nobj;
    Cell *(*proc)(Node **, int);
    Cell *x;
    Node *a;

    if (u == NULL)
        return(True);
    for (a = u; ; a = a->nnext) {
        curnode = a;
        if (isvalue(a)) {
            x = (Cell *) (a->narg[0]);
            if (isfld(x) && !donefld)
```

```
                fldbld();
            else if (isrec(x) && !donerec)
                recbld();
            return(x);
        }
        nobj = a->nobj;
        if (notlegal(nobj)) /* probably a Cell* but too risky to print */
            FATAL("illegal statement");
        proc = proctab[nobj-FIRSTTOKEN];
        x = (*proc)(a->narg, nobj);
        if (isfld(x) && !donefld)
            fldbld();
        else if (isrec(x) && !donerec)
            recbld();
        if (isexpr(a))
            return(x);
        if (isjump(x))
            return(x);
        if (a->nnext == NULL)
            return(x);
        tempfree(x);
    }
}


Cell *program(Node **a, int n)  /* execute an awk program */
{               /* a[0] = BEGIN, a[1] = body, a[2] = END */
    Cell *x;

    if (setjmp(env) != 0)
        goto ex;
    if (a[0]) {     /* BEGIN */
        x = execute(a[0]);
        if (isexit(x))
            return(True);
        if (isjump(x))
            FATAL("illegal break, continue, next or nextfile from BEGIN");
        tempfree(x);
    }
    if (a[1] || a[2])
        while (getrec(&record, &recsize, 1) > 0) {
            x = execute(a[1]);
            if (isexit(x))
                break;
            tempfree(x);
        }
  ex:
    if (setjmp(env) != 0)   /* handles exit within END */
        goto ex1;
    if (a[2]) {     /* END */
        x = execute(a[2]);
        if (isbreak(x) || isnext(x) || iscont(x))
            FATAL("illegal break, continue, next or nextfile from END");
        tempfree(x);
    }
  ex1:
    return(True);
}

struct Frame {  /* stack frame for awk function calls */
```

```
    int nargs;  /* number of arguments in this call */
    Cell *fcncell;  /* pointer to Cell for function */
    Cell **args;    /* pointer to array of arguments after execute */
    Cell *retval;   /* return value */
};

#define NARGS   50  /* max args in a call */

struct Frame *frame = NULL; /* base of stack frames; dynamically allocated */
int nframe = 0;     /* number of frames allocated */
struct Frame *fp = NULL;    /* frame pointer. bottom level unused */

Cell *call(Node **a, int n) /* function call.  very kludgy and fragile */
{
    static Cell newcopycell = { OCELL, CCOPY, 0, "", 0.0, NUM|STR|DONTFREE };
    int i, ncall, ndef;
    Node *x;
    Cell *args[NARGS], *oargs[NARGS];   /* BUG: fixed size arrays */
    Cell *y, *z, *fcn;
    char *s;

    fcn = execute(a[0]);    /* the function itself */
    s = fcn->nval;
    if (!isfcn(fcn))
        FATAL("calling undefined function %s", s);
    if (frame == NULL) {
        fp = frame = (struct Frame *) calloc(nframe += 100, sizeof(struct Frame));
        if (frame == NULL)
            FATAL("out of space for stack frames calling %s", s);
    }
    for (ncall = 0, x = a[1]; x != NULL; x = x->nnext)  /* args in call */
        ncall++;
    ndef = (int) fcn->fval;         /* args in defn */
       dprintf( ("calling %s, %d args (%d in defn), fp=%d\n", s, ncall, ndef, (int) (fp-frame)) );
    if (ncall > ndef)
        WARNING("function %s called with %d args, uses only %d",
            s, ncall, ndef);
    if (ncall + ndef > NARGS)
        FATAL("function %s has %d arguments, limit %d", s, ncall+ndef, NARGS);
    for (i = 0, x = a[1]; x != NULL; i++, x = x->nnext) {   /* get call args */
           dprintf( ("evaluate args[%d], fp=%d:\n", i, (int) (fp-frame)) );
        y = execute(x);
        oargs[i] = y;
           dprintf( ("args[%d]: %s %f <%s>, t=%o\n",
                i, y->nval, y->fval, isarr(y) ? "(array)" : y->sval, y->tval) );
        if (isfcn(y))
            FATAL("can't use function %s as argument in %s", y->nval, s);
        if (isarr(y))
            args[i] = y;    /* arrays by ref */
        else
            args[i] = copycell(y);
        tempfree(y);
    }
    for ( ; i < ndef; i++) {    /* add null args for ones not provided */
        args[i] = gettemp();
        *args[i] = newcopycell;
    }
    fp++;   /* now ok to up frame */
    if (fp >= frame + nframe) {
        int dfp = fp - frame;   /* old index */
```

```
        frame = (struct Frame *)
            realloc((char *) frame, (nframe += 100) * sizeof(struct Frame));
        if (frame == NULL)
            FATAL("out of space for stack frames in %s", s);
        fp = frame + dfp;
    }
    fp->fcncell = fcn;
    fp->args = args;
    fp->nargs = ndef;   /* number defined with (excess are locals) */
    fp->retval = gettemp();

       dprintf( ("start exec of %s, fp=%d\n", s, (int) (fp-frame)) );
    y = execute((Node *)(fcn->sval));   /* execute body */
       dprintf( ("finished exec of %s, fp=%d\n", s, (int) (fp-frame)) );

    for (i = 0; i < ndef; i++) {
        Cell *t = fp->args[i];
        if (isarr(t)) {
            if (t->csub == CCOPY) {
                if (i >= ncall) {
                    freesymtab(t);
                    t->csub = CTEMP;
                    tempfree(t);
                } else {
                    oargs[i]->tval = t->tval;
                    oargs[i]->tval &= ~(STR|NUM|DONTFREE);
                    oargs[i]->sval = t->sval;
                    tempfree(t);
                }
            }
        } else if (t != y) {    /* kludge to prevent freeing twice */
            t->csub = CTEMP;
            tempfree(t);
        }
    }
    tempfree(fcn);
    if (isexit(y) || isnext(y) || isnextfile(y))
        return y;
    tempfree(y);        /* this can free twice! */
    z = fp->retval;         /* return value */
       dprintf( ("%s returns %g |%s| %o\n", s, getfval(z), getsval(z), z->tval) );
    fp--;
    return(z);
}

Cell *copycell(Cell *x) /* make a copy of a cell in a temp */
{
    Cell *y;

    y = gettemp();
    y->csub = CCOPY;    /* prevents freeing until call is over */
    y->nval = x->nval;  /* BUG? */
    y->sval = x->sval ? tostring(x->sval) : NULL;
    y->fval = x->fval;
    y->tval = x->tval & ~(CON|FLD|REC|DONTFREE);    /* copy is not constant or field */
                            /* is DONTFREE right? */
    return y;
}

Cell *arg(Node **a, int n)  /* nth argument of a function */
```

```
{

    n = ptoi(a[0]); /* argument number, counting from 0 */
       dprintf( ("arg(%d), fp->nargs=%d\n", n, fp->nargs) );
    if (n+1 > fp->nargs)
        FATAL("argument #%d of function %s was not supplied",
            n+1, fp->fcncell->nval);
    return fp->args[n];
}

Cell *jump(Node **a, int n) /* break, continue, next, nextfile, return */
{
    Cell *y;

    switch (n) {
    case EXIT:
        if (a[0] != NULL) {
            y = execute(a[0]);
            errorflag = (int) getfval(y);
            tempfree(y);
        }
        longjmp(env, 1);
    case RETURN:
        if (a[0] != NULL) {
            y = execute(a[0]);
            if ((y->tval & (STR|NUM)) == (STR|NUM)) {
                setsval(fp->retval, getsval(y));
                fp->retval->fval = getfval(y);
                fp->retval->tval |= NUM;
            }
            else if (y->tval & STR)
                setsval(fp->retval, getsval(y));
            else if (y->tval & NUM)
                setfval(fp->retval, getfval(y));
            else        /* can't happen */
                FATAL("bad type variable %d", y->tval);
            tempfree(y);
        }
        return(jret);
    case NEXT:
        return(jnext);
    case NEXTFILE:
        nextfile();
        return(jnextfile);
    case BREAK:
        return(jbreak);
    case CONTINUE:
        return(jcont);
    default:    /* can't happen */
        FATAL("illegal jump type %d", n);
    }
    return 0;   /* not reached */
}

Cell *getline(Node **a, int n)  /* get next line from specific input */
{       /* a[0] is variable, a[1] is operator, a[2] is filename */
    Cell *r, *x;
    extern Cell **fldtab;
    FILE *fp;
    char *buf;
```

```
    int bufsize = recsize;
    int mode;

    if ((buf = (char *) malloc(bufsize)) == NULL)
        FATAL("out of memory in getline");

    fflush(stdout); /* in case someone is waiting for a prompt */
    r = gettemp();
    if (a[1] != NULL) {     /* getline < file */
        x = execute(a[2]);      /* filename */
        mode = ptoi(a[1]);
        if (mode == '|')        /* input pipe */
            mode = LE;  /* arbitrary flag */
        fp = openfile(mode, getsval(x));
        tempfree(x);
        if (fp == NULL)
            n = -1;
        else
            n = readrec(&buf, &bufsize, fp);
        if (n <= 0) {
            ;
        } else if (a[0] != NULL) {  /* getline var <file */
            x = execute(a[0]);
            setsval(x, buf);
            tempfree(x);
        } else {            /* getline <file */
            setsval(fldtab[0], buf);
            if (is_number(fldtab[0]->sval)) {
                fldtab[0]->fval = atof(fldtab[0]->sval);
                fldtab[0]->tval |= NUM;
            }
        }
    } else {            /* bare getline; use current input */
        if (a[0] == NULL)   /* getline */
            n = getrec(&record, &recsize, 1);
        else {          /* getline var */
            n = getrec(&buf, &bufsize, 0);
            x = execute(a[0]);
            setsval(x, buf);
            tempfree(x);
        }
    }
    setfval(r, (Awkfloat) n);
    free(buf);
    return r;
}

Cell *getnf(Node **a, int n)    /* get NF */
{
    if (donefld == 0)
        fldbld();
    return (Cell *) a[0];
}

Cell *array(Node **a, int n)    /* a[0] is symtab, a[1] is list of subscripts */
{
    Cell *x, *y, *z;
    char *s;
    Node *np;
    char *buf;
```

```
    int bufsz = recsize;
    int nsub = strlen(*SUBSEP);

    if ((buf = (char *) malloc(bufsz)) == NULL)
        FATAL("out of memory in array");

    x = execute(a[0]);  /* Cell* for symbol table */
    buf[0] = 0;
    for (np = a[1]; np; np = np->nnext) {
        y = execute(np);    /* subscript */
        s = getsval(y);
        if (!adjbuf(&buf, &bufsz, strlen(buf)+strlen(s)+nsub+1, recsize, 0, 0))
            FATAL("out of memory for %s[%s...]", x->nval, buf);
        strcat(buf, s);
        if (np->nnext)
            strcat(buf, *SUBSEP);
        tempfree(y);
    }
    if (!isarr(x)) {
           dprintf( ("making %s into an array\n", x->nval) );
        if (freeable(x))
            xfree(x->sval);
        x->tval &= ~(STR|NUM|DONTFREE);
        x->tval |= ARR;
        x->sval = (char *) makesymtab(NSYMTAB);
    }
    z = setsymtab(buf, "", 0.0, STR|NUM, (Array *) x->sval);
    z->ctype = OCELL;
    z->csub = CVAR;
    tempfree(x);
    free(buf);
    return(z);
}

Cell *awkdelete(Node **a, int n)    /* a[0] is symtab, a[1] is list of subscripts */
{
    Cell *x, *y;
    Node *np;
    char *s;
    int nsub = strlen(*SUBSEP);

    x = execute(a[0]);  /* Cell* for symbol table */
    if (!isarr(x))
        return True;
    if (a[1] == 0) {    /* delete the elements, not the table */
        freesymtab(x);
        x->tval &= ~STR;
        x->tval |= ARR;
        x->sval = (char *) makesymtab(NSYMTAB);
    } else {
        int bufsz = recsize;
        char *buf;
        if ((buf = (char *) malloc(bufsz)) == NULL)
            FATAL("out of memory in adelete");
        buf[0] = 0;
        for (np = a[1]; np; np = np->nnext) {
            y = execute(np);    /* subscript */
            s = getsval(y);
            if (!adjbuf(&buf, &bufsz, strlen(buf)+strlen(s)+nsub+1, recsize, 0, 0))
                FATAL("out of memory deleting %s[%s...]", x->nval, buf);
```

```
            strcat(buf, s);
            if (np->nnext)
                strcat(buf, *SUBSEP);
            tempfree(y);
        }
        freeelem(x, buf);
        free(buf);
    }
    tempfree(x);
    return True;
}

Cell *intest(Node **a, int n)   /* a[0] is index (list), a[1] is symtab */
{
    Cell *x, *ap, *k;
    Node *p;
    char *buf;
    char *s;
    int bufsz = recsize;
    int nsub = strlen(*SUBSEP);

    ap = execute(a[1]); /* array name */
    if (!isarr(ap)) {
           dprintf( ("making %s into an array\n", ap->nval) );
        if (freeable(ap))
            xfree(ap->sval);
        ap->tval &= ~(STR|NUM|DONTFREE);
        ap->tval |= ARR;
        ap->sval = (char *) makesymtab(NSYMTAB);
    }
    if ((buf = (char *) malloc(bufsz)) == NULL) {
        FATAL("out of memory in intest");
    }
    buf[0] = 0;
    for (p = a[0]; p; p = p->nnext) {
        x = execute(p); /* expr */
        s = getsval(x);
        if (!adjbuf(&buf, &bufsz, strlen(buf)+strlen(s)+nsub+1, recsize, 0, 0))
            FATAL("out of memory deleting %s[%s...]", x->nval, buf);
        strcat(buf, s);
        tempfree(x);
        if (p->nnext)
            strcat(buf, *SUBSEP);
    }
    k = lookup(buf, (Array *) ap->sval);
    tempfree(ap);
    free(buf);
    if (k == NULL)
        return(False);
    else
        return(True);
}


Cell *matchop(Node **a, int n)  /* ~ and match() */
{
    Cell *x, *y;
    char *s, *t;
    int i;
    void *p;
```

```
    x = execute(a[1]);  /* a[1] = target text */
    s = getsval(x);
    if (a[0] == 0)      /* a[1] == 0: already-compiled reg expr */
        p = (void *) a[2];
    else {
        y = execute(a[2]);  /* a[2] = regular expr */
        t = getsval(y);
        p = compre(t);
        tempfree(y);
    }
    if (n == MATCHFCN)
        i = pmatch(p, s, s);
    else
        i = match(p, s, s);
    tempfree(x);
    if (n == MATCHFCN) {
        int start = countposn(s, patbeg-s)+1;
        if (patlen < 0)
            start = 0;
        setfval(rstartloc, (Awkfloat) start);
        setfval(rlengthloc, (Awkfloat) countposn(patbeg, patlen));
        x = gettemp();
        x->tval = NUM;
        x->fval = start;
        return x;
    } else if ((n == MATCH && i == 1) || (n == NOTMATCH && i == 0))
        return(True);
    else
        return(False);
}


Cell *boolop(Node **a, int n)   /* a[0] || a[1], a[0] && a[1], !a[0] */
{
    Cell *x, *y;
    int i;

    x = execute(a[0]);
    i = istrue(x);
    tempfree(x);
    switch (n) {
    case BOR:
        if (i) return(True);
        y = execute(a[1]);
        i = istrue(y);
        tempfree(y);
        if (i) return(True);
        else return(False);
    case AND:
        if ( !i ) return(False);
        y = execute(a[1]);
        i = istrue(y);
        tempfree(y);
        if (i) return(True);
        else return(False);
    case NOT:
        if (i) return(False);
        else return(True);
    default:    /* can't happen */
```

```
        FATAL("unknown boolean operator %d", n);
    }
    return 0;   /*NOTREACHED*/
}

Cell *relop(Node **a, int n)    /* a[0 < a[1], etc. */
{
    int i;
    Cell *x, *y;
    Awkfloat j;

    x = execute(a[0]);
    y = execute(a[1]);
    if (x->tval&NUM && y->tval&NUM) {
        j = x->fval - y->fval;
        i = j<0? -1: (j>0? 1: 0);
    } else {
        i = strcmp(getsval(x), getsval(y));
    }
    tempfree(x);
    tempfree(y);
    switch (n) {
    case LT:    if (i<0) return(True);
            else return(False);
    case LE:    if (i<=0) return(True);
            else return(False);
    case NE:    if (i!=0) return(True);
            else return(False);
    case EQ:    if (i == 0) return(True);
            else return(False);
    case GE:    if (i>=0) return(True);
            else return(False);
    case GT:    if (i>0) return(True);
            else return(False);
    default:    /* can't happen */
        FATAL("unknown relational operator %d", n);
    }
    return 0;   /*NOTREACHED*/
}

void tfree(Cell *a) /* free a tempcell */
{
    if (freeable(a)) {
           dprintf( ("freeing %s %s %o\n", a->nval, a->sval, a->tval) );
        xfree(a->sval);
    }
    if (a == tmps)
        FATAL("tempcell list is curdled");
    a->cnext = tmps;
    tmps = a;
}

Cell *gettemp(void) /* get a tempcell */
{   int i;
    Cell *x;

    if (!tmps) {
        tmps = (Cell *) calloc(100, sizeof(Cell));
        if (!tmps)
            FATAL("out of space for temporaries");
```

```
        for(i = 1; i < 100; i++)
            tmps[i-1].cnext = &tmps[i];
        tmps[i-1].cnext = 0;
    }
    x = tmps;
    tmps = x->cnext;
    *x = tempcell;
    return(x);
}

Cell *indirect(Node **a, int n) /* $( a[0] ) */
{
    Cell *x;
    int m;
    char *s;

    x = execute(a[0]);
    m = (int) getfval(x);
    if (m == 0 && !is_number(s = getsval(x)))   /* suspicion! */
        FATAL("illegal field $(%s), name \"%s\"", s, x->nval);
        /* BUG: can x->nval ever be null??? */
    tempfree(x);
    x = fieldadr(m);
    x->ctype = OCELL;   /* BUG?  why are these needed? */
    x->csub = CFLD;
    return(x);
}

Cell *substr(Node **a, int nnn)     /* substr(a[0], a[1], a[2]) */
{
    int k, m, n;
    char *s, *p;
    int temp;
    Cell *x, *y, *z = 0;

    x = execute(a[0]);
    y = execute(a[1]);
    if (a[2] != 0)
        z = execute(a[2]);
    s = getsval(x);
    k = countposn(s, strlen(s)) + 1;
    if (k <= 1) {
        tempfree(x);
        tempfree(y);
        if (a[2] != 0)
            tempfree(z);
        x = gettemp();
        setsval(x, "");
        return(x);
    }
    m = (int) getfval(y);
    if (m <= 0)
        m = 1;
    else if (m > k)
        m = k;
    tempfree(y);
    if (a[2] != 0) {
        n = (int) getfval(z);
        tempfree(z);
    } else
```

```
        n = k - 1;
    if (n < 0)
        n = 0;
    else if (n > k - m)
        n = k - m;
       dprintf( ("substr: m=%d, n=%d, s=%s\n", m, n, s) );
    y = gettemp();
    while (*s && --m)
         s += mblen(s, k);
    for (p = s; *p && n--; p += mblen(p, k))
            ;
    temp = *p;  /* with thanks to John Linderman */
    *p = '\0';
    setsval(y, s);
    *p = temp;
    tempfree(x);
    return(y);
}

Cell *sindex(Node **a, int nnn)     /* index(a[0], a[1]) */
{
    Cell *x, *y, *z;
    char *s1, *s2, *p1, *p2, *q;
    Awkfloat v = 0.0;

    x = execute(a[0]);
    s1 = getsval(x);
    y = execute(a[1]);
    s2 = getsval(y);

    z = gettemp();
    for (p1 = s1; *p1 != '\0'; p1++) {
        for (q=p1, p2=s2; *p2 != '\0' && *q == *p2; q++, p2++)
            ;
        if (*p2 == '\0') {
            v = (Awkfloat) countposn(s1, p1-s1) + 1;    /* origin 1 */
            break;
        }
    }
    tempfree(x);
    tempfree(y);
    setfval(z, v);
    return(z);
}

#define MAXNUMSIZE  50

int format(char **pbuf, int *pbufsize, char *s, Node *a)    /* printf-like conversions */
{
    char *fmt;
    char *p, *t, *os;
    Cell *x;
    int flag = 0, n;
    int fmtwd; /* format width */
    int fmtsz = recsize;
    char *buf = *pbuf;
    int bufsize = *pbufsize;

    os = s;
    p = buf;
```

```
if ((fmt = (char *) malloc(fmtsz)) == NULL)
    FATAL("out of memory in format()");
while (*s) {
    adjbuf(&buf, &bufsize, MAXNUMSIZE+1+p-buf, recsize, &p, "format");
    if (*s != '%') {
        *p++ = *s++;
        continue;
    }
    if (*(s+1) == '%') {
        *p++ = '%';
        s += 2;
        continue;
    }
    /* have to be real careful in case this is a huge number, eg, %100000d */
    fmtwd = atoi(s+1);
    if (fmtwd < 0)
        fmtwd = -fmtwd;
    adjbuf(&buf, &bufsize, fmtwd+1+p-buf, recsize, &p, "format");
    for (t = fmt; (*t++ = *s) != '\0'; s++) {
        if (!adjbuf(&fmt, &fmtsz, MAXNUMSIZE+1+t-fmt, recsize, &t, 0))
            FATAL("format item %.30s... ran format() out of memory", os);
        if (isalpha(*s) && *s != 'l' && *s != 'h' && *s != 'L')
            break;  /* the ansi panoply */
        if (*s == '*') {
            x = execute(a);
            a = a->nnext;
            sprintf(t-1, "%d", fmtwd=(int) getfval(x));
            if (fmtwd < 0)
                fmtwd = -fmtwd;
            adjbuf(&buf, &bufsize, fmtwd+1+p-buf, recsize, &p, "format");
            t = fmt + strlen(fmt);
            tempfree(x);
        }
    }
    *t = '\0';
    if (fmtwd < 0)
        fmtwd = -fmtwd;
    adjbuf(&buf, &bufsize, fmtwd+1+p-buf, recsize, &p, "format");

    switch (*s) {
    case 'f': case 'e': case 'g': case 'E': case 'G':
        flag = 1;
        break;
    case 'd': case 'i':
        flag = 2;
        if(*(s-1) == 'l') break;
        *(t-1) = 'l';
        *t = 'd';
        *++t = '\0';
        break;
    case 'o': case 'x': case 'X': case 'u':
        flag = *(s-1) == 'l' ? 2 : 3;
        break;
    case 's':
        flag = 4;
        break;
    case 'c':
        flag = 5;
        break;
    default:
```

```
            WARNING("weird printf conversion %s", fmt);
            flag = 0;
            break;
        }
        if (a == NULL)
            FATAL("not enough args in printf(%s)", os);
        x = execute(a);
        a = a->nnext;
        n = MAXNUMSIZE;
        if (fmtwd > n)
            n = fmtwd;
        adjbuf(&buf, &bufsize, 1+n+p-buf, recsize, &p, "format");
        switch (flag) {
        case 0: sprintf(p, "%s", fmt);  /* unknown, so dump it too */
            t = getsval(x);
            n = strlen(t);
            if (fmtwd > n)
                n = fmtwd;
            adjbuf(&buf, &bufsize, 1+strlen(p)+n+p-buf, recsize, &p, "format");
            p += strlen(p);
            sprintf(p, "%s", t);
            break;
        case 1: sprintf(p, fmt, getfval(x)); break;
        case 2: sprintf(p, fmt, (long) getfval(x)); break;
        case 3: sprintf(p, fmt, (int) getfval(x)); break;
        case 4:
            t = getsval(x);
            n = strlen(t);
            if (fmtwd > n)
                n = fmtwd;
            if (!adjbuf(&buf, &bufsize, 1+n+p-buf, recsize, &p, 0))
                FATAL("huge string/format (%d chars) in printf %.30s... ran format() out of memory", n, t);
            sprintf(p, fmt, t);
            break;
        case 5:
            if (isnum(x)) {
                if (getfval(x))
                    sprintf(p, fmt, (int) getfval(x));
                else{
                    *p++ = '\0';
                    *p = '\0';
                }
            } else
                sprintf(p, fmt, getsval(x)[0]);
            break;
        }
        tempfree(x);
        p += strlen(p);
        s++;
    }
    *p = '\0';
    free(fmt);
    for ( ; a; a = a->nnext)        /* evaluate any remaining args */
        execute(a);
    *pbuf = buf;
    *pbufsize = bufsize;
    return p - buf;
}

Cell *awksprintf(Node **a, int n)       /* sprintf(a[0]) */
```

```
{
    Cell *x;
    Node *y;
    char *buf;
    int bufsz=3*recsize;

    if ((buf = (char *) malloc(bufsz)) == NULL)
        FATAL("out of memory in awksprintf");
    y = a[0]->nnext;
    x = execute(a[0]);
    if (format(&buf, &bufsz, getsval(x), y) == -1)
        FATAL("sprintf string %.30s... too long.  can't happen.", buf);
    tempfree(x);
    x = gettemp();
    x->sval = buf;
    x->tval = STR;
    return(x);
}

Cell *awkprintf(Node **a, int n)        /* printf */
{   /* a[0] is list of args, starting with format string */
    /* a[1] is redirection operator, a[2] is redirection file */
    FILE *fp;
    Cell *x;
    Node *y;
    char *buf;
    int len;
    int bufsz=3*recsize;

    if ((buf = (char *) malloc(bufsz)) == NULL)
        FATAL("out of memory in awkprintf");
    y = a[0]->nnext;
    x = execute(a[0]);
    if ((len = format(&buf, &bufsz, getsval(x), y)) == -1)
        FATAL("printf string %.30s... too long.  can't happen.", buf);
    tempfree(x);
    if (a[1] == NULL) {
        /* fputs(buf, stdout); */
        fwrite(buf, len, 1, stdout);
        if (ferror(stdout))
            FATAL("write error on stdout");
    } else {
        fp = redirect(ptoi(a[1]), a[2]);
        /* fputs(buf, fp); */
        fwrite(buf, len, 1, fp);
        fflush(fp);
        if (ferror(fp))
            FATAL("write error on %s", filename(fp));
    }
    free(buf);
    return(True);
}

Cell *arith(Node **a, int n)    /* a[0] + a[1], etc.  also -a[0] */
{
    Awkfloat i, j = 0;
    double v;
    Cell *x, *y, *z;

    x = execute(a[0]);
```

```
    i = getfval(x);
    tempfree(x);
    if (n != UMINUS) {
        y = execute(a[1]);
        j = getfval(y);
        tempfree(y);
    }
    z = gettemp();
    switch (n) {
    case ADD:
        i += j;
        break;
    case MINUS:
        i -= j;
        break;
    case MULT:
        i *= j;
        break;
    case DIVIDE:
        if (j == 0)
            FATAL("division by zero");
        i /= j;
        break;
    case MOD:
        if (j == 0)
            FATAL("division by zero in mod");
        modf(i/j, &v);
        i = i - j * v;
        break;
    case UMINUS:
        i = -i;
        break;
    case POWER:
        if (j >= 0 && modf(j, &v) == 0.0)   /* pos integer exponent */
            i = ipow(i, (int) j);
        else
            i = errcheck(pow(i, j), "pow");
        break;
    default:    /* can't happen */
        FATAL("illegal arithmetic operator %d", n);
    }
    setfval(z, i);
    return(z);
}

double ipow(double x, int n)    /* x**n.  ought to be done by pow, but isn't always */
{
    double v;

    if (n <= 0)
        return 1;
    v = ipow(x, n/2);
    if (n % 2 == 0)
        return v * v;
    else
        return x * v * v;
}

Cell *incrdecr(Node **a, int n)     /* a[0]++, etc. */
{
```

```
    Cell *x, *z;
    int k;
    Awkfloat xf;

    x = execute(a[0]);
    xf = getfval(x);
    k = (n == PREINCR || n == POSTINCR) ? 1 : -1;
    if (n == PREINCR || n == PREDECR) {
        setfval(x, xf + k);
        return(x);
    }
    z = gettemp();
    setfval(z, xf);
    setfval(x, xf + k);
    tempfree(x);
    return(z);
}

Cell *assign(Node **a, int n)   /* a[0] = a[1], a[0] += a[1], etc. */
{       /* this is subtle; don't muck with it. */
    Cell *x, *y;
    Awkfloat xf, yf;
    double v;

    y = execute(a[1]);
    x = execute(a[0]);
    if (n == ASSIGN) {  /* ordinary assignment */
        if (x == y && !(x->tval & (FLD|REC)))   /* self-assignment: */
            ;       /* leave alone unless it's a field */
        else if ((y->tval & (STR|NUM)) == (STR|NUM)) {
            setsval(x, getsval(y));
            x->fval = getfval(y);
            x->tval |= NUM;
        }
        else if (isstr(y))
            setsval(x, getsval(y));
        else if (isnum(y))
            setfval(x, getfval(y));
        else
            funnyvar(y, "read value of");
        tempfree(y);
        return(x);
    }
    xf = getfval(x);
    yf = getfval(y);
    switch (n) {
    case ADDEQ:
        xf += yf;
        break;
    case SUBEQ:
        xf -= yf;
        break;
    case MULTEQ:
        xf *= yf;
        break;
    case DIVEQ:
        if (yf == 0)
            FATAL("division by zero in /=");
        xf /= yf;
        break;
```

```
    case MODEQ:
        if (yf == 0)
            FATAL("division by zero in %%=");
        modf(xf/yf, &v);
        xf = xf - yf * v;
        break;
    case POWEQ:
        if (yf >= 0 && modf(yf, &v) == 0.0) /* pos integer exponent */
            xf = ipow(xf, (int) yf);
        else
            xf = errcheck(pow(xf, yf), "pow");
        break;
    default:
        FATAL("illegal assignment operator %d", n);
        break;
    }
    tempfree(y);
    setfval(x, xf);
    return(x);
}

Cell *cat(Node **a, int q)  /* a[0] cat a[1] */
{
    Cell *x, *y, *z;
    int n1, n2;
    char *s;

    x = execute(a[0]);
    y = execute(a[1]);
    getsval(x);
    getsval(y);
    n1 = strlen(x->sval);
    n2 = strlen(y->sval);
    s = (char *) malloc(n1 + n2 + 1);
    if (s == NULL)
        FATAL("out of space concatenating %.15s... and %.15s...",
            x->sval, y->sval);
    strcpy(s, x->sval);
    strcpy(s+n1, y->sval);
    tempfree(y);
    z = gettemp();
    z->sval = s;
    z->tval = STR;
    tempfree(x);
    return(z);
}

Cell *pastat(Node **a, int n)   /* a[0] { a[1] } */
{
    Cell *x;

    if (a[0] == 0)
        x = execute(a[1]);
    else {
        x = execute(a[0]);
        if (istrue(x)) {
            tempfree(x);
            x = execute(a[1]);
        }
    }
```

```
    return x;
}

Cell *dopa2(Node **a, int n)    /* a[0], a[1] { a[2] } */
{
    Cell *x;
    int pair;

    pair = ptoi(a[3]);
    if (pairstack[pair] == 0) {
        x = execute(a[0]);
        if (istrue(x))
            pairstack[pair] = 1;
        tempfree(x);
    }
    if (pairstack[pair] == 1) {
        x = execute(a[1]);
        if (istrue(x))
            pairstack[pair] = 0;
        tempfree(x);
        x = execute(a[2]);
        return(x);
    }
    return(False);
}

Cell *split(Node **a, int nnn)  /* split(a[0], a[1], a[2]); a[3] is type */
{
    Cell *x = 0, *y, *ap;
    char *s, *t, *fs = 0;
    char temp, num[50];
    int n, nb, sep, tempstat, arg3type;

    y = execute(a[0]);  /* source string */
    s = getsval(y);
    arg3type = ptoi(a[3]);
    if (a[2] == 0)      /* fs string */
        fs = *FS;
    else if (arg3type == STRING) {  /* split(str,arr,"string") */
        x = execute(a[2]);
        fs = getsval(x);
    } else if (arg3type == REGEXPR)
        fs = "(regexpr)";   /* split(str,arr,/regexpr/) */
    else
        FATAL("illegal type of split");
    sep = *fs;
    ap = execute(a[1]); /* array name */
    freesymtab(ap);
       dprintf( ("split: s=|%s|, a=%s, sep=|%s|\n", s, ap->nval, fs) );
    ap->tval &= ~STR;
    ap->tval |= ARR;
    ap->sval = (char *) makesymtab(NSYMTAB);

    n = 0;
    if ((*s != '\0' && strlen(fs) > 1) || arg3type == REGEXPR) {    /* reg expr */
        void *p;
        if (arg3type == REGEXPR) {  /* it's ready already */
            p = (void *) a[2];
        } else {
            p = compre(fs);
```

```
        }
        t = s;
        if (nematch(p,s,t)) {
            do {
                n++;
                sprintf(num, "%d", n);
                temp = *patbeg;
                *patbeg = '\0';
                if (is_number(t))
                    setsymtab(num, t, atof(t), STR|NUM, (Array *) ap->sval);
                else
                    setsymtab(num, t, 0.0, STR, (Array *) ap->sval);
                *patbeg = temp;
                t = patbeg + patlen;
                if (t[-1] == 0 || *t == 0) {
                    n++;
                    sprintf(num, "%d", n);
                    setsymtab(num, "", 0.0, STR, (Array *) ap->sval);
                    goto spdone;
                }
            } while (nematch(p,s,t));
        }
        n++;
        sprintf(num, "%d", n);
        if (is_number(t))
            setsymtab(num, t, atof(t), STR|NUM, (Array *) ap->sval);
        else
            setsymtab(num, t, 0.0, STR, (Array *) ap->sval);
spdone:
        p = NULL;
    } else if (sep == ' ') {
        for (n = 0; ; ) {
            while (*s == ' ' || *s == '\t' || *s == '\n')
                s++;
            if (*s == 0)
                break;
            n++;
            t = s;
            do
                s++;
            while (*s!=' ' && *s!='\t' && *s!='\n' && *s!='\0');
            temp = *s;
            *s = '\0';
            sprintf(num, "%d", n);
            if (is_number(t))
                setsymtab(num, t, atof(t), STR|NUM, (Array *) ap->sval);
            else
                setsymtab(num, t, 0.0, STR, (Array *) ap->sval);
            *s = temp;
            if (*s != 0)
                s++;
        }
    } else if (sep == 0) {  /* new: split(s, a, "") => 1 char/elem */
        for (n = 0; *s != 0; s += nb) {
            Rune r;
            char buf[UTFmax+1];

            n++;
            snprintf(num, sizeof num, "%d", n);
            nb = chartorune(&r, s);
```

```
            memmove(buf, s, nb);
            buf[nb] = '\0';
            if (isdigit(buf[0]))
                setsymtab(num, buf, atof(buf), STR|NUM, (Array *) ap->sval);
            else
                setsymtab(num, buf, 0.0, STR, (Array *) ap->sval);
        }
    } else if (*s != 0) {
        for (;;) {
            n++;
            t = s;
            while (*s != sep && *s != '\n' && *s != '\0')
                s++;
            temp = *s;
            *s = '\0';
            sprintf(num, "%d", n);
            if (is_number(t))
                setsymtab(num, t, atof(t), STR|NUM, (Array *) ap->sval);
            else
                setsymtab(num, t, 0.0, STR, (Array *) ap->sval);
            *s = temp;
            if (*s++ == 0)
                break;
        }
    }
    tempfree(ap);
    tempfree(y);
    if (a[2] != 0 && arg3type == STRING)
        tempfree(x);
    x = gettemp();
    x->tval = NUM;
    x->fval = n;
    return(x);
}

Cell *condexpr(Node **a, int n) /* a[0] ? a[1] : a[2] */
{
    Cell *x;

    x = execute(a[0]);
    if (istrue(x)) {
        tempfree(x);
        x = execute(a[1]);
    } else {
        tempfree(x);
        x = execute(a[2]);
    }
    return(x);
}

Cell *ifstat(Node **a, int n)   /* if (a[0]) a[1]; else a[2] */
{
    Cell *x;

    x = execute(a[0]);
    if (istrue(x)) {
        tempfree(x);
        x = execute(a[1]);
    } else if (a[2] != 0) {
        tempfree(x);
```

```
        x = execute(a[2]);
    }
    return(x);
}

Cell *whilestat(Node **a, int n)    /* while (a[0]) a[1] */
{
    Cell *x;

    for (;;) {
        x = execute(a[0]);
        if (!istrue(x))
            return(x);
        tempfree(x);
        x = execute(a[1]);
        if (isbreak(x)) {
            x = True;
            return(x);
        }
        if (isnext(x) || isexit(x) || isret(x))
            return(x);
        tempfree(x);
    }
}

Cell *dostat(Node **a, int n)   /* do a[0]; while(a[1]) */
{
    Cell *x;

    for (;;) {
        x = execute(a[0]);
        if (isbreak(x))
            return True;
        if (isnext(x) || isnextfile(x) || isexit(x) || isret(x))
            return(x);
        tempfree(x);
        x = execute(a[1]);
        if (!istrue(x))
            return(x);
        tempfree(x);
    }
}

Cell *forstat(Node **a, int n)  /* for (a[0]; a[1]; a[2]) a[3] */
{
    Cell *x;

    x = execute(a[0]);
    tempfree(x);
    for (;;) {
        if (a[1]!=0) {
            x = execute(a[1]);
            if (!istrue(x)) return(x);
            else tempfree(x);
        }
        x = execute(a[3]);
        if (isbreak(x))     /* turn off break */
            return True;
        if (isnext(x) || isexit(x) || isret(x))
            return(x);
```

```
        tempfree(x);
        x = execute(a[2]);
        tempfree(x);
    }
}

Cell *instat(Node **a, int n)   /* for (a[0] in a[1]) a[2] */
{
    Cell *x, *vp, *arrayp, *cp, *ncp;
    Array *tp;
    int i;

    vp = execute(a[0]);
    arrayp = execute(a[1]);
    if (!isarr(arrayp)) {
        return True;
    }
    tp = (Array *) arrayp->sval;
    tempfree(arrayp);
    for (i = 0; i < tp->size; i++) {    /* this routine knows too much */
        for (cp = tp->tab[i]; cp != NULL; cp = ncp) {
            setsval(vp, cp->nval);
            ncp = cp->cnext;
            x = execute(a[2]);
            if (isbreak(x)) {
                tempfree(vp);
                return True;
            }
            if (isnext(x) || isexit(x) || isret(x)) {
                tempfree(vp);
                return(x);
            }
            tempfree(x);
        }
    }
    return True;
}

Cell *bltin(Node **a, int n)    /* builtin functions. a[0] is type, a[1] is arg list */
{
    Cell *x, *y;
    Awkfloat u;
    int t;
    wchar_t wc;
    char *p, *buf;
    char mbc[50];
    Node *nextarg;
    FILE *fp;
    void flush_all(void);

    t = ptoi(a[0]);
    x = execute(a[1]);
    nextarg = a[1]->nnext;
    switch (t) {
    case FLENGTH:
        if (isarr(x))
            u = ((Array *) x->sval)->nelem; /* GROT. should be function*/
        else {
            p = getsval(x);
            u = (Awkfloat) countposn(p, strlen(p));
```

```
    }
    break;
case FLOG:
    u = errcheck(log(getfval(x)), "log"); break;
case FINT:
    modf(getfval(x), &u); break;
case FEXP:
    u = errcheck(exp(getfval(x)), "exp"); break;
case FSQRT:
    u = errcheck(sqrt(getfval(x)), "sqrt"); break;
case FSIN:
    u = sin(getfval(x)); break;
case FCOS:
    u = cos(getfval(x)); break;
case FATAN:
    if (nextarg == 0) {
        WARNING("atan2 requires two arguments; returning 1.0");
        u = 1.0;
    } else {
        y = execute(a[1]->nnext);
        u = atan2(getfval(x), getfval(y));
        tempfree(y);
        nextarg = nextarg->nnext;
    }
    break;
case FSYSTEM:
    fflush(stdout);     /* in case something is buffered already */
    u = (Awkfloat) system(getsval(x)) / 256;   /* 256 is unix-dep */
    break;
case FRAND:
    /* in principle, rand() returns something in 0..RAND_MAX */
    u = (Awkfloat) (rand() % RAND_MAX) / RAND_MAX;
    break;
case FSRAND:
    if (isrec(x))   /* no argument provided */
        u = time((time_t *)0);
    else
        u = getfval(x);
    srand((unsigned int) u);
    break;
case FTOUPPER:
case FTOLOWER:
    buf = tostring(getsval(x));
    if (t == FTOUPPER) {
        for (p = buf; *p; p++)
            if (islower(*p))
                *p = toupper(*p);
    } else {
        for (p = buf; *p; p++)
            if (isupper(*p))
                *p = tolower(*p);
    }
    tempfree(x);
    x = gettemp();
    setsval(x, buf);
    free(buf);
    return x;
case FFLUSH:
    if (isrec(x) || strlen(getsval(x)) == 0) {
        flush_all();    /* fflush() or fflush("") -> all */
```

```
            u = 0;
        } else if ((fp = openfile(FFLUSH, getsval(x))) == NULL)
            u = EOF;
        else
            u = fflush(fp);
        break;
    case FUTF:
        wc = (int)getfval(x);
        mbc[wctomb(mbc, wc)] = 0;
        tempfree(x);
        x = gettemp();
        setsval(x, mbc);
        return x;
    default:    /* can't happen */
        FATAL("illegal function type %d", t);
        break;
    }
    tempfree(x);
    x = gettemp();
    setfval(x, u);
    if (nextarg != 0) {
        WARNING("warning: function has too many arguments");
        for ( ; nextarg; nextarg = nextarg->nnext)
            execute(nextarg);
    }
    return(x);
}

Cell *printstat(Node **a, int n)    /* print a[0] */
{
    int r;
    Node *x;
    Cell *y;
    FILE *fp;

    if (a[1] == 0)  /* a[1] is redirection operator, a[2] is file */
        fp = stdout;
    else
        fp = redirect(ptoi(a[1]), a[2]);
    for (x = a[0]; x != NULL; x = x->nnext) {
        y = execute(x);
        fputs(getsval(y), fp);
        tempfree(y);
        if (x->nnext == NULL)
            r = fputs(*ORS, fp);
        else
            r = fputs(*OFS, fp);
        if (r == EOF)
            FATAL("write error on %s", filename(fp));
    }
    if (a[1] != 0)
        if (fflush(fp) == EOF)
            FATAL("write error on %s", filename(fp));
    return(True);
}

Cell *nullproc(Node **a, int n)
{
    n = n;
    a = a;
```

```
    return 0;
}


FILE *redirect(int a, Node *b)  /* set up all i/o redirections */
{
    FILE *fp;
    Cell *x;
    char *fname;

    x = execute(b);
    fname = getsval(x);
    fp = openfile(a, fname);
    if (fp == NULL)
        FATAL("can't open file %s", fname);
    tempfree(x);
    return fp;
}

struct files {
    FILE    *fp;
    char    *fname;
    int mode;   /* '|', 'a', 'w' => LE/LT, GT */
} files[FOPEN_MAX] ={
    { NULL,  "/dev/stdin",  LT },   /* watch out: don't free this! */
    { NULL, "/dev/stdout", GT },
    { NULL, "/dev/stderr", GT }
};

void stdinit(void)  /* in case stdin, etc., are not constants */
{
    files[0].fp = stdin;
    files[1].fp = stdout;
    files[2].fp = stderr;
}

FILE *openfile(int a, char *us)
{
    char *s = us;
    int i, m;
    FILE *fp = 0;

    if (*s == '\0')
        FATAL("null file name in print or getline");
    for (i=0; i < FOPEN_MAX; i++)
        if (files[i].fname && strcmp(s, files[i].fname) == 0) {
            if (a == files[i].mode || (a==APPEND && files[i].mode==GT))
                return files[i].fp;
            if (a == FFLUSH)
                return files[i].fp;
        }
    if (a == FFLUSH)    /* didn't find it, so don't create it! */
        return NULL;

    for (i=0; i < FOPEN_MAX; i++)
        if (files[i].fp == 0)
            break;
    if (i >= FOPEN_MAX)
        FATAL("%s makes too many open files", s);
    fflush(stdout); /* force a semblance of order */
```

```
    m = a;
    if (a == GT) {
        fp = fopen(s, "w");
    } else if (a == APPEND) {
        fp = fopen(s, "a");
        m = GT; /* so can mix > and >> */
    } else if (a == '|') {  /* output pipe */
        fp = popen(s, "w");
    } else if (a == LE) {   /* input pipe */
        fp = popen(s, "r");
    } else if (a == LT) {   /* getline <file */
        fp = strcmp(s, "-") == 0 ? stdin : fopen(s, "r");   /* "-" is stdin */
    } else  /* can't happen */
        FATAL("illegal redirection %d", a);
    if (fp != NULL) {
        files[i].fname = tostring(s);
        files[i].fp = fp;
        files[i].mode = m;
    }
    return fp;
}

char *filename(FILE *fp)
{
    int i;

    for (i = 0; i < FOPEN_MAX; i++)
        if (fp == files[i].fp)
            return files[i].fname;
    return "???";
}

Cell *closefile(Node **a, int n)
{
    Cell *x;
    int i, stat;

    n = n;
    x = execute(a[0]);
    getsval(x);
    for (i = 0; i < FOPEN_MAX; i++)
        if (files[i].fname && strcmp(x->sval, files[i].fname) == 0) {
            if (ferror(files[i].fp))
                WARNING( "i/o error occurred on %s", files[i].fname );
            if (files[i].mode == '|' || files[i].mode == LE)
                stat = pclose(files[i].fp);
            else
                stat = fclose(files[i].fp);
            if (stat == EOF)
                WARNING( "i/o error occurred closing %s", files[i].fname );
            if (i > 2)  /* don't do /dev/std... */
                xfree(files[i].fname);
            files[i].fname = NULL;  /* watch out for ref thru this */
            files[i].fp = NULL;
        }
    tempfree(x);
    return(True);
}

void closeall(void)
```

```
{
    int i, stat;

    for (i = 0; i < FOPEN_MAX; i++)
        if (files[i].fp) {
            if (ferror(files[i].fp))
                WARNING( "i/o error occurred on %s", files[i].fname );
            if (files[i].mode == '|' || files[i].mode == LE)
                stat = pclose(files[i].fp);
            else
                stat = fclose(files[i].fp);
            if (stat == EOF)
                WARNING( "i/o error occurred while closing %s", files[i].fname );
        }
}

void flush_all(void)
{
    int i;

    for (i = 0; i < FOPEN_MAX; i++)
        if (files[i].fp)
            fflush(files[i].fp);
}

void backsub(char **pb_ptr, char **sptr_ptr);

Cell *sub(Node **a, int nnn)    /* substitute command */
{
    char *sptr, *pb, *q;
    Cell *x, *y, *result;
    char *t, *buf;
    void *p;
    int bufsz = recsize;

    if ((buf = (char *) malloc(bufsz)) == NULL)
        FATAL("out of memory in sub");
    x = execute(a[3]);  /* target string */
    t = getsval(x);
    if (a[0] == 0)      /* 0 => a[1] is already-compiled regexpr */
        p = (void *) a[1];  /* regular expression */
    else {
        y = execute(a[1]);
        p = compre(getsval(y));
        tempfree(y);
    }
    y = execute(a[2]);  /* replacement string */
    result = False;
    if (pmatch(p, t, t)) {
        sptr = t;
        adjbuf(&buf, &bufsz, 1+patbeg-sptr, recsize, 0, "sub");
        pb = buf;
        while (sptr < patbeg)
            *pb++ = *sptr++;
        sptr = getsval(y);
        while (*sptr != 0) {
            adjbuf(&buf, &bufsz, 5+pb-buf, recsize, &pb, "sub");
            if (*sptr == '\\') {
                backsub(&pb, &sptr);
            } else if (*sptr == '&') {
```

```
                sptr++;
                adjbuf(&buf, &bufsz, 1+patlen+pb-buf, recsize, &pb, "sub");
                for (q = patbeg; q < patbeg+patlen; )
                    *pb++ = *q++;
            } else
                *pb++ = *sptr++;
        }
        *pb = '\0';
        if (pb > buf + bufsz)
            FATAL("sub result1 %.30s too big; can't happen", buf);
        sptr = patbeg + patlen;
        if ((patlen == 0 && *patbeg) || (patlen && *(sptr-1))) {
            adjbuf(&buf, &bufsz, 1+strlen(sptr)+pb-buf, 0, &pb, "sub");
            while ((*pb++ = *sptr++) != 0)
                ;
        }
        if (pb > buf + bufsz)
            FATAL("sub result2 %.30s too big; can't happen", buf);
        setsval(x, buf);    /* BUG: should be able to avoid copy */
        result = True;;
    }
    tempfree(x);
    tempfree(y);
    free(buf);
    return result;
}

Cell *gsub(Node **a, int nnn)   /* global substitute */
{
    Cell *x, *y;
    char *rptr, *sptr, *t, *pb, *c;
    char *buf;
    void *p;
    int mflag, num;
    int bufsz = recsize;

    if ((buf = (char *)malloc(bufsz)) == NULL)
        FATAL("out of memory in gsub");
    mflag = 0;  /* if mflag == 0, can replace empty string */
    num = 0;
    x = execute(a[3]);  /* target string */
    c = t = getsval(x);
    if (a[0] == 0)      /* 0 => a[1] is already-compiled regexpr */
        p = (void *) a[1];  /* regular expression */
    else {
        y = execute(a[1]);
        p = compre(getsval(y));
        tempfree(y);
    }
    y = execute(a[2]);  /* replacement string */
    if (pmatch(p, t, c)) {
        pb = buf;
        rptr = getsval(y);
        do {
            if (patlen == 0 && *patbeg != 0) {  /* matched empty string */
                if (mflag == 0) {   /* can replace empty */
                    num++;
                    sptr = rptr;
                    while (*sptr != 0) {
                        adjbuf(&buf, &bufsz, 5+pb-buf, recsize, &pb, "gsub");
```

```
                    if (*sptr == '\\') {
                        backsub(&pb, &sptr);
                    } else if (*sptr == '&') {
                        char *q;
                        sptr++;
                        adjbuf(&buf, &bufsz, 1+patlen+pb-buf, recsize, &pb, "gsub");
                        for (q = patbeg; q < patbeg+patlen; )
                            *pb++ = *q++;
                    } else
                        *pb++ = *sptr++;
                }
            }
            if (*c == 0)    /* at end */
                goto done;
            adjbuf(&buf, &bufsz, 2+pb-buf, recsize, &pb, "gsub");
            *pb++ = *c++;
            if (pb > buf + bufsz)   /* BUG: not sure of this test */
                FATAL("gsub result0 %.30s too big; can't happen", buf);
            mflag = 0;
        }
        else {  /* matched nonempty string */
            num++;
            sptr = c;
            adjbuf(&buf, &bufsz, 1+(patbeg-sptr)+pb-buf, recsize, &pb, "gsub");
            while (sptr < patbeg)
                *pb++ = *sptr++;
            sptr = rptr;
            while (*sptr != 0) {
                adjbuf(&buf, &bufsz, 5+pb-buf, recsize, &pb, "gsub");
                if (*sptr == '\\') {
                    backsub(&pb, &sptr);
                } else if (*sptr == '&') {
                    char *q;
                    sptr++;
                    adjbuf(&buf, &bufsz, 1+patlen+pb-buf, recsize, &pb, "gsub");
                    for (q = patbeg; q < patbeg+patlen; )
                        *pb++ = *q++;
                } else
                    *pb++ = *sptr++;
            }
            c = patbeg + patlen;
            if ((c[-1] == 0) || (*c == 0))
                goto done;
            if (pb > buf + bufsz)
                FATAL("gsub result1 %.30s too big; can't happen", buf);
            mflag = 1;
        }
    } while (pmatch(p, t, c));
    sptr = c;
    adjbuf(&buf, &bufsz, 1+strlen(sptr)+pb-buf, 0, &pb, "gsub");
    while ((*pb++ = *sptr++) != 0)
        ;
done:   if (pb > buf + bufsz)
        FATAL("gsub result2 %.30s too big; can't happen", buf);
    *pb = '\0';
    setsval(x, buf);    /* BUG: should be able to avoid copy + free */
}
tempfree(x);
tempfree(y);
x = gettemp();
```

```
    x->tval = NUM;
    x->fval = num;
    free(buf);
    return(x);
}

void backsub(char **pb_ptr, char **sptr_ptr)    /* handle \\& variations */
{                       /* sptr[0] == '\\' */
    char *pb = *pb_ptr, *sptr = *sptr_ptr;

    if (sptr[1] == '\\') {
        if (sptr[2] == '\\' && sptr[3] == '&') { /* \\\& -> \& */
            *pb++ = '\\';
            *pb++ = '&';
            sptr += 4;
        } else if (sptr[2] == '&') {    /* \\& -> \ + matched */
            *pb++ = '\\';
            sptr += 2;
        } else {            /* \\x -> \\x */
            *pb++ = *sptr++;
            *pb++ = *sptr++;
        }
    } else if (sptr[1] == '&') {    /* literal & */
        sptr++;
        *pb++ = *sptr++;
    } else              /* literal \ */
        *pb++ = *sptr++;

    *pb_ptr = pb;
    *sptr_ptr = sptr;
}
```

A.9.10 awk/tran.c

⟨awk/tran.c 408⟩≡

```
⟨awk copyright lucent 343a⟩
#define DEBUG
#include <stdio.h>
#include <math.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include "awk.h"
#include "y.tab.h"

#define FULLTAB 2   /* rehash when table gets this x full */
#define GROWTAB 4   /* grow table by this factor */

Array   *symtab;    /* main symbol table */

char    **FS;       /* initial field sep */
char    **RS;       /* initial record sep */
char    **OFS;      /* output field sep */
char    **ORS;      /* output record sep */
char    **OFMT;     /* output format for numbers */
char    **CONVFMT;  /* format for conversions in getsval */
Awkfloat *NF;       /* number of fields in current record */
Awkfloat *NR;       /* number of current record */
Awkfloat *FNR;      /* number of current record in current file */
```

```
char    **FILENAME; /* current filename argument */
Awkfloat *ARGC;     /* number of arguments from command line */
char    **SUBSEP;   /* subscript separator for a[i,j,k]; default \034 */
Awkfloat *RSTART;   /* start of re matched with ~; origin 1 (!) */
Awkfloat *RLENGTH;  /* length of same */

Cell    *nrloc;     /* NR */
Cell    *nfloc;     /* NF */
Cell    *fnrloc;    /* FNR */
Array   *ARGVtab;   /* symbol table containing ARGV[...] */
Array   *ENVtab;    /* symbol table containing ENVIRON[...] */
Cell    *rstartloc; /* RSTART */
Cell    *rlengthloc;    /* RLENGTH */
Cell    *symtabloc; /* SYMTAB */

Cell    *nullloc;   /* a guaranteed empty cell */
Node    *nullnode;  /* zero&null, converted into a node for comparisons */
Cell    *literal0;

extern Cell **fldtab;

void syminit(void)  /* initialize symbol table with builtin vars */
{
    literal0 = setsymtab("0", "0", 0.0, NUM|STR|CON|DONTFREE, symtab);
    /* this is used for if(x)... tests: */
    nullloc = setsymtab("$zero&null", "", 0.0, NUM|STR|CON|DONTFREE, symtab);
    nullnode = celltonode(nullloc, CCON);

    FS = &setsymtab("FS", " ", 0.0, STR|DONTFREE, symtab)->sval;
    RS = &setsymtab("RS", "\n", 0.0, STR|DONTFREE, symtab)->sval;
    OFS = &setsymtab("OFS", " ", 0.0, STR|DONTFREE, symtab)->sval;
    ORS = &setsymtab("ORS", "\n", 0.0, STR|DONTFREE, symtab)->sval;
    OFMT = &setsymtab("OFMT", "%.6g", 0.0, STR|DONTFREE, symtab)->sval;
    CONVFMT = &setsymtab("CONVFMT", "%.6g", 0.0, STR|DONTFREE, symtab)->sval;
    FILENAME = &setsymtab("FILENAME", "", 0.0, STR|DONTFREE, symtab)->sval;
    nfloc = setsymtab("NF", "", 0.0, NUM, symtab);
    NF = &nfloc->fval;
    nrloc = setsymtab("NR", "", 0.0, NUM, symtab);
    NR = &nrloc->fval;
    fnrloc = setsymtab("FNR", "", 0.0, NUM, symtab);
    FNR = &fnrloc->fval;
    SUBSEP = &setsymtab("SUBSEP", "\034", 0.0, STR|DONTFREE, symtab)->sval;
    rstartloc = setsymtab("RSTART", "", 0.0, NUM, symtab);
    RSTART = &rstartloc->fval;
    rlengthloc = setsymtab("RLENGTH", "", 0.0, NUM, symtab);
    RLENGTH = &rlengthloc->fval;
    symtabloc = setsymtab("SYMTAB", "", 0.0, ARR, symtab);
    symtabloc->sval = (char *) symtab;
}

void arginit(int ac, char **av) /* set up ARGV and ARGC */
{
    Cell *cp;
    int i;
    char temp[50];

    ARGC = &setsymtab("ARGC", "", (Awkfloat) ac, NUM, symtab)->fval;
    cp = setsymtab("ARGV", "", 0.0, ARR, symtab);
    ARGVtab = makesymtab(NSYMTAB);  /* could be (int) ARGC as well */
    cp->sval = (char *) ARGVtab;
```

```
    for (i = 0; i < ac; i++) {
        sprintf(temp, "%d", i);
        if (is_number(*av))
            setsymtab(temp, *av, atof(*av), STR|NUM, ARGVtab);
        else
            setsymtab(temp, *av, 0.0, STR, ARGVtab);
        av++;
    }
}

void envinit(char **envp)   /* set up ENVIRON variable */
{
    Cell *cp;
    char *p;

    cp = setsymtab("ENVIRON", "", 0.0, ARR, symtab);
    ENVtab = makesymtab(NSYMTAB);
    cp->sval = (char *) ENVtab;
    for ( ; *envp; envp++) {
        if ((p = strchr(*envp, '=')) == NULL)
            continue;
        *p++ = 0;   /* split into two strings at = */
        if (is_number(p))
            setsymtab(*envp, p, atof(p), STR|NUM, ENVtab);
        else
            setsymtab(*envp, p, 0.0, STR, ENVtab);
        p[-1] = '=';    /* restore in case env is passed down to a shell */
    }
}

Array *makesymtab(int n)    /* make a new symbol table */
{
    Array *ap;
    Cell **tp;

    ap = (Array *) malloc(sizeof(Array));
    tp = (Cell **) calloc(n, sizeof(Cell *));
    if (ap == NULL || tp == NULL)
        FATAL("out of space in makesymtab");
    ap->nelem = 0;
    ap->size = n;
    ap->tab = tp;
    return(ap);
}

void freesymtab(Cell *ap)   /* free a symbol table */
{
    Cell *cp, *temp;
    Array *tp;
    int i;

    if (!isarr(ap))
        return;
    tp = (Array *) ap->sval;
    if (tp == NULL)
        return;
    for (i = 0; i < tp->size; i++) {
        for (cp = tp->tab[i]; cp != NULL; cp = temp) {
            xfree(cp->nval);
            if (freeable(cp))
```

```
                xfree(cp->sval);
            temp = cp->cnext;   /* avoids freeing then using */
            free(cp);
        }
        tp->tab[i] = 0;
    }
    free(tp->tab);
    free(tp);
}

void freeelem(Cell *ap, char *s)    /* free elem s from ap (i.e., ap["s"] */
{
    Array *tp;
    Cell *p, *prev = NULL;
    int h;

    tp = (Array *) ap->sval;
    h = hash(s, tp->size);
    for (p = tp->tab[h]; p != NULL; prev = p, p = p->cnext)
        if (strcmp(s, p->nval) == 0) {
            if (prev == NULL)   /* 1st one */
                tp->tab[h] = p->cnext;
            else            /* middle somewhere */
                prev->cnext = p->cnext;
            if (freeable(p))
                xfree(p->sval);
            free(p->nval);
            free(p);
            tp->nelem--;
            return;
        }
}

Cell *setsymtab(char *n, char *s, Awkfloat f, unsigned t, Array *tp)
{
    int h;
    Cell *p;

    if (n != NULL && (p = lookup(n, tp)) != NULL) {
           dprintf( ("setsymtab found %p: n=%s s=\"%s\" f=%g t=%o\n",
            p, p->nval, p->sval, p->fval, p->tval) );
        return(p);
    }
    p = (Cell *) malloc(sizeof(Cell));
    if (p == NULL)
        FATAL("out of space for symbol table at %s", n);
    p->nval = tostring(n);
    p->sval = s ? tostring(s) : tostring("");
    p->fval = f;
    p->tval = t;
    p->csub = CUNK;
    p->ctype = OCELL;
    tp->nelem++;
    if (tp->nelem > FULLTAB * tp->size)
        rehash(tp);
    h = hash(n, tp->size);
    p->cnext = tp->tab[h];
    tp->tab[h] = p;
       dprintf( ("setsymtab set %p: n=%s s=\"%s\" f=%g t=%o\n",
        p, p->nval, p->sval, p->fval, p->tval) );
```

```
    return(p);
}

int hash(char *s, int n)    /* form hash value for string s */
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = (*s + 31 * hashval);
    return hashval % n;
}

void rehash(Array *tp)  /* rehash items in small table into big one */
{
    int i, nh, nsz;
    Cell *cp, *op, **np;

    nsz = GROWTAB * tp->size;
    np = (Cell **) calloc(nsz, sizeof(Cell *));
    if (np == NULL)     /* can't do it, but can keep running. */
        return;     /* someone else will run out later. */
    for (i = 0; i < tp->size; i++) {
        for (cp = tp->tab[i]; cp; cp = op) {
            op = cp->cnext;
            nh = hash(cp->nval, nsz);
            cp->cnext = np[nh];
            np[nh] = cp;
        }
    }
    free(tp->tab);
    tp->tab = np;
    tp->size = nsz;
}

Cell *lookup(char *s, Array *tp)    /* look for s in tp */
{
    Cell *p;
    int h;

    h = hash(s, tp->size);
    for (p = tp->tab[h]; p != NULL; p = p->cnext)
        if (strcmp(s, p->nval) == 0)
            return(p);  /* found it */
    return(NULL);           /* not found */
}

Awkfloat setfval(Cell *vp, Awkfloat f)  /* set float val of a Cell */
{
    int fldno;

    if ((vp->tval & (NUM | STR)) == 0)
        funnyvar(vp, "assign to");
    if (isfld(vp)) {
        donerec = 0;    /* mark $0 invalid */
        fldno = atoi(vp->nval);
        if (fldno > *NF)
            newfld(fldno);
           dprintf( ("setting field %d to %g\n", fldno, f) );
    } else if (isrec(vp)) {
        donefld = 0;    /* mark $1... invalid */
```

```
        donerec = 1;
    }
    if (freeable(vp))
        xfree(vp->sval); /* free any previous string */
    vp->tval &= ~STR;   /* mark string invalid */
    vp->tval |= NUM;    /* mark number ok */
       dprintf( ("setfval %p: %s = %g, t=%o\n", vp, vp->nval, f, vp->tval) );
    return vp->fval = f;
}

void funnyvar(Cell *vp, char *rw)
{
    if (isarr(vp))
        FATAL("can't %s %s; it's an array name.", rw, vp->nval);
    if (vp->tval & FCN)
        FATAL("can't %s %s; it's a function.", rw, vp->nval);
    WARNING("funny variable %p: n=%s s=\"%s\" f=%g t=%o",
        vp, vp->nval, vp->sval, vp->fval, vp->tval);
}

char *setsval(Cell *vp, char *s)    /* set string val of a Cell */
{
    char *t;
    int fldno;

       dprintf( ("starting setsval %p: %s = \"%s\", t=%o\n", vp, vp->nval, s, vp->tval) );
    if ((vp->tval & (NUM | STR)) == 0)
        funnyvar(vp, "assign to");
    if (isfld(vp)) {
        donerec = 0;    /* mark $0 invalid */
        fldno = atoi(vp->nval);
        if (fldno > *NF)
            newfld(fldno);
           dprintf( ("setting field %d to %s (%p)\n", fldno, s, s) );
    } else if (isrec(vp)) {
        donefld = 0;    /* mark $1... invalid */
        donerec = 1;
    }
    t = tostring(s);    /* in case it's self-assign */
    vp->tval &= ~NUM;
    vp->tval |= STR;
    if (freeable(vp))
        xfree(vp->sval);
    vp->tval &= ~DONTFREE;
       dprintf( ("setsval %p: %s = \"%s (%p)\", t=%o\n", vp, vp->nval, t,t, vp->tval) );
    return(vp->sval = t);
}

Awkfloat getfval(Cell *vp)  /* get float val of a Cell */
{
    if ((vp->tval & (NUM | STR)) == 0)
        funnyvar(vp, "read value of");
    if (isfld(vp) && donefld == 0)
        fldbld();
    else if (isrec(vp) && donerec == 0)
        recbld();
    if (!isnum(vp)) {   /* not a number */
        vp->fval = atof(vp->sval);  /* best guess */
        if (is_number(vp->sval) && !(vp->tval&CON))
            vp->tval |= NUM;    /* make NUM only sparingly */
```

```
    }
       dprintf( ("getfval %p: %s = %g, t=%o\n", vp, vp->nval, vp->fval, vp->tval) );
    return(vp->fval);
}

char *getsval(Cell *vp) /* get string val of a Cell */
{
    char s[100];    /* BUG: unchecked */
    double dtemp;

    if ((vp->tval & (NUM | STR)) == 0)
        funnyvar(vp, "read value of");
    if (isfld(vp) && donefld == 0)
        fldbld();
    else if (isrec(vp) && donerec == 0)
        recbld();
    if (isstr(vp) == 0) {
        if (freeable(vp))
            xfree(vp->sval);
        if (modf(vp->fval, &dtemp) == 0)    /* it's integral */
            sprintf(s, "%.30g", vp->fval);
        else
            sprintf(s, *CONVFMT, vp->fval);
        vp->sval = tostring(s);
        vp->tval &= ~DONTFREE;
        vp->tval |= STR;
    }
       dprintf( ("getsval %p: %s = \"%s (%p)\", t=%o\n", vp, vp->nval, vp->sval, vp->sval, vp->tval) );
    return(vp->sval);
}

char *tostring(char *s) /* make a copy of string s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1);
    if (p == NULL)
        FATAL("out of space in tostring on %s", s);
    strcpy(p, s);
    return(p);
}

char *qstring(char *s, int delim)   /* collect string up to next delim */
{
    char *os = s;
    int c, n;
    char *buf, *bp;

    if ((buf = (char *) malloc(strlen(s)+3)) == NULL)
        FATAL( "out of space in qstring(%s)", s);
    for (bp = buf; (c = *s) != delim; s++) {
        if (c == '\n')
            SYNTAX( "newline in string %.20s...", os );
        else if (c != '\\')
            *bp++ = c;
        else {  /* \something */
            c = *++s;
            if (c == 0) {   /* \ at end */
                *bp++ = '\\';
                break;  /* for loop */
```

```
            }
            switch (c) {
            case '\\':  *bp++ = '\\'; break;
            case 'n':   *bp++ = '\n'; break;
            case 't':   *bp++ = '\t'; break;
            case 'b':   *bp++ = '\b'; break;
            case 'f':   *bp++ = '\f'; break;
            case 'r':   *bp++ = '\r'; break;
            default:
                if (!isdigit(c)) {
                    *bp++ = c;
                    break;
                }
                n = c - '0';
                if (isdigit(s[1])) {
                    n = 8 * n + *++s - '0';
                    if (isdigit(s[1]))
                        n = 8 * n + *++s - '0';
                }
                *bp++ = n;
                break;
            }
        }
    }
    *bp++ = 0;
    return buf;
}
```

A.9.11 awk/main.c

⟨awk/main.c 415⟩≡

```
⟨awk copyright lucent 343a⟩
char    *version = "version 19990602";

#define DEBUG
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include "awk.h"
#include "y.tab.h"

extern  char    **environ;
extern  int nfields;

int dbg = 0;
char    *cmdname;   /* gets argv[0] for error messages */
extern  FILE    *yyin;  /* lex input file */
char    *lexprog;   /* points to program argument if it exists */
extern  int errorflag;  /* non-zero if any syntax errors; set by yyerror */
int compile_time = 2;   /* for error printing: */
                /* 2 = cmdline, 1 = compile, 0 = running */

char    *pfile[20]; /* program filenames from -f's */
int npfile = 0; /* number of filenames */
int curpfile = 0;   /* current filename */

int safe    = 0;    /* 1 => "safe" mode */
```

⟨*function* `main`*(awk)* 64⟩

```
int pgetc(void)     /* get 1 character from awk program */
{
    int c;

    for (;;) {
        if (yyin == NULL) {
            if (curpfile >= npfile)
                return EOF;
            if (strcmp(pfile[curpfile], "-") == 0)
                yyin = stdin;
            else if ((yyin = fopen(pfile[curpfile], "r")) == NULL)
                FATAL("can't open file %s", pfile[curpfile]);
            lineno = 1;
        }
        if ((c = getc(yyin)) != EOF)
            return c;
        if (yyin != stdin)
            fclose(yyin);
        yyin = NULL;
        curpfile++;
    }
}

char *cursource(void)   /* current source file name */
{
    if (npfile > 0)
        return pfile[curpfile];
    else
        return NULL;
}
```

A.10 archive/

A.10.1 archive/tar.c

⟨globals tar.c 416a⟩+≡ (416b) ◁94n
⟨*global* `verb`*(tar.c)* 86c⟩

⟨archive/tar.c 416b⟩≡

```
/*
 * tar - 'tape archiver', actually usable on any medium.
 *  POSIX "ustar" compliant when extracting, and by default when creating.
 *  this tar attempts to read and write multiple Tblock-byte blocks
 *  at once to and from the filesystem, and does not copy blocks
 *  around internally.
 */
⟨plan9 includes 14⟩
#include <ctype.h>
//#include <fcall.h>      /* for %M */
#include <str.h>

⟨macros TARGxxx(tar.c) 91j⟩

#define HOWMANY(a, size)    (((a) + (size) - 1) / (size))
#define BYTES2TBLKS(bytes)  HOWMANY(bytes, Tblock)
```

```
/* read big-endian binary integers; args must be (uchar *) */
#define G2BEBYTE(x) (((x)[0]<<8)  |  (x)[1])
#define G3BEBYTE(x) (((x)[0]<<16) | ((x)[1]<<8)  |  (x)[2])
#define G4BEBYTE(x) (((x)[0]<<24) | ((x)[1]<<16) | ((x)[2]<<8) | (x)[3])
#define G8BEBYTE(x) (((vlong)G4BEBYTE(x)<<32) | (u32int)G4BEBYTE((x)+4))

typedef vlong Off;
typedef char *(*Refill)(int ar, char *bufs, int justhdr);

// nice enums! tar is modern! types!
enum { Rd, Wr };            /* pipe fd-array indices */
enum { Output, Input };
⟨enum Verb 86b⟩
enum { Alldata, Justnxthdr };
enum {
    ⟨constants tar.c 86f⟩
};

/* POSIX link flags */
⟨enum LinkFlag(tar.c) 87d⟩
⟨macros islinkxxx(tar.c) 88a⟩

⟨union Hdr(tar.c) 87b⟩

⟨struct Compress(tar.c) 89a⟩
⟨constant comps(tar.c) 89b⟩

⟨struct Pushstate(tar.c) 90a⟩

#define OTHER(rdwr) ((rdwr) == Rd? Wr: Rd)

⟨globals tar.c 86e⟩
⟨global flags tar.c 86d⟩
static int debug;
static int fixednblock;
static int keepexisting;
static Off blkoff;      /* offset of the current archive block (not Tblock) */
static Off nexthdr;

static char *arname = "archive";
static char origdir[Maxname*2];
static Hdr *curblk;

⟨function usage(tar.c) 86a⟩

/* I/O, with error retry or exit */

static int
cope(char *name, int fd, void *buf, long len, Off off)
{
    fprint(2, "%s: %serror reading %s: %r\n", argv0,
        (ignerrs? "ignoring ": ""), name);
    if (!ignerrs)
        exits("read error");

    /* pretend we read len bytes of zeroes */
    memset(buf, 0, len);
    if (off >= 0)           /* seekable? */
        seek(fd, off + len, 0);
```

```
    return len;
}

static int
eread(char *name, int fd, void *buf, long len)
{
    int rd;
    Off off;

    off = seek(fd, 0, 1);       /* for coping with errors */
    rd = read(fd, buf, len);
    if (rd < 0)
        rd = cope(name, fd, buf, len, off);
    return rd;
}

static int
ereadn(char *name, int fd, void *buf, long len)
{
    int rd;
    Off off;

    off = seek(fd, 0, 1);
    rd = readn(fd, buf, len);
    if (rd < 0)
        rd = cope(name, fd, buf, len, off);
    return rd;
}

static int
ewrite(char *name, int fd, void *buf, long len)
{
    int rd;

    werrstr("");
    rd = write(fd, buf, len);
    if (rd != len)
        sysfatal("error writing %s: %r", name);
    return rd;
}

/* compression */
```

⟨*function* compmethod*(tar.c)* 89c⟩

```
/*
 * push a filter, cmd, onto fd.  if input, it's an input descriptor.
 * returns a descriptor to replace fd, or -1 on error.
 */
static int
push(int fd, char *cmd, int input, Pushstate *ps)
{
    int nfd, pifds[2];
    String *s;

    ps->open = 0;
    ps->fd = fd;
    ps->input = input;
    if (fd < 0 || pipe(pifds) < 0)
        return -1;
```

```
    ps->kid = fork();
    switch (ps->kid) {
    case -1:
        return -1;
    case 0:
        if (input)
            dup(pifds[Wr], STDOUT);
        else
            dup(pifds[Rd], STDIN);
        close(pifds[input? Rd: Wr]);
        dup(fd, (input? STDIN: STDOUT));
        s = s_new();
        if (cmd[0] != '/')
            s_append(s, "/bin/");
        s_append(s, cmd);
        execl(s_to_c(s), cmd, nil);
        sysfatal("can't exec %s: %r", cmd);
    default:
        nfd = pifds[input? Rd: Wr];
        close(pifds[input? Wr: Rd]);
        break;
    }
    ps->rfd = nfd;
    ps->open = 1;
    return nfd;
}

static char *
pushclose(Pushstate *ps)
{
    Waitmsg *wm;

    if (ps->fd < 0 || ps->rfd < 0 || !ps->open)
        return "not open";
    close(ps->rfd);
    ps->rfd = -1;
    ps->open = 0;
    while ((wm = wait()) != nil && wm->pid != ps->kid)
        continue;
    return wm? wm->msg: nil;
}

/*
 * block-buffer management
 */
```

⟨*function* `initblks`*(tar.c)* 91i⟩

```
/*
 * (re)fill block buffers from archive.  'justhdr' means we don't care
 * about the data before the next header block.
 */
static char *
refill(int ar, char *bufs, int justhdr)
{
    int i, n;
    unsigned bytes = Tblock * nblock;
    static int done, first = 1, seekable;

    if (done)
```

```
        return nil;

    blkoff = seek(ar, 0, 1);        /* note position for ‘tar r’ */
    if (first)
        seekable = blkoff >= 0;
    /* try to size non-pipe input at first read */
    if (first && usefile && !fixednblock) {
        n = eread(arname, ar, bufs, bytes);
        if (n == 0)
            sysfatal("EOF reading archive %s: %r", arname);
        i = n;
        if (i % Tblock != 0)
            sysfatal("%s: archive block size (%d) error", arname, i);
        i /= Tblock;
        if (i != nblock) {
            nblock = i;
            fprint(2, "%s: blocking = %d\n", argv0, nblock);
            endblk = (Hdr *)bufs + nblock;
            bytes = n;
        }
    } else if (justhdr && seekable && nexthdr - blkoff >= bytes) {
        /* optimisation for huge archive members on seekable media */
        if (seek(ar, bytes, 1) < 0)
            sysfatal("can’t seek on archive %s: %r", arname);
        n = bytes;
    } else
        n = ereadn(arname, ar, bufs, bytes);
    first = 0;

    if (n == 0)
        sysfatal("unexpected EOF reading archive %s", arname);
    if (n % Tblock != 0)
        sysfatal("partial block read from archive %s", arname);
    if (n != bytes) {
        done = 1;
        memset(bufs + n, 0, bytes - n);
    }
    return bufs;
}

static Hdr *
getblk(int ar, Refill rfp, int justhdr)
{
    if (curblk == nil || curblk >= endblk) {  /* input block exhausted? */
        if (rfp != nil && (*rfp)(ar, (char *)tpblk, justhdr) == nil)
            return nil;
        curblk = tpblk;
    }
    return curblk++;
}

static Hdr *
getblkrd(int ar, int justhdr)
{
    return getblk(ar, refill, justhdr);
}

static Hdr *
getblke(int ar)
{
```

```
    return getblk(ar, nil, Alldata);
}

static Hdr *
getblkz(int ar)
{
    Hdr *hp = getblke(ar);

    if (hp != nil)
        memset(hp->data, 0, Tblock);
    return hp;
}

/*
 * how many block buffers are available, starting at the address
 * just returned by getblk*?
 */
static int
gothowmany(int max)
{
    int n = endblk - (curblk - 1);

    return n > max? max: n;
}

/*
 * indicate that one is done with the last block obtained from getblke
 * and it is now available to be written into the archive.
 */
static void
putlastblk(int ar)
{
    unsigned bytes = Tblock * nblock;

    /* if writing end-of-archive, aid compression (good hygiene too) */
    if (curblk < endblk)
        memset(curblk, 0, (char *)endblk - (char *)curblk);
    ewrite(arname, ar, tpblk, bytes);
}

static void
putblk(int ar)
{
    if (curblk >= endblk)
        putlastblk(ar);
}

static void
putbackblk(int ar)
{
    curblk--;
    USED(ar);
}

static void
putreadblks(int ar, int blks)
{
    curblk += blks - 1;
    USED(ar);
}
```

```
static void
putblkmany(int ar, int blks)
{
    assert(blks > 0);
    curblk += blks - 1;
    putblk(ar);
}

/*
 * common routines
 */

/*
 * modifies hp->chksum but restores it; important for the last block of the
 * old archive when updating with ‘tar rf archive’
 */
static long
chksum(Hdr *hp)
{
    int n = Tblock;
    long i = 0;
    uchar *cp = hp->data;
    char oldsum[sizeof hp->chksum];

    memmove(oldsum, hp->chksum, sizeof oldsum);
    memset(hp->chksum, ' ', sizeof hp->chksum);
    while (n-- > 0)
        i += *cp++;
    memmove(hp->chksum, oldsum, sizeof oldsum);
    return i;
}
```

⟨*function* `isustar` *(tar.c)* 87c⟩

```
/*
 * s is at most n bytes long, but need not be NUL-terminated.
 * if shorter than n bytes, all bytes after the first NUL must also
 * be NUL.
 */
static int
strnlen_(char *s, int n)
{
    return s[n - 1] != '\0'? n: strlen(s);
}

/* set fullname from header */
static char *
name(Hdr *hp)
{
    int pfxlen, namlen;
    char *fullname;
    static char fullnamebuf[2+Maxname+1];  /* 2+ for ./ on relative names */

    fullname = fullnamebuf+2;
    namlen = strnlen_(hp->name, sizeof hp->name);
    if (hp->prefix[0] == '\0' || !isustar(hp)) {    /* old-style name? */
        memmove(fullname, hp->name, namlen);
        fullname[namlen] = '\0';
        return fullname;
```

```
    }

    /* name is in two pieces */
    pfxlen = strnlen_(hp->prefix, sizeof hp->prefix);
    memmove(fullname, hp->prefix, pfxlen);
    fullname[pfxlen] = '/';
    memmove(fullname + pfxlen + 1, hp->name, namlen);
    fullname[pfxlen + 1 + namlen] = '\0';
    return fullname;
}

static int
isdir(Hdr *hp)
{
    /* the mode test is ugly but sometimes necessary */
    return hp->linkflag == LF_DIR ||
        strrchr(name(hp), '\0')[-1] == '/' ||
        (strtoul(hp->mode, nil, 8)&0170000) == 040000;
}

static int
eotar(Hdr *hp)
{
    return name(hp)[0] == '\0';
}

/*
static uvlong
getbe(uchar *src, int size)
{
    uvlong vl = 0;

    while (size-- > 0) {
        vl <<= 8;
        vl |= *src++;
    }
    return vl;
}
 */

static void
putbe(uchar *dest, uvlong vl, int size)
{
    for (dest += size; size-- > 0; vl >>= 8)
        *--dest = vl;
}

/*
 * cautious parsing of octal numbers as ascii strings in
 * a tar header block.  this is particularly important for
 * trusting the checksum when trying to resync.
 */
static uvlong
hdrotoull(char *st, char *end, uvlong errval, char *name, char *field)
{
    char *numb;

    for (numb = st; (*numb == ' ' || *numb == '\0') && numb < end; numb++)
        ;
    if (numb < end && isascii(*numb) && isdigit(*numb))
```

```
        return strtoull(numb, nil, 8);
    else if (numb >= end)
        fprint(2, "%s: %s: empty %s in header\n", argv0, name, field);
    else
        fprint(2, "%s: %s: %s: non-numeric %s in header\n",
            argv0, name, numb, field);
    return errval;
}

/*
 * return the nominal size from the header block, which is not always the
 * size in the archive (the archive size may be zero for some file types
 * regardless of the nominal size).
 *
 * gnu and freebsd tars are now recording vlongs as big-endian binary
 * with a flag in byte 0 to indicate this, which permits file sizes up to
 * 2^64-1 (actually 2^80-1 but our file sizes are vlongs) rather than 2^33-1.
 */
static Off
hdrsize(Hdr *hp)
{
    uchar *p;

    if((uchar)hp->size[0] == Binnegsz) {
        fprint(2, "%s: %s: negative length, which is insane\n",
            argv0, name(hp));
        return 0;
    } else if((uchar)hp->size[0] == Binsize) {
        p = (uchar *)hp->size + sizeof hp->size - 1 -
            sizeof(vlong);      /* -1 for terminating space */
        return G8BEBYTE(p);
    }

    return hdrotoull(hp->size, hp->size + sizeof hp->size, 0,
        name(hp), "size");
}

/*
 * return the number of bytes recorded in the archive.
 */
static Off
arsize(Hdr *hp)
{
    if(isdir(hp) || islink(hp->linkflag))
        return 0;
    return hdrsize(hp);
}

static long
parsecksum(char *cksum, char *name)
{
    Hdr *hp;

    return hdrotoull(cksum, cksum + sizeof hp->chksum, (uvlong)-1LL,
        name, "checksum");
}
```

⟨*function* `readhdr` *(tar.c)* 88c⟩

```
/*
```

```
 * tar r[c]
 */

/*
 * if name is longer than Namsiz bytes, try to split it at a slash and fit the
 * pieces into hp->prefix and hp->name.
 */
static int
putfullname(Hdr *hp, char *name)
{
    int namlen, pfxlen;
    char *sl, *osl;
    String *slname = nil;

    if (isdir(hp)) {
        slname = s_new();
        s_append(slname, name);
        s_append(slname, "/");      /* posix requires this */
        name = s_to_c(slname);
    }

    namlen = strlen(name);
    if (namlen <= Namsiz) {
        strncpy(hp->name, name, Namsiz);
        hp->prefix[0] = '\0';       /* ustar paranoia */
        return 0;
    }

    if (!posix || namlen > Maxname) {
        fprint(2, "%s: name too long for tar header: %s\n",
            argv0, name);
        return -1;
    }
    /*
     * try various splits until one results in pieces that fit into the
     * appropriate fields of the header.  look for slashes from right
     * to left, in the hopes of putting the largest part of the name into
     * hp->prefix, which is larger than hp->name.
     */
    sl = strrchr(name, '/');
    while (sl != nil) {
        pfxlen = sl - name;
        if (pfxlen <= sizeof hp->prefix && namlen-1 - pfxlen <= Namsiz)
            break;
        osl = sl;
        *osl = '\0';
        sl = strrchr(name, '/');
        *osl = '/';
    }
    if (sl == nil) {
        fprint(2, "%s: name can't be split to fit tar header: %s\n",
            argv0, name);
        return -1;
    }
    *sl = '\0';
    strncpy(hp->prefix, name, sizeof hp->prefix);
    *sl++ = '/';
    strncpy(hp->name, sl, sizeof hp->name);
    if (slname)
        s_free(slname);
```

```
    return 0;
}
```

⟨*function* `mkhdr` *(tar.c)* 88b⟩

```
static void addtoar(int ar, char *file, char *shortf);

static void
addtreetoar(int ar, char *file, char *shortf, int fd)
{
    int n;
    Dir *dent, *dirents;
    String *name = s_new();

    n = dirreadall(fd, &dirents);
    if (n < 0)
        fprint(2, "%s: dirreadall %s: %r\n", argv0, file);
    close(fd);
    if (n <= 0)
        return;

    if (chdir(shortf) < 0)
        sysfatal("chdir %s: %r", file);
    if (Debug)
        fprint(2, "chdir %s\t# %s\n", shortf, file);

    for (dent = dirents; dent < dirents + n; dent++) {
        s_reset(name);
        s_append(name, file);
        s_append(name, "/");
        s_append(name, dent->name);
        addtoar(ar, s_to_c(name), dent->name);
    }
    s_free(name);
    free(dirents);

    /*
     * this assumes that shortf is just one component, which is true
     * during directory descent, but not necessarily true of command-line
     * arguments.  Our caller (or addtoar's) must reset the working
     * directory if necessary.
     */
    if (chdir("..") < 0)
        sysfatal("chdir %s/..: %r", file);
    if (Debug)
        fprint(2, "chdir ..\n");
}

static void
addtoar(int ar, char *file, char *shortf)
{
    int n, fd, isdir;
    long bytes, blksread;
    ulong blksleft;
    Hdr *hbp;
    Dir *dir;
    String *name = nil;

    if (shortf[0] == '#') {
        name = s_new();
```

```
        s_append(name, "./");
        s_append(name, shortf);
        shortf = s_to_c(name);
    }

    if (Debug)
        fprint(2, "opening %s   # %s\n", shortf, file);
    fd = open(shortf, OREAD);
    if (fd < 0) {
        fprint(2, "%s: can't open %s: %r\n", argv0, file);
        if (name)
            s_free(name);
        return;
    }
    dir = dirfstat(fd);
    if (dir == nil)
        sysfatal("can't fstat %s: %r", file);

    hbp = getblkz(ar);
    isdir = (dir->qid.type & QTDIR) != 0;
    if (mkhdr(hbp, dir, file) < 0) {
        putbackblk(ar);
        free(dir);
        close(fd);
        if (name)
            s_free(name);
        return;
    }
    putblk(ar);

    blksleft = BYTES2TBLKS(dir->length);
    free(dir);

    if (isdir)
        addtreetoar(ar, file, shortf, fd);
    else {
        for (; blksleft > 0; blksleft -= blksread) {
            hbp = getblke(ar);
            blksread = gothowmany(blksleft);
            assert(blksread >= 0);
            bytes = blksread * Tblock;
            n = ereadn(file, fd, hbp->data, bytes);
            assert(n >= 0);
            /*
             * ignore EOF.  zero any partial block to aid
             * compression and emergency recovery of data.
             */
            if (n < Tblock)
                memset(hbp->data + n, 0, bytes - n);
            putblkmany(ar, blksread);
        }
        close(fd);
        if (verbose)
            fprint(2, "%s\n", file);
    }
    if (name)
        s_free(name);
}

static void
```

```
skip(int ar, Hdr *hp, char *msg)
{
    ulong blksleft, blksread;
    Off bytes;

    bytes = arsize(hp);
    for (blksleft = BYTES2TBLKS(bytes); blksleft > 0; blksleft -= blksread) {
        if (getblkrd(ar, Justnxthdr) == nil)
            sysfatal("unexpected EOF on archive %s %s", arname, msg);
        blksread = gothowmany(blksleft);
        putreadblks(ar, blksread);
    }
}

static void
skiptoend(int ar)
{
    Hdr *hp;

    while ((hp = readhdr(ar)) != nil)
        skip(ar, hp, "skipping to end");

    /*
     * we have just read the end-of-archive Tblock.
     * now seek back over the (big) archive block containing it,
     * and back up curblk ptr over end-of-archive Tblock in memory.
     */
    if (seek(ar, blkoff, 0) < 0)
        sysfatal("can't seek back over end-of-archive in %s: %r", arname);
    curblk--;
}
```

⟨*function* `replace` *(tar.c)* 92c⟩

```
/*
 * tar [xt]
 */

/* is pfx a file-name prefix of name? */
static int
prefix(char *name, char *pfx)
{
    int pfxlen = strlen(pfx);
    char clpfx[Maxname+1];

    if (pfxlen > Maxname)
        return 0;
    strcpy(clpfx, pfx);
    cleanname(clpfx);
    return strncmp(clpfx, name, pfxlen) == 0 &&
        (name[pfxlen] == '\0' || name[pfxlen] == '/');
}

static int
match(char *name, char **argv)
{
    int i;
    char clname[Maxname+1];

    if (argv[0] == nil)
```

```
        return 1;
    strcpy(clname, name);
    cleanname(clname);
    for (i = 0; argv[i] != nil; i++)
        if (prefix(clname, argv[i]))
            return 1;
    return 0;
}

static void
cantcreate(char *s, int mode)
{
    int len;
    static char *last;

    /*
     * Always print about files.  Only print about directories
     * we haven't printed about.  (Assumes archive is ordered
     * nicely.)
     */
    if(mode&DMDIR){
        if(last){
            /* already printed this directory */
            if(strcmp(s, last) == 0)
                return;
            /* printed a higher directory, so printed this one */
            len = strlen(s);
            if(memcmp(s, last, len) == 0 && last[len] == '/')
                return;
        }
        /* save */
        free(last);
        last = strdup(s);
    }
    fprint(2, "%s: can't create %s: %r\n", argv0, s);
}

static int
makedir(char *s)
{
    int f;

    if (access(s, AEXIST) == 0)
        return -1;
    f = create(s, OREAD, DMDIR | 0777);
    if (f >= 0)
        close(f);
    else
        cantcreate(s, DMDIR);
    return f;
}

static int
mkpdirs(char *s)
{
    int err;
    char *p;

    p = s;
    err = 0;
```

```
    while (!err && (p = strchr(p+1, '/')) != nil) {
        *p = '\0';
        err = (access(s, AEXIST) < 0 && makedir(s) < 0);
        *p = '/';
    }
    return -err;
}

/* Call access but preserve the error string. */
static int
xaccess(char *name, int mode)
{
    char err[ERRMAX];
    int rv;

    err[0] = 0;
    errstr(err, sizeof err);
    rv = access(name, mode);
    errstr(err, sizeof err);
    return rv;
}

static int
openfname(Hdr *hp, char *fname, int dir, int mode)
{
    int fd;

    fd = -1;
    cleanname(fname);
    switch (hp->linkflag) {
    case LF_LINK:
    case LF_SYMLINK1:
    case LF_SYMLINK2:
        fprint(2, "%s: can't make (sym)link %s\n",
            argv0, fname);
        break;
    case LF_FIFO:
        fprint(2, "%s: can't make fifo %s\n", argv0, fname);
        break;
    default:
        if (!keepexisting || access(fname, AEXIST) < 0) {
            int rw = (dir? OREAD: OWRITE);

            fd = create(fname, rw, mode);
            if (fd < 0) {
                mkpdirs(fname);
                fd = create(fname, rw, mode);
            }
            if (fd < 0 && (!dir || xaccess(fname, AEXIST) < 0))
                    cantcreate(fname, mode);
        }
        if (fd >= 0 && verbose)
            fprint(2, "%s\n", fname);
        break;
    }
    return fd;
}

/* copy from archive to file system (or nowhere for table-of-contents) */
static void
```

```
copyfromar(int ar, int fd, char *fname, ulong blksleft, Off bytes)
{
    int wrbytes;
    ulong blksread;
    Hdr *hbp;

    if (blksleft == 0 || bytes < 0)
        bytes = 0;
    for (; blksleft > 0; blksleft -= blksread) {
        hbp = getblkrd(ar, (fd >= 0? Alldata: Justnxthdr));
        if (hbp == nil)
            sysfatal("unexpected EOF on archive extracting %s from %s",
                fname, arname);
        blksread = gothowmany(blksleft);
        if (blksread <= 0) {
            fprint(2, "%s: got %ld blocks reading %s!\n",
                argv0, blksread, fname);
            blksread = 0;
        }
        wrbytes = Tblock*blksread;
        assert(bytes >= 0);
        if(wrbytes > bytes)
            wrbytes = bytes;
        assert(wrbytes >= 0);
        if (fd >= 0)
            ewrite(fname, fd, hbp->data, wrbytes);
        putreadblks(ar, blksread);
        bytes -= wrbytes;
        assert(bytes >= 0);
    }
    if (bytes > 0)
        fprint(2, "%s: %lld bytes uncopied at EOF on archive %s; "
            "%s not fully extracted\n", argv0, bytes, arname, fname);
}

static void
wrmeta(int fd, Hdr *hp, long mtime, int mode)       /* update metadata */
{
    Dir nd;

    nulldir(&nd);
    nd.mtime = mtime;
    nd.mode = mode;
    dirfwstat(fd, &nd);
    if (isustar(hp)) {
        nulldir(&nd);
        nd.gid = hp->gname;
        dirfwstat(fd, &nd);
        nulldir(&nd);
        nd.uid = hp->uname;
        dirfwstat(fd, &nd);
    }
}

/*
 * copy a file from the archive into the filesystem.
 * fname is result of name(), so has two extra bytes at beginning.
 */
static void
extract1(int ar, Hdr *hp, char *fname)
```

```
{
    int fd = -1, dir = 0;
    long mtime = strtol(hp->mtime, nil, 8);
    ulong mode = strtoul(hp->mode, nil, 8) & 0777;
    Off bytes = hdrsize(hp);        /* for printing */
    ulong blksleft = BYTES2TBLKS(arsize(hp));

    /* fiddle name, figure out mode and blocks */
    if (isdir(hp)) {
        mode |= DMDIR|0700;
        dir = 1;
    }
    switch (hp->linkflag) {
    case LF_LINK:
    case LF_SYMLINK1:
    case LF_SYMLINK2:
    case LF_FIFO:
        blksleft = 0;
        break;
    }
    if (relative)
        if(fname[0] == '/')
            *--fname = '.';
        else if(fname[0] == '#'){
            *--fname = '/';
            *--fname = '.';
        }

    if (verb == Xtract)
        fd = openfname(hp, fname, dir, mode);
    else if (verbose) {
        char *cp = ctime(mtime);

        print("%M %8lld %-12.12s %-4.4s %s\n",
            mode, bytes, cp+4, cp+24, fname);
    } else
        print("%s\n", fname);

    copyfromar(ar, fd, fname, blksleft, bytes);

    /* touch up meta data and close */
    if (fd >= 0) {
        /*
         * directories should be wstated *after* we're done
         * creating files in them, but we don't do that.
         */
        if (settime)
            wrmeta(fd, hp, mtime, mode);
        close(fd);
    }
}
```

⟨*function* `extract` *(tar.c)* 92b⟩

⟨*function* `main` *(tar.c)* 90b⟩

A.10.2 archive/gzip/gzip.h

⟨`archive/gzip/gzip.h` 432⟩≡

```
/*
 * gzip header fields
 */
enum
{
    ⟨gzip constants 95g⟩
    ⟨gzip flags 97c⟩
    GZXFAST     = 2,            /* used fast algorithm, little compression */
    GZXBEST     = 4,            /* used maximum compression algorithm */
    ⟨gzip fs type 97d⟩
};
```

A.10.3 archive/gzip/gunzip.c

⟨archive/gzip/gunzip.c 433a⟩≡

```
⟨plan9 includes 14⟩
#include <bio.h>
#include <flate.h>
#include "gzip.h"

typedef struct  GZHead  GZHead;
⟨struct GZHead 100f⟩

// forward decls
static  int crcwrite(void *bout, void *buf, int n);
static  int     get1(Biobuf *b);
static  ulong   get4(Biobuf *b);
static  int gunzipf(char *file, bool stdout);
static  int gunzip(fdt ofd, char *ofile, Biobuf *bin);
static  void    header(Biobuf *bin, GZHead *h);
static  void    trailer(Biobuf *bin, long wlen);
static  void    error(char*, ...);

#pragma varargck    argpos  error   1

⟨globals flags gunzip.c 99e⟩
⟨globals gunzip.c 100a⟩

⟨function usage(gunzip.c) 99d⟩
⟨function main(gunzip.c) 100g⟩

⟨function gunzipf 101b⟩
⟨function gunzip 103a⟩

⟨function header(gunzip.c) 104⟩
⟨function trailer(gunzip.c) 105a⟩

⟨function get4(gunzip.c) 105c⟩
⟨function get1(gunzip.c) 105b⟩
⟨function crcwrite(gunzip.c) 103b⟩

⟨function error(gunzip.c) 100d⟩
```

A.10.4 archive/gzip.c

⟨archive/gzip/gzip.c 433b⟩≡

```
⟨plan9 includes 14⟩
#include <bio.h>
```

```
#include <flate.h>
#include "gzip.h"

// forward decls
static  error0 gzipf(char*, bool);
static  int gzip(char*, long, int, Biobuf*);
static  int crcread(void *fd, void *buf, int n);
static  int gzwrite(void *bout, void *buf, int n);
```

⟨*global flags gzip.c* 95d⟩
⟨*globals gzip.c* 95e⟩

⟨*function* `usage` *(gzip.c)* 95c⟩
⟨*function* `main` *(gzip.c)* 95h⟩

⟨*function* `gzipf` 96⟩
⟨*function* `gzip` 98b⟩

⟨*function* `crcread` *(gzip.c)* 99b⟩
⟨*function* `gzwrite` *(gzip.c)* 99c⟩

A.10.5 archive/zip/zip.h

⟨zip/zip.h 434⟩≡

```
typedef struct ZipHead  ZipHead;

enum
{
    /*
     * magic numbers
     */
    ZHeader     = 0x04034b50,
    ZCHeader    = 0x02014b50,
    ZECHeader   = 0x06054b50,

    /*
     * "general purpose flag" bits
     */
    ZEncrypted  = 1 << 0,
    ZTrailInfo  = 1 << 3,   /* uncsize, csize, and crc are in trailer */
    ZCompPatch  = 1 << 5,   /* compression patched data */

    ZCrcPoly    = 0xedb88320,

    /*
     * compression method
     */
    ZDeflate    = 8,

    /*
     * internal file attributes
     */
    ZIsText     = 1 << 0,

    /*
     * file attribute interpretation, from high byte of version
     */
    ZDos        = 0,
    ZAmiga      = 1,
```

```
    ZVMS        = 2,
    ZUnix       = 3,
    ZVMCMS      = 4,
    ZAtariST    = 5,
    ZOS2HPFS    = 6,
    ZMac        = 7,
    ZZsys       = 8,
    ZCPM        = 9,
    ZNtfs       = 10,

    /*
     * external attribute flags for ZDos
     */
    ZDROnly     = 0x01,
    ZDHidden    = 0x02,
    ZDSystem    = 0x04,
    ZDVLable    = 0x08,
    ZDDir       = 0x10,
    ZDArch      = 0x20,

    ZHeadSize   = 4 + 2 + 2 + 2 + 2 + 2 + 4 + 4 + 4 + 2 + 2,
    ZHeadCrc    = 4 + 2 + 2 + 2 + 2 + 2,
    ZTrailSize  = 4 + 4 + 4,
    ZCHeadSize  = 4 + 2 + 2 + 2 + 2 + 2 + 2 + 4 + 4 + 4 + 2 + 2 + 2 + 2 + 2 + 4 + 4,
    ZECHeadSize = 4 + 2 + 2 + 2 + 2 + 4 + 4 + 2,
};

/*
 * interesting info from a zip header
 */
struct ZipHead
{
    int madeos;         /* version made by */
    int madevers;
    int extos;          /* version needed to extract */
    int extvers;
    int flags;          /* general purpose bit flag */
    int meth;
    int modtime;
    int moddate;
    ulong   crc;
    ulong   csize;
    ulong   uncsize;
    int iattr;
    ulong   eattr;
    ulong   off;
    char    *file;
};
```

A.10.6 archive/zip/zip.c

⟨zip/zip.c 435⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <flate.h>
#include "zip.h"

enum
```

```
{
    HeadAlloc   = 64,
};

static  void    zip(Biobuf *bout, char *file, int stdout);
static  void    zipDir(Biobuf *bout, int fd, ZipHead *zh, int stdout);
static  int crcread(void *fd, void *buf, int n);
static  int zwrite(void *bout, void *buf, int n);
static  void    put4(Biobuf *b, ulong v);
static  void    put2(Biobuf *b, int v);
static  void    put1(Biobuf *b, int v);
static  void    header(Biobuf *bout, ZipHead *zh);
static  void    trailer(Biobuf *bout, ZipHead *zh, vlong off);
static  void    putCDir(Biobuf *bout);

static  void    error(char*, ...);
#pragma varargck    argpos  error   1

static  Biobuf  bout;
static  ulong   crc;
static  ulong   *crctab;
static  int debug;
static  int eof;
static  int level;
static  int nzheads;
static  ulong   totr;
static  ulong   totw;
static  int verbose;
static  int zhalloc;
static  ZipHead *zheads;
static  jmp_buf zjmp;

void
usage(void)
{
    fprint(2, "usage: zip [-vD] [-1-9] [-f zipfile] file ...\n");
    exits("usage");
}

void
main(int argc, char *argv[])
{
    char *zfile;
    int i, fd, err;

    zfile = nil;
    level = 6;
    ARGBEGIN{
    case 'D':
        debug++;
        break;
    case 'f':
        zfile = ARGF();
        if(zfile == nil)
            usage();
        break;
    case 'v':
        verbose++;
        break;
    case '1': case '2': case '3': case '4':
```

```
    case '5': case '6': case '7': case '8': case '9':
        level = ARGC() - '0';
        break;
    default:
        usage();
        break;
    }ARGEND

    if(argc == 0)
        usage();

    crctab = mkcrctab(ZCrcPoly);
    err = deflateinit();
    if(err != FlateOk)
        sysfatal("deflateinit failed: %s", flateerr(err));

    if(zfile == nil)
        fd = 1;
    else{
        fd = create(zfile, OWRITE, 0664);
        if(fd < 0)
            sysfatal("can't create %s: %r", zfile);
    }
    Binit(&bout, fd, OWRITE);

    if(setjmp(zjmp)){
        if(zfile != nil){
            fprint(2, "zip: removing output file %s\n", zfile);
            remove(zfile);
        }
        exits("errors");
    }

    for(i = 0; i < argc; i++)
        zip(&bout, argv[i], zfile == nil);

    putCDir(&bout);

    exits(nil);
}

static void
zip(Biobuf *bout, char *file, int stdout)
{
    Tm *t;
    ZipHead *zh;
    Dir *dir;
    vlong off;
    int fd, err;

    fd = open(file, OREAD);
    if(fd < 0)
        error("can't open %s: %r", file);
    dir = dirfstat(fd);
    if(dir == nil)
        error("can't stat %s: %r", file);

    /*
     * create the header
     */
```

```
    if(nzheads >= zhalloc){
        zhalloc += HeadAlloc;
        zheads = realloc(zheads, zhalloc * sizeof(ZipHead));
        if(zheads == nil)
            error("out of memory");
    }
    zh = &zheads[nzheads++];
    zh->madeos = ZDos;
    zh->madevers = (2 * 10) + 0;
    zh->extos = ZDos;
    zh->extvers = (2 * 10) + 0;

    t = localtime(dir->mtime);
    zh->modtime = (t->hour<<11) | (t->min<<5) | (t->sec>>1);
    zh->moddate = ((t->year-80)<<9) | ((t->mon+1)<<5) | t->mday;

    zh->flags = 0;
    zh->crc = 0;
    zh->csize = 0;
    zh->uncsize = 0;
    zh->file = strdup(file);
    if(zh->file == nil)
        error("out of memory");
    zh->iattr = 0;
    zh->eattr = ZDArch;
    if((dir->mode & 0700) == 0)
        zh->eattr |= ZDROnly;
    zh->off = Boffset(bout);

    if(dir->mode & DMDIR){
        zh->eattr |= ZDDir;
        zh->meth = 0;
        zipDir(bout, fd, zh, stdout);
    }else{
        zh->meth = 8;
        if(stdout)
            zh->flags |= ZTrailInfo;
        off = Boffset(bout);
        header(bout, zh);

        crc = 0;
        eof = 0;
        totr = 0;
        totw = 0;
        err = deflate(bout, zwrite, (void*)fd, crcread, level, debug);
        if(err != FlateOk)
            error("deflate failed: %s: %r", flateerr(err));

        zh->csize = totw;
        zh->uncsize = totr;
        zh->crc = crc;
        trailer(bout, zh, off);
    }
    close(fd);
    free(dir);
}

static void
zipDir(Biobuf *bout, int fd, ZipHead *zh, int stdout)
{
```

```
    Dir *dirs;
    char *file, *pfile;
    int i, nf, nd;

    nf = strlen(zh->file) + 1;
    if(strcmp(zh->file, ".") == 0){
        nzheads--;
        free(zh->file);
        pfile = "";
        nf = 1;
    }else{
        nf++;
        pfile = malloc(nf);
        if(pfile == nil)
            error("out of memory");
        snprint(pfile, nf, "%s/", zh->file);
        free(zh->file);
        zh->file = pfile;
        header(bout, zh);
    }

    nf += 256;  /* plenty of room */
    file = malloc(nf);
    if(file == nil)
        error("out of memory");
    while((nd = dirread(fd, &dirs)) > 0){
        for(i = 0; i < nd; i++){
            snprint(file, nf, "%s%s", pfile, dirs[i].name);
            zip(bout, file, stdout);
        }
        free(dirs);
    }
}

static void
header(Biobuf *bout, ZipHead *zh)
{
    int flen;

    if(verbose)
        fprint(2, "adding %s\n", zh->file);
    put4(bout, ZHeader);
    put1(bout, zh->extvers);
    put1(bout, zh->extos);
    put2(bout, zh->flags);
    put2(bout, zh->meth);
    put2(bout, zh->modtime);
    put2(bout, zh->moddate);
    put4(bout, zh->crc);
    put4(bout, zh->csize);
    put4(bout, zh->uncsize);
    flen = strlen(zh->file);
    put2(bout, flen);
    put2(bout, 0);
    if(Bwrite(bout, zh->file, flen) != flen)
        error("write error");
}

static void
trailer(Biobuf *bout, ZipHead *zh, vlong off)
```

```
{
    vlong coff;

    coff = -1;
    if(!(zh->flags & ZTrailInfo)){
        coff = Boffset(bout);
        if(Bseek(bout, off + ZHeadCrc, 0) < 0)
            error("can't seek in archive");
    }
    put4(bout, zh->crc);
    put4(bout, zh->csize);
    put4(bout, zh->uncsize);
    if(!(zh->flags & ZTrailInfo)){
        if(Bseek(bout, coff, 0) < 0)
            error("can't seek in archive");
    }
}

static void
cheader(Biobuf *bout, ZipHead *zh)
{
    int flen;

    put4(bout, ZCHeader);
    put1(bout, zh->madevers);
    put1(bout, zh->madeos);
    put1(bout, zh->extvers);
    put1(bout, zh->extos);
    put2(bout, zh->flags & ~ZTrailInfo);
    put2(bout, zh->meth);
    put2(bout, zh->modtime);
    put2(bout, zh->moddate);
    put4(bout, zh->crc);
    put4(bout, zh->csize);
    put4(bout, zh->uncsize);
    flen = strlen(zh->file);
    put2(bout, flen);
    put2(bout, 0);
    put2(bout, 0);
    put2(bout, 0);
    put2(bout, zh->iattr);
    put4(bout, zh->eattr);
    put4(bout, zh->off);
    if(Bwrite(bout, zh->file, flen) != flen)
        error("write error");
}

static void
putCDir(Biobuf *bout)
{
    vlong hoff, ecoff;
    int i;

    hoff = Boffset(bout);

    for(i = 0; i < nzheads; i++)
        cheader(bout, &zheads[i]);

    ecoff = Boffset(bout);
```

```
    if(nzheads >= (1 << 16))
        error("too many entries in zip file: max %d", (1 << 16) - 1);
    put4(bout, ZECHeader);
    put2(bout, 0);
    put2(bout, 0);
    put2(bout, nzheads);
    put2(bout, nzheads);
    put4(bout, ecoff - hoff);
    put4(bout, hoff);
    put2(bout, 0);
}

static int
crcread(void *fd, void *buf, int n)
{
    int nr, m;

    nr = 0;
    for(; !eof && n > 0; n -= m){
        m = read((int)(uintptr)fd, (char*)buf+nr, n);
        if(m <= 0){
            eof = 1;
            if(m < 0)
{
fprint(2, "input error %r\n");
                return -1;
}
            break;
        }
        nr += m;
    }
    crc = blockcrc(crctab, crc, buf, nr);
    totr += nr;
    return nr;
}

static int
zwrite(void *bout, void *buf, int n)
{
    if(n != Bwrite(bout, buf, n)){
        eof = 1;
        return -1;
    }
    totw += n;
    return n;
}

static void
put4(Biobuf *b, ulong v)
{
    int i;

    for(i = 0; i < 4; i++){
        if(Bputc(b, v) < 0)
            error("write error");
        v >>= 8;
    }
}

static void
```

```
put2(Biobuf *b, int v)
{
    int i;

    for(i = 0; i < 2; i++){
        if(Bputc(b, v) < 0)
            error("write error");
        v >>= 8;
    }
}

static void
put1(Biobuf *b, int v)
{
    if(Bputc(b, v)< 0)
        error("unexpected eof reading file information");
}

static void
error(char *fmt, ...)
{
    va_list arg;

    fprint(2, "zip: ");
    va_start(arg, fmt);
    vfprint(2, fmt, arg);
    va_end(arg);
    fprint(2, "\n");

    longjmp(zjmp, 1);
}
```

A.10.7 archive/zip/unzip.c

⟨zip/unzip.c 442⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <flate.h>
#include "zip.h"

enum
{
    BufSize = 4096
};

static  int cheader(Biobuf *bin, ZipHead *zh);
static  int copyout(int ofd, Biobuf *bin, long len);
static  int crcwrite(void *ofd, void *buf, int n);
static  int findCDir(Biobuf *bin, char *file);
static  int get1(Biobuf *b);
static  int get2(Biobuf *b);
static  ulong   get4(Biobuf *b);
static  char    *getname(Biobuf *b, int len);
static  int header(Biobuf *bin, ZipHead *zh);
static  long    msdos2time(int time, int date);
static  int sunzip(Biobuf *bin);
static  int sunztable(Biobuf *bin);
static  void    trailer(Biobuf *bin, ZipHead *zh);
```

```
static  int unzip(Biobuf *bin, char *file);
static  int unzipEntry(Biobuf *bin, ZipHead *czh);
static  int unztable(Biobuf *bin, char *file);
static  int wantFile(char *file);

static  void    *emalloc(ulong);
static  void    error(char*, ...);
#pragma varargck    argpos  error   1

static  Biobuf  bin;
static  ulong   crc;
static  ulong   *crctab;
static  int debug;
static  char    *delfile;
static  int lower;
static  int nwant;
static  ulong   rlen;
static  int settimes;
static  int stdout;
static  int verbose;
static  char    **want;
static  int wbad;
static  ulong   wlen;
static  jmp_buf zjmp;
static  jmp_buf seekjmp;
static  int autodir;

static void
usage(void)
{
    fprint(2, "usage: unzip [-tsv] [-f zipfile] [file ...]\n");
    exits("usage");
}

void
main(int argc, char *argv[])
{
    char *zfile;
    int fd, ok, table, stream;

    table = 0;
    stream = 0;
    zfile = nil;
    ARGBEGIN{
    case 'a':
        autodir++;
        break;
    case 'D':
        debug++;
        break;
    case 'c':
        stdout++;
        break;
    case 'i':
        lower++;
        break;
    case 'f':
        zfile = ARGF();
        if(zfile == nil)
            usage();
```

```
        break;
    case 's':
        stream++;
        break;
    case 't':
        table++;
        break;
    case 'T':
        settimes++;
        break;
    case 'v':
        verbose++;
        break;
    default:
        usage();
        break;
    }ARGEND

    nwant = argc;
    want = argv;

    crctab = mkcrctab(ZCrcPoly);
    ok = inflateinit();
    if(ok != FlateOk)
        sysfatal("inflateinit failed: %s", flateerr(ok));

    if(zfile == nil){
        Binit(&bin, 0, OREAD);
        zfile = "<stdin>";
    }else{
        fd = open(zfile, OREAD);
        if(fd < 0)
            sysfatal("can't open %s: %r", zfile);
        Binit(&bin, fd, OREAD);
    }

    if(setjmp(seekjmp)){
        fprint(2, "trying to re-run assuming -s\n");
        stream = 1;
        Bseek(&bin, 0, 0);
    }

    if(table){
        if(stream)
            ok = sunztable(&bin);
        else
            ok = unztable(&bin, zfile);
    }else{
        if(stream)
            ok = sunzip(&bin);
        else
            ok = unzip(&bin, zfile);
    }

    exits(ok ? nil: "errors");
}

/*
 * print the table of contents from the "central directory structure"
 */
```

```
static int
unztable(Biobuf *bin, char *file)
{
    ZipHead zh;
    int entries;

    entries = findCDir(bin, file);
    if(entries < 0)
        return 0;

    if(verbose > 1)
        print("%d items in the archive\n", entries);
    while(entries-- > 0){
        if(setjmp(zjmp)){
            free(zh.file);
            return 0;
        }

        memset(&zh, 0, sizeof(zh));
        if(!cheader(bin, &zh))
            return 1;

        if(wantFile(zh.file)){
            if(verbose)
                print("%-32s %10lud %s", zh.file, zh.uncsize, ctime(msdos2time(zh.modtime, zh.moddate)));
            else
                print("%s\n", zh.file);

            if(verbose > 1){
                print("\tmade by os %d vers %d.%d\n", zh.madeos, zh.madevers/10, zh.madevers % 10);
                print("\textract by os %d vers %d.%d\n", zh.extos, zh.extvers/10, zh.extvers % 10);
                print("\tflags %x\n", zh.flags);
                print("\tmethod %d\n", zh.meth);
                print("\tmod time %d\n", zh.modtime);
                print("\tmod date %d\n", zh.moddate);
                print("\tcrc %lux\n", zh.crc);
                print("\tcompressed size %lud\n", zh.csize);
                print("\tuncompressed size %lud\n", zh.uncsize);
                print("\tinternal attributes %ux\n", zh.iattr);
                print("\texternal attributes %lux\n", zh.eattr);
                print("\tstarts at %ld\n", zh.off);
            }
        }

        free(zh.file);
        zh.file = nil;
    }

    return 1;
}

/*
 * print the "local file header" table of contents
 */
static int
sunztable(Biobuf *bin)
{
    ZipHead zh;
    vlong off;
    ulong hcrc, hcsize, huncsize;
```

```
int ok, err;

ok = 1;
for(;;){
    if(setjmp(zjmp)){
        free(zh.file);
        return 0;
    }

    memset(&zh, 0, sizeof(zh));
    if(!header(bin, &zh))
        return ok;

    hcrc = zh.crc;
    hcsize = zh.csize;
    huncsize = zh.uncsize;

    wlen = 0;
    rlen = 0;
    crc = 0;
    wbad = 0;

    if(zh.meth == 0){
        if(!copyout(-1, bin, zh.csize))
            error("reading data for %s failed: %r", zh.file);
    }else if(zh.meth == 8){
        off = Boffset(bin);
        err = inflate((void*)-1, crcwrite, bin, (int(*)(void*))Bgetc);
        if(err != FlateOk)
            error("inflate %s failed: %s", zh.file, flateerr(err));
        rlen = Boffset(bin) - off;
    }else
        error("can't handle compression method %d for %s", zh.meth, zh.file);

    trailer(bin, &zh);

    if(wantFile(zh.file)){
        if(verbose)
            print("%-32s %10lud %s", zh.file, zh.uncsize, ctime(msdos2time(zh.modtime, zh.moddate)));
        else
            print("%s\n", zh.file);

        if(verbose > 1){
            print("\textract by os %d vers %d.%d\n", zh.extos, zh.extvers / 10, zh.extvers % 10);
            print("\tflags %x\n", zh.flags);
            print("\tmethod %d\n", zh.meth);
            print("\tmod time %d\n", zh.modtime);
            print("\tmod date %d\n", zh.moddate);
            print("\tcrc %lux\n", zh.crc);
            print("\tcompressed size %lud\n", zh.csize);
            print("\tuncompressed size %lud\n", zh.uncsize);
            if((zh.flags & ZTrailInfo) && (hcrc || hcsize || huncsize)){
                print("\theader crc %lux\n", zh.crc);
                print("\theader compressed size %lud\n", zh.csize);
                print("\theader uncompressed size %lud\n", zh.uncsize);
            }
        }
    }

    if(zh.crc != crc)
```

```
            error("crc mismatch for %s", zh.file);
        if(zh.uncsize != wlen)
            error("output size mismatch for %s", zh.file);
        if(zh.csize != rlen)
            error("input size mismatch for %s", zh.file);


        free(zh.file);
        zh.file = nil;
    }
}

/*
 * extract files using the info in the central directory structure
 */
static int
unzip(Biobuf *bin, char *file)
{
    ZipHead zh;
    vlong off;
    int ok, eok, entries;

    entries = findCDir(bin, file);
    if(entries < 0)
        return 0;

    ok = 1;
    while(entries-- > 0){
        if(setjmp(zjmp)){
            free(zh.file);
            return 0;
        }
        memset(&zh, 0, sizeof(zh));
        if(!cheader(bin, &zh))
            return ok;


        off = Boffset(bin);
        if(wantFile(zh.file)){
            if(Bseek(bin, zh.off, 0) < 0){
                fprint(2, "unzip: can't seek to start of %s, skipping\n", zh.file);
                ok = 0;
            }else{
                eok = unzipEntry(bin, &zh);
                if(eok <= 0){
                    fprint(2, "unzip: skipping %s\n", zh.file);
                    ok = 0;
                }
            }
        }

        free(zh.file);
        zh.file = nil;

        if(Bseek(bin, off, 0) < 0){
            fprint(2, "unzip: can't seek to start of next entry, terminating extraction\n");
            return 0;
        }
    }
```

```
    return ok;
}

/*
 * extract files using the info the "local file headers"
 */
static int
sunzip(Biobuf *bin)
{
    int eok;

    for(;;){
        eok = unzipEntry(bin, nil);
        if(eok == 0)
            return 1;
        if(eok < 0)
            return 0;
    }
}

static int mkdirs(char *);

/*
 * if any directories leading up to path don't exist, create them.
 * modifies but restores path.
 */
static int
mkpdirs(char *path)
{
    int rv = 0;
    char *sl = strrchr(path, '/');
print("%s\n", path);
    if (sl != nil) {
        *sl = '\0';
        rv = mkdirs(path);
        *sl = '/';
    }
    return rv;
}

/*
 * if path or any directories leading up to it don't exist, create them.
 * modifies but restores path.
 */
static int
mkdirs(char *path)
{
    int fd;

    if (access(path, AEXIST) >= 0)
        return 0;

    /* make presumed-missing intermediate directories */
    if (mkpdirs(path) < 0)
        return -1;

    /* make final directory */
    fd = create(path, OREAD, 0755|DMDIR);
    if (fd < 0)
        /*
```

```
         * we may have lost a race; if the directory now exists,
         * it's okay.
         */
        return access(path, AEXIST) < 0? -1: 0;
    close(fd);
    return 0;
}


/*
 * extracts a single entry from a zip file
 * czh is the optional corresponding central directory entry
 */
static int
unzipEntry(Biobuf *bin, ZipHead *czh)
{
    Dir *d;
    ZipHead zh;
    char *p;
    vlong off;
    int fd, isdir, ok, err;

    zh.file = nil;
    if(setjmp(zjmp)){
        delfile = nil;
        free(zh.file);
        return -1;
    }

    memset(&zh, 0, sizeof(zh));
    if(!header(bin, &zh))
        return 0;

    ok = 1;
    isdir = 0;

    fd = -1;
    if(wantFile(zh.file)){
        if(verbose)
            fprint(2, "extracting %s\n", zh.file);

        if(czh != nil && czh->extos == ZDos){
            isdir = czh->eattr & ZDDir;
            if(isdir && zh.uncsize != 0)
                fprint(2, "unzip: ignoring directory data for %s\n", zh.file);
        }
        if(zh.meth == 0 && zh.uncsize == 0){
            p = strchr(zh.file, '\0');
            if(p > zh.file && p[-1] == '/')
                isdir = 1;
        }

        if(stdout){
            if(ok && !isdir)
                fd = 1;
        }else if(isdir){
            fd = create(zh.file, OREAD, DMDIR | 0775);
            if(fd < 0){
                d = dirstat(zh.file);
                if(d == nil || (d->mode & DMDIR) != DMDIR){
```

```
                fprint(2, "unzip: can't create directory %s: %r\n", zh.file);
                ok = 0;
            }
            free(d);
        }
    }else if(ok){
        if(autodir)
            mkpdirs(zh.file);
        fd = create(zh.file, OWRITE, 0664);
        if(fd < 0){
            fprint(2, "unzip: can't create %s: %r\n", zh.file);
            ok = 0;
        }else
            delfile = zh.file;
    }
}

wlen = 0;
rlen = 0;
crc = 0;
wbad = 0;

if(zh.meth == 0){
    if(!copyout(fd, bin, zh.csize))
        error("copying data for %s failed: %r", zh.file);
}else if(zh.meth == 8){
    off = Boffset(bin);
    err = inflate((void*)fd, crcwrite, bin, (int(*)(void*))Bgetc);
    if(err != FlateOk)
        error("inflate failed: %s", flateerr(err));
    rlen = Boffset(bin) - off;
}else
    error("can't handle compression method %d for %s", zh.meth, zh.file);

trailer(bin, &zh);

if(zh.crc != crc)
    error("crc mismatch for %s", zh.file);
if(zh.uncsize != wlen)
    error("output size mismatch for %s", zh.file);
if(zh.csize != rlen)
    error("input size mismatch for %s", zh.file);

delfile = nil;
free(zh.file);

if(fd >= 0 && !stdout){
    if(settimes){
        d = dirfstat(fd);
        if(d != nil){
            d->mtime = msdos2time(zh.modtime, zh.moddate);
            if(d->mtime)
                dirfwstat(fd, d);
        }
    }
    close(fd);
}

return ok;
}
```

```
static int
wantFile(char *file)
{
    int i, n;

    if(nwant == 0)
        return 1;
    for(i = 0; i < nwant; i++){
        if(strcmp(want[i], file) == 0)
            return 1;
        n = strlen(want[i]);
        if(strncmp(want[i], file, n) == 0 && file[n] == '/')
            return 1;
    }
    return 0;
}

/*
 * find the start of the central directory
 * returns the number of entries in the directory,
 * or -1 if there was an error
 */
static int
findCDir(Biobuf *bin, char *file)
{
    vlong ecoff;
    long off, size, m;
    int entries, zclen, dn, ds, de;

    ecoff = Bseek(bin, -ZECHeadSize, 2);
    if(ecoff < 0){
        fprint(2, "unzip: can't seek to contents of %s\n", file);
        longjmp(seekjmp, 1);
        return -1;
    }
    if(setjmp(zjmp))
        return -1;

    if((m=get4(bin)) != ZECHeader){
        fprint(2, "unzip: bad magic number for table of contents of %s: %#.8lx\n", file, m);
        longjmp(seekjmp, 1);
        return -1;
    }
    dn = get2(bin);
    ds = get2(bin);
    de = get2(bin);
    entries = get2(bin);
    size = get4(bin);
    off = get4(bin);
    zclen = get2(bin);
    while(zclen-- > 0)
        get1(bin);

    if(verbose > 1){
        print("table starts at %ld for %ld bytes\n", off, size);
        if(ecoff - size != off)
            print("\ttable should start at %lld-%ld=%lld\n", ecoff, size, ecoff-size);
        if(dn || ds || de != entries)
            print("\tcurrent disk=%d start disk=%d table entries on this disk=%d\n", dn, ds, de);
```

```
    }

    if(Bseek(bin, off, 0) != off){
        fprint(2, "unzip: can't seek to start of contents of %s\n", file);
        longjmp(seekjmp, 1);
        return -1;
    }

    return entries;
}

static int
cheader(Biobuf *bin, ZipHead *zh)
{
    ulong v;
    int flen, xlen, fclen;

    v = get4(bin);
    if(v != ZCHeader){
        if(v == ZECHeader)
            return 0;
        error("bad magic number %lux", v);
    }
    zh->madevers = get1(bin);
    zh->madeos = get1(bin);
    zh->extvers = get1(bin);
    zh->extos = get1(bin);
    zh->flags = get2(bin);
    zh->meth = get2(bin);
    zh->modtime = get2(bin);
    zh->moddate = get2(bin);
    zh->crc = get4(bin);
    zh->csize = get4(bin);
    zh->uncsize = get4(bin);
    flen = get2(bin);
    xlen = get2(bin);
    fclen = get2(bin);
    get2(bin);      /* disk number start */
    zh->iattr = get2(bin);
    zh->eattr = get4(bin);
    zh->off = get4(bin);

    zh->file = getname(bin, flen);

    while(xlen-- > 0)
        get1(bin);

    while(fclen-- > 0)
        get1(bin);

    return 1;
}

static int
header(Biobuf *bin, ZipHead *zh)
{
    ulong v;
    int flen, xlen;

    v = get4(bin);
```

```
    if(v != ZHeader){
        if(v == ZCHeader)
            return 0;
        error("bad magic number %lux at %lld", v, Boffset(bin)-4);
    }
    zh->extvers = get1(bin);
    zh->extos = get1(bin);
    zh->flags = get2(bin);
    zh->meth = get2(bin);
    zh->modtime = get2(bin);
    zh->moddate = get2(bin);
    zh->crc = get4(bin);
    zh->csize = get4(bin);
    zh->uncsize = get4(bin);
    flen = get2(bin);
    xlen = get2(bin);

    zh->file = getname(bin, flen);

    while(xlen-- > 0)
        get1(bin);

    return 1;
}

static void
trailer(Biobuf *bin, ZipHead *zh)
{
    if(zh->flags & ZTrailInfo){
        zh->crc = get4(bin);
        zh->csize = get4(bin);
        zh->uncsize = get4(bin);
    }
}

static char*
getname(Biobuf *bin, int len)
{
    char *s;
    int i, c;

    s = emalloc(len + 1);
    for(i = 0; i < len; i++){
        c = get1(bin);
        if(lower)
            c = tolower(c);
        s[i] = c;
    }
    s[i] = '\0';
    return s;
}

static int
crcwrite(void *out, void *buf, int n)
{
    int fd, nw;

    wlen += n;
    crc = blockcrc(crctab, crc, buf, n);
    fd = (int)(uintptr)out;
```

```
    if(fd < 0)
        return n;
    nw = write(fd, buf, n);
    if(nw != n)
        wbad = 1;
    return nw;
}

static int
copyout(int ofd, Biobuf *bin, long len)
{
    char buf[BufSize];
    int n;

    for(; len > 0; len -= n){
        n = len;
        if(n > BufSize)
            n = BufSize;
        n = Bread(bin, buf, n);
        if(n <= 0)
            return 0;
        rlen += n;
        if(crcwrite((void*)ofd, buf, n) != n)
            return 0;
    }
    return 1;
}

static ulong
get4(Biobuf *b)
{
    ulong v;
    int i, c;

    v = 0;
    for(i = 0; i < 4; i++){
        c = Bgetc(b);
        if(c < 0)
            error("unexpected eof reading file information");
        v |= c << (i * 8);
    }
    return v;
}

static int
get2(Biobuf *b)
{
    int i, c, v;

    v = 0;
    for(i = 0; i < 2; i++){
        c = Bgetc(b);
        if(c < 0)
            error("unexpected eof reading file information");
        v |= c << (i * 8);
    }
    return v;
}

static int
```

```
get1(Biobuf *b)
{
    int c;

    c = Bgetc(b);
    if(c < 0)
        error("unexpected eof reading file information");
    return c;
}

static long
msdos2time(int time, int date)
{
    Tm tm;

    tm.hour = time >> 11;
    tm.min = (time >> 5) & 63;
    tm.sec = (time & 31) << 1;
    tm.year = 80 + (date >> 9);
    tm.mon = ((date >> 5) & 15) - 1;
    tm.mday = date & 31;
    tm.zone[0] = '\0';
    tm.yday = 0;

    return tm2sec(&tm);
}

static void*
emalloc(ulong n)
{
    void *p;

    p = malloc(n);
    if(p == nil)
        sysfatal("out of memory");
    return p;
}

static void
error(char *fmt, ...)
{
    va_list arg;

    fprint(2, "unzip: ");
    va_start(arg, fmt);
    vfprint(2, fmt, arg);
    va_end(arg);
    fprint(2, "\n");

    if(delfile != nil){
        fprint(2, "unzip: removing output file %s\n", delfile);
        remove(delfile);
        delfile = nil;
    }

    longjmp(zjmp, 1);
}
```

A.11 calc/

A.11.1 calc/dc.c

⟨macros dc.c 456a⟩≡ (457)
```
#define OUTC(x)     {Bputc(&bout,x); if(--count == 0){Bprint(&bout,"\\\n"); count=ll;} }
#define TEST2       {if((count -= 2) <=0){Bprint(&bout,"\\\n");count=ll;}}
#define EMPTY       if(stkerr != 0){Bprint(&bout,"stack empty\n"); continue; }
#define EMPTYR(x)   if(stkerr != 0){pushp(x);Bprint(&bout,"stack empty\n");continue;}
#define EMPTYS      if(stkerr != 0){Bprint(&bout,"stack empty\n"); return(1);}
#define EMPTYSR(x)  if(stkerr != 0){Bprint(&bout,"stack empty\n");pushp(x);return(1);}
#define error(p)    {Bprint(&bout,p); continue; }
#define errorrt(p)  {Bprint(&bout,p); return(1); }
```

⟨constants dc.c 456b⟩≡ (457)
```
#define HEADSZ 1024
#define STKSZ 100
#define RDSKSZ 100
#define ARRAYST 221
#define MAXIND 2048
```

⟨globals dc.c 456c⟩+≡ (457) ◁135f
```
Biobuf *fsave;
Blk *arg1, *arg2;
uchar   savk;
int ifile;
Blk *scalptr, *basptr, *tenptr, *inbas;
Blk *sqtemp, *chptr, *strptr, *divxyz;
Blk *stack[STKSZ];
Blk **stkptr,**stkbeg;
Blk **stkend;
Blk *hfree;
int stkerr;
int lastchar;
Blk *readstk[RDSKSZ];
Blk **readptr;
Blk *rem;
int k;
Blk *irem;
int skd,skr;
int neg;
Sym symlst[TBLSZ];
Sym *stable[TBLSZ];
Sym *sptr, *sfree;
long    rel;
long    nbytes;
long    all;
long    headmor;
long    obase;
int fw,fw1,ll;
void    (*outdit)(Blk *p, int flg);
int logo;
int logten;
int count;
char    *pp;
char    *dummy;
long    longest, maxsize, active;
int lall, lrel, lcopy, lmore, lbytes;
int inside;
```

⟨calc/dc.c 457⟩≡

```
⟨plan9 includes 14⟩
#include <bio.h>

typedef void*   pointer;
#pragma varargck    type    "lx"    pointer

#define FATAL 0
#define NFATAL 1
#define BLK sizeof(Blk)
#define PTRSZ sizeof(int*)
#define TBLSZ 256           /* 1<<BI2BY */

⟨constants dc.c 456b⟩

#define NL 1
#define NG 2
#define NE 3

⟨macros on Blk dc.c 134b⟩
⟨macros dc.c 456a⟩

#define LASTFUN 026

typedef struct  Blk Blk;
⟨struct Blk(dc.c) 134a⟩
typedef struct  Sym Sym;
⟨struct Sym(dc.c) 135b⟩
typedef struct  Wblk    Wblk;
⟨struct Wblk(dc.c) 135a⟩

⟨global flags dc.c 135e⟩

⟨globals dc.c 135c⟩

// forward decls
void    main(int argc, char *argv[]);
void    commnds(void);
Blk*    readin(void);
Blk*    div_(Blk *ddivd, Blk *ddivr);
int     dscale(void);
Blk*    removr(Blk *p, int n);
Blk*    dcsqrt(Blk *p);
void    init(int argc, char *argv[]);
void    onintr(void);
void    pushp(Blk *p);
Blk*    pop(void);
Blk*    readin(void);
Blk*    add0(Blk *p, int ct);
Blk*    mult(Blk *p, Blk *q);
void    chsign(Blk *p);
int     readc(void);
void    unreadc(char c);
void    binop(char c);
void    dcprint(Blk *hptr);
Blk*    dcexp(Blk *base, Blk *ex);
Blk*    getdec(Blk *p, int sc);
void    tenot(Blk *p, int sc);
void    oneot(Blk *p, int sc, char ch);
void    hexot(Blk *p, int flg);
```

```
void    bigot(Blk *p, int flg);
Blk*    add(Blk *a1, Blk *a2);
int     eqk(void);
Blk*    removc(Blk *p, int n);
Blk*    scalint(Blk *p);
Blk*    scale(Blk *p, int n);
int     subt(void);
int     command(void);
int     cond(char c);
void    load(void);
int     log2_(long n);
Blk*    salloc(int size);
Blk*    morehd(void);
Blk*    copy(Blk *hptr, int size);
void    sdump(char *s1, Blk *hptr);
void    seekc(Blk *hptr, int n);
void    salterwd(Blk *hptr, Blk *n);
void    more(Blk *hptr);
void    ospace(char *s);
void    garbage(char *s);
void    release(Blk *p);
Blk*    dcgetwd(Blk *p);
void    putwd(Blk *p, Blk *c);
Blk*    lookwd(Blk *p);
int     getstk(void);

/********debug only**/
void
tpr(char *cp, Blk *bp)
{
    print("%s-> ", cp);
    print("beg: %lx rd: %lx wt: %lx last: %lx\n", bp->beg, bp->rd,
        bp->wt, bp->last);
    for (cp = bp->beg; cp != bp->wt; cp++) {
        print("%d", *cp);
        if (cp != bp->wt-1)
            print("/");
    }
    print("\n");
}
/************/
```

⟨*function* `main`*(dc.c)* 135d⟩

⟨*function* `commnds`*(dc.c)* 137c⟩

```
Blk*
div_(Blk *ddivd, Blk *ddivr)
{
    int divsign, remsign, offset, divcarry,
        carry, dig, magic, d, dd, under, first;
    long c, td, cc;
    Blk *ps, *px, *p, *divd, *divr;

    dig = 0;
    under = 0;
    divcarry = 0;
    rem = 0;
    p = salloc(0);
    if(length(ddivr) == 0) {
```

```
    pushp(ddivr);
    Bprint(&bout,"divide by 0\n");
    return(p);
}
divsign = remsign = first = 0;
divr = ddivr;
fsfile(divr);
if(sbackc(divr) == -1) {
    divr = copy(ddivr,length(ddivr));
    chsign(divr);
    divsign = ~divsign;
}
divd = copy(ddivd,length(ddivd));
fsfile(divd);
if(sfbeg(divd) == 0 && sbackc(divd) == -1) {
    chsign(divd);
    divsign = ~divsign;
    remsign = ~remsign;
}
offset = length(divd) - length(divr);
if(offset < 0)
    goto ddone;
seekc(p,offset+1);
sputc(divd,0);
magic = 0;
fsfile(divr);
c = sbackc(divr);
if(c < 10)
    magic++;
c = c * 100 + (sfbeg(divr)?0:sbackc(divr));
if(magic>0){
    c = (c * 100 +(sfbeg(divr)?0:sbackc(divr)))*2;
    c /= 25;
}
while(offset >= 0) {
    first++;
    fsfile(divd);
    td = sbackc(divd) * 100;
    dd = sfbeg(divd)?0:sbackc(divd);
    td = (td + dd) * 100;
    dd = sfbeg(divd)?0:sbackc(divd);
    td = td + dd;
    cc = c;
    if(offset == 0)
        td++;
    else
        cc++;
    if(magic != 0)
        td = td<<3;
    dig = td/cc;
    under=0;
    if(td%cc < 8  && dig > 0 && magic) {
        dig--;
        under=1;
    }
    rewind(divr);
    rewind(divxyz);
    carry = 0;
    while(sfeof(divr) == 0) {
        d = sgetc(divr)*dig+carry;
```

```
        carry = d / 100;
        salterc(divxyz,d%100);
    }
    salterc(divxyz,carry);
    rewind(divxyz);
    seekc(divd,offset);
    carry = 0;
    while(sfeof(divd) == 0) {
        d = slookc(divd);
        d = d-(sfeof(divxyz)?0:sgetc(divxyz))-carry;
        carry = 0;
        if(d < 0) {
            d += 100;
            carry = 1;
        }
        salterc(divd,d);
    }
    divcarry = carry;
    backc(p);
    salterc(p,dig);
    backc(p);
    fsfile(divd);
    d=sbackc(divd);
    if((d != 0) && /*!divcarry*/ (offset != 0)) {
        d = sbackc(divd) + 100;
        salterc(divd,d);
    }
    if(--offset >= 0)
        divd->wt--;
}
if(under) { /* undershot last - adjust*/
    px = copy(divr,length(divr));   /*11/88 don't corrupt ddivr*/
    chsign(px);
    ps = add(px,divd);
    fsfile(ps);
    if(length(ps) > 0 && sbackc(ps) < 0) {
        release(ps);    /*only adjust in really undershot*/
    } else {
        release(divd);
        salterc(p, dig+1);
        divd=ps;
    }
}
if(divcarry != 0) {
    salterc(p,dig-1);
    salterc(divd,-1);
    ps = add(divr,divd);
    release(divd);
    divd = ps;
}

rewind(p);
divcarry = 0;
while(sfeof(p) == 0){
    d = slookc(p)+divcarry;
    divcarry = 0;
    if(d >= 100){
        d -= 100;
        divcarry = 1;
    }
```

```
        salterc(p,d);
    }
    if(divcarry != 0)salterc(p,divcarry);
    fsfile(p);
    while(sfbeg(p) == 0) {
        if(sbackc(p) != 0)
            break;
        truncate(p);
    }
    if(divsign < 0)
        chsign(p);
    fsfile(divd);
    while(sfbeg(divd) == 0) {
        if(sbackc(divd) != 0)
            break;
        truncate(divd);
    }
ddone:
    if(remsign<0)
        chsign(divd);
    if(divr != ddivr)
        release(divr);
    rem = divd;
    return(p);
}

int
dscale(void)
{
    Blk *dd, *dr, *r;
    int c;

    dr = pop();
    EMPTYS;
    dd = pop();
    EMPTYSR(dr);
    fsfile(dd);
    skd = sunputc(dd);
    fsfile(dr);
    skr = sunputc(dr);
    if(sfbeg(dr) == 1 || (sfbeg(dr) == 0 && sbackc(dr) == 0)) {
        sputc(dr,skr);
        pushp(dr);
        Bprint(&bout,"divide by 0\n");
        return(1);
    }
    if(sfbeg(dd) == 1 || (sfbeg(dd) == 0 && sbackc(dd) == 0)) {
        sputc(dd,skd);
        pushp(dd);
        return(1);
    }
    c = k-skd+skr;
    if(c < 0)
        r = removr(dd,-c);
    else {
        r = add0(dd,c);
        irem = 0;
    }
    arg1 = r;
    arg2 = dr;
```

```
    savk = k;
    return(0);
}

Blk*
removr(Blk *p, int n)
{
    int nn, neg;
    Blk *q, *s, *r;

    fsfile(p);
    neg = sbackc(p);
    if(neg < 0)
        chsign(p);
    rewind(p);
    nn = (n+1)/2;
    q = salloc(nn);
    while(n>1) {
        sputc(q,sgetc(p));
        n -= 2;
    }
    r = salloc(2);
    while(sfeof(p) == 0)
        sputc(r,sgetc(p));
    release(p);
    if(n == 1){
        s = div_(r,tenptr);
        release(r);
        rewind(rem);
        if(sfeof(rem) == 0)
            sputc(q,sgetc(rem));
        release(rem);
        if(neg < 0){
            chsign(s);
            chsign(q);
            irem = q;
            return(s);
        }
        irem = q;
        return(s);
    }
    if(neg < 0) {
        chsign(r);
        chsign(q);
        irem = q;
        return(r);
    }
    irem = q;
    return(r);
}

Blk*
dcsqrt(Blk *p)
{
    Blk *t, *r, *q, *s;
    int c, n, nn;

    n = length(p);
    fsfile(p);
    c = sbackc(p);
```

```
    if((n&1) != 1)
        c = c*100+(sfbeg(p)?0:sbackc(p));
    n = (n+1)>>1;
    r = salloc(n);
    zero(r);
    seekc(r,n);
    nn=1;
    while((c -= nn)>=0)
        nn+=2;
    c=(nn+1)>>1;
    fsfile(r);
    backc(r);
    if(c>=100) {
        c -= 100;
        salterc(r,c);
        sputc(r,1);
    } else
        salterc(r,c);
    for(;;){
        q = div_(p,r);
        s = add(q,r);
        release(q);
        release(rem);
        q = div_(s,sqtemp);
        release(s);
        release(rem);
        s = copy(r,length(r));
        chsign(s);
        t = add(s,q);
        release(s);
        fsfile(t);
        nn = sfbeg(t)?0:sbackc(t);
        if(nn>=0)
            break;
        release(r);
        release(t);
        r = q;
    }
    release(t);
    release(q);
    release(p);
    return(r);
}

Blk*
dcexp(Blk *base, Blk *ex)
{
    Blk *r, *e, *p, *e1, *t, *cp;
    int temp, c, n;

    r = salloc(1);
    sputc(r,1);
    p = copy(base,length(base));
    e = copy(ex,length(ex));
    fsfile(e);
    if(sfbeg(e) != 0)
        goto edone;
    temp=0;
    c = sbackc(e);
    if(c<0) {
```

```
        temp++;
        chsign(e);
    }
    while(length(e) != 0) {
        e1=div_(e,sqtemp);
        release(e);
        e = e1;
        n = length(rem);
        release(rem);
        if(n != 0) {
            e1=mult(p,r);
            release(r);
            r = e1;
        }
        t = copy(p,length(p));
        cp = mult(p,t);
        release(p);
        release(t);
        p = cp;
    }
    if(temp != 0) {
        if((c = length(base)) == 0) {
            goto edone;
        }
        if(c>1)
            create_(r);
        else {
            rewind(base);
            if((c = sgetc(base))<=1) {
                create_(r);
                sputc(r,c);
            } else
                create_(r);
        }
    }
edone:
    release(p);
    release(e);
    return(r);
}
```

⟨*function* `init`*(dc.c)* 135g⟩

⟨*function* `pushp`*(dc.c)* 139a⟩
⟨*function* `pop`*(dc.c)* 139b⟩

⟨*function* `readin`*(dc.c)* 138b⟩

```
/*
 * returns pointer to struct with ct 0's & p
 */
Blk*
add0(Blk *p, int ct)
{
    Blk *q, *t;

    q = salloc(length(p)+(ct+1)/2);
    while(ct>1) {
        sputc(q,0);
```

```
        ct -= 2;
    }
    rewind(p);
    while(sfeof(p) == 0) {
        sputc(q,sgetc(p));
    }
    release(p);
    if(ct == 1) {
        t = mult(tenptr,q);
        release(q);
        return(t);
    }
    return(q);
}

Blk*
mult(Blk *p, Blk *q)
{
    Blk *mp, *mq, *mr;
    int sign, offset, carry;
    int cq, cp, mt, mcr;

    offset = sign = 0;
    fsfile(p);
    mp = p;
    if(sfbeg(p) == 0) {
        if(sbackc(p)<0) {
            mp = copy(p,length(p));
            chsign(mp);
            sign = ~sign;
        }
    }
    fsfile(q);
    mq = q;
    if(sfbeg(q) == 0){
        if(sbackc(q)<0) {
            mq = copy(q,length(q));
            chsign(mq);
            sign = ~sign;
        }
    }
    mr = salloc(length(mp)+length(mq));
    zero(mr);
    rewind(mq);
    while(sfeof(mq) == 0) {
        cq = sgetc(mq);
        rewind(mp);
        rewind(mr);
        mr->rd += offset;
        carry=0;
        while(sfeof(mp) == 0) {
            cp = sgetc(mp);
            mcr = sfeof(mr)?0:slookc(mr);
            mt = cp*cq + carry + mcr;
            carry = mt/100;
            salterc(mr,mt%100);
        }
        offset++;
        if(carry != 0) {
            mcr = sfeof(mr)?0:slookc(mr);
```

```
            salterc(mr,mcr+carry);
        }
    }
    if(sign < 0) {
        chsign(mr);
    }
    if(mp != p)
        release(mp);
    if(mq != q)
        release(mq);
    return(mr);
}

void
chsign(Blk *p)
{
    int carry;
    char ct;

    carry=0;
    rewind(p);
    while(sfeof(p) == 0) {
        ct=100-slookc(p)-carry;
        carry=1;
        if(ct>=100) {
            ct -= 100;
            carry=0;
        }
        salterc(p,ct);
    }
    if(carry != 0) {
        sputc(p,-1);
        fsfile(p);
        backc(p);
        ct = sbackc(p);
        if(ct == 99 /*&& !sfbeg(p)*/) {
            truncate(p);
            sputc(p,-1);
        }
    } else{
        fsfile(p);
        ct = sbackc(p);
        if(ct == 0)
            truncate(p);
    }
    return;
}
```

⟨*function* `readc` *(dc.c)* 137d⟩
⟨*function* `unreadc` *(dc.c)* 138a⟩

⟨*function* `binop` *(dc.c)* 142b⟩

⟨*function* `dcprint` *(dc.c)* 145e⟩

```
Blk*
getdec(Blk *p, int sc)
{
    int cc;
    Blk *q, *t, *s;
```

```
    rewind(p);
    if(length(p)*2 < sc) {
        q = copy(p,length(p));
        return(q);
    }
    q = salloc(length(p));
    while(sc >= 1) {
        sputc(q,sgetc(p));
        sc -= 2;
    }
    if(sc != 0) {
        t = mult(q,tenptr);
        s = salloc(cc = length(q));
        release(q);
        rewind(t);
        while(cc-- > 0)
            sputc(s,sgetc(t));
        sputc(s,0);
        release(t);
        t = div_(s,tenptr);
        release(s);
        release(rem);
        return(t);
    }
    return(q);
}

void
tenot(Blk *p, int sc)
{
    int c, f;

    fsfile(p);
    f=0;
    while((sfbeg(p) == 0) && ((p->rd-p->beg-1)*2 >= sc)) {
        c = sbackc(p);
        if((c<10) && (f == 1))
            Bprint(&bout,"0%d",c);
        else
            Bprint(&bout,"%d",c);
        f=1;
        TEST2;
    }
    if(sc == 0) {
        Bprint(&bout,"\n");
        release(p);
        return;
    }
    if((p->rd-p->beg)*2 > sc) {
        c = sbackc(p);
        Bprint(&bout,"%d.",c/10);
        TEST2;
        OUTC(c%10 +'0');
        sc--;
    } else {
        OUTC('.');
    }
    while(sc>(p->rd-p->beg)*2) {
        OUTC('0');
```

```
        sc--;
    }
    while(sc > 1) {
        c = sbackc(p);
        if(c<10)
            Bprint(&bout,"0%d",c);
        else
            Bprint(&bout,"%d",c);
        sc -= 2;
        TEST2;
    }
    if(sc == 1) {
        OUTC(sbackc(p)/10 +'0');
    }
    Bprint(&bout,"\n");
    release(p);
}

void
oneot(Blk *p, int sc, char ch)
{
    Blk *q;

    q = removc(p,sc);
    create_(strptr);
    sputc(strptr,-1);
    while(length(q)>0) {
        p = add(strptr,q);
        release(q);
        q = p;
        OUTC(ch);
    }
    release(q);
    Bprint(&bout,"\n");
}

void
hexot(Blk *p, int flg)
{
    int c;

    USED(flg);
    rewind(p);
    if(sfeof(p) != 0) {
        sputc(strptr,'0');
        release(p);
        return;
    }
    c = sgetc(p);
    release(p);
    if(c >= 16) {
        Bprint(&bout,"hex digit > 16");
        return;
    }
    sputc(strptr,c<10?c+'0':c-10+'a');
}

void
bigot(Blk *p, int flg)
{
```

```
    Blk *t, *q;
    int neg, l;

    if(flg == 1) {
        t = salloc(0);
        l = 0;
    } else {
        t = strptr;
        l = length(strptr)+fw-1;
    }
    neg=0;
    if(length(p) != 0) {
        fsfile(p);
        if(sbackc(p)<0) {
            neg=1;
            chsign(p);
        }
        while(length(p) != 0) {
            q = div_(p,tenptr);
            release(p);
            p = q;
            rewind(rem);
            sputc(t,sfeof(rem)?'0':sgetc(rem)+'0');
            release(rem);
        }
    }
    release(p);
    if(flg == 1) {
        l = fw1-length(t);
        if(neg != 0) {
            l--;
            sputc(strptr,'-');
        }
        fsfile(t);
        while(l-- > 0)
            sputc(strptr,'0');
        while(sfbeg(t) == 0)
            sputc(strptr,sbackc(t));
        release(t);
    } else {
        l -= length(strptr);
        while(l-- > 0)
            sputc(strptr,'0');
        if(neg != 0) {
            sclobber(strptr);
            sputc(strptr,'-');
        }
    }
    sputc(strptr,' ');
}

Blk*
add(Blk *a1, Blk *a2)
{
    Blk *p;
    int carry, n, size, c, n1, n2;

    size = length(a1)>length(a2)?length(a1):length(a2);
    p = salloc(size);
    rewind(a1);
```

```
    rewind(a2);
    carry=0;
    while(--size >= 0) {
        n1 = sfeof(a1)?0:sgetc(a1);
        n2 = sfeof(a2)?0:sgetc(a2);
        n = n1 + n2 + carry;
        if(n>=100) {
            carry=1;
            n -= 100;
        } else
        if(n<0) {
            carry = -1;
            n += 100;
        } else
            carry = 0;
        sputc(p,n);
    }
    if(carry != 0)
        sputc(p,carry);
    fsfile(p);
    if(sfbeg(p) == 0) {
        c = 0;
        while(sfbeg(p) == 0 && (c = sbackc(p)) == 0)
            ;
        if(c != 0)
            salterc(p,c);
        truncate(p);
    }
    fsfile(p);
    if(sfbeg(p) == 0 && sbackc(p) == -1) {
        while((c = sbackc(p)) == 99) {
            if(c == -1)
                break;
        }
        skipc(p);
        salterc(p,-1);
        truncate(p);
    }
    return(p);
}

int
eqk(void)
{
    Blk *p, *q;
    int skp, skq;

    p = pop();
    EMPTYS;
    q = pop();
    EMPTYSR(p);
    skp = sunputc(p);
    skq = sunputc(q);
    if(skp == skq) {
        arg1=p;
        arg2=q;
        savk = skp;
        return(0);
    }
    if(skp < skq) {
```

```
        savk = skq;
        p = add0(p,skq-skp);
    } else {
        savk = skp;
        q = add0(q,skp-skq);
    }
    arg1=p;
    arg2=q;
    return(0);
}

Blk*
removc(Blk *p, int n)
{
    Blk *q, *r;

    rewind(p);
    while(n>1) {
        skipc(p);
        n -= 2;
    }
    q = salloc(2);
    while(sfeof(p) == 0)
        sputc(q,sgetc(p));
    if(n == 1) {
        r = div_(q,tenptr);
        release(q);
        release(rem);
        q = r;
    }
    release(p);
    return(q);
}

Blk*
scalint(Blk *p)
{
    int n;

    n = sunputc(p);
    p = removc(p,n);
    return(p);
}

Blk*
scale(Blk *p, int n)
{
    Blk *q, *s, *t;

    t = add0(p,n);
    q = salloc(1);
    sputc(q,n);
    s = dcexp(inbas,q);
    release(q);
    q = div_(t,s);
    release(t);
    release(s);
    release(rem);
    sputc(q,n);
    return(q);
```

```
}
```

⟨*function* `subt` *(dc.c)* 142d⟩

⟨*function* `command` *(dc.c)* 155c⟩

```
int
cond(char c)
{
    Blk *p;
    int cc;

    if(subt() != 0)
        return(1);
    p = pop();
    sclobber(p);
    if(length(p) == 0) {
        release(p);
        if(c == '<' || c == '>' || c == NE) {
            getstk();
            return(0);
        }
        load();
        return(1);
    }
    if(c == '='){
        release(p);
        getstk();
        return(0);
    }
    if(c == NE) {
        release(p);
        load();
        return(1);
    }
    fsfile(p);
    cc = sbackc(p);
    release(p);
    if((cc<0 && (c == '<' || c == NG)) ||
       (cc >0) && (c == '>' || c == NL)) {
        getstk();
        return(0);
    }
    load();
    return(1);
}

void
load(void)
{
    int c;
    Blk *p, *q, *t, *s;

    c = getstk() & 0377;
    sptr = stable[c];
    if(sptr != 0) {
        p = sptr->val;
        if(c >= ARRAYST) {
            q = salloc(length(p));
            rewind(p);
```

```
            while(sfeof(p) == 0) {
                s = dcgetwd(p);
                if(s == 0) {
                    putwd(q, (Blk*)0);
                } else {
                    t = copy(s,length(s));
                    putwd(q,t);
                }
            }
            pushp(q);
        } else {
            q = copy(p,length(p));
            pushp(q);
        }
    } else {
        q = salloc(1);
        if(c <= LASTFUN) {
            Bprint(&bout,"function %c undefined\n",c+'a'-1);
            sputc(q,'c');
            sputc(q,'0');
            sputc(q,' ');
            sputc(q,'1');
            sputc(q,'Q');
        }
        else
            sputc(q,0);
        pushp(q);
    }
}

int
log2_(long n)
{
    int i;

    if(n == 0)
        return(0);
    i=31;
    if(n<0)
        return(i);
    while((n <<= 1) > 0)
        i--;
    return i-1;
}
```

⟨*function* salloc*(dc.c)* 139c⟩
⟨*function* morehd*(dc.c)* 140a⟩
⟨*function* copy*(dc.c)* 140b⟩

⟨*function* sdump*(dc.c)* 147b⟩

```
void
seekc(Blk *hptr, int n)
{
    char *nn,*p;

    nn = hptr->beg+n;
    if(nn > hptr->last) {
        nbytes += nn - hptr->last;
        if(nbytes > maxsize)
```

```
            maxsize = nbytes;
        lbytes += nn - hptr->last;
        if(n > longest)
            longest = n;
/*      free(hptr->beg); /**/
        p = realloc(hptr->beg, n);
        if(p == 0) {
/*          hptr->beg = realloc(hptr->beg, hptr->last-hptr->beg);
**          garbage("seekc");
**          if((p = realloc(hptr->beg, n)) == 0)
*/              ospace("seekc");
        }
        hptr->beg = p;
        hptr->wt = hptr->last = hptr->rd = p+n;
        return;
    }
    hptr->rd = nn;
    if(nn>hptr->wt)
        hptr->wt = nn;
}

void
salterwd(Blk *ahptr, Blk *n)
{
    Wblk *hptr;

    hptr = (Wblk*)ahptr;
    if(hptr->rdw == hptr->lastw)
        more(ahptr);
    *hptr->rdw++ = n;
    if(hptr->rdw > hptr->wtw)
        hptr->wtw = hptr->rdw;
}

void
more(Blk *hptr)
{
    unsigned size;
    char *p;

    if((size=(hptr->last-hptr->beg)*2) == 0)
        size=2;
    nbytes += size/2;
    if(nbytes > maxsize)
        maxsize = nbytes;
    if(size > longest)
        longest = size;
    lbytes += size/2;
    lmore++;
/*  free(hptr->beg);/**/
    p = realloc(hptr->beg, size);

    if(p == 0) {
/*      hptr->beg = realloc(hptr->beg, (hptr->last-hptr->beg));
**      garbage("more");
**      if((p = realloc(hptr->beg,size)) == 0)
*/          ospace("more");
    }
    hptr->rd = p + (hptr->rd - hptr->beg);
    hptr->wt = p + (hptr->wt - hptr->beg);
```

```
    hptr->beg = p;
    hptr->last = p+size;
}
```

⟨*function* `ospace` *(dc.c)* 141a⟩
⟨*function* `garbage` *(dc.c)* 141b⟩
⟨*function* `release` *(dc.c)* 141c⟩

```
Blk*
dcgetwd(Blk *p)
{
    Wblk *wp;

    wp = (Wblk*)p;
    if(wp->rdw == wp->wtw)
        return(0);
    return(*wp->rdw++);
}

void
putwd(Blk *p, Blk *c)
{
    Wblk *wp;

    wp = (Wblk*)p;
    if(wp->wtw == wp->lastw)
        more(p);
    *wp->wtw++ = c;
}

Blk*
lookwd(Blk *p)
{
    Wblk *wp;

    wp = (Wblk*)p;
    if(wp->rdw == wp->wtw)
        return(0);
    return(*wp->rdw);
}

int
getstk(void)
{
    int n;
    uchar c;

    c = readc();
    if(c != '<')
        return c;
    n = 0;
    while(1) {
        c = readc();
        if(c == '>')
            break;
        n = n*10+c-'0';
    }
    return n;
}
```

A.11.2 calc/bc.y

⟨*prelude bc.y* 476a⟩≡ (476b)

```
⟨plan9 includes 14⟩
#include        <bio.h>

#define bsp_max 5000

⟨globals bc.y 156⟩
⟨global flags bc.y 157a⟩

// forward decls
char*   bundle(int, ...);
void    conout(char*, char*);
int     cpeek(int, int, int);
int     getch(void);
char*   geta(char*);
char*   getf(char*);
void    getout(void);
void    output(char*);
void    pp(char*);
void    routput(char*);
void    tp(char*);
void    yyerror(char*, ...);
int     yyparse(void);

typedef void*   pointer;
#pragma varargck        type    "lx"    pointer
```

⟨*calc/bc.y* 476b⟩≡

```
%{
        ⟨prelude bc.y 476a⟩
%}
⟨union directive bc.y 158b⟩

⟨type directives bc.y 162a⟩
⟨token directives bc.y 159a⟩
⟨priority directives bc.y 162b⟩
%%
⟨grammar bc.y 162c⟩
%%

⟨function yylex(bc.y) 159c⟩

int
cpeek(int c, int yes, int no)
{

        peekc = getch();
        if(peekc == c) {
                peekc = -1;
                return yes;
        }
        return no;
}

int
getch(void)
{
        long ch;
```

```
loop:
        ch = peekc;
        if(ch < 0){
                if(in == 0)
                        ch = -1;
                else
                        ch = Bgetc(in);
        }
        peekc = -1;
        if(ch >= 0)
                return ch;
        ifile++;
        if(ifile > sargc) {
                if(ifile >= sargc+2)
                        getout();
                in = &bstdin;
                Binit(in, 0, OREAD);
                ln = 0;
                goto loop;
        }
        if(in)
                Bterm(in);
        if((in = Bopen(sargv[ifile], OREAD)) != 0){
                ln = 0;
                ss = sargv[ifile];
                goto loop;
        }
        yyerror("cannot open input file");
        return 0;               /* shut up ken */
}

char*
bundle(int a, ...)
{
        int i;
        char **q;
        va_list arg;

        i = a;
        va_start(arg, a);
        q = bsp_nxt;
        if(bdebug)
                fprint(2, "bundle %d elements at %lx\n", i, q);
        while(i-- > 0) {
                if(bsp_nxt >= &bspace[bsp_max])
                        yyerror("bundling space exceeded");
                *bsp_nxt++ = va_arg(arg, char*);
        }
        *bsp_nxt++ = 0;
        va_end(arg);
        //old: does not work with bison, need to use $$ in caller
        //yyval.cptr = (char*)q;
        return (char*)q;
}

void
routput(char *p)
{
        char **pp;
```

```
        if(bdebug)
                fprint(2, "routput(%lx)\n", p);
        if((char**)p >= &bspace[0] && (char**)p < &bspace[bsp_max]) {
                /* part of a bundle */
                pp = (char**)p;
                while(*pp != 0)
                        routput(*pp++);
        } else
                Bprint(&bstdout, p);    /* character string */
}

void
output(char *p)
{
        routput(p);
        bsp_nxt = &bspace[0];
        Bprint(&bstdout, "\n");
        Bflush(&bstdout);
        cp = cary;
        crs = rcrs;
}

void
conout(char *p, char *s)
{
        Bprint(&bstdout, "[");
        routput(p);
        Bprint(&bstdout, "]s%s\n", s);
        Bflush(&bstdout);
        lev--;
}
```

⟨*function* `yyerror` *(bc.y)* 158a⟩

```
void
pp(char *s)
{
        /* puts the relevant stuff on pre and post for the letter s */
        pre = bundle(3, "S", s, pre);
        post = bundle(4, post, "L", s, "s.");
}

void
tp(char *s)
{
        /* same as pp, but for temps */
        pre = bundle(3, "0S", s, pre);
        post = bundle(4, post, "L", s, "s.");
}
```

⟨*function* `yyinit` *(bc.y)* 159b⟩

```
void
getout(void)
{
        Bprint(&bstdout, "q");
        Bflush(&bstdout);
        exits(0);
}
```

```
char*
getf(char *p)
{
        return funtab[*p - 'a'];
}

char*
geta(char *p)
{
        return atab[*p - 'a'];
}
```

⟨*function* `main`*(bc.y)* 157b⟩

A.11.3 calc/hoc/hoc.y

⟨*calc/hoc.y* 479⟩≡

```
%{
#include "hoc.h"
#define code2(c1,c2)    code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
        Symbol  *sym;   /* symbol table pointer */
        Inst    *inst;  /* machine instruction */
        int     narg;   /* number of arguments */
        Formal  *formals;       /* list of formal parameters */
}
%token  <sym>   NUMBER STRING PRINT VAR BLTIN UNDEF WHILE FOR IF ELSE
%token  <sym>   FUNCTION PROCEDURE RETURN FUNC PROC READ
%type   <formals>       formals
%type   <inst>  expr stmt asgn prlist stmtlist
%type   <inst>  cond while for if begin end
%type   <sym>   procname
%type   <narg>  arglist
%right  '=' ADDEQ SUBEQ MULEQ DIVEQ MODEQ
%left   OR
%left   AND
%left   GT GE LT LE EQ NE
%left   '+' '-'
%left   '*' '/' '%'
%left   UNARYMINUS NOT INC DEC
%right  '^'
%%
list:     /* nothing */
        | list '\n'
        | list defn '\n'
        | list asgn '\n'  { code2(xpop, STOP); return 1; }
        | list stmt '\n'  { code(STOP); return 1; }
        | list expr '\n'  { code2(printtop, STOP); return 1; }
        | list error '\n' { yyerrok; }
        ;
asgn:     VAR '=' expr { code3(varpush,(Inst)$1,assign); $$=$3; }
        | VAR ADDEQ expr        { code3(varpush,(Inst)$1,addeq); $$=$3; }
        | VAR SUBEQ expr        { code3(varpush,(Inst)$1,subeq); $$=$3; }
        | VAR MULEQ expr        { code3(varpush,(Inst)$1,muleq); $$=$3; }
        | VAR DIVEQ expr        { code3(varpush,(Inst)$1,diveq); $$=$3; }
        | VAR MODEQ expr        { code3(varpush,(Inst)$1,modeq); $$=$3; }
```

```
        ;
stmt:     expr  { code(xpop); }
        | RETURN { defnonly("return"); code(procret); }
        | RETURN expr
              { defnonly("return"); $$=$2; code(funcret); }
        | PROCEDURE begin '(' arglist ')'
              { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
        | PRINT prlist  { $$ = $2; }
        | while '(' cond ')' stmt end {
              ($1)[1] = (Inst)$5;     /* body of loop */
              ($1)[2] = (Inst)$6; }   /* end, if cond fails */
        | for '(' cond ';' cond ';' cond ')' stmt end {
              ($1)[1] = (Inst)$5;     /* condition */
              ($1)[2] = (Inst)$7;     /* post loop */
              ($1)[3] = (Inst)$9;     /* body of loop */
              ($1)[4] = (Inst)$10; }  /* end, if cond fails */
        | if '(' cond ')' stmt end {    /* else-less if */
              ($1)[1] = (Inst)$5;     /* thenpart */
              ($1)[3] = (Inst)$6; }   /* end, if cond fails */
        | if '(' cond ')' stmt end ELSE stmt end {      /* if with else */
              ($1)[1] = (Inst)$5;     /* thenpart */
              ($1)[2] = (Inst)$8;     /* elsepart */
              ($1)[3] = (Inst)$9; }   /* end, if cond fails */
        | '{' stmtlist '}'      { $$ = $2; }
        ;
cond:      expr         { code(STOP); }
        ;
while:    WHILE { $$ = code3(whilecode,STOP,STOP); }
        ;
for:      FOR   { $$ = code(forcode); code3(STOP,STOP,STOP); code(STOP); }
        ;
if:       IF    { $$ = code(ifcode); code3(STOP,STOP,STOP); }
        ;
begin:    /* nothing */         { $$ = progp; }
        ;
end:      /* nothing */         { code(STOP); $$ = progp; }
        ;
stmtlist: /* nothing */         { $$ = progp; }
        | stmtlist '\n'
        | stmtlist stmt
        ;
expr:     NUMBER { $$ = code2(constpush, (Inst)$1); }
        | VAR    { $$ = code3(varpush, (Inst)$1, eval); }
        | asgn
        | FUNCTION begin '(' arglist ')'
              { $$ = $2; code3(call,(Inst)$1,(Inst)$4); }
        | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
        | BLTIN '(' expr ')' { $$=$3; code2(bltin, (Inst)$1->u.ptr); }
        | '(' expr ')'  { $$ = $2; }
        | expr '+' expr { code(add); }
        | expr '-' expr { code(sub); }
        | expr '*' expr { code(mul); }
        | expr '/' expr { code(div); }
        | expr '%' expr { code(mod); }
        | expr '^' expr { code (power); }
        | '-' expr   %prec UNARYMINUS   { $$=$2; code(negate); }
        | expr GT expr  { code(gt); }
        | expr GE expr  { code(ge); }
        | expr LT expr  { code(lt); }
        | expr LE expr  { code(le); }
```

```
        | expr EQ expr  { code(eq); }
        | expr NE expr  { code(ne); }
        | expr AND expr { code(and); }
        | expr OR expr  { code(or); }
        | NOT expr      { $$ = $2; code(not); }
        | INC VAR       { $$ = code2(preinc,(Inst)$2); }
        | DEC VAR       { $$ = code2(predec,(Inst)$2); }
        | VAR INC       { $$ = code2(postinc,(Inst)$1); }
        | VAR DEC       { $$ = code2(postdec,(Inst)$1); }
        ;
prlist:   expr                  { code(prexpr); }
        | STRING                { $$ = code2(prstr, (Inst)$1); }
        | prlist ',' expr       { code(prexpr); }
        | prlist ',' STRING     { code2(prstr, (Inst)$3); }
        ;
defn:     FUNC procname { $2->type=FUNCTION; indef=1; }
            '(' formals ')' stmt { code(procret); define($2, $5); indef=0; }
        | PROC procname { $2->type=PROCEDURE; indef=1; }
            '(' formals ')' stmt { code(procret); define($2, $5); indef=0; }
        ;
formals:        { $$ = 0; }
        | VAR                   { $$ = formallist($1, 0); }
        | VAR ',' formals       { $$ = formallist($1, $3); }
        ;
procname: VAR
        | FUNCTION
        | PROCEDURE
        ;
arglist:  /* nothing */         { $$ = 0; }
        | expr                  { $$ = 1; }
        | arglist ',' expr      { $$ = $1 + 1; }
        ;
%%
        /* end of grammar */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
jmp_buf begin;
int     indef;
char    *infile;        /* input file name */
Biobuf  *bin;           /* input file descriptor */
Biobuf  binbuf;
char    **gargv;        /* global argument list */
int     gargc;

int c = '\n';   /* global for use by warning() */

int     backslash(int), follow(int, int, int);
void    defnonly(char*), run(void);
void    warning(char*, char*);

yylex(void)             /* hoc6 */
{
        while ((c=Bgetc(bin)) == ' ' || c == '\t')
                ;
        if (c < 0)
                return 0;
```

```
if (c == '\\') {
        c = Bgetc(bin);
        if (c == '\n') {
                lineno++;
                return yylex();
        }
}
if (c == '#') {         /* comment */
        while ((c=Bgetc(bin)) != '\n' && c >= 0)
                ;
        if (c == '\n')
                lineno++;
        return c;
}
if (c == '.' || isdigit(c)) {   /* number */
        double d;
        Bungetc(bin);
        Bgetd(bin, &d);
        yylval.sym = install("", NUMBER, d);
        return NUMBER;
}
if (isalpha(c) || c == '_' || c >= 0x80) {
        Symbol *s;
        char sbuf[100], *p = sbuf;
        do {
                if (p >= sbuf + sizeof(sbuf) - 1) {
                        *p = '\0';
                        execerror("name too long", sbuf);
                }
                *p++ = c;
        } while ((c=Bgetc(bin)) >= 0 && (isalnum(c) || c == '_' || c >= 0x80));
        Bungetc(bin);
        *p = '\0';
        if ((s=lookup(sbuf)) == 0)
                s = install(sbuf, UNDEF, 0.0);
        yylval.sym = s;
        return s->type == UNDEF ? VAR : s->type;
}
if (c == '"') { /* quoted string */
        char sbuf[100], *p;
        for (p = sbuf; (c=Bgetc(bin)) != '"'; p++) {
                if (c == '\n' || c == Beof)
                        execerror("missing quote", "");
                if (p >= sbuf + sizeof(sbuf) - 1) {
                        *p = '\0';
                        execerror("string too long", sbuf);
                }
                *p = backslash(c);
        }
        *p = 0;
        yylval.sym = (Symbol *)emalloc(strlen(sbuf)+1);
        strcpy((char*)yylval.sym, sbuf);
        return STRING;
}
switch (c) {
case '+':       return follow('+', INC, follow('=', ADDEQ, '+'));
case '-':       return follow('-', DEC, follow('=', SUBEQ, '-'));
case '*':       return follow('=', MULEQ, '*');
case '/':       return follow('=', DIVEQ, '/');
case '%':       return follow('=', MODEQ, '%');
```

```
        case '>':       return follow('=', GE, GT);
        case '<':       return follow('=', LE, LT);
        case '=':       return follow('=', EQ, '=');
        case '!':       return follow('=', NE, NOT);
        case '|':       return follow('|', OR, '|');
        case '&':       return follow('&', AND, '&');
        case '\n':      lineno++; return '\n';
        default:        return c;
        }
}

backslash(int c)        /* get next char with \'s interpreted */
{
        static char transtab[] = "b\bf\fn\nr\rt\t";
        if (c != '\\')
                return c;
        c = Bgetc(bin);
        if (islower(c) && strchr(transtab, c))
                return strchr(transtab, c)[1];
        return c;
}

follow(int expect, int ifyes, int ifno) /* look ahead for >=, etc. */
{
        int c = Bgetc(bin);

        if (c == expect)
                return ifyes;
        Bungetc(bin);
        return ifno;
}

void
yyerror(char* s)        /* report compile-time error */
{
/*rob
        warning(s, (char *)0);
        longjmp(begin, 0);
rob*/
        execerror(s, (char *)0);
}

void
execerror(char* s, char* t)     /* recover from run-time error */
{
        warning(s, t);
        Bseek(bin, 0L, 2);              /* flush rest of file */
        restoreall();
        longjmp(begin, 0);
}

void
fpecatch(void)  /* catch floating point exceptions */
{
        execerror("floating point exception", (char *) 0);
}

void
intcatch(void)  /* catch interrupts */
{
```

```
        execerror("interrupt", 0);
}

void
run(void)       /* execute until EOF */
{
        setjmp(begin);
        for (initcode(); yyparse(); initcode())
                execute(progbase);
}

void
main(int argc, char* argv[])    /* hoc6 */
{
        static int first = 1;
#ifdef YYDEBUG
        extern int yydebug;
        yydebug=3;
#endif
        progname = argv[0];
        init();
        if (argc == 1) {        /* fake an argument list */
                static char *stdinonly[] = { "-" };

                gargv = stdinonly;
                gargc = 1;
        } else if (first) {     /* for interrupts */
                first = 0;
                gargv = argv+1;
                gargc = argc-1;
        }
        Binit(&binbuf, 0, OREAD);
        bin = &binbuf;
        while (moreinput())
                run();
        exits(0);
}

moreinput(void)
{
        char *expr;
        static char buf[64];
        int fd;
        static Biobuf b;

        if (gargc-- <= 0)
                return 0;
        if (bin && bin != &binbuf)
                Bterm(bin);
        infile = *gargv++;
        lineno = 1;
        if (strcmp(infile, "-") == 0) {
                bin = &binbuf;
                infile = 0;
                return 1;
        }
        if(strncmp(infile, "-e", 2) == 0) {
                if(infile[2]==0){
                        if(gargc == 0){
                                fprint(2, "%s: no argument for -e\n", progname);
```

```
                        return 0;
                }
                gargc--;
                expr = *gargv++;
        }else
                expr = infile+2;
        sprint(buf, "/tmp/hocXXXXXXX");
        infile = mktemp(buf);
        fd = create(infile, ORDWR|ORCLOSE, 0600);
        if(fd < 0){
                fprint(2, "%s: can't create temp. file: %r\n", progname);
                return 0;
        }
        fprint(fd, "%s\n", expr);
        /* leave fd around; file will be removed on exit */
        /* the following looks weird but is required for unix version */
        bin = &b;
        seek(fd, 0, 0);
        Binit(bin, fd, OREAD);
    } else {
        bin=Bopen(infile, OREAD);
        if (bin == 0) {
                fprint(2, "%s: can't open %s\n", progname, infile);
                return moreinput();
        }
    }
    return 1;
}

void
warning(char* s, char* t)       /* print warning message */
{
    fprint(2, "%s: %s", progname, s);
    if (t)
        fprint(2, " %s", t);
    if (infile)
        fprint(2, " in %s", infile);
    fprint(2, " near line %d\n", lineno);
    while (c != '\n' && c != Beof)
        if((c = Bgetc(bin)) == '\n')    /* flush rest of input line */
            lineno++;
}

void
defnonly(char *s)       /* warn if illegal definition */
{
    if (!indef)
        execerror(s, "used outside definition");
}
```

A.11.4 calc/hoc/code.c

⟨*constant* NSTACK*(hoc)* 485a⟩≡ (497c)

```
#define NSTACK  256
```

⟨*constant* NPROG*(hoc)* 485b⟩≡ (497c)

```
#define NPROG   2000
```

⟨*constant* NFRAME*(hoc)* 485c⟩≡ (497c)

```
#define NFRAME  100
```

⟨*function* `initcode`*(hoc)* 486a⟩≡ (497c)

```
void
initcode(void)
{
    progp = progbase;
    stackp = stack;
    fp = frame;
    returning = 0;
    indef = 0;
}
```

⟨*function* `push`*(hoc)* 486b⟩≡ (497c)

```
void
push(Datum d)
{
    if (stackp >= &stack[NSTACK])
        execerror("stack too deep", 0);
    *stackp++ = d;
}
```

⟨*function* `pop`*(hoc)* 486c⟩≡ (497c)

```
Datum
pop(void)
{
    if (stackp == stack)
        execerror("stack underflow", 0);
    return *--stackp;
}
```

⟨*function* `xpop`*(hoc)* 486d⟩≡ (497c)

```
void
xpop(void)      /* for when no value is wanted */
{
    if (stackp == stack)
        execerror("stack underflow", (char *)0);
    --stackp;
}
```

⟨*function* `constpush`*(hoc)* 486e⟩≡ (497c)

```
void
constpush(void)
{
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d);
}
```

⟨*function* `varpush`*(hoc)* 486f⟩≡ (497c)

```
void
varpush(void)
{
    Datum d;
    d.sym = (Symbol *)(*pc++);
    push(d);
}
```

⟨*function* `whilecode`*(hoc)* 487a⟩≡ (497c)

```
void
whilecode(void)
{
    Datum d;
    Inst *savepc = pc;

    execute(savepc+2);  /* condition */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc)));  /* body */
        if (returning)
            break;
        execute(savepc+2);      /* condition */
        d = pop();
    }
    if (!returning)
        pc = *((Inst **)(savepc+1)); /* next stmt */
}
```

⟨*function* `forcode`*(hoc)* 487b⟩≡ (497c)

```
void
forcode(void)
{
    Datum d;
    Inst *savepc = pc;

    execute(savepc+4);          /* precharge */
    pop();
    execute(*((Inst **)(savepc)));      /* condition */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc+2)));        /* body */
        if (returning)
            break;
        execute(*((Inst **)(savepc+1)));        /* post loop */
        pop();
        execute(*((Inst **)(savepc)));  /* condition */
        d = pop();
    }
    if (!returning)
        pc = *((Inst **)(savepc+3)); /* next stmt */
}
```

⟨*function* `ifcode`*(hoc)* 487c⟩≡ (497c)

```
void
ifcode(void)
{
    Datum d;
    Inst *savepc = pc;  /* then part */

    execute(savepc+3);  /* condition */
    d = pop();
    if (d.val)
        execute(*((Inst **)(savepc)));
    else if (*((Inst **)(savepc+1))) /* else part? */
        execute(*((Inst **)(savepc+1)));
    if (!returning)
        pc = *((Inst **)(savepc+2)); /* next stmt */
}
```

⟨*function* `define`*(hoc)* 488a⟩≡ (497c)

```
void
define(Symbol* sp, Formal *f)   /* put func/proc in symbol table */
{
    Fndefn *fd;
    int n;

    fd = emalloc(sizeof(Fndefn));
    fd->code = progbase;        /* start of code */
    progbase = progp;   /* next code starts here */
    fd->formals = f;
    for(n=0; f; f=f->next)
        n++;
    fd->nargs = n;
    sp->u.defn = fd;
}
```

⟨*function* `call`*(hoc)* 488b⟩≡ (497c)

```
void
call(void)              /* call a function */
{
    Formal *f;
    Datum *arg;
    Saveval *s;
    int i;

    Symbol *sp = (Symbol *)pc[0]; /* symbol table entry */
                    /* for function */
    if (fp >= &frame[NFRAME-1])
        execerror(sp->name, "call nested too deeply");
    fp++;
    fp->sp = sp;
    fp->nargs = (int)(uintptr)pc[1];
    fp->retpc = pc + 2;
    fp->argn = stackp - 1;      /* last argument */
    if(fp->nargs != sp->u.defn->nargs)
        execerror(sp->name, "called with wrong number of arguments");
    /* bind formals */
    f = sp->u.defn->formals;
    arg = stackp - fp->nargs;
    while(f){
        s = emalloc(sizeof(Saveval));
        s->val = f->sym->u;
        s->type = f->sym->type;
        s->next = f->save;
        f->save = s;
        f->sym->u.val = arg->val;
        f->sym->type = VAR;
        f = f->next;
        arg++;
    }
    for (i = 0; i < fp->nargs; i++)
        pop();  /* pop arguments; no longer needed */
    execute(sp->u.defn->code);
    returning = 0;
}
```

⟨*function* `restore`*(hoc)* 488c⟩≡ (497c)

```
void
restore(Symbol *sp)     /* restore formals associated with symbol */
```

```
{
    Formal *f;
    Saveval *s;

    f = sp->u.defn->formals;
    while(f){
        s = f->save;
        if(s == 0)      /* more actuals than formals */
            break;
        f->sym->u = s->val;
        f->sym->type = s->type;
        f->save = s->next;
        free(s);
        f = f->next;
    }
}
```

⟨*function* `restoreall` *(hoc)* 489a⟩≡ (497c)

```
void
restoreall(void)        /* restore all variables in case of error */
{
    while(fp>=frame && fp->sp){
        restore(fp->sp);
        --fp;
    }
    fp = frame;
}
```

⟨*function* `ret` *(hoc)* 489b⟩≡ (497c)

```
static void
ret(void)               /* common return from func or proc */
{
    /* restore formals */
    restore(fp->sp);
    pc = (Inst *)fp->retpc;
    --fp;
    returning = 1;
}
```

⟨*function* `funcret` *(hoc)* 489c⟩≡ (497c)

```
void
funcret(void)   /* return from a function */
{
    Datum d;
    if (fp->sp->type == PROCEDURE)
        execerror(fp->sp->name, "(proc) returns value");
    d = pop();  /* preserve function return value */
    ret();
    push(d);
}
```

⟨*function* `procret` *(hoc)* 489d⟩≡ (497c)

```
void
procret(void)   /* return from a procedure */
{
    if (fp->sp->type == FUNCTION)
        execerror(fp->sp->name,
            "(func) returns no value");
    ret();
}
```

⟨*function* `bltin`*(hoc)* 490a⟩≡ (497c)

```
void
bltin(void)
{

    Datum d;
    d = pop();
    d.val = (*(double (*)(double))*pc++)(d.val);
    push(d);
}
```

⟨*function* `add`*(hoc)* 490b⟩≡ (497c)

```
void
add(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push(d1);
}
```

⟨*function* `sub`*(hoc)* 490c⟩≡ (497c)

```
void
sub(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val -= d2.val;
    push(d1);
}
```

⟨*function* `mul`*(hoc)* 490d⟩≡ (497c)

```
void
mul(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val *= d2.val;
    push(d1);
}
```

⟨*function* `div`*(hoc)* 490e⟩≡ (497c)

```
void
div(void)
{
    Datum d1, d2;
    d2 = pop();
    if (d2.val == 0.0)
        execerror("division by zero", (char *)0);
    d1 = pop();
    d1.val /= d2.val;
    push(d1);
}
```

⟨*function* `mod`*(hoc)* 491a⟩≡ (497c)

```
void
mod(void)
{
    Datum d1, d2;
    d2 = pop();
    if (d2.val == 0.0)
        execerror("division by zero", (char *)0);
    d1 = pop();
    /* d1.val %= d2.val; */
    d1.val = fmod(d1.val, d2.val);
    push(d1);
}
```

⟨*function* `negate`*(hoc)* 491b⟩≡ (497c)

```
void
negate(void)
{
    Datum d;
    d = pop();
    d.val = -d.val;
    push(d);
}
```

⟨*function* `verify`*(hoc)* 491c⟩≡ (497c)

```
void
verify(Symbol* s)
{
    if (s->type != VAR && s->type != UNDEF)
        execerror("attempt to evaluate non-variable", s->name);
    if (s->type == UNDEF)
        execerror("undefined variable", s->name);
}
```

⟨*function* `eval`*(hoc)* 491d⟩≡ (497c)

```
void
eval(void)              /* evaluate variable on stack */
{
    Datum d;
    d = pop();
    verify(d.sym);
    d.val = d.sym->u.val;
    push(d);
}
```

⟨*function* `preinc`*(hoc)* 491e⟩≡ (497c)

```
void
preinc(void)
{
    Datum d;
    d.sym = (Symbol *)(*pc++);
    verify(d.sym);
    d.val = d.sym->u.val += 1.0;
    push(d);
}
```

⟨*function* `predec`*(hoc)* 491f⟩≡ (497c)

```
void
predec(void)
{
```

```
    Datum d;
    d.sym = (Symbol *)(*pc++);
    verify(d.sym);
    d.val = d.sym->u.val -= 1.0;
    push(d);
}
```

⟨*function* `postinc` *(hoc)* 492a⟩≡ (497c)

```
void
postinc(void)
{
    Datum d;
    double v;
    d.sym = (Symbol *)(*pc++);
    verify(d.sym);
    v = d.sym->u.val;
    d.sym->u.val += 1.0;
    d.val = v;
    push(d);
}
```

⟨*function* `postdec` *(hoc)* 492b⟩≡ (497c)

```
void
postdec(void)
{
    Datum d;
    double v;
    d.sym = (Symbol *)(*pc++);
    verify(d.sym);
    v = d.sym->u.val;
    d.sym->u.val -= 1.0;
    d.val = v;
    push(d);
}
```

⟨*function* `gt` *(hoc)* 492c⟩≡ (497c)

```
void
gt(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val > d2.val);
    push(d1);
}
```

⟨*function* `lt` *(hoc)* 492d⟩≡ (497c)

```
void
lt(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val < d2.val);
    push(d1);
}
```

⟨*function* `ge` *(hoc)* 493a⟩≡ (497c)

```
void
ge(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val >= d2.val);
    push(d1);
}
```

⟨*function* `le` *(hoc)* 493b⟩≡ (497c)

```
void
le(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val <= d2.val);
    push(d1);
}
```

⟨*function* `eq` *(hoc)* 493c⟩≡ (497c)

```
void
eq(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val == d2.val);
    push(d1);
}
```

⟨*function* `ne` *(hoc)* 493d⟩≡ (497c)

```
void
ne(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != d2.val);
    push(d1);
}
```

⟨*function* `and` *(hoc)* 493e⟩≡ (497c)

```
void
and(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != 0.0 && d2.val != 0.0);
    push(d1);
}
```

⟨*function* `or` *(hoc)* 493f⟩≡ (497c)

```
void
or(void)
{
    Datum d1, d2;
```

```
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != 0.0 || d2.val != 0.0);
    push(d1);
}
```

⟨*function* `not` *(hoc)* 494a⟩≡ (497c)

```
void
not(void)
{
    Datum d;
    d = pop();
    d.val = (double)(d.val == 0.0);
    push(d);
}
```

⟨*function* `power` *(hoc)* 494b⟩≡ (497c)

```
void
power(void)
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = Pow(d1.val, d2.val);
    push(d1);
}
```

⟨*function* `assign` *(hoc)* 494c⟩≡ (497c)

```
void
assign(void)
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    d1.sym->u.val = d2.val;
    d1.sym->type = VAR;
    push(d2);
}
```

⟨*function* `addeq` *(hoc)* 494d⟩≡ (497c)

```
void
addeq(void)
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    d2.val = d1.sym->u.val += d2.val;
    d1.sym->type = VAR;
    push(d2);
}
```

⟨*function* subeq*(hoc)* 495a⟩≡ (497c)

```
void
subeq(void)
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    d2.val = d1.sym->u.val -= d2.val;
    d1.sym->type = VAR;
    push(d2);
}
```

⟨*function* muleq*(hoc)* 495b⟩≡ (497c)

```
void
muleq(void)
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    d2.val = d1.sym->u.val *= d2.val;
    d1.sym->type = VAR;
    push(d2);
}
```

⟨*function* diveq*(hoc)* 495c⟩≡ (497c)

```
void
diveq(void)
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    d2.val = d1.sym->u.val /= d2.val;
    d1.sym->type = VAR;
    push(d2);
}
```

⟨*function* modeq*(hoc)* 495d⟩≡ (497c)

```
void
modeq(void)
{
    Datum d1, d2;
    long x;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    /* d2.val = d1.sym->u.val %= d2.val; */
    x = d1.sym->u.val;
    x %= (long) d2.val;
    d2.val = d1.sym->u.val = x;
```

```
    d1.sym->type = VAR;
    push(d2);
}
```

⟨*function* printtop*(hoc)* 496a⟩≡ (497c)

```
void
printtop(void)  /* pop top value from stack, print it */
{
    Datum d;
    static Symbol *s;   /* last value computed */
    if (s == 0)
        s = install("_", VAR, 0.0);
    d = pop();
    print("%.12g\n", d.val);
    s->u.val = d.val;
}
```

⟨*function* prexpr*(hoc)* 496b⟩≡ (497c)

```
void
prexpr(void)    /* print numeric value */
{
    Datum d;
    d = pop();
    print("%.12g ", d.val);
}
```

⟨*function* prstr*(hoc)* 496c⟩≡ (497c)

```
void
prstr(void)             /* print string value */
{
    print("%s", (char *) *pc++);
}
```

⟨*function* varread*(hoc)* 496d⟩≡ (497c)

```
void
varread(void)   /* read into variable */
{
    Datum d;
    extern Biobuf *bin;
    Symbol *var = (Symbol *) *pc++;
    int c;

  Again:
    do
        c = Bgetc(bin);
    while(c==' ' || c=='\t' || c=='\n');
    if(c == Beof){
  Iseof:
        if(moreinput())
            goto Again;
        d.val = var->u.val = 0.0;
        goto Return;
    }

    if(strchr("+-.0123456789", c) == 0)
        execerror("non-number read into", var->name);
    Bungetc(bin);
    if(Bgetd(bin, &var->u.val) == Beof)
        goto Iseof;
    else
```

```
        d.val = 1.0;
    Return:
      var->type = VAR;
      push(d);
  }
```

⟨*function* code*(hoc)* 497a⟩≡ (497c)

```
  Inst*
  code(Inst f)    /* install one instruction or operand */
  {
      Inst *oprogp = progp;
      if (progp >= &prog[NPROG])
          execerror("program too big", (char *)0);
      *progp++ = f;
      return oprogp;
  }
```

⟨*function* execute*(hoc)* 497b⟩≡ (497c)

```
  void
  execute(Inst* p)
  {
      for (pc = p; *pc != STOP && !returning; )
          (*((++pc)[-1]))();
  }
```

⟨hoc/code.c 497c⟩≡

```
  #include <u.h>
  #include <libc.h>
  #include <bio.h>
  #include "hoc.h"
  #include "y.tab.h"

  ⟨constant NSTACK(hoc) 485a⟩

  static Datum stack[NSTACK];     /* the stack */
  static Datum *stackp;           /* next free spot on stack */

  ⟨constant NPROG(hoc) 485b⟩
  Inst    prog[NPROG];    /* the machine */
  Inst    *progp;         /* next free spot for code generation */
  Inst    *pc;            /* program counter during execution */
  Inst    *progbase = prog; /* start of current subprogram */
  int     returning;      /* 1 if return stmt seen */
  int     indef;  /* 1 if parsing a func or proc */

  typedef struct Frame {  /* proc/func call stack frame */
      Symbol      *sp;    /* symbol table entry */
      Inst        *retpc; /* where to resume after return */
      Datum       *argn;  /* n-th argument on stack */
      int nargs;  /* number of arguments */
  } Frame;
  ⟨constant NFRAME(hoc) 485c⟩
  Frame   frame[NFRAME];
  Frame   *fp;            /* frame pointer */

  ⟨function initcode(hoc) 486a⟩

  ⟨function push(hoc) 486b⟩

  ⟨function pop(hoc) 486c⟩
```

⟨*function* `xpop`*(hoc)* 486d⟩

⟨*function* `constpush`*(hoc)* 486e⟩

⟨*function* `varpush`*(hoc)* 486f⟩

⟨*function* `whilecode`*(hoc)* 487a⟩

⟨*function* `forcode`*(hoc)* 487b⟩

⟨*function* `ifcode`*(hoc)* 487c⟩

⟨*function* `define`*(hoc)* 488a⟩

⟨*function* `call`*(hoc)* 488b⟩

⟨*function* `restore`*(hoc)* 488c⟩

⟨*function* `restoreall`*(hoc)* 489a⟩

⟨*function* `ret`*(hoc)* 489b⟩

⟨*function* `funcret`*(hoc)* 489c⟩

⟨*function* `procret`*(hoc)* 489d⟩

⟨*function* `bltin`*(hoc)* 490a⟩

⟨*function* `add`*(hoc)* 490b⟩

⟨*function* `sub`*(hoc)* 490c⟩

⟨*function* `mul`*(hoc)* 490d⟩

⟨*function* `div`*(hoc)* 490e⟩

⟨*function* `mod`*(hoc)* 491a⟩

⟨*function* `negate`*(hoc)* 491b⟩

⟨*function* `verify`*(hoc)* 491c⟩

⟨*function* `eval`*(hoc)* 491d⟩

⟨*function* `preinc`*(hoc)* 491e⟩

⟨*function* `predec`*(hoc)* 491f⟩

⟨*function* `postinc`*(hoc)* 492a⟩

⟨*function* `postdec`*(hoc)* 492b⟩

⟨*function* `gt`*(hoc)* 492c⟩

⟨*function* `lt`*(hoc)* 492d⟩

⟨*function* `ge`*(hoc)* 493a⟩

⟨*function* `le`*(hoc)* 493b⟩

⟨*function* `eq`*(hoc)* 493c⟩

⟨*function* `ne`*(hoc)* 493d⟩

⟨*function* `and`*(hoc)* 493e⟩

⟨*function* `or`*(hoc)* 493f⟩

⟨*function* `not`*(hoc)* 494a⟩

⟨*function* `power`*(hoc)* 494b⟩

⟨*function* `assign`*(hoc)* 494c⟩

⟨*function* `addeq`*(hoc)* 494d⟩

⟨*function* `subeq`*(hoc)* 495a⟩

⟨*function* `muleq`*(hoc)* 495b⟩

⟨*function* `diveq`*(hoc)* 495c⟩

⟨*function* `modeq`*(hoc)* 495d⟩

⟨*function* `printtop`*(hoc)* 496a⟩

⟨*function* `prexpr`*(hoc)* 496b⟩

⟨*function* `prstr`*(hoc)* 496c⟩

⟨*function* `varread`*(hoc)* 496d⟩

⟨*function* `code`*(hoc)* 497a⟩

⟨*function* `execute`*(hoc)* 497b⟩

A.11.5 calc/hoc/hoc.h

⟨*constant* `STOP`*(hoc)* 499a⟩≡ (499b)

```
#define STOP    (Inst) 0
```

⟨`hoc/hoc.h` 499b⟩≡

```
typedef void (*Inst)(void);
⟨constant STOP(hoc) 499a⟩

typedef struct Symbol   Symbol;
typedef union Datum     Datum;
typedef struct Formal   Formal;
typedef struct Saveval  Saveval;
typedef struct Fndefn   Fndefn;
typedef union Symval    Symval;

union Symval { /* value of a symbol */
    double      val;            /* VAR */
    double      (*ptr)(double); /* BLTIN */
    Fndefn      *defn;          /* FUNCTION, PROCEDURE */
    char        *str;           /* STRING */
};
```

```
struct Symbol { /* symbol table entry */
    char        *name;
    long        type;
    Symval u;
    struct Symbol       *next;  /* to link to another */
};
Symbol  *install(char*, int, double), *lookup(char*);

union Datum {   /* interpreter stack type */
    double      val;
    Symbol      *sym;
};

struct Saveval {        /* saved value of variable */
    Symval      val;
    long                type;
    Saveval     *next;
};

struct Formal { /* formal parameter */
    Symbol      *sym;
    Saveval     *save;
    Formal      *next;
};

struct Fndefn { /* formal parameter */
    Inst        *code;
    Formal      *formals;
    int nargs;
};

extern  Formal *formallist(Symbol*, Formal*);
extern  double Fgetd(int);
extern  int moreinput(void);
extern  void restore(Symbol*);
extern  void restoreall(void);
extern  void execerror(char*, char*);
extern  void define(Symbol*, Formal*), verify(Symbol*);
extern  Datum pop(void);
extern  void initcode(void), push(Datum), xpop(void), constpush(void);
extern  void varpush(void);
extern  void eval(void), add(void), sub(void), mul(void), div(void), mod(void);
extern  void negate(void), power(void);
extern  void addeq(void), subeq(void), muleq(void), diveq(void), modeq(void);

extern  Inst *progp, *progbase, prog[], *code(Inst);
extern  void assign(void), bltin(void), varread(void);
extern  void prexpr(void), prstr(void);
extern  void gt(void), lt(void), eq(void), ge(void), le(void), ne(void);
extern  void and(void), or(void), not(void);
extern  void ifcode(void), whilecode(void), forcode(void);
extern  void call(void), arg(void), argassign(void);
extern  void funcret(void), procret(void);
extern  void preinc(void), predec(void), postinc(void), postdec(void);
extern  void execute(Inst*);
extern  void printtop(void);

extern double   Log(double), Log10(double), Gamma(double), Sqrt(double), Exp(double);
extern double   Asin(double), Acos(double), Sinh(double), Cosh(double), integer(double);
```

```
extern double   Pow(double, double);

extern  void init(void);
extern  int yyparse(void);
extern  void execerror(char*, char*);
extern  void *emalloc(unsigned);
```

A.11.6 calc/hoc/init.c

⟨*function* `init` *(hoc)* 501a⟩≡ (501b)

```
void
init(void)      /* install constants and built-ins in table */
{
    int i;
    Symbol *s;
    for (i = 0; keywords[i].name; i++)
        install(keywords[i].name, keywords[i].kval, 0.0);
    for (i = 0; consts[i].name; i++)
        install(consts[i].name, VAR, consts[i].cval);
    for (i = 0; builtins[i].name; i++) {
        s = install(builtins[i].name, BLTIN, 0.0);
        s->u.ptr = builtins[i].func;
    }
}
```

⟨hoc/init.c 501b⟩≡

```
#include <u.h>
#include <libc.h>
#include "hoc.h"
#include "y.tab.h"

static struct {         /* Keywords */
    char        *name;
    int kval;
} keywords[] = {
    "proc",             PROC,
    "func",             FUNC,
    "return",   RETURN,
    "if",               IF,
    "else",             ELSE,
    "while",    WHILE,
    "for",              FOR,
    "print",    PRINT,
    "read",             READ,
    0,          0,
};

static struct {         /* Constants */
    char *name;
    double cval;
} consts[] = {
    "PI",        3.14159265358979323846,
    "E",         2.71828182845904523536,
    "GAMMA", 0.57721566490153286060,  /* Euler */
    "DEG",      57.29577951308232087680,  /* deg/radian */
    "PHI",   1.61803398874989484820,  /* golden ratio */
    0,   0
};
```

```
static struct {         /* Built-ins */
    char *name;
    double      (*func)(double);
} builtins[] = {
    "sin",      sin,
    "cos",      cos,
    "tan",      tan,
    "atan",     atan,
    "asin",     Asin,   /* checks range */
    "acos", Acos,       /* checks range */
    "sinh",     Sinh,   /* checks range */
    "cosh",     Cosh,   /* checks range */
    "tanh",     tanh,
    "log",      Log,    /* checks range */
    "log10", Log10,     /* checks range */
    "exp",      Exp,    /* checks range */
    "sqrt",     Sqrt,   /* checks range */
    "int",      integer,
    "abs",      fabs,
    0,  0
};
```

⟨*function* `init` *(hoc)* 501a⟩

A.11.7 calc/hoc/math.c

⟨*function* `Log` *(hoc)* 502a⟩≡ (503g)
```
double
Log(double x)
{
    return errcheck(log(x), "log");
}
```

⟨*function* `Log10` *(hoc)* 502b⟩≡ (503g)
```
double
Log10(double x)
{
    return errcheck(log10(x), "log10");
}
```

⟨*function* `Sqrt` *(hoc)* 502c⟩≡ (503g)
```
double
Sqrt(double x)
{
    return errcheck(sqrt(x), "sqrt");
}
```

⟨*function* `Exp` *(hoc)* 502d⟩≡ (503g)
```
double
Exp(double x)
{
    return errcheck(exp(x), "exp");
}
```

⟨*function* `Asin` *(hoc)* 502e⟩≡ (503g)
```
double
Asin(double x)
{
    return errcheck(asin(x), "asin");
}
```

⟨*function* Acos*(hoc)* 503a⟩≡ (503g)
```
double
Acos(double x)
{
    return errcheck(acos(x), "acos");
}
```

⟨*function* Sinh*(hoc)* 503b⟩≡ (503g)
```
double
Sinh(double x)
{
    return errcheck(sinh(x), "sinh");
}
```

⟨*function* Cosh*(hoc)* 503c⟩≡ (503g)
```
double
Cosh(double x)
{
    return errcheck(cosh(x), "cosh");
}
```

⟨*function* Pow*(hoc)* 503d⟩≡ (503g)
```
double
Pow(double x, double y)
{
    return errcheck(pow(x,y), "exponentiation");
}
```

⟨*function* integer*(hoc)* 503e⟩≡ (503g)
```
double
integer(double x)
{
    if(x<-2147483648.0 || x>2147483647.0)
        execerror("argument out of domain", 0);
    return (double)(long)x;
}
```

⟨*function* errcheck*(hoc)* 503f⟩≡ (503g)
```
double
errcheck(double d, char* s)     /* check result of library call */
{
    if(isNaN(d))
        execerror(s, "argument out of domain");
    if(isInf(d, 0))
        execerror(s, "result out of range");
    return d;
}
```

⟨hoc/math.c 503g⟩≡
```
#include <u.h>
#include <libc.h>

#include "hoc.h"

double  errcheck(double, char*);

⟨function Log(hoc) 502a⟩
⟨function Log10(hoc) 502b⟩

⟨function Sqrt(hoc) 502c⟩
```

⟨*function* `Exp`*(hoc)* 502d⟩

⟨*function* `Asin`*(hoc)* 502e⟩

⟨*function* `Acos`*(hoc)* 503a⟩

⟨*function* `Sinh`*(hoc)* 503b⟩
⟨*function* `Cosh`*(hoc)* 503c⟩
⟨*function* `Pow`*(hoc)* 503d⟩

⟨*function* `integer`*(hoc)* 503e⟩

⟨*function* `errcheck`*(hoc)* 503f⟩

A.11.8 calc/hoc/symbol.c

⟨*function* `lookup`*(hoc)* 504a⟩≡ (505b)

```
Symbol*
lookup(char* s) /* find s in symbol table */
{
    Symbol *sp;

    for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
        if (strcmp(sp->name, s) == 0)
            return sp;
    return 0;   /* 0 ==> not found */
}
```

⟨*function* `install`*(hoc)* 504b⟩≡ (505b)

```
Symbol*
install(char* s, int t, double d)  /* install s in symbol table */
{
    Symbol *sp;

    sp = emalloc(sizeof(Symbol));
    sp->name = emalloc(strlen(s)+1); /* +1 for '\0' */
    strcpy(sp->name, s);
    sp->type = t;
    sp->u.val = d;
    sp->next = symlist; /* put at front of list */
    symlist = sp;
    return sp;
}
```

⟨*function* `emalloc`*(hoc)* 504c⟩≡ (505b)

```
void*
emalloc(unsigned n)     /* check return from malloc */
{
    char *p;

    p = malloc(n);
    if (p == 0)
        execerror("out of memory", (char *) 0);
    return p;
}
```

⟨*function* `formallist` *(hoc)* 505a⟩≡ (505b)

```
Formal*
formallist(Symbol *formal, Formal *list)        /* add formal to list */
{
    Formal *f;

    f = emalloc(sizeof(Formal));
    f->sym = formal;
    f->save = 0;
    f->next = list;
    return f;
}
```

⟨`hoc/symbol.c` 505b⟩≡

```
#include <u.h>
#include <libc.h>
#include "hoc.h"
#include "y.tab.h"

static Symbol *symlist = 0;  /* symbol table: linked list */
```

⟨*function* `lookup` *(hoc)* 504a⟩

⟨*function* `install` *(hoc)* 504b⟩

⟨*function* `emalloc` *(hoc)* 504c⟩

⟨*function* `formallist` *(hoc)* 505a⟩

Glossary

LOC = Lines Of Code
CLI = Command-Line Interface

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 9

[KP76] Brian W. Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley, 1976. cited page(s) 7

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984. cited page(s) 7, 9

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999. cited page(s) 7

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 9

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 9

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 9, 174

[Pad16] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 9, 95, 175

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 174