Principia Softwarica: The Build System `mk`

version 1.0

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Andrew Hume

June 10, 2026

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

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

Contents

1 Introduction **7**
- 1.1 Motivations . . . 7
- 1.2 The Plan 9 build system: `mk` . . . 8
- 1.3 Other build systems . . . 8
- 1.4 Getting started . . . 10
- 1.5 Requirements . . . 12
- 1.6 About this document . . . 12
- 1.7 Copyright . . . 12
- 1.8 Acknowledgments . . . 12

2 Overview **13**
- 2.1 Build system principles . . . 13
  - 2.1.1 A domain-specific language . . . 14
  - 2.1.2 A graph of dependencies . . . 17
  - 2.1.3 A job scheduler . . . 22
- 2.2 `mk` command-line interface . . . 23
- 2.3 `hello.mk` . . . 24
- 2.4 Code organization . . . 24
- 2.5 Software architecture . . . 24
- 2.6 Book structure . . . 27

3 Core Data Structures **28**
- 3.1 Symbol table . . . 28
  - 3.1.1 `Symtab` . . . 28
  - 3.1.2 `hash` . . . 29
  - 3.1.3 Namespaces . . . 31
- 3.2 `Words` . . . 32
- 3.3 `Rules` . . . 34
  - 3.3.1 `Rule` . . . 34
  - 3.3.2 Simple `rules` . . . 35
  - 3.3.3 `metarules` . . . 35
  - 3.3.4 Adding rules . . . 36
- 3.4 Graph . . . 39
  - 3.4.1 `Node` . . . 39
  - 3.4.2 `Arc` . . . 40
- 3.5 `Jobs` . . . 41
- 3.6 Putting it together: the data structures for `hello.mk` . . . 43

4 main() 45
4.1 main() skeleton . . . 45
4.2 mk -<flag> arguments processing . . . 46
4.3 mk <var>=<values> arguments processing . . . 46
4.4 mk remaining arguments processing . . . 48
4.5 Using the mkfile or mk -f <file> . . . 48
4.6 Building the target(s) . . . 48
4.6.1 Default target: target1 . . . 49
4.6.2 Building sequentiallly . . . 49
4.6.3 Building in parallel . . . 50

5 Parsing the mkfile 51
5.1 parse() . . . 51
5.1.1 Assembling a line: assline() . . . 53
5.1.2 Parsing the head of a line: rhead() . . . 57
5.1.3 Splitting a string in words: stow() . . . 60
5.2 Parsing rules: <target>:<prereqs> . . . 67
5.2.1 Parsing the recipe: rbody() . . . 69
5.2.2 Parsing rule attributes . . . 70
5.3 Parsing included files: < <file> . . . 71
5.4 Parsing Variable definitions: <var>=<values> . . . 73
5.4.1 Overriding variable definitions . . . 74
5.4.2 Parsing variable attributes . . . 74

6 Building the Graph of Dependencies 76
6.1 graph() and applyrules() . . . 76
6.2 Finding the simple rule(s) for a target . . . 77
6.3 Finding matching metarules . . . 78
6.3.1 Matching a pattern: match() . . . 80
6.3.2 Substituting the stem: subst() . . . 80
6.4 Node cache . . . 81
6.5 timeof() . . . 82
6.6 Checking the graph and the rules . . . 83
6.6.1 Cycle detection . . . 83
6.6.2 Infinite rule detection . . . 85
6.6.3 Ambiguous rules detection . . . 86

7 Finding Outdated Files 92
7.1 mk() . . . 92
7.2 Initializing nodes: clrmade() . . . 94
7.3 Exploring the graph: work() . . . 95
7.3.1 The leaf case . . . 95
7.3.2 The parent case . . . 95
7.4 Scheduling recipes: dorecipe() . . . 97
7.4.1 Building the list of target nodes . . . 99
7.4.2 Building the list of prerequisites . . . 100

8 Scheduling Jobs **101**
8.1 Enqueuing jobs: `run()` . . . 101
8.2 Scheduling jobs . . . 102
8.2.1 `RunEvent` and `events` . . . 102
8.2.2 `sched()` . . . 103
8.3 Executing Jobs: `execsh()` . . . 105
8.4 Waiting for jobs to finish . . . 107
8.4.1 `waitup()` . . . 107
8.4.2 `update()` . . . 108
8.4.3 `waitup()` edge cases . . . 109
8.5 Process lifecycle: `Exit()` . . . 110
8.6 Notes (signals) management . . . 110

9 The Shell Environment **112**
9.1 `ShellEnvVar` and `shellenv` . . . 112
9.2 Initializing the shell environment: `initenv()` . . . 114
9.3 Importing the environment: `readenv()` . . . 115
9.4 Adjusting the shell environment: `buildenv()` . . . 117
9.5 Exporting the shell environment: `exportenv()` . . . 118

10 Debugging and Profiling Support **120**
10.1 Printing jobs: `shprint()` . . . 120
10.1.1 Expanding and printing variables . . . 121
10.1.2 Printing quoted strings . . . 123
10.2 Explain mode: `mk -e` . . . 123
10.3 Dry mode: `mk -n` . . . 124
10.4 What-if mode: `mk -w <file>` . . . 125
10.5 Processor utilization: `mk -u` . . . 126

11 Advanced Features **128**
11.1 Regular-expression rules: `:R:` . . . 128
11.2 Shell-command expansion: `` `<cmd>` `` . . . 130
11.2.1 Parsing backquotes: `bquote()` . . . 131
11.2.2 Adjusting `execsh()` . . . 132
11.3 Dynamic `mkfile`: `<| <prog>` . . . 133
11.4 Substitution variables: `$<name>:<pattern>=<subst>` . . . 135
11.4.1 Parsing adjustments . . . 135
11.4.2 Substitutions: `subsub()` . . . 137
11.5 Rule attributes . . . 139
11.5.1 Virtual targets: `:V:` . . . 140
11.5.2 Deleting a target when the recipe returns an error: `:D:` . . . 141
11.5.3 Quiet mode (not printing the recipe): `:Q:` . . . 141
11.5.4 Running a shell script without `-e`: `:E:` . . . 142
11.5.5 Disabling the no-recipe warning, `:N:` . . . 142
11.5.6 Forbidding metarules to match virtual targets: `:n:` . . . 142
11.5.7 Interactive recipes: `:I:` . . . 143
11.5.8 Custom-dependency comparison program: `:P:` . . . 143
11.6 Variable attributes . . . 146
11.6.1 Private variables: `=U=` . . . 146
11.7 Advanced `mk` variables . . . 147

11.7.1 $target versus $alltarget . . . 147
11.7.2 $prereq versus $newprereq . . . 148
11.7.3 $NREP . . . 148
11.7.4 $pid . . . 149
11.7.5 $nproc . . . 149
11.8 Shell choice: $MKSHELL . . . 149
11.9 Dealing with archives (libraries) . . . 150
11.10 Optimizations . . . 153
11.10.1 Missing-intermediates optimization: mk -I . . . 154
11.10.2 Touching-mode optimization: mk -t . . . 156
11.10.3 Time cache . . . 157
11.10.4 Bulk time optimisation . . . 157
11.11 Recompiling everything: mk -a . . . 159
11.12 Recursive mk: $MKFLAGS and $MKARGS . . . 160
11.13 Keep-going mode: mk -k . . . 161
11.13.1 kflag and runerrs . . . 161
11.13.2 Adjusting mk(), work(), and waitup() . . . 161

12 Conclusion **163**
12.1 Patterns and techniques . . . 163
12.2 Connections to other books . . . 163
12.3 Beyond mk . . . 164

A Debugging **165**
A.1 Dumping the rules: mk -dp . . . 166
A.2 Dumping the graph: mk -dg . . . 167
A.3 Tracing jobs: mk -de . . . 167
A.4 Tracing function calls: mk -dt . . . 168

B Profiling **170**

C Utilities **171**
C.1 Memory management . . . 171
C.2 Buffer management . . . 171
C.3 File management . . . 174

D Examples of mkfiles **175**
D.1 The mkfile of mk . . . 175
D.2 The mkfiles of Principia Softwarica . . . 176
D.2.1 $objtype/mkfile for the ARM . . . 176
D.2.2 mkfiles/mkfile.proto . . . 176
D.2.3 mkfiles/mkone . . . 177
D.2.4 mkfiles/mklib . . . 178
D.2.5 mkfiles/mkdirs . . . 179

E Extra Code **180**
E.1 mk/ . . . 180
E.1.1 mk/fns.h . . . 180
E.1.2 mk/mk.h . . . 182
E.1.3 mk/globals.c . . . 184
E.1.4 mk/utils.c . . . 185

E.1.5 mk/bufblock.c . . . 185
E.1.6 mk/symtab.c . . . 185
E.1.7 mk/rc.c . . . 186
E.1.8 mk/word.c . . . 186
E.1.9 mk/var.c . . . 186
E.1.10 mk/archive.c . . . 187
E.1.11 mk/match.c . . . 187
E.1.12 mk/env.c . . . 187
E.1.13 mk/parse.c . . . 188
E.1.14 mk/shprint.c . . . 188
E.1.15 mk/job.c . . . 188
E.1.16 mk/arc.c . . . 188
E.1.17 mk/rule.c . . . 189
E.1.18 mk/lex.c . . . 189
E.1.19 mk/file.c . . . 189
E.1.20 mk/run.c . . . 190
E.1.21 mk/graph.c . . . 191
E.1.22 mk/mk.c . . . 191
E.1.23 mk/recipe.c . . . 192
E.1.24 mk/varsub.c . . . 192
E.1.25 mk/dumpers.c . . . 193
E.1.26 mk/main.c . . . 193
E.1.27 mk/Posix.c . . . 193
E.1.28 mk/Plan9.c . . . 199

Glossary **201**

Index **202**

References **208**

Chapter 1

Introduction

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

1.1 Motivations

Why a build system? Because I think you are a better programmer if you fully understand how things work under the hood, and a build system is one of the tools a programmer uses the most. Indeed, it allows programmers to assemble, compile, link, test, check, package, deploy, and distribute software with one simple command ("the one command that rules them all"), from very simple programs to entire operating systems.

A build system allows to describe and maintain dependencies between files. Those dependencies are usually represented by rules, which are stored in a special configuration file, for instance, a `Makefile` with the build system Make [Fel79] (one of the most popular build systems).

Even though a build system is not as interesting as a kernel or a compiler, it is a necessary piece in the programmer's toolbox. Indeed, all programs, including kernels and compilers, rely on a build system to be built. In fact, a build system usually also relies on itself to be built, leading to bootstrapping issues similar to the ones found in compilers. Moreover, build systems contain components that are useful in many contexts, for example a job scheduler. Finally, build systems such as Make use an original approach to solve problems. Indeed, to describe dependencies, Make provides a domain-specific language (DSL). The author of Make designed this DSL to require as few syntax as possible, in order to remove as much overhead as possible for the programmer when writing rules. Moreover, this specific language, because it is restricted, because it is not as powerful as a general-purpose programming language, allows in counterpart special checks that would be impossible in a general language.

The industry certainly takes build systems seriously: Google and Facebook each developed their own—Bazel and Buck—and invest heavily in them, because at the scale of a giant monorepo every second shaved off a build is multiplied across thousands of waiting engineers.

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

- What are the fundamental concepts of a build system? What is the core algorithm behind a build system?
- What are the kinds of dependencies a build system needs to represent in order to cover all the use-cases? Can a file depend only on one other file (one-to-one dependency), or on many files (one-to-many)? Can many files depend on the same single file (many-to-one), or on many files (many-to-many)?
- What is the minimal syntax you need to describe dependencies, and to describe the commands to maintain those dependencies? If this syntax uses special symbols, how do you reference filenames containing those symbols?
- What are the mistakes a build system can detect? Can it detect ambiguous rules? Infinite rules? Can it detect cycles in dependencies?

- What kind of help can a build system offer to help debug a `Makefile`? How can you visualize the dependencies?
- How does a build system maintain dependencies efficiently? Can it compile projects incrementally? Can different parts of a project be compiled in parallel?
- How does the build system run different tasks in parallel, and how does it coordinate them? How do you write a simple job scheduler?

1.2 The Plan 9 build system: `mk`

I will explain in this book the code of the Plan 9 build system `mk` [Hum87][1], which contains about 4500 lines of code (LOC). `mk` is written entirely in C.

Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. Moreover, `mk` is the "spiritual successor" [Hum87] to Make. Indeed, it was designed in the same lab (Bell Labs), and even Stuart Feldman, the original author of Make, recommends `mk` as a better system.

`mk` is modeled after Make, like many other build systems, but `mk` is both simpler and more powerful than Make. For instance, `mk` does not suffer from the infamous `TAB` requirement in the first column from Make[2]; with `mk`, the programmer can use spaces and tabs interchangeably. Moreover, `mk` executes shell commands in *parallel*, an important improvement over the original Make as this speeds up greatly the building process on machines with multiple processors.

With one single command (`mk everything`) executed from the top directory of the Plan 9 fork used in Principia Softwarica, you can build all the Plan 9 libraries, the Plan 9 programs, the Plan 9 kernel, and build a disk image containing a Plan 9 distribution that can be installed on a Raspberry Pi[3] or booted through QEMU[4]; all of this with one single command.

1.3 Other build systems

Here are a few build systems that I considered for this book, but which I ultimately discarded:

- Make [Fel79], which from now on I will call UNIX Make, to differentiate it from its other variants, was the original Make part of early versions of UNIX. Stuart Feldman, its author, won the ACM System Software Award in 2003 for it: "there is probably no large software system in the world today that has not been processed by a version or offspring of MAKE." [5]. Indeed, UNIX Make has many variants, which are often confusingly called also Make (e.g., BSD Make, GNU Make), and which often use the same command-line program name: `make`.

  One of the latest versions of UNIX Make, for UNIX V7 in 1979[6], contains less than 2500 LOC. This is smaller than `mk`, but this version contains also far less features than `mk`. For instance, it lacks the ability to execute shell commands in parallel. This ability requires more than just a few lines of code. In fact, it led in `mk` to the complete redesign of the approach used by Make to compute dependencies. Indeed, `mk` computes a graph of dependencies *statically* when it starts, and then uses this graph to guide the building process.

[1] See http://plan9.bell-labs.com/magic/man2html/1/mk for the manual page of `mk`.
[2] See http://www.catb.org/esr/writings/taoup/html/ch15s04.html for the story behind the `TAB` requirement.
[3] https://www.raspberrypi.org/
[4] https://www.qemu.org/
[5] http://awards.acm.org/award_winners/feldman_1240498.cfm
[6] See http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/make/.

- pmake [dB88] (for *parallel make*) is a variant of UNIX Make developed at Berkeley that introduced the ability to run shell commands in parallel, the same feature that motivated the redesign of `mk` described above. It is the ancestor of the `make` shipped with the BSD systems, which is why it is also called BSD Make (and later `bmake` on NetBSD or `bsdmake` on FreeBSD). Unlike `mk` however, pmake bolts parallelism onto the original Make design instead of computing a static dependency graph, and it does so at the cost of a much larger codebase: around 21 000 LOC, more than four times the size of `mk`.

- GNU Make [Mec04][7] is probably the most popular variant of Make. It is used by almost every open source applications under Linux. GNU Make contains many extensions to the original UNIX Make, including the ability to run shell commands in parallel as in `mk` with `make -j` (`'j'` for jobs), inspired from pmake. It supports also many platforms, including old platforms such as DOS, VMS, or Amiga. However, its codebase is significantly larger: 37 000 LOC, which is almost one order of magnitude more code than `mk`.

  GNU Make follows closely the design of UNIX Make, and so inherits also its major flaws, for instance, the requirement to use `TAB` in the first column for shell commands. Where `mk` tries to generalize, unify, and reuse the services and syntax of other tools, GNU Make specializes, uses its own syntax, and adds an extra layer of complexity. For example, use of variables in GNU Make (and UNIX Make) requires parenthesis (e.g., `$(OBJS)`), which are not required when using variables in a shell (e.g., `$OBJS`). Moreover, the use of shell variables inside a `Makefile` requires an extra dollar (e.g., `$$i`). In `mk`, the syntax for variables is the same than in the shell (e.g., `$OBJS`, `$i`). Finally, `mk` replaces cryptic Make variables such as `$@` or `$^` (which do not require parenthesis for once) with clearer names such as `$target` or `$prereq`.

- Ant [HL02][8] (for Another Neat Tool) was a build system used by many Java projects. Instead of using a DSL to express dependencies, and of relying on a shell and command-line programs to maintain those dependencies, an Ant user writes dependency rules in XML in a `build.xml` configuration file. For instance, here is a rule to clean files in Ant:

  ```
  <target name="clean" <delete dir="classes"></target>
  ```

  To contrast, here is the same rule with Make:

  ```
  clean:
      rm -rf classes/
  ```

  Ant supports many XML tags to perform various tasks such as deleting files (`<delete>`), creating directories (`<mkdir>`), or calling the Java compiler (`<javac>`). The main advantage of Ant over Make is *portability*. Indeed, with Make, the command-line programs used in a `Makefile` may be specific to an operating system. For instance, the default C compiler, linker, and file utilities under Microsoft Windows are not the same than in Linux or macOS. In fact, even the shell and `make` programs are different under Linux, macOS, and Microsoft Windows. However, the portability of Ant comes at a price; its codebase is very large with more than 200 000 LOC (without the tests), which is 40 times more code than `mk`.

  The main advantage of Make (and `mk`) over Ant is *generality*. Indeed, you can call any command-line programs from the `Makefile`. Moreover, the shell, which is the language used to write commands in a `Makefile`, is an expressive language (see the Shell book [Pad18]). With Ant, if there is no XML tags for a certain task, you need to extend Ant itself. Finally, XML is an extremely verbose language; writing XML dependency rules by hand is tedious.

[7] https://www.gnu.org/software/make/
[8] http://ant.apache.org/

- CMake[9] is a cross-platform build system used in many large open source C++ projects (e.g., LLVM). It was designed, like Ant, to overcome the lack of portability of `Makefiles`. Unlike Ant, CMake acts as a frontend to Make (and other build systems). Indeed, CMake is a *meta build system*. Instead of writing `Makefiles`, the user of CMake creates `CMakeLists.txt` files containing builtin (portable) commands to compile source code. CMake then generates from those configuration files regular `Makefiles` that can be processed by Make. CMake can also generate files for IDEs such as Apple's Xcode or Microsoft's Visual Studio.

  CMake contains thousands of builtin commands, offers a graphical user interface, and supports many IDEs. However, its codebase contains more than 250 000 LOC (not including the tests). This is extremely large, especially considering the fact that CMake is just a frontend to other build systems.

- Cargo[10] is the build tool of the Rust language, and it illustrates a more recent trend: the build system and the *package manager* have become the same program. Where an `mkfile` only knows how to rebuild files already present on your disk, Cargo also resolves, downloads, and version-locks the external libraries your code depends on before building them. This fusion of building and dependency management was pioneered by Maven for Java; Cargo did not invent it but turned it into an ergonomic template—a `Cargo.toml` manifest, a committed lock file, and semantic versioning—that many languages have since copied (e.g., Go, Swift, Elixir, OCaml). The price for this extra responsibility is size: Cargo is around 100 000 LOC, twenty times the size of `mk`. Moreover, a package manager fused with the build tool is also necessarily tied to one language: Cargo builds Rust and nothing else, whereas `mk` (like Make) stays general and can drive the build of any language by orchestrating arbitrary command-line programs.

- Bazel[11] is the open source release of *Blaze* [WMW20], the build system Google uses on its giant internal monorepo, and it is built for *scale*: hermetic and reproducible builds, fine-grained caching, and the ability to distribute a single build across a fleet of remote machines. This power comes at an extreme cost in complexity; Bazel is around 640 000 LOC, more than a hundred times the size of `mk`. Facebook wrote its own near-equivalent, Buck, with around 510 000 LOC (Blaze was still a Google-internal secret when they needed it, as Bazel was only open sourced in 2015, two years after Buck). Meta has since rewritten Buck from scratch in Rust as Buck2, which remains just as enormous (around 760 000 LOC). These build systems solve real problems for codebases with millions of files, but for the kind of programs studied in Principia Softwarica they are massively over-engineered; `mk` does the essential job in 4500 LOC.

Note that the lack of portability of `Makefiles` mentioned before has been partially fixed in the last ten years. Indeed, with the availability of GNU utilities for operating systems other than Linux (e.g., through cygwin[12] for Microsoft Windows, and macports[13] or Homebrew[14] for macOS), `Makefiles` are now more portable because the same command-line programs are available on more platforms.

Figure 1.1 presents a timeline of major build systems. I think `mk` represents the best compromise for this book: it implements the essential features of a build system while still having a small and understandable codebase (4500 LOC).

1.4 Getting started

To play with `mk`, you will first need to install the Plan 9 fork used in Principia Softwarica (see https://www.principia-softwarica.org/getting-started.html). Once installed you can test `mk` under Plan 9 with the following commands:

[9] https://cmake.org/
[10] https://doc.rust-lang.org/cargo/
[11] https://bazel.build/
[12] https://www.cygwin.com/
[13] https://www.macports.org/
[14] http://brew.sh/

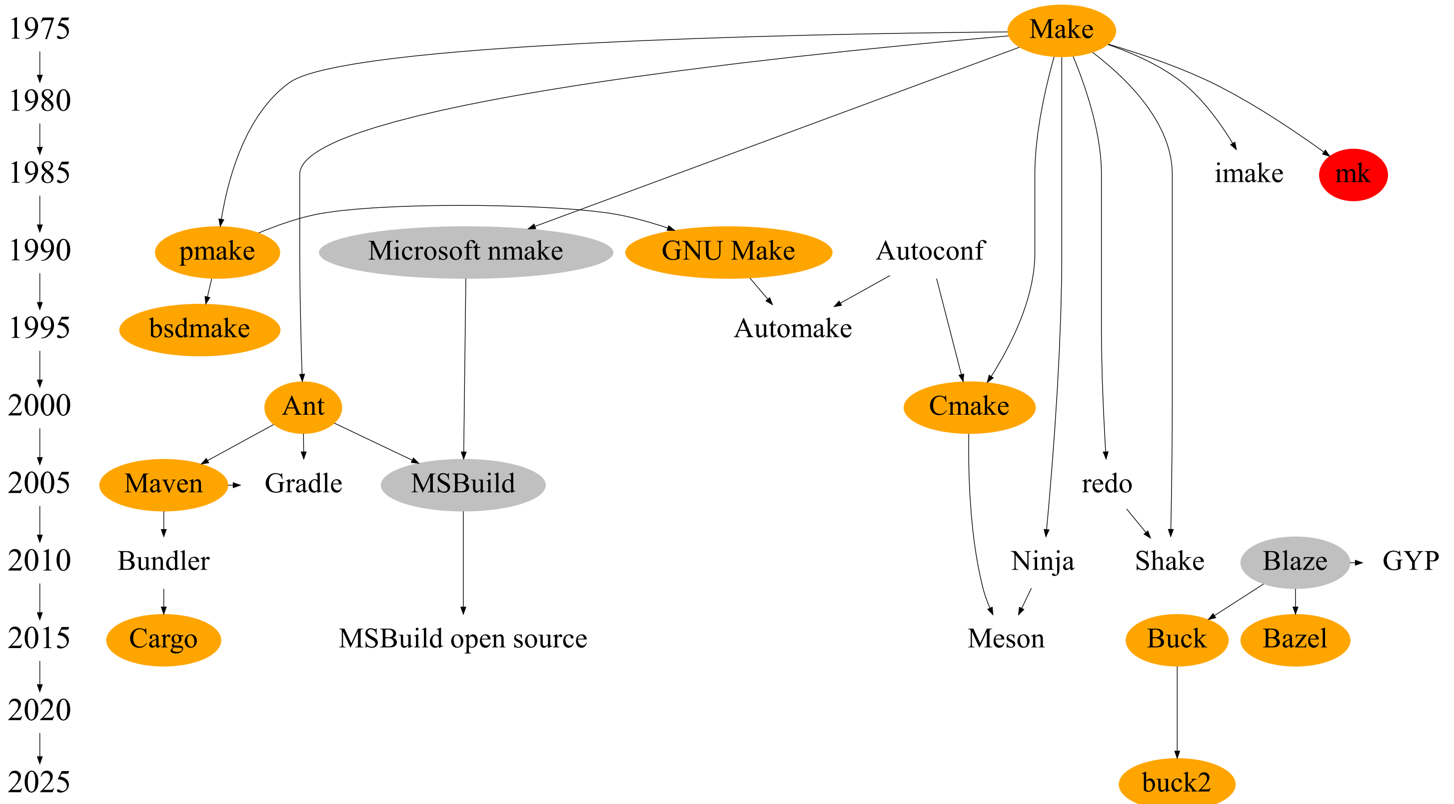


Figure 1.1: Build systems timeline

```
1   $ cd /tests/mk
2   $ mk -f hello.mk
3   5c -c hello.c
4   5c -c world.c
5   5l -o hello hello.5 world.5
6   $ mk -f hello.mk
7   mk: 'hello' is up to date
8   $ touch world.c
9   $ mk -f hello.mk
10  5c -c world.c
11  5l -o hello hello.5 world.5
12  $
```

Line 2 runs `mk` with the `-f hello.mk` argument to tell `mk` to use the rules in the `hello.mk` file instead of the default `mkfile` (`mk`'s equivalent of a `Makefile`). Line 3 through 5 are the shell commands ran by `mk` given the rules contained in `hello.mk`. Those commands compile and link a simple program called `hello` using the ARM C Compiler `5c` (see the COMPILER book [Pad16b]) and ARM linker `5l` (see the LINKER book [Pad15b]). Remember that `'.5'` is the filename extension of ARM object files under Plan 9, hence the use of `hello.5` and `world.5` in the linking command Line 5.

Line 6 re-runs `mk`, which should not recompile or relink anything because nothing has changed. Instead, `mk` should indicate that `hello` is already up to date. Line 8 modifies the date of `world.c`. This time, re-running `mk` at Line 9 should trigger the recompilation of `world.c` and relinking of `hello`. Note that `mk` should not recompile `hello.c` because `hello.5` depends only on `hello.c`, not `world.c`.

With those commands, you can see the main purpose of a build system: to maintain dependencies between files (here source files, objects, and binaries) automatically, and efficiently by running only the minimum number

of commands.

You can also use mk on Linux, macOS, or Windows through plan9port[15] or Goken9cc[16], where it is compiled natively using gcc or clang.

1.5 Requirements

Because most of this book is made of C source code, you will need a good knowledge of the C programming language [KR88] to understand it. I also assume you are already familiar with at least one build system, for instance, a variant of Make, and so are familiar with concepts such as a Makefile, a rule, a target, a prerequisite, or a shell command. If not, I suggest you to read one of the books about GNU Make [Mec04, OL96, SMS16].

If, while reading this book, you have specific questions on the interface of mk, I suggest you to consult the man page of mk at docs/man/1/mk in my Plan 9 repository. Note that the builders/docs/ directory in my Plan 9 repository contains documents describing either mk [Hum87], or the mkfiles used in Plan 9 [HF95]. Those documents are useful to understand some of the design decisions presented in this book, especially how and why mk differs from Make.

1.6 About this document

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

1.7 Copyright

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

1.8 Acknowledgments

I would like to acknowledge of course the author of mk, Andrew Hume, who wrote in some sense most of this book.

[15] https://9fans.github.io/plan9port/
[16] https://github.com/aryx/goken9cc

Chapter 2

Overview

Before showing the source code of `mk` in the following chapters, I first give an overview in this chapter of the general principles of a build system. I also quickly describe the command-line interface of `mk` and show a simple `mkfile` for a toy application. Finally, I define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Build system principles

To understand the goal of a build system, it is useful to remember how programmers were compiling projects before Make was invented. Before Make, programmers were using shell scripts to record the compiling and linking commands for a project. For instance, here is one such script called `make.rc`[1] to build the toy program mentioned in Section 1.4:

⟨*tests/mk/make.rc* 13⟩≡
```
#!/bin/rc

5c -c hello.c
5c -c world.c
5l -o hello hello.5 world.5
```

As a program grows larger, and the number of files grows, the shell-script approach becomes too inefficient[2]. Indeed, in the previous example, if only `hello.c` is modified, there is no need to recompile also `world.c`, but that is what the script would do if it was run again to rebuild the program. An alternative is to keep track in your head of all the modifications, and to recompile manually only what is necessary. However, again, as programs grow larger, and dependencies between files become more complex, it becomes difficult to remember which files need to be recompiled and what are the precise flags used to recompile or link those files.

Thus, the goal of a build system is to describe *concisely* and maintain *efficiently* dependencies between files. For a programmer, those files are C or Assembly source files, object files, and executables. For a writer, those files are Troff or TeX documents, pictures, and PDFs. A build system can be used in many contexts, not just for programming.

Note that every words in the first sentence of the previous paragraph are important. The word "concisely" led in `mk` to the creation of a domain-specific language. The word "dependencies" led to the use of a graph to represent the relationships between files. Finally, the word "efficiently" led to the use of a job scheduler to run in parallel multiple tasks. All of this will be explained in the following sections.

Note that even if in the next sections I will use examples using the syntax of `mk`, which is very close to the syntax of Make, the principles apply to most build systems.

[1] This script is using the Plan 9 shell, which is called `rc` (see the SHELL book [Pad18]), hence the use of the '`.rc`' filename extension.

[2] This approach can still be useful to bootstrap the build system itself.

2.1.1 A domain-specific language

A *domain-specific language* (DSL), as its name suggests, is a language specialized for one particular task. In the case of a build system, the task is (1) to describe file dependencies, and (2) to describe the commands to maintain those dependencies. A DSL uses a special syntax to describe more concisely than with a general-purpose language the solution to a particular problem.

The shell script `make.rc` above is already a good solution for (2). Indeed, a shell can almost be considered a DSL for running commands: it uses a special syntax for launching programs (by just typing the name of a command, without the need to call `fork()` or `exec()`), for creating pipes (with `'|'`), and for file redirection (with `'>'` or `'<'`). Thus, the main idea behind the DSL of `mk` (and Make before) was to extend slightly the shell syntax to accommodate also (1), with special constructs to express dependencies between files. Those constructs are the rule, the pattern, the variable, and the file inclusion, as explained in the next sections.

The rule

The most important construct in the DSL of a build system is the rule. A *rule* describes a relation between two or more files, for instance, the relation between an object file `hello.5` and its source `hello.c`. In `mk`'s terminology, the object file is called the *target*, and the source the *prerequisite*. A rule describes also the command to maintain the dependency between those two files, that is the shell command to generate the target from the prerequisite. In mk's terminology, this command is called the *recipe*.

Here is the syntax of a rule in `mk`'s DSL:

```
<target> : <prerequisite 1>...<prerequisite n>
    <recipe>
```

Here are the concrete rules corresponding to the script `make.rc` shown above for the toy program mentioned in Section 1.4:

⟨*tests/mk/mkfile version 1* 14⟩≡

```
# rule 1
hello.5: hello.c
    5c -c hello.c
# rule 2
world.5: world.c
    5c -c world.c
# rule 3
hello: hello.5 world.5
    5l -o hello hello.5 world.5
```

Note that a rule can have multiple prerequisites, like in the third rule above with the multiple object files. In fact, a rule can also have multiple targets, as you will see in Section 2.1.2. Moreover, the same file can be a target in one rule and a prerequisite in another rule, for instance, `hello.5` in respectively the first and third rules above.

The rules are usually stored in a special *configuration file*: an `mkfile` for `mk` (and a `Makefile` for Make). The syntax of an `mkfile` is very similar to the syntax of a shell script. `mk` even uses the same syntax for comments, which are prefixed with a `'#'`. The only syntactical addition is the use of `':'` to separate the target from the prerequisites, and the newline and indentation to separate the prerequisites from the recipe. This syntax is *minimalist*. Indeed, there is no quotes around filenames or around commands. Moreover, the different elements in the list of prerequisites are simply separated by spaces.

The semantic of a rule is also very simple. `mk` will check if the *modification time* of a target is more recent than all its prerequisites. If not, it will run the recipe, which hopefully will update the modification time of the target.

As I said before, the recipe is simply a shell command. Even though the commands in the `mkfile` above are simple, `mk` allows to use the full language of the Plan 9 shell `rc` (see the SHELL book [Pad18]), with loops, conditionals, functions, etc. Indeed, one of the design principles of `mk` was to leverage existing tools.

The shell language is said to be *embedded* inside `mk`'s DSL. This is similar to other UNIX DSLs, for example Lex [LS79] and Yacc [Joh79]. With those DSLs the programmer can use some special syntax to define respectively regexps and grammar rules; he can also use the full C language for the actions triggered when a regexp or grammar rule is recognized (See the COMPILERGENERATOR book [Pad16a]). In `mk`, the actions are not written in C but in the shell language of `rc`, and those actions are embedded not inside the definition of a regexp or of a grammar but inside the definition of a *graph*. Indeed, the targets and prerequisites are similar to the sources and destinations of *arcs* in a graph. The recipe corresponds to the *label* on an arc. In fact, as you will see in Section 2.1.2, `mk` internally uses a graph to represent the dependencies between files.

The pattern

The `mkfile` in the previous section allows to maintain efficiently dependencies between files: if only `hello.c` is modified, then `world.c` will not be recompiled by `mk`. However, the `mkfile` is not very concise. Fortunately, the first two rules are very similar: they differ only by the name of the file. This is why, to factorize rules, most build systems provide a way to use patterns inside a rule.

In `mk`, a *pattern* is a sequence of characters where the special character `'%'` can match any sequence of characters. Here is an example of a pattern that matches any C source files:

```
%.c
```

A rule using a pattern in his target is called a *meta rule* in `mk`'s terminology. Here is a better version of the `mkfile` for the same toy program:

⟨*tests/mk/mkfile version 2* 15⟩≡
```
# simple rule
hello: hello.5 world.5
    5l -o hello hello.5 world.5

# meta rule
%.5: %.c
    5c -c $stem.c
```

During the processing of the first rule above, `mk` will recognize that `hello.5` and `world.5`, which are the prerequisites, *match* the target in the second rule if the percent is set to `hello` or `world`. `mk` will then *instantiate* the meta rule to generate a specific rule for `hello.5`, and another one for `world.5`. The percent in the prerequisite is then *substituted* by the matched string in the target (`hello` or `world`), and the special variable `$stem` can be used from the shell command to access the matched string.

The use of `'%'` to match any sequence of characters is similar to the use of `'.*'` in a *regular expression* (see the LIBCORE book [Pad16c]), or the use of `'*'` in shell *globbing* (see the SHELL book [Pad18]). In fact, `mk` supports also meta rules using regular expressions, as explained in Section 11.1. However, in most cases, patterns using `'%'` are expressive enough and simpler to write.

GNU Make supports meta rules with percents, but not with regular expressions. UNIX Make and GNU Make support also *suffix rules* (e.g., `.5.o: ...`). However, suffix rules are less intuitive to write and less expressive than meta rules. This is why they are not supported by `mk`.

The variable

In the previous section, I have shown the use of a variable in a recipe (`$stem`). This variable was set by `mk`. Most build systems offer a way to define and use your own variables to factorize things. In `mk`, those variables can contain a list of *words* (just like in `rc`), which can correspond to anything: filenames, compiler flags, etc. Here is an example of a variable containing two filenames:

```
OBJS=hello.5 world.5
```

Here is a slightly different version of the previous `mkfile` using a variable:

⟨*tests/mk/mkfile version 3* 16⟩≡

```
OBJS=hello.5 world.5

hello: $OBJS
    5l -o $target $OBJS

%.5: %.c
    5c -c $stem.c
```

One of the design principles of `mk` was to leverage existing tools, but also existing syntax. Thus, the syntax to define and use variables in `mk` is exactly the same than in the shell. This syntax is again minimalist: to define a variable, type a variable name, followed by an equal sign, followed by a list of words simply separated by space. There is no braces, brackets, quotes, commas, types, or semicolons like in other languages (e.g., `char* OBJS[] = {"hello.5", "world.5"};` in C). The next newline marks the end of the variable definition[3]. To use a variable, prefix the variable name with the dollar sign (e.g., `$OBJS` in the rule for `hello` above).

Note that `mk`, like the shell `rc`, treats the content of a variable as a list. `mk` offers also a special syntax to concatenate lists by simply juxtaposing a variable with other elements or other variables, as in the following example:

```
X= b c d
# Y will contain a b c d e f
Y=a $X d e f
```

`mk` offers also a special syntax to transform lists, as explained in Section 11.4

Once a variable is defined, you can use it in other variable definitions, or in a rule (in the target, in the prerequisites, or even in the recipe). You can also use variables in patterns in meta rules. Moreover, `mk` imports variables from the environment, so you can also use variables such as `$HOME` in your `mkfile`.

Note that the term "variable" in the context of `mk` is slightly misleading. Indeed, in `mk` variables are actually constants. Variable definitions are more *binding* definitions. As we will see soon, `mk` needs to know statically the value of a variable to be able to compute the graph of dependencies.

File inclusion

The last construct found in most build systems is the file inclusion. In `mk`, a *file inclusion* is an instruction in the `mkfile`, starting with `'<'`. You can include files to load the rules and variables defined in those files, which can themselves include other files, recursively. Here is an example of a file inclusion:

```
</$objtype/mkfile
```

The effect is similar to a `#include` in C. Note that the filename can contain variables defined previously in the `mkfile` or in the environment. Here, `$objtype` is a special Plan 9 environment variable containing the type of architecture you are targeting (e.g., `arm`, `386`). You can then define for each architecture a specific `mkfile` with variables such as `$CC` or `$LD` containing the name of the corresponding compiler and linker. Appendix D.2.1 presents such an `mkfile` for the ARM architecture. It is good practice to include `/$objtype/mkfile` at the beginning of an `mkfile` for better portability. Note that again `mk` reuses the syntax of other tools by using the `'<'` symbol used for input redirection in the shell.

Just like the rule, the pattern, and the variable, a file inclusion allows to factorize things. Indeed, you can store a library of meta rules in a separate file (e.g., `/shared/mkgeneric`) and reuse this file in multiple projects.

[3] You can also *escape* newlines, to spread variable definitions over multiple lines, as explained in Section 5.1.1.

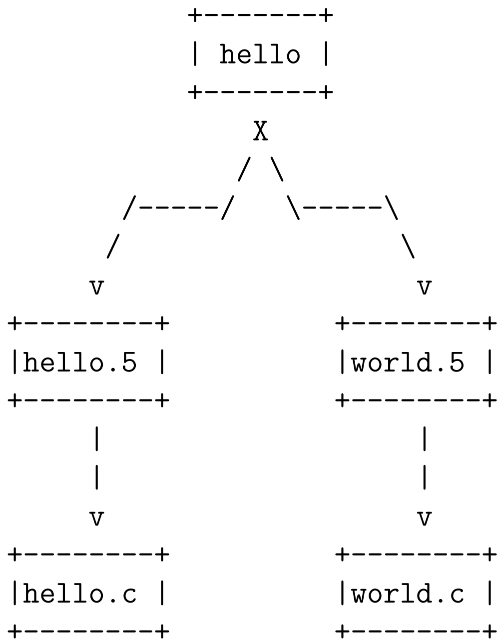


Figure 2.1: Graph of dependencies for `hello`.

In fact, by combining variables and file inclusion, you can also have a library of simple rules shared by multiple projects, as shown by the example below:

⟨*tests/mk/mkfile version 4* 17a⟩≡
```
OBJS=hello.5 world.5
PROG=hello

</shared/mkone
```

⟨*/shared/mkone* 17b⟩≡
```
$PROG: $OBJS
    5l -o $PROG $OBJS
%.5: %.c
    5c -c $stem.c
```

`mk` offers a few more constructs, but those I just presented are the main constructs of a build system. See Chapter 11 for the list of advanced constructs supported by `mk`.

2.1.2 A graph of dependencies

The pattern, the variable, and the file inclusion are nice features, but they are not the essence of a build system; rules are the essence of a build system. Indeed, once the build system has *loaded* included files, *expanded* variables, and *substituted* patterns, what remains is a set of rules with concrete filenames as targets and prerequisites. Moreover, as I mentioned briefly in Section 2.1.1, the rules in a build system define essentially the nodes and arcs of a graph. Thus, the essence of a build system is also this *graph of dependencies*.

In the next sections, I will present different examples of graph of dependencies, with increasing complexity.

A simple tree

Figure 2.1 presents the graph of dependencies for the `hello` program of Section 1.4. In this example, the graph is simply a *tree*. The *nodes* in the graph of dependencies correspond to concrete filenames. The *arcs* represent

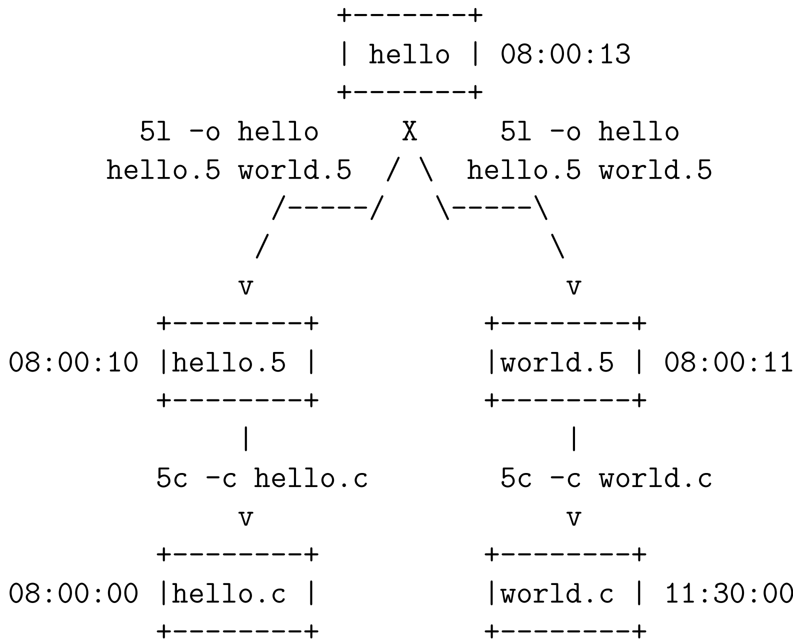


Figure 2.2: Graph of dependencies for `hello`, with labels.

dependencies between files. For instance, `world.5` depends on `world.c`, hence the arc between the two nodes in Figure 2.1. Note that a rule with two prerequisites leads to the creation of two arcs in the graph of dependencies.

Given the `mkfile` in Section 2.1.1, `mk` will internally build the graph of dependencies of Figure 2.1. The use of either simple rules or meta rules to describe the dependencies (or both) will result in the same graph. Once `mk` matched and substituted patterns, what remains will be a set of nodes with concrete filenanes.

A labeled tree

Figure 2.2 presents also the graph of dependencies for the `hello` program where nodes ands arcs are annotated also with *labels*. Arcs are labeled with a recipe whereas nodes are labeled with the modification time of the file they represent. If the file does not exist, the modification time is set to zero. In Figure 2.2, the day, month, and year of the modification time of the files are omitted for simplification purpose; only the hours, minutes, and seconds are shown. I assume all the files were modified in the same day.

The scenario that led to the modification times in Figure 2.2 is as follows: The programmer of the `hello` program finished modifying `hello.c` and `world.c` at 8am. He then ran `mk` to build the program. `mk` finished compiling `hello.5` at 8am and 10 seconds, and `world.5` at 8am and 11 seconds. `mk` finished linking `hello` at 8am and 13 seconds. Finally, the programmer modified `world.c` at 11:30am and stopped. At this point, running `mk` should recompile `world.c` (and relink `hello`), but should not recompile `hello.c`.

Depth-first search

Once the graph of dependencies is built, the main algorithm behind `mk` is to perform a *depth-first search* (DFS) traversal on this graph. Figure 2.3 presents the order in which the DFS visits the nodes on the previous graph. The DFS starts from the *root* (step 1 at the top of Figure 2.3) and goes as deep as possible along a *branch* (steps 2 and 3). When it reaches a *leaf*, for instance `hello.c` (step 3), the algorithm just checks whether the file exists. If the file does not exist, then `mk` should report an error because there is no instruction on how to generate this file (there is no prerequisite connected to the node and so no recipe). If the file exists, then the algorithm can continue and the DFS can backtrack by going back up in the tree (step 4). At this point, the algorithm checks whether the modification time of the node is more recent than *all* its prerequisites, which have all been already visited by the DFS by now. If the node is more recent, for instance, `hello.5` is more recent than `hello.c` in

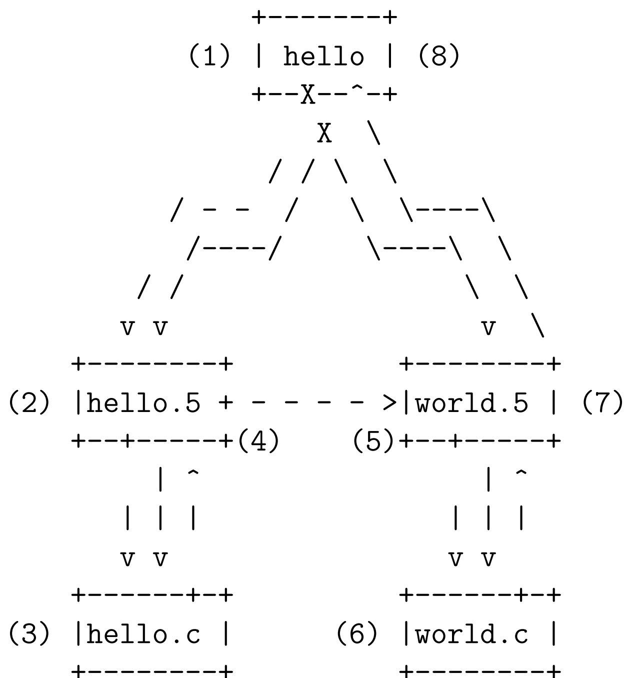


Figure 2.3: Depth-first search traversal of the graph dependencies for `hello`.

Figure 2.2, then there is nothing to do but to continue the DFS (steps 5 and 6). If the node is older than one or more of its prerequisites, for instance, `world.5` is older than `world.c` in Figure 2.2, then the algorithm should run the associated recipe in a separate shell process. Hopefully, running this process will modify the time of the node. This will in turn trigger the recompilation of all the ancestors of the node while going back up to the root of the graph (step 8 in Figure 2.3), because the ancestors should now be older than this newly-generated file.

The choice of DFS over BFS (Breath-first search) is not arbitrary. A BFS traversal would visit the root first, then its children, then grandchildren—but you cannot decide whether the root needs rebuilding until you know the status of every descendant. DFS naturally solves this: by going deep first, it reaches the leaves (source files), confirms they exist, then backtracks, checking timestamps at each level. By the time the algorithm returns to a node, all its prerequisites have already been visited and their status is known.

A direct acyclic graph

most build systems support a more general form of graphs: *direct acyclic graphs* (DAGs), where the same node can be referenced multiple times from different branches. Figure 2.4 presents such as graph for the same `hello` program, but where an additional header file, `common.h`, is shared and included by the two source files.

Note that even though `common.h` is included by `hello.c` and `world.c`, there is no arc between those files in the graph of dependencies. Indeed, modifying `common.h` should not cause the regeneration of `hello.c` or `world.c`. However, the modification of `common.h` should cause the regeneration of the object files `hello.5` and `world.5`, hence the arcs from those files to `common.h` in Figure 2.4. Indeed, the header file may define data structures that have an impact on the object code generation, hence the two arcs from the object files to the header file.

There are multiple situations where the same file can be referenced multiple times in the graph of dependencies: multiple executables may depend on and reuse the same library, multiple libraries may use the same object file, multiple object files may depend on the same header file, etc. Those shared files add arcs in the graph. However, the graph must remain acyclic. Indeed, a file can not depend on itself, directly or indirectly[4].

[4] Section 6.6.1 presents the code to check for cycles in the graph of dependencies.

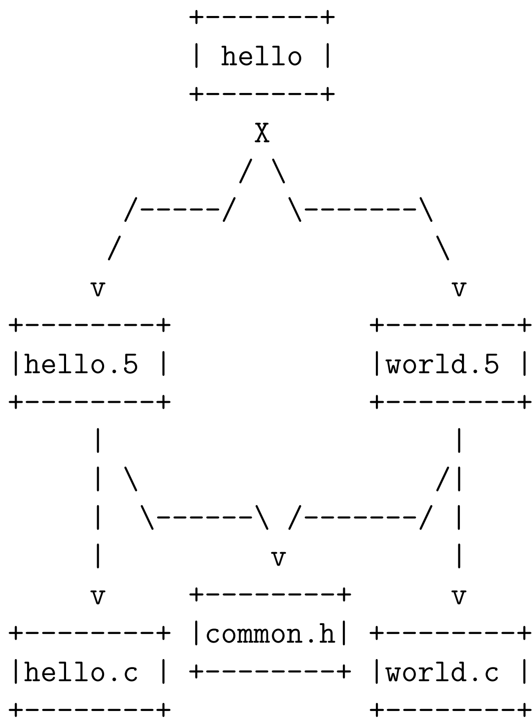


Figure 2.4: Graph of dependencies for `hello` with `common.h`.

There are multiple ways to add the dependency to `common.h` from `hello.5` (and `world.5`) in the `mkfiles` of Section 2.1.1:

1. You can add `common.h` in the list of prerequisites in the rule for `hello.5`:

```
hello.5: hello.c   common.h
    5c -c hello.c
```

   However, this approach does not work well when the rule to compile `hello.c` is a meta-rule such as `%.5: %.c`. Indeed, each source file has its own header file dependencies, which are impossible to factorize in a single meta-rule.

2. You can add a separate rule using the same target and the same recipe:

```
hello.5: common.h
    5c -c hello.c
```

   It is important to impose to have the same recipe. If the recipe was different, `mk` would be confronted with an *ambiguity* when both the source file `hello.c` and the header `common.h` are modified: which recipe to choose to update the target `hello.5`?[5] However, it is difficult for `mk` to check whether the recipe of a meta-rule such as `5c -c $stem.c` is equivalent to the recipe of a simple rule such as `5c -c hello.c`.

3. You can add a separate rule using the same target but without any recipe:

```
hello.5: common.h
```

[5] Section 6.6.3 presents the code to check for the presence of ambiguities.

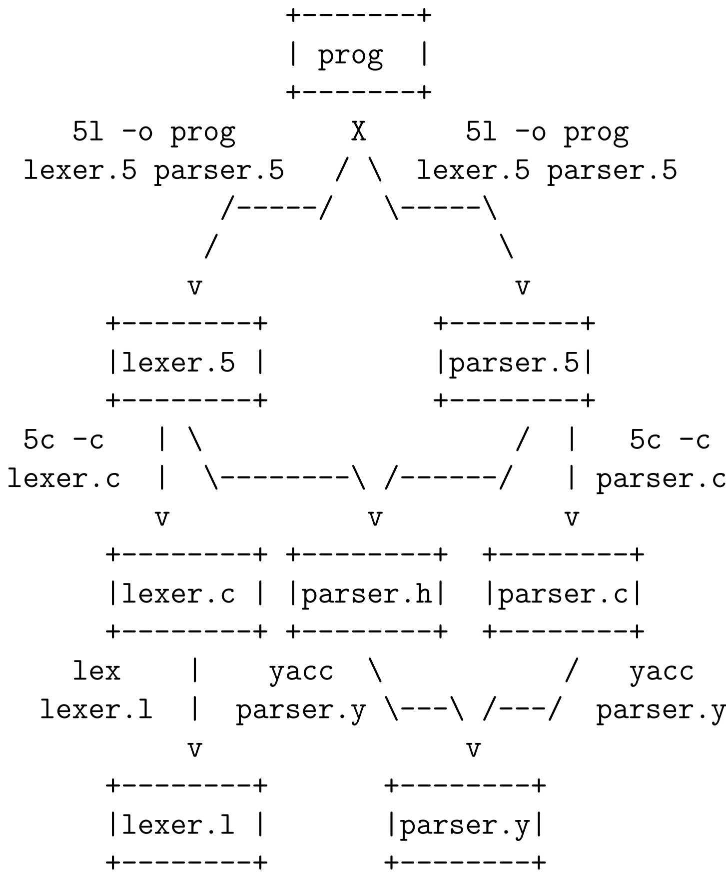


Figure 2.5: Graph with many-to-one dependencies.

This works if the build system imposes that there must be another single rule, called the *master rule*, with the same target but including a recipe. In that case, there is no ambiguity and no need to check if two recipes are equivalent.

`mk` supports (1) and (3) but not (2). (3) is more convenient for the programmer because it works well with meta-rules. Moreover, when combined with file inclusion, (3) allows to leverage programs that automatically extract header dependencies from source files (e.g., `gcc -MM` for C files, `ocamldep` for OCaml files). Indeed, the output of such programs can simply be redirected in a `.depend` file that can be included from the `mkfile` (hence the need of commands such as `make depend` in many projects such as the Linux kernel).

Many-to-one dependencies

In Figure 2.5, two object files, `lexer.5` and `parser.5`, depend on the same file: `parser.h`. They also depend on other files (`lexer.c` and `parser.c`) and have different recipes (`5c -c lexer.c` and `5c -c parser.c`). This is similar to the situation depicted by Figure 2.4 with the shared file `common.h`. However, in Figure 2.5, two files, `parser.h` and `parser.c`, depend also exclusively on the same file, `parser.y`, with the same recipe (`yacc parser.y`). This last file is a Yacc [Joh79] grammar file. The `yacc` program generates both a header file (`.h`) and a source file (`.c`) from a single grammar file (`.y`). There are multiple ways to express this *many-to-one* dependency:

1. You could create two separate rules for each target:

```
parser.h: parser.y
    yacc parser.y
```

```
parser.c: parser.y
    yacc parser.y
```

2. You could create a single rule with *multiple targets*:

```
parser.h parser.c: parser.y
    yacc parser.y
```

However, the semantics for (1) and (2) are different. Indeed, with (1), if you modify `parser.y`, then `mk` will create two shell processes and execute two times the `yacc` command, which is useless (and could even be incorrect if the two commands are run in parallel and the writes on the same file are intertwined). With (2), it will create a single process and execute only one time the `yacc` command.

The use of multiple targets in one rule has implications on the DFS traversal of the graph of dependencies. Indeed, in Figure 2.5, once the DFS has processed `parser.y`, backtracked on `parser.h`, and ran the recipe to update `parser.h`, it is important that the algorithm adjusts the modification time of both the `parser.h` and `parser.c` nodes. If only the `parser.h` node is updated, `mk` would run another time the `yacc` command when the DFS reaches the `parser.c` node with an obsolete modification time. This is why, as you will see later in Section 3.3.4, the arc from `parser.h` to `parser.y` contains also a reference to the `parser.c` node (via an `alltargets` field).

2.1.3 A job scheduler

In the previous sections I have described the main features of the DSL of a build system, the underlying representation of dependencies in a build system, as well as the basic algorithm behind a build system (the DFS). I will now focus on the efficiency of a build system.

A build system maintains dependencies between files efficiently firstly by being an *incremental* program. Indeed, if you modify only one source file, the build system will recompile and relink only what is necessary. This is made possible by comparing the modification times of nodes in the graph of dependencies. In fact, this graph enables also the build system to be more efficient by running recipes in *parallel*. Indeed, with a graph, it is easy to detect whether two commands can be run in parallel when they belong to two independent branches in the graph. For instance, in Figure 2.5, the recipes with the `lex` and `yacc` commands can be run in parallel. However, if only the `lex` recipe finished, it is not possible to run `5c -c lexer.c` in parallel with `yacc` because there is an arc between `lexer.5` and `parser.h` in the graph; you must also wait for the `yacc` recipe to finish.

To run recipes in parallel, a build system should not wait during the DFS that a shell process finishes executing a recipe. This is why, during the DFS, `mk` adds instead the recipe in a *queue* and continues the DFS. Each element of this queue contains, in addition to the recipe, a pointer to the target node (or target nodes) associated with the recipe. `mk` can add multiple recipes in the queue during the DFS, and can then execute in parallel those recipes once the DFS finished. The recipe and associated target node(s) stored in the queue is called a *job* in `mk`'s terminology. The queue is also called the *job queue*. Thus, `mk` is also a *job scheduler*.

The use of a job queue has implications on the graph and the DFS. Indeed, in Figure 2.3, if the job to regenerate `world.5` from `world.c` is enqueued at step 7 (instead of being executed synchronously), the modification time of the `world.5` node will not be updated directly. Then, when the DFS backtracks on the root node at step 8, the modification time of the root node may still be more recent than all its prerequisites (as shown in Figure 2.2), so `mk` will not detect that it needs to re-link too `hello`. However, `mk` should not stop there and should *not* declare that `hello` is up-to-date. Once the job to generate `world.5` has finished, `hello` will not be up-to-date. This is why the modification time of nodes associated with a job should be marked specially in the graph (as `BeingMade` as explained below).

To keep track of the nodes involved in a job, `mk` uses an extra label on each node to indicate the building status of the node: `NotMade`, `Made`, and `BeingMade`, as shown in Figure 2.6. The algorithm behind `mk`, exposed

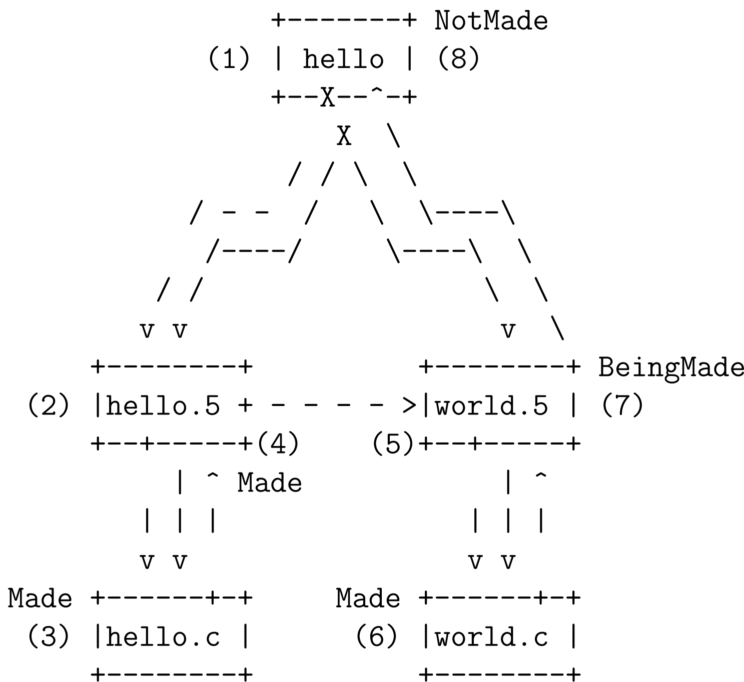


Figure 2.6: Graph of dependencies with building-status labels.

previously in Section 2.1.2, is modified as follows. After the graph is built, every status labels in every nodes is set to `NotMade` (by `clrmade()`[94a]). During the DFS, if the algorithm reaches a leaf containing an existing file, the node is marked as `Made` (e.g., `hello.c` at step 3, and `world.c` at step 6 in Figure 2.6). During backtracking, the algorithm will use different marks depending on the situation:

- If the node is more recent than all its prerequisite nodes, and all those nodes are marked as `Made`, then this node is also marked as `Made` (e.g., `hello.5` at step 4 in Figure 2.6).
- If the node is older than one or more of its prerequisites, and all the prerequisites are marked as `Made`, then a new job will be enqueued and this node is marked as `BeingMade` (e.g., `world.5` at step 7 in Figure 2.6).
- If the node is older or more recent, but one of its prerequisites is marked as `BeingMade`, then the status should be kept as `NotMade`.

`mk` will run the DFS in a loop multiple times until the root node is marked as `Made`. During each of those loops, of those *waves*, the DFS will find new jobs to run in parallel.

2.2 `mk` command-line interface

The command-line interface of `mk` is very simple: just go in a directory and type `mk`[6]. However, this assumes the directory contains a file named `mkfile`. Moreover, it assumes the first target in this `mkfile` is the target you want to build. To change the default behavior, you can use the `-f` flag, as shown in Section 1.4, to specify another configuration file. Finally, you can change the default target by specifying a target from the command line (e.g., `mk hello.5`). In fact, you can even give a list of targets on the command line.

`mk` supports also a few extra options to provide advanced features or to help debug `mk` itself. I will present gradually those options in this book. Here is the full command-line interface of `mk`:

[6] This interface is similar to the one in Make, except `mk` is even shorter to type than `make`, which is useful as `mk` is a command you will type a lot (the two letters are even next to each other on a QWERTY keyboard).

```
$ mk -help
Usage: mk [-f file] [-n] [-a] [-e] [-t] [-k] [-i] [-d[egp]] [targets ...]
```

2.3 hello.mk

Here is finally the content of the hello.mk file mentioned in Section 1.4:

⟨*tests/mk/hello.mk* 24⟩≡
```
OBJS=hello.5 world.5
CFLAGS=
LDFLAGS=

hello: $OBJS
 5l $LDFLAGS -o $target $prereq

%.5: %.c
 5c $CFLAGS -c $stem.c
```

This file is named hello.mk to illustrate the -f command-line flag of mk, but a common practice is to name mk's configuration file mkfile instead.

I have described most of the features used in hello.mk in Section 2.1.1, so I will not repeat the explanations here. The only new feature is the use of the *special variables* $target and $prereq. Those variables are set by mk in the environment of the shell process executing the recipe. As their names suggest, they contain respectively the name of the target and the list of prerequisites of the rule in which they occur.

The compilation and linking flags ($CFLAGS and $LDFLAGS) are set to an empty list in the example above, but those flags can be *overriden* from the command-line. Indeed, mk can take a list of variable definitions as arguments (e.g., mk CFLAGS=-g). Those definitions override any definition contained in the mkfile (or in files included from the mkfile). This is convenient because you can simply cross-compile a project under Plan 9 by overriding the definition of $objtype from the command line (e.g., mk objtype=arm on a machine where $objtype is by default set to 386).

For more examples of mkfiles, notably the mkfile of the mk project itself, see Appendix D. The examples in this chapter were used just to illustrate the main features of mk.

2.4 Code organization

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

2.5 Software architecture

Figure 2.7 describes the main control flow of mk, whereas Figure 2.8 describes the main data flow of mk. The main steps of the building pipeline of mk are as follows:

1. *Parse* the mkfile (via parse()[51e]) to extract the rules and meta rules in the file
2. *Build* the graph of dependencies (via graph()[76]) for a specific target given the rules extracted previously
3. *Find* outdated files in the graph (via work()[95a])
4. *Schedule* jobs (via sched()[103e]) that will run the shell recipes (via execsh()[193c]) to update the outdated files

```
              +------+
              | main |
              +------+
                  |
     +------------+------------+
     v                         v
 +--------+                +--------+
 | parse  |                |   mk   |
 +--------+                +--------+
     |                         |
     v             +-----------+
+---------+        v           +----------------+
|assline  |   +---------+      |                |
+---------+   |  graph  |      v                v
              +---------+ +---------+      +---------+
                   |      |  work   |      | waitup  |
             +-----+      +---------+      +---------+
             v                 |                |
       +----------+     +------+-----+      +----+-+
       |applyrules|     |           |      |      v
       +----------+     v           v      | +---------+
                   +---------+ +---------+ | | update  |
                   |outofdate| |dorecipe | | +---------+
                   +---------+ +----+----+ |
                                    |      |
                               +----v----+ |
                               |   run   | |
                               +------+--+ |
                                      |    |
                                   +--v----v-+
                                   |  sched  |
                                   +----+----+
                                        |
                                   +----v----+
                                   | execsh  |
                                   +---------+
```

Figure 2.7: Control flow diagram of mk.

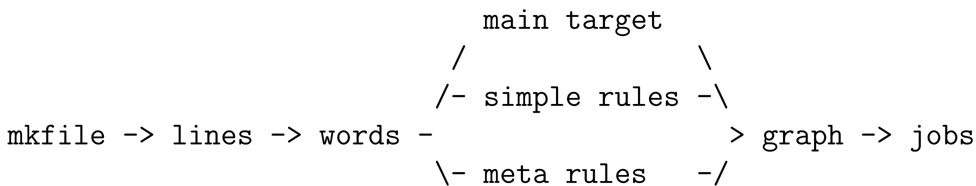


Figure 2.8: Data flow diagram of mk.

Function	Ch.	File	Entities	LOC
data structures and constants	3	`mk.h`	`Symtab`[28] `Word`[32b] `Rule`[34] `Node`[39d] `Arc`[40f] `Job`[41d]	246
symbol table and cache	3	`symtab.c`	`hash`[29b] `symlook()`[30a] `symtraverse()`[31a]	59
variables	3	`var.c`	`setvar()`[31b]	19
list of strings (words)	3	`word.c`	`newword()`[32d] `wtos()`[33d]	65
globals	3	`globals.c`	`rules`[35b] `metarules`[35e] `jobs`[42c]	23
adding rules	3	`rule.c`	`addrule()`[36b]	106
function prototypes	3	`fns.h`		91
entry point	4	`main.c`	`main()`[45a]	228
lexer	5	`lex.c`	`assline()`[53b] `nextrune()`[54a]	130
parser	5	`parse.c`	`parse()`[51e] `rhead()`[57b] `rbody()`[69a]	318
escaping methods for `rc`	5	`rc.c`	`escapetoken()`[56b] `charin()`[58c] `squote()`[59b]	166
parsing and expanding variables	5	`varsub.c`	`stow()`[60b] `varsub()`[63d] `varname()`[64d]	354
building and checking the graph	6	`graph.c`	`graph()`[76] `applyrules()`[77a] `cyclechk()`[84b]	311
pattern matching and substituting	6	`match.c`	`match()`[80a] `subst()`[80c]	54
file and time management	6	`file.c`	`timeof()`[82d]	78
finding outdated files in the graph	7	`mk.c`	`mk()`[92] `work()`[95a] `outofdate()`[96c] `update()`[108b]	247
constructing a job	7	`recipe.c`	`dorecipe()`[97e]	146
scheduling jobs	8	`run.c`	`run()`[101a] `RunEvent`[102c] `sched()`[103e] `waitup()`[107c]	316
shell environment	9	`env.c`	`buildenv()`[117c] `shellenv`[112b] `envinsert()`[113a]	149
printing shell commands	10	`shprint.c`	`shprint()`[121b]	86
handling archives (libraries)	11	`archive.c`	`atimeof()`[150g]	152
dumpers	A	`dumpers.c`	`dumpv()`[166d] `dumpr()`[166c] `dumpn()`[167b]	76
memory management	C	`utils.c`	`Malloc()`[171a] `Realloc()`[171b]	26
string buffer management	C	`bufblock.c`	`newbuf()`[172d] `insert()`[173a]	76
Plan 9 host-specific code	E	`Plan9.c`	`execsh()` `bulkmtime()`[158e] `notifyf()`[193c]	443
POSIX host-specific code	E	`Posix.c`	`execsh()` `readenv()` `xwaitfor()`[193c]	315
Total				4280

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

Starting from the top of Figure 2.7, the function `main()`[45a], after some basic command-line processing and initializations, calls `parse()`, with the file to parse as an argument (by default `mkfile`, unless you specified another filename with the `-f` command-line flag). `parse()` first calls the lexer to assemble a line (via `assline()`[53b]), which is then split in words, which are then analyzed to populate important globals such as `rules`[35b] and `metarules`[35e], which contain the lists of extracted rules. `parse()` also sets the global `target1`[49b] with the name of the first target found in the `mkfile`, unless you gave a specific target on the command line (e.g., with `mk hello.5`). Finally, `parse()` updates and uses a global symbol table containing the values for the variables defined in the `mkfile` and in the environment.

After the rules have been extracted, `main()` calls `mk()`[92] (at the top right in Figure 2.7) with the name of a target as an argument (the name stored in `target1` by default). `mk()` then calls `graph()` to build the graph of dependencies for this target given the rules and metarules extracted during parsing. This graph contains nodes and arcs, as explained in Section 2.1.2. The nodes correspond to concrete files (e.g., `hello.5`, `hello.c`), and the arcs connect two nodes when a node depends on another node (e.g., `hello.5` is connected to `hello.c`). Those arcs are also labeled with the rule containing the recipe to generate the target node.

`graph()` works by first creating a node for the target parameter, called the root of the graph, and by then calling `applyrules()`[77a] on this node. `applyrules()` then finds a rule or meta rule with the node as a target, and creates new nodes for all the prerequisites found in the rule. It then calls recursively `applyrules()` on those new nodes. Note that as opposed to Make, in `mk` the graph of dependencies is computed statically once and for all at the very beginning.

`mk()` then calls `work()` to find outdated files in the graph. Starting from the root, `work()` performs a depth-first search and goes down recursively in the graph to find nodes corresponding to inexistent files, or to files that are older than the files in the nodes they are connected to. Once it found such a node, `work()` calls `dorecipe()`[97e] with the outdated node as a parameter. `dorecipe()` then finds the arc containing the master rule with the recipe to regenerate the file in the node. `dorecipe()` then calls `run()`[101a] (at the bottom of Figure 2.7) to add in a queue the job to run the recipe. `run()` possibly calls `sched()` to schedule the job if there was a free processor to run the job in parallel. `sched()` then calls `execsh()` to fork and execute in a shell the recipe.

`mk()` calls `work()` in a loop, to schedule jobs in parallel until the root node is up-to-date (`MADE`[40d]). However, during those loops, `work()` may not be able to schedule any job. Indeed, all the processors may already be in use, or certain jobs may not be able to start until other jobs are finished. This is why `mk()` also calls sometimes `waitup()`[107c] (at the right in Figure 2.7) to wait for those jobs to finish. Once a job is finished, `waitup()` calls `update()`[108b] to update the node in the graph associated with the job, and `sched()` to schedule another job.

2.6 Book structure

You now have enough background to understand the source code of `mk`. The rest of the book is organized as follows. I will start by describing the core data structures of `mk` in Chapter 3. Then, I will use a top-down approach, starting with Chapter 4 with the description of `main()`[45a] and the initialization of `mk`. The following chapters will describe the main components of the building pipeline: Chapter 5 will present the code to parse an `mkfile`, Chapter 6 the code to build the graph of dependencies, Chapter 7 the code to find outdated files in the graph, Chapter 8 the code to schedule jobs, and finally Chapter 9 the code to communicate with the shell through the environment. In Chapter 10, I will present code to help you debug and profile your `mkfile`. Chapter 11 presents advanced features of `mk` that I did not present before to simplify the explanations, for instance, rule attributes. Finally, Chapter 12 concludes and gives pointers to other books in the Principia Softwarica series.

Some appendices present the code of non-functional properties: code to help debug `mk` itself in Appendix A and code to profile `mk` itself in Appendix B. Appendix C contains the code of utility functions used by `mk` but that are not specific to `mk` (e.g., a library to manage string buffers). Finally, Appendix D presents examples of `mkfile`s.

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

In this chapter, I will present the core data structures of `mk`: the symbol table (containing among other things the value of variables), the list of rules and meta rules, the graph of dependencies, and the description of a job. All those data structures are defined in the `mk.h`[182d] header file.

3.1 Symbol table

`mk` uses internally a *symbol table* to keep track of different things: the value of variables, the set of rules associated with a target, the modification time of a file in the graph of dependencies, etc.

3.1.1 Symtab

The structure below represents a symbol (e.g., a variable, a target, a file) and its property. It essentially associates a *key* to a *value*:

⟨*struct* `Symtab` 28⟩≡ (182d)

```
struct Symtab
{
    // the key: (name x space)

    // ref_own<string>
    char  *name;
    // enum<Namespace>, the ``namespace''
    short  space;

    // the value (generic)

    union{
        void* ptr;
        uintptr value;
    } u;

    // Extra
    ⟨Symtab extra fields 29d⟩
};
```

Uses `__anon_struct_2` 28.

Because the same string can denote a target, a file, or a variable, the key in `Symtab` is a pair made of a string and an enumeration constant called a *namespace* (stored in `Symtab.space`). The first namespace is the one for variables:

⟨*enum* `Namespace` 29a⟩≡ (182d)
```
enum Namespace {
    S_VAR, /* variable -> value */ // value is a list of words
    ⟨Sxxx cases 31c⟩
};
```

To look for the value of the variable `$OBJS`, you must use the key `("OBJS", S_VAR)`. I will gradually describe the other namespaces in the following chapters.

The value associated to a key can also be different things: a list of words for a variable, an integer representing a time for the modification time of a file, etc. Thus, the value in `Symtab` is a union containing either an integer (in `Symtab.u.value`) or a generic pointer (in `Symtab.u.ptr`).

3.1.2 hash

The symbol table itself is stored in a global hash table called `hash`. It makes sense to use a global because the symbol table will be accessed by different components of the building pipeline.

One way to implement a hash table in C is to use a big array of lists, also known as an array of *buckets*:

⟨*global* `hash` 29b⟩≡ (185c)
```
// hash<(string * enum<Namespace>), 'a> (next = Symtab.next in bucket)
static Symtab *hash[NHASH];
```
Uses `NHASH-1` 29c.

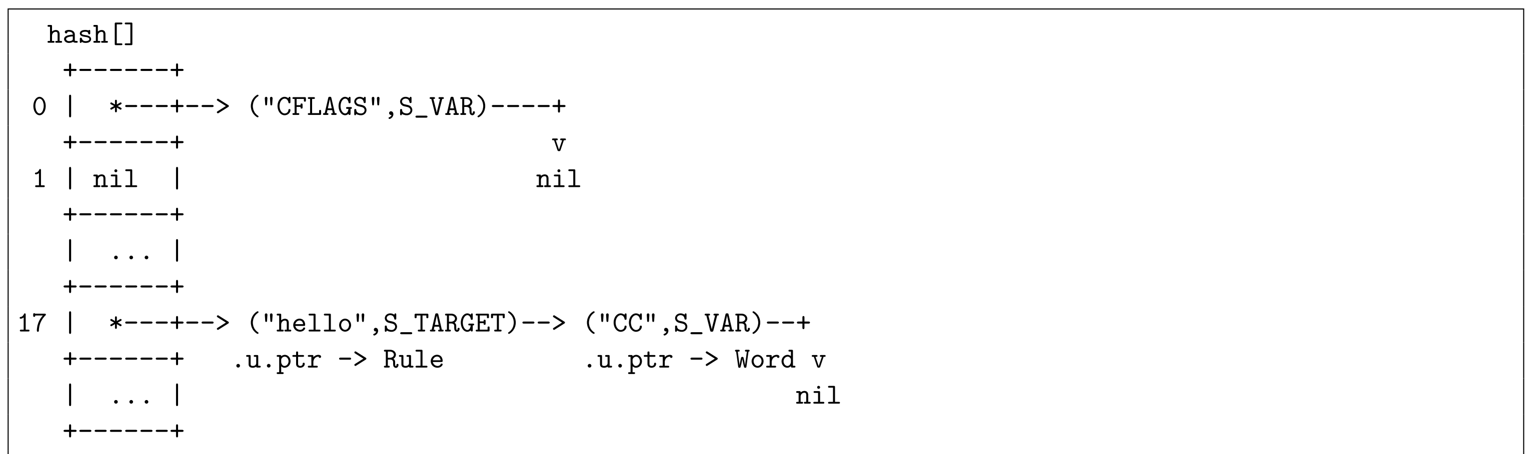
⟨*constant* `NHASH` 29c⟩≡ (185c)
```
#define NHASH 4099
```

One way to implement a list of something in C is to embed in this something a `next` field pointing to the next element in the list:

⟨`Symtab` *extra fields* 29d⟩≡ (28)
```
// list<ref_own<Symtab>> (head = hash)
struct Symtab *next;
```
Uses `Symtab` 28.

The end of the list is represented by the null pointer (`nil` in Plan 9).

A single bucket can hold entries from different namespaces, because the namespace is folded into the hash key (see the code of `symlook()`[30a] further below). Two symbols that hash to the same slot may belong to entirely different namespaces and still chain together, as in the diagram below where a variable (`S_VAR`) and a target (`S_TARGET`) coexist in bucket 17:



The main interface to the symbol table is the function `symlook()` below, which internally uses the global `hash`[29b]. `symlook()` takes a symbol name and a namespace, forming a full key, and returns the `Symtab`[28] in the symbol table `hash` associated with this key.

⟨*function* symlook 30a⟩≡ (185c)
```
Symtab*
symlook(char *sym, int space, void *install)
{
    Symtab *s;
    long h;
    ⟨symlook() other locals 30b⟩

    ⟨symlook() compute hash value h of sym 30c⟩

    // s = hash_lookup((sym, space), h, hash)
    for(s = hash[h]; s; s = s->next)
        if((s->space == space) && (strcmp(s->name, sym) == 0))
            return s;
    // else
    ⟨symlook() if symbol not found 30e⟩
}
```
Uses `hash-3` 29b.

⟨symlook() *other locals* 30b⟩≡ (30a)
```
char *p;
```

⟨symlook() *compute hash value* h *of* sym 30c⟩≡ (30a)
```
//h = hash(sym, space)
for(p = sym, h = space; *p; h += *p++)
    h *= HASHMUL;
if(h < 0)
    h = ~h;
h %= NHASH;
```
Uses `HASHMUL-2` 30d and `NHASH-1` 29c.

⟨*constant* HASHMUL 30d⟩≡ (185c)
```
#define HASHMUL 79L /* this is a good value */
```

If `symlook()` does not find the key and the install parameter is not `nil`, it creates a new symbol:

⟨symlook() *if symbol not found* 30e⟩≡ (30a)
```
if(install == nil)
    return nil;
// else
s = (Symtab *)Malloc(sizeof(Symtab));
s->name = sym;
s->space = space;
s->u.ptr = install;

// add_list(s, hash)
s->next = hash[h];
hash[h] = s;

return s;
```
Uses `Malloc()` 171a and `hash-3` 29b.

`Malloc()`[171a], called above, is a small wrapper around `malloc()` from the C library (see the LIBCORE book [Pad16c]). `Malloc()` offers some `mk`-specific error management services, as explained in Appendix C.

In addition to `symlook()`, `mk` relies also on the generic function `symtraverse()` below to apply a function `fn` to all the elements in a specific namespace:

⟨*function* symtraverse 31a⟩≡ (185c)
```
void
symtraverse(int space, void (*fn)(Symtab*))
{
    Symtab **s, *ss;

    for(s = hash; s < &hash[NHASH]; s++)
        for(ss = *s; ss; ss = ss->next)
            if(ss->space == space)
                (*fn)(ss);
}
```
Uses `NHASH-1` 29c and `hash-3` 29b.

Finally, because setting (or overriding) the value of a variable is a common operation in `mk`, the function below provides a convenient wrapper around `symlook()`:

⟨*function* setvar 31b⟩≡ (186c)
```
void
setvar(char *name, void *value)
{
    symlook(name, S_VAR, value)->u.ptr = value;
}
```
Uses `S_VAR` 29a and `symlook()` 30a.

3.1.3 Namespaces

As I explained in Section 2.1.1, `mk` allows the user to define variables. `mk` also defines special variables such as `$stem` or `$target`. To clearly separate those two kinds of variables, `mk` stores them in different namespaces: `S_VAR`[29a] for the variables set by the user (and environment), and `S_INTERNAL` for the special (internal) variables set by `mk`.

⟨Sxxx *cases* 31c⟩≡ (29a) 37b ▷
```
S_INTERNAL, /* an internal mk variable (e.g., stem, target) */
```

Thus, to get the value of the special variable `$stem`, call the `symlook()`[30a] function with the pair `("stem", S_INTERNAL)`.

The private global `specialvars` below stores the list of special variables:

⟨*global* specialvars 31d⟩≡ (187c)
```
static char *specialvars[] =
{
    "target",
    "prereq",
    "stem",

    ⟨specialvars other array elements 130a⟩
    0,
};
```

I will gradually describe the other internal variables used by `mk` in the following chapters. I will also gradually describe more namespaces.

specialvars[31d] is used to initialize entries in the symbol table in `inithash()` (called from `main()`[45a]):

⟨*function* `inithash` 32a⟩≡ (187c)
```
/// main -> <>
void
inithash(void)
{
    char **p;

    for(p = specialvars; *p; p++)
        symlook(*p, S_INTERNAL, (void *)"");

    readenv();    /* o.s. dependent */
}
```
Uses `S_INTERNAL` 31c, `specialvars-8` 31d, and `symlook()` 30a.

`readenv()`[193c] called above initializes the symbol table with the environment variables (e.g., `$HOME`, `$objtype`). `mk` will store those environment variables in the `S_VAR` namespace.

3.2 Words

There are many places in the code of `mk` manipulating list of words: when `mk` processes a list of prerequisites, a list of targets, or the content of a variable. C does not have any builtin support for lists, so `mk` uses the following structure to represent a list of words:

⟨*struct* `Word` 32b⟩≡ (182d)
```
struct Word
{
    // ref_own<string>
    char   *s;

    // Extra
    ⟨Word extra fields 32c⟩
};
```

⟨`Word` *extra fields* 32c⟩≡ (32b)
```
// list<ref_own<Word>>
struct Word  *next;
```
Uses `Word` 32b.

`mk` defines also a few convenient functions to manipulate those lists. `newword()` below constructs a list with a single element from a string `s`:

⟨*constructor* `newword` 32d⟩≡ (186b)
```
Word*
newword(char *s)
{
    Word *w;

    w = (Word *)Malloc(sizeof(Word));
    w->s = strdup(s);
    w->next = nil;
    return w;
}
```
Uses `Malloc()` 171a.

freewords() frees a list of words:

⟨*destructor* freewords 33a⟩≡ (186b)
```
void
freewords(Word *w)
{
    Word *v;

    while(v = w){
        w = w->next;
        if(v->s)
            free(v->s);
        free(v);
    }
}
```

addw() adds in a list of words w a word s (a string) if it was not already in:

⟨*function* addw 33b⟩≡ (186b)
```
void
addw(Word *w, char *s)
{
    Word *lastw;

    for(lastw = w; w = w->next; lastw = w){
        if(strcmp(s, w->s) == 0)
            return;
    }
    lastw->next = newword(s);
}
```
Uses newword() 32d.

wdup() copies (duplicates) a list of words:

⟨*function* wdup 33c⟩≡ (186b)
```
Word*
wdup(Word *w)
{
    Word *lastw, *new, *head;

    head = lastw = nil;
    while(w){
        new = newword(w->s);
        if(lastw)
            lastw->next = new;
        else
            head = new;
        lastw = new;
        w = w->next;
    }
    return head;
}
```
Uses newword() 32d.

Finally, wtos() (for "words to string") concatenates together the words in a list of words with a special character sep (for separator):

⟨*function* wtos 33d⟩≡ (186b)
```
char *
wtos(Word *w, int sep)
{
    Bufblock *buf;
```

```
    char *cp;

    buf = newbuf();
    for(; w; w = w->next){
        for(cp = w->s; *cp; cp++)
            insert(buf, *cp);
        if(w->next)
            insert(buf, sep);
    }
    insert(buf, '\0');

    cp = strdup(buf->start);
    freebuf(buf);
    return cp;
}
```

Uses `freebuf()` 172e, `insert()` 173a, and `newbuf()` 172d.

`wtos()`[33d] relies on the `Bufblock`[171c] data structure described in Appendix C.2, which is an implementation of a *string buffer*. It implements efficiently string concatenation to avoid quadratic complexity when concatenating a list of strings together. The code for `Bufblock` is in `mk/bufblock.c`[185b], but this code could be put in a library and used by other projects because it is a general-purpose data structure. This is why I describe it in Appendix C and not here.

3.3 Rules

As mentioned in Section 2.1.2, the rule is the most important concept in a build system, and so the most important data structure in `mk`. It represents the content of an `mkfile` and it guides the creation of the graph of dependencies.

3.3.1 Rule

The structure `Rule` below represents a rule in memory. You can see that the first fields represent the major elements of a rule I mentioned in Section 2.1.1: the target, the prerequisites, and the recipe.

⟨*struct* `Rule` 34⟩≡ (182d)

```
struct Rule
{
    // ref_own<string>
    char   *target; /* one target */
    // list<ref_own<Word>>
    Word   *prereqs;  /* constituents of targets */
    // ref_own<string>, never nil, but can be the empty string (just '\0')
    char   *recipe; /* do it ! */

    ⟨Rule other fields 35h⟩
    ⟨Rule debug fields 35a⟩

    // Extra
    ⟨Rule extra fields 35c⟩
};
```

`Rule.prereqs` contains a list of words, hence the use of a pointer to a `Word`[32b]. `Rule.target` is a single string, not a list of words, even though some rules have multiple targets. I will explain later how `mk` represents internally rules with multiple targets. `Rule.recipe` is a string. This string can contain variables using the dollar sign. However, the strings in `Rule.target` and `Rule.prereqs` do not contain any variable. Indeed, as I will show in Section 5.1.3, `mk` expands variables used outside a recipe at parsing time.

In addition to the major fields mentioned above, `Rule` contains also information about where a rule comes from:

⟨`Rule` *debug fields* 35a⟩≡ (34)
```
char*   file;  /* source file */
short   line;  /* source line */
```

Those fields will be useful when reporting errors in the `mkfile` to the user (see Section 5.1).

3.3.2 Simple `rules`

The list of all *simple rules*, that is all non-meta rules, is stored in the global `rules`:

⟨*global* `rules` 35b⟩≡ (184)
```
// list<ref_own<Rule>> (next = Rule.next, end = lr)
Rule *rules;
```

When `mk` parses an `mkfile` (and possibly some included files), it populates this global (using the `addrule()`[36b] function).

Again, in C, you can embed a `next` field in a structure to make it a list:

⟨`Rule` *extra fields* 35c⟩≡ (34) 37c▷
```
// list<ref_own<Rule>> (head = rules | metarules)
struct Rule *next;
```
Uses `Rule` 34.

To quickly add a rule to the end of the list `rules`, `mk` maintains another global `lr` pointing to the last rule:

⟨*global* `lr` 35d⟩≡ (189a)
```
// option<ref<Rule>> (head = rules)
static Rule *lr;
```

3.3.3 `metarules`

The list of all *meta rules* is stored instead in an another global:

⟨*global* `metarules` 35e⟩≡ (184)
```
// list<ref_own<Rule>> (next = Rule.next, end = lmr)
Rule *metarules;
```

`mk` relies also on a global pointing to the last meta rule:

⟨*global* `lmr` 35f⟩≡ (189a)
```
// option<ref<Rule>> (head = metarules)
static Rule *lmr;
```

Remember that a meta rule is a rule using the special character `'%'` to specify a *pattern* in the target or prerequisites of a rule (see Section 2.1.1). In fact, `mk` supports another special character to represent a pattern: `'&'`, hence the code in the macro below:

⟨*function* `PERCENT` 35g⟩≡ (182d)
```
#define PERCENT(ch) (((ch) == '%') || ((ch) == '&'))
```

The difference between those two special characters is explained in the manual page of `mk` (in `docs/man/1/mk`):

- `'%'` matches a maximal length string of any characters
- `'&'` matches a maximal length string of any characters except period or slash

Section 6.6.2 gives an example where the difference between the two characters matter.

In addition to being stored in `metarules` instead of `rules`[35b], a meta rule contains also the `META` *rule attribute* in `Rule.attr`:

⟨`Rule` *other fields* 35h⟩≡ (34) 39b▷
```
// bitset<Rule_attr>
short   attr;  /* attributes */
```

⟨*enum* Rule_attr 36a⟩≡ (182d)

```
enum Rule_attr {
    META   = 0x0001,
    ⟨Rule_attr cases 128a⟩
};
```

Most of the other rule attributes correspond to advanced features of `mk` I will describe in Section 11.5.

3.3.4 Adding rules

Now that I described the data structures and globals related to the rules and meta rules, I can explain the code to add rules.

One rule with one target, `addrule()`

`addrule()` below adds a rule with a single target (I will explain later the code to support rules with multiple targets):

⟨*function* addrule 36b⟩≡ (189a)

```
/// (main -> parse | main) -> <>
void
addrule(char *target, Word *prereqs, char *recipe,
        Word *alltargets, int attr, int hline, char *prog)
{
    Rule *r = nil;
    bool reuse = false;
    ⟨addrule() other locals 37d⟩

    ⟨addrule() find if rule already exists, set reuse, update r 38a⟩

    if(r == nil)
        r = (Rule *)Malloc(sizeof(Rule));

    r->target = target;
    r->prereqs = prereqs;
    r->recipe = recipe;

    r->attr = attr;
    r->line = hline;
    ⟨addrule() set more fields 38c⟩
    ⟨addrule() indexing r by target in S_TARGET 37e⟩

    ⟨addrule() if meta rule 37a⟩
    else {
        ⟨addrule() if simple rule 36c⟩
    }
}
```

Uses `Malloc()` 171a.

If the rule is a simple rule, `mk` populates `rules`[35b]:

⟨`addrule()` *if simple rule* 36c⟩≡ (36b)

```
⟨addrule() return if reuse, to not add the rule in a list 38d⟩
// else
// add_list(r, rules, lr)
if(rules == nil)
    rules = lr = r;
else {
    lr->next = r;
    lr = r;
```

```
}
```

Uses `lr-22` 35d and `rules` 35b.

If the rule is a meta rule, `mk` populates `metarules`[35e]. `mk` detects if the rule is a meta rule simply by looking whether the target contains one of the special pattern character:

⟨`addrule()` *if meta rule* 37a⟩≡ (36b)

```
if(charin(target, "%&") || (attr&REGEXP)){
    r->attr |= META;
    ⟨addrule() return if reuse, to not add the rule in a list 38d⟩
    // else
    ⟨addrule() if REGEXP attribute 128e⟩
    // add_list(r, metarules, lmr)
    if(metarules == nil)
        metarules = lmr = r;
    else {
        lmr->next = r;
        lmr = r;
    }
}
```

Uses `META` 36a, `REGEXP` 128a, `charin()` 58c, `lmr-23` 35f, and `metarules` 35e.

I will describe the function `charin()`[58c] and the rule attribute `REGEXP`[128a] later. Note that `charin()` does not just search for a set of character in a string. Indeed, `charin()` must also handle *escaped characters*, a feature I will explain in Section 5.1.1. For example, when the target name is put inside a quote as in `'myfile%has%weird%characters.doc'`, the target should not be considered a pattern. For the rule attribute `REGEXP` used above, see Section 11.1.

One target with multiple rules, `S_TARGET`

It is useful when building the graph of dependencies to quickly know the rule associated to a specific target. Thus, `mk` uses another namespace, `S_TARGET`, to store such information in the symbol table.

⟨`Sxxx` *cases* 37b⟩+≡ (29a) ◁31c 74c▷

```
S_TARGET,  /* target -> rules */
```

In fact, as I mentioned in Section 2.1.2, `mk` allows the user to write multiple rules using the same target. For instance, an `mkfile` can contain a master rule such as `foo.5:  foo.c ...`, and a `.depend` file included by this `mkfile` can contain another rule without any recipe but with extra dependencies such as `foo.5:  foo.h bar.h`. Thus, the symbol table and the namespace `S_TARGET` map a target to a set of rules chained together by an extra field in `Rule`[34]:

⟨`Rule` *extra fields* 37c⟩+≡ (34) ◁35c

```
// list<ref<Rule>> (head = symlook(x, S_TARGET))
struct Rule *chain;  /* hashed per target */
```

Uses `Rule` 34.

Here is the code to update the symbol table in `addrule()`[36b]:

⟨`addrule()` *other locals* 37d⟩≡ (36b)

```
Symtab *sym;
Rule *rr;
```

⟨`addrule()` *indexing* `r` *by target in* `S_TARGET` 37e⟩≡ (36b)

```
if(!reuse){
    sym = symlook(target, S_TARGET, r);
    rr = sym->u.ptr;
    if(rr != r){ // target had already a rule
        r->chain = rr->chain;
        rr->chain = r;
```

```
    } else
        r->chain = nil;
}
```
Uses S_TARGET 37b and symlook() 30a.

Remember that the last parameter of `symlook()`[30a], called `install` (and here set to the argument `r`), is used to initialize a new symbol if the symbol was not already in the symbol table. Thus, if the test `if (rr != r)` above succeeds, this means a symbol was already there, in which case `mk` needs to add the rule `r` to the chain.

I will explain the guard using the variable `reuse` above in the next section.

Overwriting a previous rule

`mk` allows to overwrite the recipe of a rule when another rule uses exactly the same target and prerequisites. This can be useful when a generic `mkfile.generic` file defines some default targets and recipes, but the user wants to overwrite those defaults in his own `mkfile` (which can include `mkfile.generic`).

The code below detects whether a previous rule was using the same target and prerequisites, in which case `mk` needs to reuse and overwrite this previously allocated rule:

⟨`addrule()` *find if rule already exists, set* `reuse`*, update* `r` 38a⟩≡ (36b)
```
sym = symlook(target, S_TARGET, nil);
if(sym){
    for(r = sym->u.ptr; r; r = r->chain)
        if(rcmp(r, target, prereqs) == 0){
            reuse = true;
            break;
        }
}
```
Uses S_TARGET 37b and symlook() 30a.

Note that the code above relies on the indexing of rules in `S_TARGET`[37b] from the previous section.

⟨*function* `rcmp` 38b⟩≡ (189a)
```
static int
rcmp(Rule *r, char *target, Word *prereqs)
{
    Word *w;

    if(strcmp(r->target, target))
        return 1;
    for(w = r->prereqs; w && prereqs; w = w->next, prereqs = prereqs->next)
        if(strcmp(w->s, prereqs->s))
            return 1;
    return (w || prereqs);
}
```

If `addrule()`[36b] overwrites (reuses) a previous rule, the `Rule.next`[35c] field of this rule should not be modified. Otherwise, `Rule.next` should be set to `nil`:

⟨`addrule()` *set more fields* 38c⟩≡ (36b) 39c ▷
```
if(!reuse){
    r->next = nil;
}
```

⟨`addrule()` *return if* `reuse`*, to not add the rule in a list* 38d⟩≡ (37a 36c)
```
if(reuse)
    return;
```

One rule with multiple targets, `addrules()`

I can now show the code to handle rules with multiple targets. `mk` uses the function `addrules()` below to add separate rules for each target in the original rule:

⟨*function* `addrules` 39a⟩≡ (188a)
```
void
addrules(Word *targets, Word *prereqs, char *recipe,
         int attr, int hline, char *prog)
{
    Word *w;

    assert(/*addrules args*/ targets && recipe);

    ⟨addrules() set target1 67d⟩
    for(w = targets; w; w = w->next)
        addrule(w->s, prereqs, recipe, targets, attr, hline, prog);
}
```
Uses `addrule()` 36b.

As I mentioned in Section 2.1.2, the use of multiple targets in a rule has implications on the DFS traversal of the graph of dependencies: `mk` needs to remember the other targets associated with a rule. This is why in addition to passing `w->s` above, `addrules()`[39a] passes also the set of targets in the fourth argument to `addrule()`[36b]. This argument is then stored in a special field in the rule:

⟨`Rule` *other fields* 39b⟩+≡ (34) ◁35h 85b▷
```
// ref<list<ref_own<Word>>
Word   *alltargets; /* all the targets */
```

⟨`addrule()` *set more fields* 39c⟩+≡ (36b) ◁38c 51b▷
```
r->alltargets = alltargets;
```

3.4 Graph

The graph of dependencies is represented in `mk` essentially by a set of nodes linked together through pointers. `mk` does not use a matrix or an array of adjacent lists to represent a graph; it just uses pointers, as you will see in the following sections.

3.4.1 `Node`

A node represents a file in the graph of dependencies. As I mentioned in Section 2.1.2 and Figure 2.2, a node is also labeled with the modification time of the file. That way, the DFS can find out-of-date files by comparing the `Node.time` fields of different nodes.

⟨*struct* `Node` 39d⟩≡ (182d)
```
struct Node
{
    // ref_own<string>, usually a filename, or a virtual target like 'clean'
    char*  name;
    // option<Time> (None = 0, for nonexistent files and virtual targets)
    ulong  time; // last mtime of file

    ⟨Node arcs field 40e⟩
    ⟨Node other fields 40b⟩

    // Extra
    ⟨Node extra fields 42a⟩
};
```

The function below constructs a new node:

⟨*constructor* newnode 40a⟩≡ (191a)
```
/// main -> mk -> graph -> applyrules -> <>
static Node*
newnode(char *name)
{
    Node *node;

    node = (Node *)Malloc(sizeof(Node));
    ⟨newnode() update node cache 82b⟩

    node->name = name;
    // call to timeof()!
    node->time = timeof(name, false);
    node->flags = 0;
    ⟨newnode() adjust flags of node 90a⟩

    node->arcs = nil;
    node->next = nil;
    ⟨newnode() debug 168i⟩
    return node;
}
```
Uses Malloc() 171a and timeof() 82d.

A node is also labeled with a set of *node attributes*:

⟨Node *other fields* 40b⟩≡ (39d)
```
// bitset<enum<Node_flag>>
ushort  flags;
```

⟨*enum* Node_flag 40c⟩≡ (182d)
```
enum Node_flag {
    ⟨Node_flag cases 40d⟩
};
```

The building status of a node (`Made`, `NotMade`, and `BeingMade`), which I introduced in Section 2.1.3, is stored in `Node.flags` (as well as other information used for advanced features of `mk`):

⟨Node_flag *cases* 40d⟩≡ (40c) 84a ▷
```
NOTMADE    = 0x0020,
BEINGMADE  = 0x0040,
MADE       = 0x0080,
```

I will gradually describe the other node attributes in the following chapters.

3.4.2 Arc

A `Node`[39d] contains also a set of arcs where each arc contains a pointer to another node (a prerequisite):

⟨Node *arcs field* 40e⟩≡ (39d)
```
// list<ref_own<Arc>> (next = Arc.next)
Arc  *arcs;
```

⟨*struct* Arc 40f⟩≡ (182d)
```
struct Arc
{
    // option<ref<Node>>, the other node in the arc (the dependency, None when virtual)
    struct Node *n;
    // ref<Rule>, to generate the target node from the dependent node
    Rule *r;
```

```
        ⟨Arc other fields 41b⟩

        //Extra
        ⟨Arc extra fields 41a⟩
  };
```
Uses `Node` 39d.

As I mentioned in Section 2.1.2 and Figure 2.2, an arc is labeled with a rule, hence the field `Arc.r` above. Note that `Arc.n` can sometimes be `nil` when a rule does not have any prerequisite (for instance, because it is a virtual target, as explained in Section 11.5.1). In that case, we still want the node corresponding to the target of the rule to be connected to a rule, especially its recipe.

The head of the list of arcs of a node is stored in `Node.arcs`, but the arcs are chained together with the following field:

⟨`Arc` *extra fields* 41a⟩≡ (40f)
```
  // list<ref_own<arc> (head = Node.arcs)
  struct Arc *next;
```
Uses `Arc` 40f.

Some nodes and arcs are derived from meta rules. For instance, in Figure 2.1, the nodes `hello.5` and `hello.c` could come from a meta rule such as `%.5: %.c ...`. In that case, `mk` needs to remember in the arc connecting `hello.5` to `hello.c` the *stem* that was used to instantiate the meta rule (here `hello`):

⟨`Arc` *other fields* 41b⟩≡ (40f) 88e ▷
```
  // option<ref_own<string>>, what '%' matched?
  char  *stem;
```

The function below constructs a new arc that can be added later to the list of arcs of a source node. This arc will connect the source node to a destination node `n`, with the rule `r`, possibly instantiated with the stem `stem` if the rule was a meta rule (the last parameter `match` is used for regexp rules, as explained in Section 11.1):

⟨*constructor* `newarc` 41c⟩≡ (191a)
```
  /// main -> mk -> graph -> applyrules -> <>
  Arc*
  newarc(Node *n, Rule *r, char *stem, Resub *match)
  {
      Arc *a;

      a = (Arc *)Malloc(sizeof(Arc));
      a->n = n;
      a->r = r;
      a->stem = strdup(stem);

      a->next = nil;
      ⟨newarc() set other fields 89a⟩
      return a;
  }
```
Uses `Malloc()` 171a.

3.5 Jobs

Finally, the last core data structure of `mk` is the description of a job. As I mentioned in Section 2.1.3, a job must contain all the information needed to run a recipe and to update the graph of dependencies: a rule (and its recipe), a list of nodes to update, and the value of special variables such as `$target`, `$prereq`, or `$stem`:

⟨*struct* `Job` 41d⟩≡ (182d)
```
  struct Job
  {
```

```
    // ref<Rule>
    Rule  *r; /* master rule for job */

    // list<ref<Node>> (next = Node.next)
    Node  *n; /* list of node targets */

    // $target and $prereq
    // list<ref<Word>>
    Word  *t; /* targets */
    // list<ref<Word>>
    Word  *p; /* prerequisites */
    // ref<string> ($stem)
    char  *stem;

    ⟨Job other fields 129g⟩

    // Extra
    ⟨Job extra fields 42d⟩
};
```

The list of target nodes of a job are chained together through an extra field in `Node`[39d]:

⟨`Node` *extra fields* 42a⟩≡ (39d)

```
// list<ref<Node>> (head = Job.n)
struct Node *next;  /* list for a rule */
```

Uses `Node` 39d.

The function below constructs a new job:

⟨*constructor* `newjob` 42b⟩≡ (192a)

```
Job*
newjob(Rule *r, Node *nlist, char *stem, char **match,
       Word *allprereqs, Word *newprereqs,
       Word *alltargets, Word *oldtargets)
{
    Job *j;

    j = (Job *)Malloc(sizeof(Job));
    j->r = r;
    j->n = nlist;
    j->p = allprereqs;
    j->t = oldtargets;

    j->stem = stem;
    j->match = match;

    ⟨newjob() setting other fields 147g⟩

    j->next = nil;
    return j;
}
```

Uses `Malloc()` 171a.

This job can then be added to the job queue, which is stored in the global `jobs`:

⟨*global* `jobs` 42c⟩≡ (184)

```
// list<ref_own<jobs>> (next = Job.next)
Job *jobs;
```

⟨`Job` *extra fields* 42d⟩≡ (41d)

```
// list<ref_own<Job>> (head = jobs)
struct Job *next;
```

Uses `Job` 41d.

3.6 Putting it together: the data structures for hello.mk

It is worth pausing here to see how the core data structures fit together before main()[45a] starts using them. Rather than listing fields in the abstract, Figure 3.1 shows the actual instances mk creates when it processes the example hello.mk (see Section 2.3) to build hello.

Everything is reached through the global symbol table hash[29b], keyed by a (name, space) pair: the name hello resolves to its Rule[34] under S_TARGET[37b] and, separately, to its Node[39d] under S_NODE[82a]. From there hang the static Rule lists (rules[35b] and metarules[35e], with tail pointers lr[35d] and lmr[35f] for O(1) append) and the dynamic Node/Arc[40f] graph instantiated once hello is requested. In the figure, a * is a pointer field that an arrow follows; a Word[32b] list is written in square brackets (e.g. [prereqs]) and a plain char* string in quotes (e.g. "target").

One way to read Figure 3.1 is to follow what happens when the user types mk hello. First, mk looks up hello in the S_TARGET space and finds Rule0; it then instantiates a Node for hello and for each file it depends on, caching them in S_NODE so that a file is never visited twice (see Section 6.4 for more info on S_NODE and the Node cache). The regular rule yields the two arcs out of hello (both tagged r:Rule0), while the meta rule %.5: %.c matches twice, producing one arc per concrete pair with the stem set to hello and world respectively. Once the graph is complete, the build walk work()[95a] descends it depth-first from the requested target, recursing into a node's prerequisites before the node itself; a Job[41d] is dispatched for a node only once all of its prerequisites are up to date. So hello.5 and world.5 are compiled before hello can be linked, turning Rule0 and RuleM's recipes into the concrete commands 5c -c hello.c, 5c -c world.c, and finally 5l -o hello hello.5 world.5.

The static/dynamic split in Figure 3.1 is deliberate. Rules describe the mkfile verbatim and never change after parsing. Nodes and Arcs are an instantiation of those rules against the actual requested targets and files in the project: a meta rule %.5: %.c becomes one arc per concrete pair (hello.5→hello.c, world.5→world.c). A Job then captures one in-flight execution—it points back into the rule for the recipe and into the node list for graph updates.

One detail of Figure 3.1 may surprise you: the Rule for hello lists its prerequisites as the literal words hello.5 world.5, even though hello.mk (See Section 2.3) writes them as the variable $OBJS (with OBJS=hello.5 world.5 defined above in the hello.mk file). This is because mk expands variables in targets and prerequisites at parse time (See Section 5.1.3); by the time a Rule is built, its prereqs Word list already holds the expanded filenames, so no variable ever reaches the graph.

Recipes are the one exception: they are expanded only later by the shell at recipe-execution time. That is why Rule0's recipe still shows $target and $prereq; an $OBJS appearing inside a recipe would likewise survive untouched all the way into the Job.

```
SYMBOL TABLE: global hash[NHASH=4099], key=(name,space)

  hash[h1] --> Symtab{name:"hello", space:S_TARGET} u.ptr *--> Rule0
  hash[h2] --> Symtab{name:"hello", space:S_NODE  } u.ptr *--> Node "hello"
  hash[h3] --> Symtab{name:"OBJS",  space:S_VAR   } u.ptr *--> [hello.5 world.5]
       (Symtab.next chains hash collisions within one bucket)

STATIC: the rule lists  (head pointer + tail pointer lr/lmr)

                          ("hello", S_TARGET)
                                   v
                    +--------------------------------+
  rules     *--+--> | Rule0                          | --next--> nil
  lr        *--+    | target  "hello"                |
                    | prereqs [hello.5 world.5]      |
                    | recipe  "5l -o $target $prereq"|
                    +--------------------------------+


                    +--------------------------------+
  metarules *--+--> | RuleM                          | --next--> nil
  lmr       *--+    | target  "%.5"                  |
                    | prereqs [%.c]                  |
                    | recipe  "5c -c $stem.c"        |
                    +--------------------------------+

DYNAMIC: graph built when "hello" is requested.

  +-----------------+ arcs +-------------+ next +-------------+
  |Node "hello"     |----->| Arc         | ---> | Arc         |--> nil
  | time=0 NOTMADE  |      | r *--> Rule0|      | r *--> Rule0|
  +-----------------+      | n *--+      |      | n *--+      |
      ^                    +------|------+      +------|------+
   ("hello",S_NODE)               v                    v
                          +-----------------+ +-----------------+
                          |Node hello.5     | |Node world.5     |
                          | time=0 NOTMADE  | | time=0 NOTMADE  |
                          +--------+--------+ +--------+--------+
                                   v arcs              v arcs
                          +-------------+     +-------------+
                          | Arc         |     | Arc         |
                          | r *--> RuleM|     | r *--> RuleM|
                          | stem:"hello"|     | stem:"world"|
                          | n *--+      |     | n *--+      |
                          +------|------+     +------|------+
                                 v                   v
                          +-----------------+ +-----------------+
                          |Node hello.c     | |Node world.c     |
                          | time=900 NOTMADE| | time=900 NOTMADE|
                          +-----------------+ +-----------------+
                              (leaves, already on disk)
```

Figure 3.1: The core data structures of `mk`, instantiated for `hello.mk`.

Chapter 4

main()

I now switch from the bottom-up approach of Chapter 3 to a top-down approach. Indeed, I will describe in the following chapters the main functions of `mk`, starting in this chapter with `main()`, the entry point of `mk`.

4.1 main() skeleton

You can see the main components of the building pipeline in the `main()` skeleton below:

```
⟨function main 45a⟩≡                                          (193b)
  void
  main(int argc, char **argv)
  {
      ⟨main() locals 47a⟩

      // Initializing
      ⟨main() initializations 45c⟩

      // Parsing the mkfile
      ⟨main() parsing mkfile, call parse() 48e⟩

      // Building the graph, finding out-of-date files
      ⟨main() initializations before building 111a⟩
      ⟨main() setting the targets, call mk() 49a⟩

      // Reporting (optional)
      ⟨main() print profiling stats if uflag 127a⟩

      // Exiting
      exits(nil);
  }
```

The next chapters will detail those different components. In this chapter, I will focus mostly on the initializations and the processing of command-line arguments.

An important global set by `main()` is `bout`:

```
⟨global bout 45b⟩≡                                            (184)
  Biobuf bout;
```

`mk` uses this global to print messages to the user (e.g., errors, job progress, profiling information). `bout` is a buffer connected to the standard output:

```
⟨main() initializations 45c⟩≡                                 (45a) 46a ▷
  Binit(&bout, STDOUT, OWRITE);
```

Uses `bout` 45b.

Biobuf, the type of bout, as well as Binit() are defined in the libbio (for "buffered IO") library, which extends the C library (see the LIBCORE book [Pad16c]).

mk processes the command-line arguments in three steps, as hinted in the code below, and as explained in the following sections.

```
⟨main() initializations 46a⟩+≡                                    (45a) ◁45c
  ⟨main() argv processing part 1, -xxx 46b⟩
  ⟨main() setup profiling 126e⟩
  inithash();
  ⟨main() argv processing part 2, xxx=yyy 47b⟩
  ⟨main() set variables for recursive mk 160d⟩
  ⟨main() argv processing part 3, skip xxx=yyy 48a⟩
  ⟨main() profile initializations 126f⟩
```

Uses inithash() 32a.

inithash()[32a] called above initializes the special variables in the symbol table (e.g., $target, $prereqs), and imports variables from the environment in the symbol table (e.g., $objtype, $HOME).

4.2 mk -<flag> arguments processing

The first step in the processing of command-line arguments is an iteration over argv:

```
⟨main() argv processing part 1, -xxx 46b⟩≡                        (46a)
  USED(argc);
  for(argv++; *argv && (**argv == '-'); argv++)
  {
      ⟨main() add argv[0] in buf 160b⟩

      switch(argv[0][1]) {
      ⟨main() -xxx switch cases 48d⟩
      default:
          badusage();
      }
  }
```

Uses badusage() 46c.

This iteration looks for command-line arguments prefixed by '-' (e.g., -f). I will gradually described the cases of the switch above in the following chapters.

```
⟨function badusage 46c⟩≡                                          (193b)
  void
  badusage(void)
  {

      fprint(STDERR,
             "Usage: mk [-f file] [-(n|a|e|t|k|i)] [-d[egp]] [targets ...]\n");
      Exit();
  }
```

4.3 mk <var>=<values> arguments processing

The second step in the processing of command-line arguments is also an iteration over argv, but this time looking for arguments containing an equal sign. Indeed, mk allows the user to overwrite variables defined in the mkfile by adding a command-line argument in the form x=y before the target, as in mk objtype=arm all.

Because the parser of mkfile I will describe in Chapter 5 contains already code to process variable definitions, mk reuses this code when dealing with command-line definitions. Indeed, after storing those definitions in a temporary file, mk can then simply call parse()[51e] on this temporary file to load those definitions.

The temporary file is first a filename (`temp`), then a file descriptor once opened (`tfd`), and finally an output buffer once initialized (`tb`):

```
⟨main() locals 47a⟩≡                                              (45a) 48c ▷
  char *temp = nil;
  fdt tfd = -1;
  Biobuf tb;
  int i;
```

```
⟨main() argv processing part 2, xxx=yyy 47b⟩≡                          (46a)
  for(i = 0; argv[i]; i++)
    if(utfrune(argv[i], '=')){
      ⟨main() add argv[i] in buf 160c⟩

      ⟨main() create temporary file if not exist yet and set tb 47c⟩
      Bprint(&tb, "%s\n", argv[i]);
      ⟨main() mark argv[i] for skipping 48b⟩
    }

  if(tfd >= 0){
      Bflush(&tb);
      seek(tfd, 0L, SEEK__START);
      parse("<command line args>", tfd, true);
      remove(temp);
  }
```

Uses `parse()` 51e.

`utfrune()`, called above, is a function looking for a certain character in a string. However, the character and the string use particular formats. Indeed, `utfrune()` looks for a *Rune* in a sequence of *UTF-8* encoded bytes (hence its name). Plan 9 uses the UTF-8 encoding everywhere text is involved: in filenames, in command-line arguments, in files on disk, etc. See the LIBCORE book [Pad16c] for more info on runes, UTF-8, and more generally Unicode.

The last argument to `parse()` above is a boolean indicating whether `parse()` should accept definitions overwriting previous definitions. Obviously, in this case `main()`[45a] passes `true` to `parse()`.

```
⟨main() create temporary file if not exist yet and set tb 47c⟩≡          (47b)
  if(tfd < 0){
      temp = maketmp();
      ⟨main() when creating temporary file, sanity check temp 47d⟩
      tfd = create(temp, ORDWR, 0600);
      ⟨main() when creating temporary file, sanity check tfd 47e⟩
      Binit(&tb, tfd, OWRITE);
  }
```

The code above omits the error management code shown below:

```
⟨main() when creating temporary file, sanity check temp 47d⟩≡            (47c)
  if(temp == nil) {
      perror("temp file");
      Exit();
  }
```

```
⟨main() when creating temporary file, sanity check tfd 47e⟩≡             (47c)
  if(tfd < 0){
      perror(temp);
      Exit();
  }
```

In the rest of this book, I will usually not comment the error-management code. Such code is necessary but often trivial.

4.4 mk remaining arguments processing

The last step in the processing of command-line arguments is to skip variable definitions:

⟨main() *argv processing part 3, skip xxx=yyy* 48a⟩≡ (46a)
```
  /* skip assignment args */
  while(*argv && (**argv == '\0'))
      argv++;
```

This is made possible because of the previous marking of assignment arguments:

⟨main() *mark* argv[i] *for skipping* 48b⟩≡ (47b)
```
  /*
   *   assignment args become null strings
   */
  *argv[i] = '\0';
```

Once mk has cleaned up `argv`, the strings remaining in `argv` are the targets the user wants to build.

4.5 Using the mkfile or mk -f <file>

I described before in Section 1.4 the use of `-f` to change the default file used by `mk`. Here is the code for it:

⟨main() *locals* 48c⟩+≡ (45a) ◁47a 49c▷
```
  char *f = nil;
```

⟨main() *-xxx switch cases* 48d⟩≡ (46b) 49f▷
```
  case 'f':
      if(*++argv == nil)
          badusage();
      f = *argv;
      ⟨main() add argv[0] in buf 160b⟩
      break;
```
Uses `badusage()` 46c.

The `parse()`[51e] function, called below, will process the `mkfile` (or another file if `-f` was used) and modify `rules`[35b], `metarules`[35e], as well as a few other globals. Note that this time `main()` passes `false` to `parse()`, so overwriting variable definitions (e.g., the ones given on the command-line) is disabled.

⟨main() *parsing mkfile, call* parse() 48e⟩≡ (45a)
```
  if(f == nil){
      if(access(MKFILE, AREAD) == OK_0)
          parse(MKFILE, open(MKFILE, OREAD), false);
  } else
      parse(f, open(f, OREAD), false);
  ⟨main() if DEBUG(D_PARSE) 166a⟩
```
Uses `MKFILE-9` 48f and `parse()` 51e.

⟨*constant* MKFILE 48f⟩≡ (193b)
```
  #define MKFILE  "mkfile"
```

4.6 Building the target(s)

Once `parse()`[51e] processed the `mkfile` and modified some globals, `mk` is ready to build a target by calling `mk()`[92]. There are multiple ways to specify the target to build and how to build it, as explained in the following sections,

and as hinted by the following code:

```
⟨main() setting the targets, call mk() 49a⟩≡                                  (45a)
  if(*argv == nil){
      ⟨main() when no target arguments 49d⟩
  } else {
      ⟨main() if sequential mode and target arguments given 49g⟩
      else {
         ⟨main() parallel mode and target arguments given 50⟩
      }
  }
```

4.6.1 Default target: `target1`

As I mentioned in Section 2.2, if the user does not specify any target on the command-line, `mk` uses the target of the first simple rule found in the `mkfile` as the default target. This default target is stored in the following global:

```
⟨global target1 49b⟩≡                                                         (184)
  // option<list<ref<Word>>>
  Word *target1;
```

Section 5.2 contains the code in `parse()`[51e] modifying `target1`. If the user did not provide a target on the command-line and `target1` was set, then `mk` builds this target by calling `mk()`[92]:

```
⟨main() locals 49c⟩+≡                                            (45a) ◁48c 49e▷
  Word *w;
```

```
⟨main() when no target arguments 49d⟩≡                                        (49a)
  if(target1)
      for(w = target1; w; w = w->next)
          // The call!
          mk(w->s);
  else {
      fprint(STDERR, "mk: nothing to mk\n");
      Exit();
  }
```

Uses `mk()` 92 and `target1` 49b.

Note that the first simple rule can contain multiple targets, which is why the code above iterates over the list of words in `target1`.

4.6.2 Building sequentiallly

The second way to build one or more targets is to specify a set of targets on the command-line. Moreover, `mk` supports a special flag, `-s` (for "sequential"), to build sequentiallly those targets:

```
⟨main() locals 49e⟩+≡                                           (45a) ◁49c 125a▷
  bool sflag = false;
```

```
⟨main() -xxx switch cases 49f⟩+≡                               (46b) ◁48d 124b▷
  case 's':
      sflag = true;
      break;
```

```
⟨main() if sequential mode and target arguments given 49g⟩≡                   (49a)
  if(sflag){
      for(; *argv; argv++)
          if(**argv)
              mk(*argv);
  }
```

Uses `mk()` 92.

4.6.3 Building in parallel

The last way to build one or more targets is to specify them on the command-line without the `-s` flag. In that case, `mk` builds the targets in parallel. To do so, `mk` creates a new rule with the command-line targets as the prerequisites of the new rule, and an arbitrary string for its target. `mk` then calls `mk()`[92] with this arbitrary string, which will trigger the DFS to build its prerequisites in parallel.

⟨`main()` *parallel mode and target arguments given* 50⟩≡ (49a)

```
Word *head;
Word *tail = nil;
Word *t = nil;

/* fake a new rule with all the args as prereqs */
for(; *argv; argv++)
    if(**argv){
        // add_list(newword(*argv), t)
        if(tail == nil)
            tail = t = newword(*argv);
        else {
            t->next = newword(*argv);
            t = t->next;
        }
    }
if(tail->next == nil)
    // a single target argument
    mk(tail->s);
else {
    head = newword("<command line arguments>");
    addrules(head, tail, strdup(""), VIR, mkinline, nil);
    mk(head->s);
}
```

Uses `VIR` 140c, `addrules()` 39a, `mk()` 92, `mkinline` 51c, and `newword()` 32d.

You can see in the code above a few calls to functions I described in Chapter 3, for instance, `newword()`[32d] and `addrules()`[39a]. I will describe `mk()`, called above, the most important function of `mk`, in Chapter 7.

The `VIR`[140c] argument above indicates that the target is a *virtual target*. `VIR` is a rule attribute I will explain fully in Section 11.5.1. The arbitrary string used in the first argument to `newword()` above is a virtual target because it does not correspond to a file. In that case, it is not an error if the target does not exist after `mk` ran the recipe.

The last two arguments to `addrules()` above are the line and file location of the rule, which are used for error reporting. Because here the rule was created artificially by `mk`, the file location is set to `nil`.

That completes `main()`[45a]. It is essentially the conductor: it reads the command line in three passes—flags, then `<var>=<value>` definitions, then the requested targets—loads the `mkfile`, and hands off to the builder.

Chapter 5

Parsing the mkfile

Now that you have seen `main()`[45a], I can explain the different components in the building pipeline, starting in this chapter with the parsing functions.

I mentioned before `parse()`[51e], which takes a path to an `mkfile` as a parameter, parses this `mkfile` to identify rules, meta rules, and variable definitions, and stores those entities in different globals (`rules`[35b], `metarules`[35e], and the symbol table `hash`[29b]). `parse()` is a complex function that relies on many helper functions to scan a file, identify rules, expand variables, process included files, or define variables, as explained in the following sections.

5.1 parse()

Before showing the code of `parse()`[51e], I describe here a few globals used by `parse()` (and a few other functions) to report errors to the user.

`infile` below contains the name of the file currently processed by `parse()` (an `mkfile` or one of its included files):

⟨*global* `infile` 51a⟩≡ (184)

```
char *infile;
```

This global is used in `addrule()`[36b]:

⟨`addrule()` *set more fields* 51b⟩+≡ (36b) ◁39c 85d▷

```
r->file = infile;
```

Uses `infile` 51a.

`mkinline` below contains the line number of the line currently processed by `parse()`:

⟨*global* `mkinline` 51c⟩≡ (184)

```
int mkinline;
```

Both globals are used in the following macro to report syntax errors to the user:

⟨*function* `SYNERR` 51d⟩≡ (182d)

```
#define SYNERR(l) (fprint(STDERR, "mk: %s:%d: syntax error; ", \
                          infile, ((l)>=0)? (l) : mkinline))
```

Here is finally the code of `parse()`:

⟨*function* `parse` 51e⟩≡ (188a)

```
void
parse(char *f, fdt fd, bool varoverride)
{
    Biobuf in;
    Bufblock *buf;
    char c; // one of : = <
    Word *head, *tail;
    int hline; // head line number
```

```
    ⟨parse() other locals 67b⟩

    ⟨parse() sanity check fd 52⟩
    ⟨parse() start, push 72a⟩

    // Initialization
    infile = strdup(f);
    mkinline = 1;
    Binit(&in, fd, OREAD);
    buf = newbuf();

    // Lexing
    while(assline(&in, buf)){
        hline = mkinline;

        // Parsing
        c = rhead(buf->start, &head, &tail,     &attr, &prog);

        // Semantic actions (they may read more lines)
        switch(c) {
        ⟨parse() switch rhead cases 53a⟩
        }
    }
    close(fd);
    freebuf(buf);
    ⟨parse() end, pop 72b⟩
}
```

Uses `assline()` 53b, `freebuf()` 172e, `infile` 51a, `mkinline` 51c, and `newbuf()` 172d.

The code of `parse()` operates in four steps (as shown by the sectioning comments in the code above): intialization, lexing, parsing, and actions. Here are a few comments about each step:

- *Initialization*: `parse()` initializes the globals `infile` and `mkinline` mentioned above, as well as two local buffers:
  1. `in`: an input buffer (using the `libbio` library; see the LIBCORE book [Pad16c]) connected to the file descriptor of the opened `mkfile`
  2. `buf`: a string buffer (see Appendix C.2) that will be used to store one line of the `mkfile`.
- *Lexing*: `parse()` reads and assembles a line from the `mkfile` in `buf` (via the function `assline()`[53b]). This is similar to the lexing phase in a compiler (see the COMPILER book [Pad16b]).
- *Parsing*: `parse()` processes a line to extract its elements: the target and prerequisites around the special character `':'` in a rule, or the variable name and values around the special character `'='` in a variable definition. `rhead()`[57b] uses the buffer containing a line from the `mkfile` as an argument and returns the special character `c` used in the line (`':'` for a rule, `'='` for a definition, and `'<'` for an inclusion). It also modifies the `head` and `tail` arguments passed by address to contain respectively the left and right parts around the special character (for `'<'`, the left part `head` is empty).
- *Actions*: Based on the special character read in the previous step, `parse()` will populate `rules`[35b] and `metarules`[35e], or the symbol table. I will describe in the next sections the cases of the `switch` above.

The skeleton of `parse()` above omits the error management code shown below:

```
⟨parse() sanity check fd 52⟩≡                                    (51e)
  if(fd < 0){
      perror(f);
      Exit();
  }
```

⟨parse() *switch rhead cases* 53a⟩≡ (51e) 67c ▷

```
  default:
      SYNERR(hline);
      fprint(STDERR, "expected one of :<=\n");
      Exit();
      break;
```

Uses SYNERR 51d.

There are two other locals in `parse()` that are passed by address to `rhead()`: `attr`, which will contain possibly a rule attribute, and `prog`. Both locals are used for advanced features of `mk` I will describe later.

5.1.1 Assembling a line: `assline()`

The `assline()` function below[1] reads characters from the input buffer `bp`, and fills the string buffer `buf` with those characters until it finds a newline. It also returns `true` when there are more lines to assemble, or `false` when `assline()` reaches the end of the file.

⟨*function* assline 53b⟩≡ (189b)

```
  /*
   * Assemble a line skipping blank lines, comments, and eliding
   * escaped newlines
   */
  bool
  assline(Biobuf *bp, Bufblock *buf)
  {
      int c;
      ⟨assline() other locals 55a⟩

      resetbuf(buf);
      while ((c = nextrune(bp, true)) >= 0){
          switch(c) {
          case '\n':
              if (!isempty(buf)) {
                  insert(buf, '\0');
                  return true;
              }
              // else
              break;  /* skip empty lines */
          ⟨assline() switch character cases 55b⟩
          default:
              rinsert(buf, c);
              break;
          }
      }
  eof:
      insert(buf, '\0');
      return *(bufcontent(buf)) != '\0';
  }
```

Uses bufcontent 172h, insert() 173a, isempty 172f, nextrune() 54a, resetbuf 172g, and rinsert() 173b.

`insert()`[173a], `rinsert()`[173b] (for "Rune insert"), `resetbuf()`[172g], `bufcontent()`[172h], and `isempty()`[172f], called above, are all functions (or macros) operating on a string buffer and are described in Appendix C.2.

The code of `assline()`[53b] may look trivial, but as its (inappropriate) name suggests, `assline()` does not just read a line: it *assembles* a line. For example, if the current line is a comment, `assline()` must skip the line and return the next meaningful line to `parse()`[51e]. Moreover, `assline()` must handle also blank lines (shown above), escaped newlines, and certain quoted characters, as explained in the following sections.

[1] This function has a confusing name.

Escaped newline, nextrune()

`assline()`[53b] relies on the function `nextrune()` below to read the next character from the input buffer `bp`. `nextrune()` is essentially a wrapper over `Bgetrune()` from the `libbio` library.

⟨*function* `nextrune` 54a⟩≡ (189b)
```
/*
 * get next character stripping escaped newlines
 * the flag specifies whether escaped newlines are to be elided or
 * replaced with a blank.
 */
int
nextrune(Biobuf *bp, bool elide)
{
    int c;

    for (;;) {
        c = Bgetrune(bp);
        ⟨nextrune() if escape character 54c⟩
        ⟨nextrune() handle mkinline 54b⟩
        return c;
    }
}
```

⟨`nextrune()` *handle mkinline* 54b⟩≡ (54a)
```
if (c == '\n')
    mkinline++;
```
Uses `mkinline` 51c.

`nextrune()`[54a] must also handle escaped newlines. An *escaped newline* is a newline character prefixed by the special *escape character* `'\'`. As I mentioned in Section 2.1.1, the syntax of `mk` (and Make) is minimalist. For example, a variable definition consists simply of a name followed by an equal sign and a set of values separated by space and terminated by a newline. Thus, spaces and newlines have a meaning in `mk` (as opposed to most programming languages). However, if the list of values is very long, it would be convenient to split the list over multiple lines. This is why `mk` allows to split such definitions on multiple lines if each newline is preceded by the special character `'\'`; the newline is then said to be *escaped*. This is similar to what the C preprocessor `cpp` provides for defining long macros over multiple lines (see the COMPILER book [Pad16b]).

⟨`nextrune()` *if escape character* 54c⟩≡ (54a)
```
if (c == '\\') {
    if (Bgetrune(bp) == '\n') {
        // an escaped newline!
        mkinline++;
        if (elide)
            continue;
        // else
        return ' ';
    }
    // else, it was just \
    Bungetrune(bp);
}
```
Uses `mkinline` 51c.

There is a small self-reference worth noticing here: the escape character this code looks for, the backslash, must itself be written `'\\'` in the C source, because the backslash is also C's own escape character.

When `nextrune()` reads an escaped newline, it does not return the newline character to the caller `assline()`. Instead, it consumes this escaped newline and returns the character after (unless this character is again an escaped newline or if the second argument to `nextrune()` is `false`). By consuming the escaped newline, `nextrune()` will cause `assline()` to read more characters until the next true (non-escaped) newline.

Note that if '\' is not followed by a newline, `nextrune()` must just return the '\' character. However, `nextrune()` already went too far in the input buffer by reading an extra character (to check whether this character was a newline). This is why `libbio` provides the function `Bungetrune()` called above to go back in the input buffer, to "unread" the character, a classic technique used in many lexers in Plan 9.

Comments

`assline()`[53b] also recognizes and skips comments. As I said in Section 2.1.1, `mk` allows the user to add comments in his `mkfile` by prefixing a line with the special character '#'.

⟨`assline()` *other locals* 55a⟩≡ (53b)
```
int prevc;
```

⟨`assline()` *switch character cases* 55b⟩≡ (53b) 56a ▷
```
case '#':
    prevc = '#';
    // skip all characters in comment until newline
    while ((c = Bgetc(bp)) != '\n') {
        if (c < 0)
            goto eof;
        prevc = c;
    }
    mkinline++;
   ⟨assline() when processing comments, if escaped newline 55e⟩
   ⟨assline() when processing comments, if not only comment on the line 55c⟩
    // else, skip lines with only a comment
    break;
```
Uses `mkinline` 51c.

A comment can be alone on its line, or it can be used at the end of a variable definition or rule, as in `FOO=1 # true`. When a comment is not alone on its line, the characters before the comments are not skipped but returned instead by `assline()`:

⟨`assline()` *when processing comments, if not only comment on the line* 55c⟩≡ (55b)
```
if (!isempty(buf)) {
     insert(buf, '\0');
     return true;
 }
```
Uses `insert()` 173a and `isempty` 172f.

The `prevc` local variable above is used to handle escaped newlines in a comment as in

⟨*example of escaped newline* 55d⟩≡
```
A=foo # this is a long definition mixed with a comment\
  bar
```

In that case, the definition will be parsed as the single line `A=foo bar`.

⟨`assline()` *when processing comments, if escaped newline* 55e⟩≡ (55b)
```
if (prevc == '\\')
    break;  /* propagate escaped newlines??*/
```

Quoted characters

Assembling a line sounds like an easy ask, but as you have just seen `assline()`[53b] is not trivial: it must handle escaped newlines (through `nextrune()`[54a]), blank lines, and comments.

The use of the special character '#' to denote comments introduces in turn another complication for `assline()`. Indeed, what if the target or prerequisite in a rule contains a '#' in its filename? We do not

want assline() to skip all the characters following that '#' in the rule. In fact, certain filenames in a project may contain other special characters used by mk such as ':', '=', '<', or even space.

To reference filenames using special characters, you must *quote* them in order for mk to not interpret them. This is a feature found in most programming languages. When assline() reads a line thats contains a quote, the '#' inside the quote has a different meaning; it is not a comment anymore.

You can configure mk to use either the Plan 9 shell rc (see the SHELL book [Pad18]) or a Bourne-alike shell (e.g., bash). However, each shell has its own escaping rules: sometimes a single quote, sometimes double quotes, or sometimes the antislash character, hence the different cases below:

⟨assline() *switch character cases* 56a⟩+≡ (53b) ◁55b 131a▷
```
case '\'':
case '"':
case '\\':
    rinsert(buf, c);
    if (escapetoken(bp, buf, true, c) == ERROR_0)
        Exit();
    break;
```
Uses escapetoken() 56b and rinsert() 173b.

As I mentioned before, mk tries to reuse as much as possible the syntax of the shell. This is why the cases of assline() above are as generic as possible and delegate instead the shell escaping policy to the shell-specific escapetoken() function.

The function below is the escapetoken() for rc in mk/rc.c[186a]. Like assline(), it reads characters from the input buffer bp, and fills the string buffer buf with those characters, but this time until the next quote (not until the next newline).

⟨*function* escapetoken*(rc.c)* 56b⟩≡ (186a)
```
/*
 * Input an escaped token.  Possible escape chars are single-quote,
 * double-quote and backslash.  Only the first is a valid escape for
 * rc; the others are just inserted into the receiving buffer.
 */
error0
escapetoken(Biobuf *bp, Bufblock *buf, bool preserve, int esc)
{
    int c;
    int line = mkinline;

    if(esc != '\'')
        return OK_1; // " and \ are not valid escape for rc, only '

    while((c = nextrune(bp, false)) > 0){
        if(c == '\''){
            if(preserve)
                rinsert(buf, c);
            ⟨escapetoken() return, unless quote quote 57a⟩
        }
        // else
        rinsert(buf, c);
    }
    // must have reached EOF
    SYNERR(line);
    fprint(STDERR, "missing closing %c\n", esc);
    return ERROR_0;
}
```
Uses SYNERR 51d, mkinline 51c, nextrune() 54a, and rinsert() 173b.

Of course, since "'" is now a special character reserved to quote special characters, how do you quote ' itself?

C itself faces this very question, and its answer is visible in the code just above: a single quote is written `'\''`, the backslash marking the quote as data rather than as syntax.

Another common technique found in many programming languages is to double the escaping or quoting character (as shown for C in the code of `assline()` above, where the antislash character is doubled). Thus, in `rc` and `mk`, two `'` inside a quoted string are interpreted as a single `'`. For example, `mk` interprets `'foo''bar'` as a filename containing a single quote between the strings `foo` and `bar` (`foo'bar`). Here is the code to handle two quotes:

⟨`escapetoken()` *return, unless quote quote* 57a⟩≡ (56b)

```
c = Bgetrune(bp);
if (c < 0)
    break; // eof
if(c != '\''){
    Bungetrune(bp);
    return OK_1;
}
// else, '', so continue the while loop
```

5.1.2 Parsing the head of a line: `rhead()`

Once `parse()`[51e] assembled a line, it can analyze the line with `rhead()`[57b] to return the special character separator involved in the line. `rhead()` also sets the second and third arguments passed by address to contain the list of words on the left (the head `h`) and right (the tail `t`) of the separator:

⟨*function* `rhead` 57b⟩≡ (188a)

```
static int
rhead(char *line, Word **h, Word **t,    int *attr, char **prog)
{
    char *p;
    int sep; // one of : = <
    ⟨rhead() other locals 70d⟩

    p = charin(line, ":=<");
    if(p == nil)
        return '?';
    // else
    sep = *p;
    *p++ = '\0';
    ⟨rhead() adjust sep if dynamic mkfile <| 134a⟩
    ⟨rhead() adjust attr and prog 70c⟩

    // potentially expand variables in head
    *h = stow(line);
    ⟨rhead() sanity check h 58a⟩
    // potentially expand variables in tail
    *t = stow(p);

    return sep;
}
```

`rhead()` relies on the function `charin()`[58c] to find one of the characters mentioned in the second argument of `charin()` in the string passed in the first argument; I will describe `charin()` soon.

The address of the separator character is stored first in the local variable `p`. The content of `p` is saved in the other local variable `sep`, before being overwritten by the end-of-string null character as illustrated in the middle of Figure 5.1. At this point, `line` and `p` point to the start of two independent strings. Both strings are then processed by `stow()`[60b] (for "string to words"). `stow()` splits the content of a string in a list of words, as shown at the bottom of Figure 5.1.

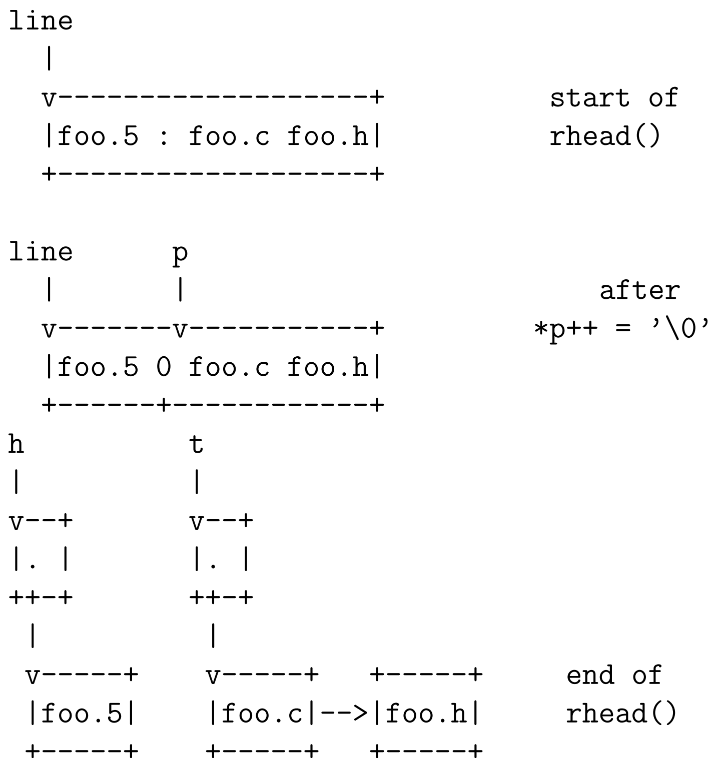


Figure 5.1: Evolution of local variables in `rhead()`.

Note that if the separator in the line is `'<'` (for an inclusion instruction), it is normal for `h` to be empty; otherwise, `mk` should report an error to the user:

⟨`rhead()` *sanity check h* 58a⟩≡ (57b)

```
if(empty_words(*h) && sep != '<' && sep != '|') {
    SYNERR(mkinline-1);
    fprint(STDERR,
           "no var (or target) on left side of assignment (or rule)\n");
    Exit();
}
```

⟨*macro* `empty_words` 58b⟩≡ (182d)

```
#define empty_words(w) ((w) == nil || (w)->s == nil || (w)->s[0] == '\0')
```

Finding special characters, `charin()`

I mentioned the functions `charin()`[58c] a few times before. `addrule()`[36b] called `charin()` to check whether the target contains a special pattern character. `rhead()`[57b] calls `charin()` to get the position of the `':'`, `'='`, or `'<'` character in a line. In both cases, `charin()` does not just look for a character in a string; it must also handle quoted characters.

`charin()` below essentially iterates over the characters in `cp` by incrementing `cp` until the start of `cp` points to one of the characters in `pat`:

⟨*function* `charin` 58c⟩≡ (186a)

```
/// addrule | rhead -> <>
/*
 * Search a string for characters in a pattern set.
 * Characters in quotes and variable generators are escaped.
 */
char*
```

```
charin(char *cp, char *pat)
{
    Rune r;
    int n;
    bool vargen = false;

    while(*cp){
        n = chartorune(&r, cp);
        switch(r){
        ⟨charin() switch rune cases 59a⟩
        default:
            if(utfrune(pat, r) && !vargen)
                return cp;
            break;
        }
        cp += n;
    }
    ⟨charin() sanity check vargen 136a⟩
    return nil;
}
```

`chartorune()`, called above, is a function from the C library (see the LIBCORE book [Pad16c]). It is similar to `Bgetrune()` used in `nextrune()`[54a] before, but operates on a plain string instead of a string buffer. In both cases, the string is a sequence of bytes using the UTF-8 encoding and terminated by the null character. `utfrune()` is another function from the C library we mentioned before. It checks whether a rune is part of one of the characters in a set of characters.

The local variable `vargen` is used to handle variable generator, an advanced feature of `mk` I will explain later in Section 11.4.

Skipping quoted characters

Just like `assline()`[53b] needs special code to handle quoted strings (because they may contain a `'#'` that needs to be treated differently), `charin()`[58c] needs also special code to handle quoted strings, because they may contain one of the special characters `rhead()`[57b] is looking for (`':'`, `'='`, or `'<'`).

⟨`charin()` *switch rune cases* 59a⟩≡ (58c) 135b ▷

```
case '\'':   /* skip quoted string */
    cp = squote(cp+1); /* n must = 1 */
    if(!cp)
        return nil;
    break;
```

Uses `squote()` 59b.

⟨*function* `squote` 59b⟩≡ (186a)

```
/*
 * skip a token in single quotes.
 */
static char *
squote(char *cp)
{
    Rune r;
    int n;

    while(*cp){
        n = chartorune(&r, cp);
        if(r == '\'') {
            ⟨squote() return, unless quote quote 60a⟩
        }
```

```
        cp += n;
    }
    SYNERR(-1);  /* should never occur */
    fprint(STDERR, "missing closing '\n");
    return nil;
}
```

Uses SYNERR 51d.

⟨squote() *return, unless quote quote* 60a⟩≡ (59b)

```
n += chartorune(&r, cp+n);
if(r != '\'')
    return cp;
// else, double '', continue while loop
```

5.1.3 Splitting a string in words: stow()

After rhead()[57b] found the position of the special character in the line assembled by assline()[53b], rhead() calls stow() to split in multiple words the strings on the left and right parts of the special character. The space character marks the boundaries between words. The code of stow() below is very simple because it delegates most of the complexity to nextword(), which I will explain after.

⟨*function* stow 60b⟩≡ (192b)

```
Word *
stow(char *s)
{
    // list<ref_own<Word>>
    Word *head, *new;
    // option<ref<Word>>
    Word *lastw;

    head = lastw = nil;
    while(*s){
        new = nextword(&s);
        if(new == nil)
            break;

        // head = concat_list(head, new)
        if (lastw)
            lastw->next = new;
        else
            head = lastw = new;

        while(lastw->next)
            lastw = lastw->next;

    }
    ⟨stow() if head still nil 60c⟩
    return head;
}
```

Uses nextword() 61.

Note that nextword() does not return a string but a list of words, for reasons I will explain soon. This is why stow() must concatenate the lists in head and new, not just adding one word to a list of words.

If head remains still nil after stow() processed s, then stow() does not return nil but instead returns the empty word.

⟨stow() *if head still nil* 60c⟩≡ (60b)

```
if (!head)
    head = newword("");
```

Uses newword() 32d.

An empty word is a list containing one word where the string is just the null character (see the code of `newword()`[32d]).

Assembling the next words, `nextword()`

The code for **`nextword()`** sounds trivial again: look for the next space character in the string as the separation marker between two words. However, again, **`nextword()`** must also handle quoted strings because they may contain a space that must not be treated as a word separator. In fact, **`nextword()`** must also handle variables and other features of **`mk`**, as explained in the following sections.

⟨*function* `nextword` 61⟩≡ (192b)

```
/// stow -> <>
/*
 * break out a word from a string handling quotes, executions,
 * and variable expansions.
 */
static Word*
nextword(char **s)
{
    char *cp = *s;
    Bufblock *buf;
    Rune r;
    // list<ref_own<Word>>
    Word *head;
    // option<ref<Word>>
    Word  *lastw;
    bool empty;
    ⟨nextword() other locals 63b⟩

    buf = newbuf();

restart:
    head = lastw = nil;
    empty = true;
    ⟨nextword() skipping leading white space 62a⟩

    while(*cp){
        cp += chartorune(&r, cp);
        switch(r)
        {
        case ' ':
        case '\t':
        case '\n':
            goto out;
        ⟨nextword() switch rune cases 62b⟩
        default:
            empty = false;
            rinsert(buf, r);
            break;
        }
    }
out:
    *s = cp;
    if(!isempty(buf)){
        ⟨nextword() when buffer not empty, if there was already an head 67a⟩
         else {
            insert(buf, '\0');
            head = newword(buf->start);
        }
```

```
    }
    freebuf(buf);
    return head;
}
```

Uses freebuf() 172e, insert() 173a, isempty 172f, newbuf() 172d, newword() 32d, and rinsert() 173b.

The presence of the `restart` label above, as well as the local variables `empty` and `lastw` will become clear later. They are needed because certain strings can expand in other strings that need to be reprocessed by `nextword()`[61].

`nextword()` first skips the leading white spaces in the string:

⟨nextword() *skipping leading white space* 62a⟩≡ (61)

```
while(*cp == ' ' || *cp == '\t')  /* leading white space */
    cp++;
```

Thus, there is no difference between writing a rule like `foo.5:foo.c` and `foo.5:  foo.c`, or between a definition like `FOO=a b` and `FOO = a b`.

Expanding quoted characters

As I mentioned before, `nextword()`[61] must handle quoted strings. As opposed to `assline()`[53b], which *inputs* a quoted string, or `charin()`[58c], which *skips* over a quoted string, `nextword()` *expands* a quoted string and stores the expansion in the buffer `b`. Indeed, `stow()`[60b] and `nextword()` are the last steps in the parsing phase; there is no need to keep the quotes (or quote quote inside those quotes) anymore.

⟨nextword() *switch rune cases* 62b⟩≡ (61) 63c▷

```
case '\'':
case '"':
case '\\':
    empty = false;
    cp = expandquote(cp, r, buf);
    if(cp == nil){
        fprint(STDERR, "missing closing quote: %s\n", *s);
        Exit();
    }
    break;
```

Uses expandquote() 62c.

⟨*function* expandquote*(rc.c)* 62c⟩≡ (186a)

```
/*
 * extract an escaped token.  Possible escape chars are single-quote,
 * double-quote, and backslash.  Only the first is valid for rc. The
 * others are just inserted into the receiving buffer.
 */
char*
expandquote(char *s, Rune r, Bufblock *buf)
{
    if (r != '\'') {
        // " and \ are not valid escape for rc, only '
        rinsert(buf, r);
        return s;
    }
    // else
    while(*s){
        s += chartorune(&r, s);
        if(r == '\'') {
            ⟨expandquote() return, unless quote quote 63a⟩
        }
        rinsert(buf, r);
```

```
    }
    return nil;
}
```
Uses rinsert() 173b.

⟨expandquote() *return, unless quote quote* 63a⟩≡ (62c)
```
if(*s == '\'')
    s++; // skip one of the two quotes
else
    return s;
```

Expanding variables

The expansion of variables in rules, variable definitions, and inclusions is done at parsing-time by `nextword()`[61], just like the expansion of quotes (and backquotes, as explained in Section 11.2). A single variable can expand to multiple words because a variable holds a list of words in `mk`.

⟨nextword() *other locals* 63b⟩≡ (61)
```
// list<ref_own<Word>
Word *w;
```

This is why `nextword()` returns a list of words, and why `stow()`[60b] concatenate a list of words; what may seem like a single word (a variable) can expand to multiple words.

The code of `nextword()` below relies on `varsub()`[63d] to return the value of a variable. `varsub()` can also substitute certain variables, as explained in Section 11.4, hence its name.

⟨nextword() *switch rune cases* 63c⟩+≡ (61) ◁62b
```
case '$':
    w = varsub(&cp);
    ⟨nextword() when in variable case, if w is nil 64b⟩
    empty = false;
    ⟨nextword() when in variable case, if non-space chars before var 65c⟩
    ⟨nextword() when in variable case, if head is not empty 65d⟩
    else
        head = lastw = w;
    while(lastw->next)
        lastw = lastw->next;
    break;
```
Uses varsub() 63d.

`varsub()` in turn relies on `varname()`[64d] to extract the name of the variable from the string pointed by `s` (`varname()` also increments `s` by side effect), and `varmatch()`[65b] to grab the value of the variable from the symbol table:

⟨*function* varsub 63d⟩≡ (192b)
```
/// nextword -> <>
Word*
varsub(char **s)
{
    Bufblock *buf;
    Word *w;

    ⟨varsub() if variable starts with open brace 136b⟩
    // else
    buf = varname(s);
    ⟨varsub() sanity check buf 64a⟩
    w = varmatch(buf->start);

    freebuf(buf);
```

```
        return w;
    }
```

Uses freebuf() 172e and varname() 64d.

Note that if the user mentions a variable not defined anywhere (neither in the mkfile nor in the environment), nextword() will skip (silently) over this variable:

⟨varsub() *sanity check buf* 64a⟩≡ (63d)
```
    if(buf == nil)
        return nil;
```

⟨nextword() *when in variable case, if w is nil* 64b⟩≡ (63c)
```
    if(w == nil){
       ⟨nextword() when in variable case, if w is nil and no char before 64c⟩
        break;
    }
```

In fact, if the variable is not defined and there was no character before the variable, as in FOO= $bar abc, then nextword() skips over the undefined variable and also skips again the possible whitespaces after the variable, hence the jump to restart below:

⟨nextword() *when in variable case, if w is nil and no char before* 64c⟩≡ (64b)
```
    if(empty)
        goto restart;
```

varname() below returns a buffer containing the name of a variable in s (without the leading $), and increments s to point after the variable name:

⟨*function* varname 64d⟩≡ (192b)
```
    /*
     * extract a variable name
     */
    static Bufblock*
    varname(char **s)
    {
        Bufblock *buf;
        char *cp = *s;
        Rune r;
        int n;

        buf = newbuf();
        for(;;){
            n = chartorune(&r, cp);
            if (!WORDCHR(r))
                break;
            rinsert(buf, r);
            cp += n;
        }
       ⟨varname() sanity check buf 65a⟩
        *s = cp;
        insert(buf, '\0');
        return buf;
    }
```

Uses WORDCHR 64e, insert() 173a, newbuf() 172d, and rinsert() 173b.

What constitutes a variable name is mostly anything except the set of special characters in the macro below:

⟨*function* WORDCHR 64e⟩≡ (182d)
```
    #define WORDCHR(r) ((r) > ' ' && !utfrune("!\"#$%&'()*+,-./:;<=>?@[\\]^`{|}~", (r)))
```

⟨varname() *sanity check buf* 65a⟩≡ (64d)
```
  if (isempty(buf)){
      SYNERR(-1);
      fprint(STDERR, "missing variable name <%s>\n", *s);
      freebuf(buf);
      return nil;
  }
```
Uses SYNERR 51d, freebuf() 172e, and isempty 172f.

Here is finally varmatch() called from varsub():

⟨*function* varmatch 65b⟩≡ (192b)
```
  static Word*
  varmatch(char *name)
  {
      Word *w;
      Symtab *sym;

      sym = symlook(name, S_VAR, nil);
      if(sym){
          /* check for at least one non-NULL value */
          for (w = sym->u.ptr; w; w = w->next)
              if(w->s && *w->s)
                  return wdup(w);
      }
      return nil;
  }
```

Edge cases

Note that if a variable is preceded directly by non-space characters, like in the middle of Figure 5.2 with '@', the first word in the list of words in the value of the variable must be adjusted:

⟨nextword() *when in variable case, if non-space chars before var* 65c⟩≡ (63c)
```
  if(!isempty(buf)){
      bufcpy(buf, w->s, strlen(w->s));
      insert(buf, '\0');
      free(w->s);
      // adjust the first word
      w->s = strdup(buf->start);

      resetbuf(buf);
  }
```
Uses bufcpy() 173c, insert() 173a, isempty 172f, and resetbuf 172g.

Moreover, if a variable is followed by another variable, as at the top of Figure 5.3, the last word of the first variable must be merged with the first word of the second variable:

⟨nextword() *when in variable case, if head is not empty* 65d⟩≡ (63c)
```
  if(head){
      // merge the last and first words
      bufcpy(buf, lastw->s, strlen(lastw->s));
      bufcpy(buf, w->s, strlen(w->s));
      insert(buf, '\0');
      free(lastw->s);
      lastw->s = strdup(buf->start);

      lastw->next = w->next;
      free(w->s);
      free(w);
```

```
  cp           where A=foo bar
   |                 B=bar foo
  +v------+
s:|@$A$B: |                                     start of
  +-------+                                    nextword()



            start
   cp         | current
    |         | |
  +-v-----+   v-v---------+                    processing
s:|@$A$B: |  b:|@          |                       '@'
  +-------+   +-----------+

           start           start
             |current      current
     cp      | |              |
      |      v-v---------+   v----+------+
  +---v---+  b:|@          | b:|@foo|      |
s:|@$A$B: |   +-----------+   +----+------+ processing
  +-------+   +---+  +---+    +----+  +---+    '$A'
             w:|foo|->|bar|  w:|@foo|->|bar|
              +---+  +---+    +----+  +---+

              (before)          (after)

                 adjusting the first
                        word
```

Figure 5.2: `nextword()` edge cases (part 1).

```
    resetbuf(buf);
}
```
Uses `bufcpy()` 173c, `insert()` 173a, and `resetbuf` 172g.

Finally, if a variable is followed directly by non-space characters, as at the bottom of Figure 5.3, the last word must be adjusted:

⟨`nextword()` *when buffer not empty, if there was already an head* 67a⟩≡ (61)
```
if(head){
    cp = buf->current;
    bufcpy(buf, lastw->s, strlen(lastw->s));
    bufcpy(buf, buf->start, cp - buf->start);
    insert(buf, '\0');
    free(lastw->s);
    // adjust the last word
    lastw->s = strdup(cp);
}
```
Uses `bufcpy()` 173c and `insert()` 173a.

5.2 Parsing rules: `<target>:<prereqs>`

Now that you have seen the generic parts of the code to parse an `mkfile`, I can describe the specific parts with the actions in `parse()`[51e] to process the rules, definitions, and inclusions. Those actions are based on the information returned by `rhead()`[57b]: the special character in the line, as well as the list of words on the left and right parts of this special character. I will start in this section with the action to manage rules, when the special character returned by `rhead()` is `':'`.

⟨`parse()` *other locals* 67b⟩≡ (51e) 70b▷
```
char *body;
```

⟨`parse()` *switch rhead cases* 67c⟩+≡ (51e) ◁53a 71b▷
```
case ':':
    body = rbody(&in);
    addrules(head, tail, body,   attr, hline, prog);
    break;
```
Uses `addrules()` 39a and `rbody()` 69a.

The action above relies on `rbody()` to read the body of a rule, that is its recipe. I will explain `rbody()` in the next section. I have described `addrules()` called above in Section 3.3.4. `addrules()` can handle rules with multiple targets by calling `addrule()`[36b] for each target.

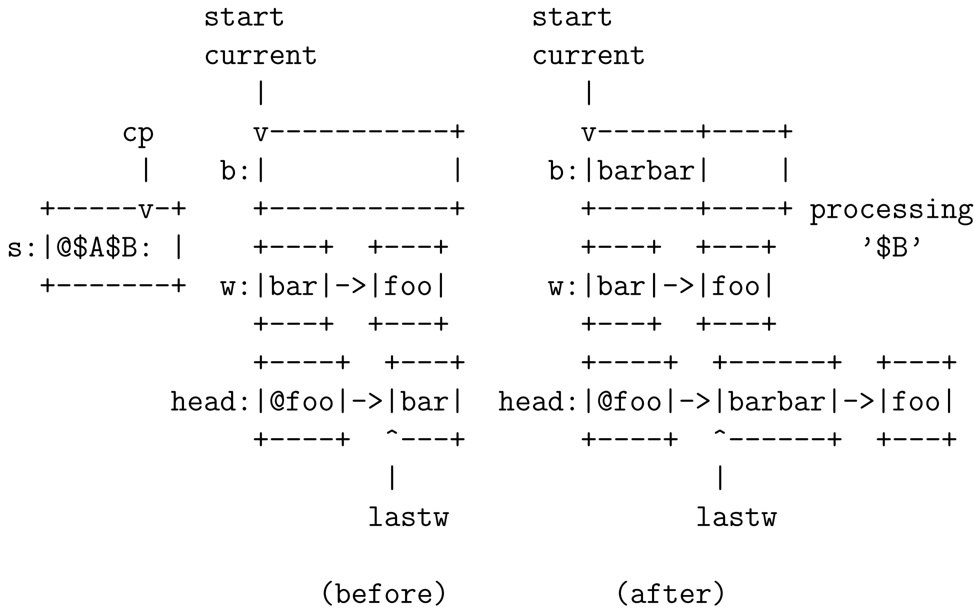
I mentioned also in Section 4.6.1 that `mk`, during parsing, sets the global `target1`[49b] to contain the first target found in the mkfile. Here is finally the code that sets `target1`:

⟨`addrules()` *set* `target1` 67d⟩≡ (39a)
```
/* tuck away first non-meta rule as default target*/
if(target1 == nil && !(attr&REGEXP)){
    for(w = targets; w; w = w->next)
        if(charin(w->s, "%&"))
            break;
    if(w == nil) // head does not contain any pattern
        target1 = wdup(targets);
}
```
Uses `REGEXP` 128a, `charin()` 58c, `target1` 49b, and `wdup()` 33c.

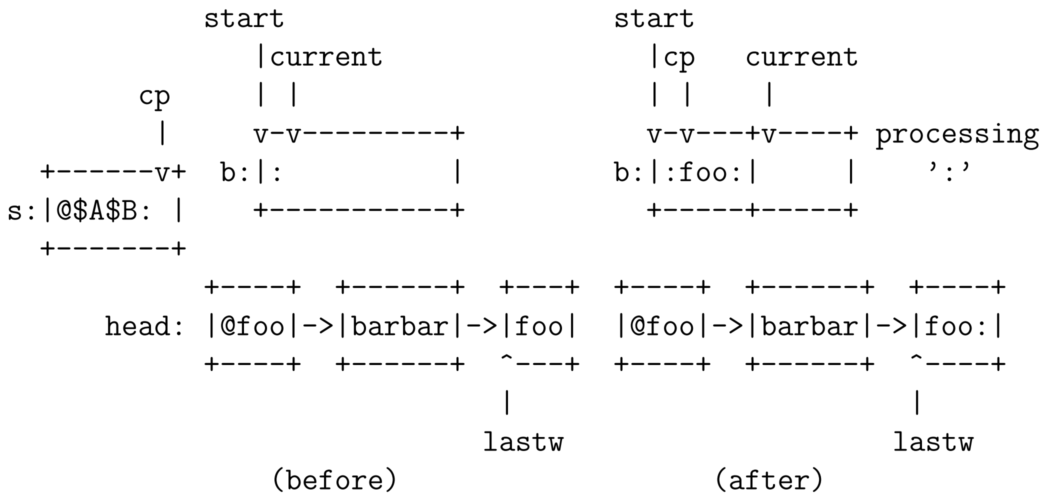
Note the use of `charin()`[58c] above to make sure the first target is not a pattern and so does not contain a special pattern character. Indeed, `mk` would not know how to instantiate this pattern to build a concrete target.

For the rule attribute `REGEXP`[128a] used above, see Section 11.1.

where A=foo bar
B=bar foo



merging the last and
first word



adjusting the last
word

Figure 5.3: `nextword()` edge cases (part 2).

5.2.1 Parsing the recipe: `rbody()`

`rbody()`[69a], like `assline()`[53b], takes as a parameter an input buffer `in`. The cursor in this buffer should now point to the character following the newline of the line containing the target and prerequisites of the rule (consumed by `assline()`). `rbody()` then fills its local string buffer `buf` until a non-spacing character is found in the first column. This character marks the end of the recipe and the start of a new rule, definition, or inclusion.

⟨*function* `rbody` 69a⟩≡ (188a)

```
static char *
rbody(Biobuf *in)
{
    Bufblock *buf;
    int r, lastr;
    char *p;

    lastr = '\n';
    buf = newbuf();

    for(;;){
        r = Bgetrune(in);
        if (r < 0)
            break; // eof
        // in first column?
        if (lastr == '\n') {
            ⟨rbody() if comment in first column 69c⟩
            else
              if (r != ' ' && r != '\t') {
                Bungetrune(in);
                break;
            }
        } else
            // not in first column
            rinsert(buf, r);

        lastr = r;
        ⟨rbody() handle mkinline 69b⟩
    }

    insert(buf, '\0');
    p = strdup(buf->start);
    freebuf(buf);

    return p;
}
```

Uses `freebuf()` 172e, `insert()` 173a, `newbuf()` 172d, and `rinsert()` 173b.

⟨`rbody()` *handle mkinline* 69b⟩≡ (69a)

```
if (r == '\n')
    mkinline++;
```

Uses `mkinline` 51c.

If `rbody()` finds a comment in the first column, this comment can not be the start of a rule, definition, or inclusion, so it is added in the buffer for the recipe:

⟨`rbody()` *if comment in first column* 69c⟩≡ (69a)

```
if (r == '#')
    rinsert(buf, r); // the shell recognize comments too
```

Uses `rinsert()` 173b.

If a rule has no recipe, `rbody()` returns the empty string to its caller `parse()`[51e]. Note that the empty string is not the same thing than `nil`. Indeed, an empty string contains one byte: the end-of-string null character `'\0'`.

Note also that `mk` does not impose to use a `TAB` in the first column like Make. A space character is also valid. Moreover, you can use more than one space. You can also write multiple shell commands on multiple lines as long as they all have a leading spacing character in the first column. You do not need to escape newlines in a recipe (as you have to do in a variable definition).

5.2.2 Parsing rule attributes

`mk` allows to customize certain rules by using *rule attributes*. An attribute often used is the attribute to indicate that the target in a rule does not correspond to a filename. In that case, `mk` should not expect from the recipe to generate such a target. For instance, many `mkfiles` use the target `clean` to cleanup a directory. In `mk`'s terminology, such a target is called a *virtual target* (see Section 11.5.1 for a full explanation).

In GNU Make, the use of `.PHONY:` followed by a string in a `Makefile` indicates that the string is a virtual target. In `mk`, the syntax to add attributes to a rule is to add non-spacing characters after the first `':'` of a rule, and to add another `':'` after the non-spacing characters, as in the following rule:

⟨*tests/mkfile/mkclean* 70a⟩≡
```
clean:V:
    rm -f *.5 $PROG $LIB
```

Each character between the two `':'` can correspond to a different attribute.

Rule attributes are stored in `Rule.attr`[35h], but before they are stored in a local variable in `parse()`[51e]:

⟨`parse()` *other locals* 70b⟩+≡ (51e) ◁67b 71a▷
```
// bitset<Rule_attr>
int attr;
```

This variable is passed by address to `rhead()`[57b] (see the call to `rhead()` in `parse()`). `rhead()` then initializes this variable and adjusts it depending on the separator:

⟨`rhead()` *adjust* `attr` *and* `prog` 70c⟩≡ (57b)
```
*attr = 0; // Nothing
*prog = nil;
// variable attributes
⟨rhead() if sep is = 74e⟩
// rule attributes
⟨rhead() if sep is : 70e⟩
```

Finally, `rhead()` modifies `attr` in the (hidden) cases of the `switch` below:

⟨`rhead()` *other locals* 70d⟩≡ (57b) 143g▷
```
Rune r;
int n;
```

⟨`rhead()` *if sep is* : 70e⟩≡ (70c)
```
if((sep == ':') && *p && (*p != ' ') && (*p != '\t')){
    while (*p) {
        n = chartorune(&r, p);
        if (r == ':')
            break;
        p += n;
        switch(r)
        {
        ⟨rhead() when parsing rule attributes, switch rune cases 128b⟩
        default:
            SYNERR(-1);
            fprint(STDERR, "unknown attribute '%c'\n", p[-1]);
```

```
            Exit();
        }
    }
    if (*p++ != ':') {
eos:
        SYNERR(-1);
        fprint(STDERR, "missing trailing :\n");
        Exit();
    }
}
```

Most of the rule attributes correspond to advanced features of `mk` I will describe in Section 11.5.

5.3 Parsing included files: `< <file>`

I will now describe the action to manage inclusions, when the special character returned by `rhead()`[57b] is `'<'`.

A file inclusion in an `mkfile` will result in the opening of a new file, hence the following additional variables in `parse()`[51e]:

⟨`parse()` *other locals* 71a⟩+≡ (51e) ◁70b 73b▷

```
char *p;
fdt newfd;
```

Here is the code using `newfd`:

⟨`parse()` *switch rhead cases* 71b⟩+≡ (51e) ◁67c 73c▷

```
case '<':
    p = wtos(tail, ' ');
    ⟨parse() when parsing included file, sanity check p 71c⟩
    newfd = open(p, OREAD);
    ⟨parse() when parsing included file, sanity check newfd 71d⟩
    else
        // recurse
        parse(p, newfd, false);
    break;
```

Uses `parse()` 51e and `wtos()` 33d.

⟨`parse()` *when parsing included file, sanity check p* 71c⟩≡ (71b)

```
if(*p == '\0'){
    SYNERR(-1);
    fprint(STDERR, "missing include file name\n");
    Exit();
}
```

Uses `SYNERR` 51d.

⟨`parse()` *when parsing included file, sanity check newfd* 71d⟩≡ (71b)

```
if(newfd < 0){
    fprint(STDERR, "warning: skipping missing include file: ");
    perror(p);
}
```

Note that `tail` above contains the list of words on the right of `'<'` while `head`, which contains the list of words on the left, should be empty. Just like for the rules, this list of words is set in `rhead()` and is the result of a call to `stow()`[60b], which performs quote and variable expansions. Thus, you can also use variables in the filename to include, as in

⟨*tests/mkfile/mkincludearc* 71e⟩≡

```
</$objtype/mkfile
```

I described wtos()[33d] called above in Section 3.2. It converts a list of words back to a string. In practice, this list should contain only one element for file inclusions.

To include a file, mk simply calls recursively parse(). However, the globals infile[51a] and mkinline[51c] must be saved before the call and restored after, hence the following calls in parse():

⟨parse() *start, push* 72a⟩≡ (51e)
```
  ipush();
```
Uses ipush() 72f.

⟨parse() *end, pop* 72b⟩≡ (51e)
```
  ipop();
```
Uses ipop() 73a.

Both functions use the following structure to remember the list of "parent" files in the stack of opened files.

⟨*struct* input 72c⟩≡ (188a)
```
  struct Input
  {
      char *file;
      int line;

      // Extra
      ⟨Input extra fields 72e⟩
  };
```

⟨*global* inputs 72d⟩≡ (188a)
```
  // list<ref_own<Input>> (next = Input.next)
  static struct Input *inputs = nil;
```
Uses Input 72c and inputs-13 72d.

⟨Input *extra fields* 72e⟩≡ (72c)
```
  // list<ref_own<Input>> (head = inputs)
  struct Input *next;
```
Uses Input 72c.

⟨*function* ipush 72f⟩≡ (188a)
```
  void
  ipush(void)
  {
      struct Input *in, *me;

      me = (struct Input *)Malloc(sizeof(*me));
      // saving globals
      me->file = infile;
      me->line = mkinline;
      me->next = nil;

      // add_list(me, inputs)
      if(inputs == nil)
          inputs = me;
      else {
          for(in = inputs; in->next; )
              in = in->next;
          in->next = me;
      }
  }
```
Uses Input 72c, Malloc() 171a, infile 51a, inputs-13 72d, and mkinline 51c.

⟨*function* `ipop` 73a⟩≡ (188a)
```
void
ipop(void)
{
    struct Input *in, *me;

    assert(/*pop input list*/ inputs != nil);
    // me = pop_list(inputs)
    if(inputs->next == nil){
        me = inputs;
        inputs = nil;
    } else {
        for(in = inputs; in->next->next; )
            in = in->next;
        me = in->next;
        in->next = nil;
    }
    // restoring globals
    infile = me->file;
    mkinline = me->line;
    free((char *)me);
}
```
Uses `Input` 72c, `infile` 51a, `inputs-13` 72d, and `mkinline` 51c.

5.4 Parsing Variable definitions: `<var>=<values>`

The final action of `parse()`[51e] manages variable definitions, when the special character returned by `rhead()`[57b] is `'='`.

⟨`parse()` *other locals* 73b⟩+≡ (51e) ◁71a 134c▷
```
bool set = true;
```

⟨`parse()` *switch rhead cases* 73c⟩+≡ (51e) ◁71b 134d▷
```
case '=':
    ⟨parse() when parsing variable definitions, sanity check head 73d⟩
    ⟨parse() when parsing variable definitions, override handling 74b⟩
    if(set){
        setvar(head->s, (void *) tail);
        ⟨parse() when parsing variable definitions, extra setting 123a⟩
    }
    ⟨parse() when parsing variable definitions, if variable with attr 146g⟩
    break;
```
Uses `setvar()` 31b.

The code above relies mainly on the function `setvar()`[31b] to add an entry in the `S_VAR`[29a] namespace in the symbol table.

Note that for variable definitions, the list of words on the left of `'='` should contain only one element:

⟨`parse()` *when parsing variable definitions, sanity check head* 73d⟩≡ (73c)
```
if(head->next){
    SYNERR(-1);
    fprint(STDERR, "multiple vars on left side of assignment\n");
    Exit();
}
```
Uses `SYNERR` 51d.

Note also that both `head` and `tail` used above are the results of calls to `stow()`[60b] in `rhead()`, which expands variables. Thus, you can use variables on the right side of `'='`, but also more surprisingly on the left as in the following example:

⟨*indirection in mkfile example* 74a⟩≡
```
A=B
D=d e f
$A = a b c $D
# => B contains a b c d e f
```

5.4.1 Overriding variable definitions

As I mentioned in Section 4.3, you can override variable definitions in an `mkfile` (or in files included from this `mkfile`) by passing definitions through the command-line, as in `mk objtype=arm CFLAGS=-g`. In that case, `mk` creates a temporary file containing those command-line definitions and calls `parse()`[51e] with `true` for its `varoverride` parameter (see the code in Section 4.3). Here is the code using this `varoverride` parameter:

⟨`parse()` *when parsing variable definitions, override handling* 74b⟩≡ (73c)
```
if(symlook(head->s, S_OVERRIDE, nil)){
    set = varoverride;
} else {
    set = true;
    if(varoverride)
        symlook(head->s, S_OVERRIDE, (void *)"");
}
```
Uses `S_OVERRIDE` 74c and `symlook()` 30a.

The code above relies on the new namespace `S_OVERRIDE`, which contains the set of variables defined through the command-line (and whose definitions can not be overriden by definitions in the `mkfile`).

⟨`Sxxx` *cases* 74c⟩+≡ (29a) ◁37b 82a▷
```
S_OVERRIDE, /* can't override */
```

5.4.2 Parsing variable attributes

Variables, likes rules, can have attributes. The syntax for variable attributes uses a scheme similar to the rule attributes (see Section 5.2.2): the special character `'='` is doubled and attributes reside between the two special characters, as in the following example:

⟨*mkfile using a private variable* 74d⟩≡
```
MYVAR=U= foo.5 bar.5 # a private variable
```

Here is the code in `rhead()`[57b] to extract variable attributes:

⟨`rhead()` *if sep is* = 74e⟩≡ (70c)
```
if(sep == '='){
    pp = charin(p, termchars);      /* termchars is shell-dependent */
    if (pp && *pp == '=') {
        while (p != pp) {
            n = chartorune(&r, p);
            switch(r)
            {
            ⟨rhead() when parsing variable attributes, switch rune cases 146f⟩
            default:
                SYNERR(-1);
                fprint(STDERR, "unknown attribute '%c'\n",*p);
                Exit();
            }
            p += n;
```

```
        }
        p++;                /* skip trailing '=' */
    }
}
```

Variable attributes correspond to advanced features of `mk` rarely used. I will describe those attributes and the cases of the `switch` above in Section 11.6.

The code above relies on the following constant to check whether a line contains a variable attribute:

⟨*global* `termchars` 75a⟩≡ (186a)

```
char *termchars = "=' \t"; /*used in parse.c to isolate assignment attribute*/
```

Uses `termchars` 75a.

Note that `termchars` does not contain just `'='`. Indeed, variable definitions can contain quoted string containing the equal character, as in `A='U=1'`, in which case the equal sign inside the quote should not be interpreted as the end mark of variable attributes. This is why the code above is looking for the first character that is either an equal or quote and stops there. In fact, `termchars` contains also spacing characters to remove some possible ambiguities as shown in the following example:

⟨*mkfile using space to disambiguate variable attributes* 75b⟩≡

```
MYVAR=U=a b c d  # private var MYVAR containing the list: a b c d
MYVAR= U=a b c d # MYVAR contains the list: U=a b c d
```

Chapter 6

Building the Graph of Dependencies

The next component in the building pipeline is the construction of the graph of dependencies. Section 2.1.2 showed a few examples of graph of dependencies for small projects. You will see in this chapter how `mk` builds those graphs.

The most important function in this chapter is `graph()`[76], which takes a target name as a parameter and returns the root of the graph of dependencies for this target. `graph()` will use the globals `rules`[35b] and `metarules`[35e] populated by `parse()`[51e] (via `addrules()`[39a]) in Section 5.2. I will also describe in this chapter the code to check for user mistakes in the rules. Those mistakes translate in issues while building the graph, for instance, the presence of cycles in the graph.

6.1 graph() and applyrules()

The graph is built *statically* before any recipe is executed. This separation is a key design choice: by computing the entire dependency graph first, `mk` can detect errors (cycles, ambiguous rules) before doing any work, and can plan parallel execution. GNU Make, by contrast, interleaves graph construction with execution, which makes parallelization harder.

`graph()` is mostly a wrapper around `applyrules()`[77a], which is the function containing the logic to build the graph of dependencies.

⟨*function* `graph` 76⟩≡ (191a)

```
/// main -> mk -> <>
Node*
graph(char *target)
{
    Node *root;
    ⟨graph() other locals 85e⟩

    ⟨graph() set cnt for infinite rule detection 86a⟩
    root = applyrules(target, cnt);
    ⟨graph() free cnt 86c⟩

    ⟨graph() checking the graph 83c⟩
    ⟨graph() propagate attributes 140a⟩

    return root;
}
```

Uses `applyrules()` 77a.

I will present the code to check the graph in Section 6.6.

I mentioned briefly in Section 2.5 the algorithm behind `applyrules()`. The algorithm first builds a node corresponding to the target (via `newnode()`[40a]), then finds the rules or meta rules with matching targets, and

finally calls recursively `applyrules()` on the prerequisites of those matching rules and meta rules. The algorithm stops when a node is a leaf, which happens when a node has no matching rules or rules with no prerequisites. Here is the skeleton of `applyrules()`:

⟨*function* applyrules 77a⟩≡ (191a)
```
static Node*
applyrules(char *target, char *cnt)
{
    Node *node;
    // list<ref<Arc> (next = Arc.next, last = lasta, head elt to skip)
    Arc head;
    // ref<Arc>
    Arc *lasta = &head;
    ⟨applyrules other locals 77b⟩

    ⟨applyrules debug 168h⟩
    ⟨applyrules check node cache if target is already there 82c⟩
    // else

    target = strdup(target);
    node = newnode(target); // calls timeof() internally
    head.next = nil;
    ⟨applyrules other initializations 130e⟩

    // apply regular rules with target as a head (modifies lasta)
    ⟨applyrules() apply regular rules 77c⟩

    // apply meta rules (modifies lasta)
    ⟨applyrules() apply meta rules 78e⟩

    node->arcs = head.next;

    return node;
}
```
Uses `newnode()` 40a.

The code above relies on a local variable `head` containing an `Arc`[40f] allocated in the stack, and another local variable `lasta` pointing originally to this arc. This is a standard idiom in C allowing later to write code adding an element in a list without having to worry whether this list is originally empty or not (as the code of `mk` does for example in `addrule()`[36b] with the list of rules, or in Section 4.6.3 with a list of words).

I will explain the code to apply rules and meta rules in the next two sections.

6.2 Finding the simple rule(s) for a target

The first step to find the dependencies of a node is to look for all the rules mentioning the node as a target. Fortunately, `applyrules()`[77a] can rely on the symbol table and the `S_TARGET`[37b] namespace to quickly access all the rules mentioning a specific target (as explained in Section 3.3.4):

⟨applyrules *other locals* 77b⟩≡ (77a) 79a ▷
```
Symtab *sym;
Rule *r;
Word *pre;
Arc *arc;
```

⟨applyrules() *apply regular rules* 77c⟩≡ (77a)
```
sym = symlook(target, S_TARGET, nil);
r = sym? sym->u.ptr : nil;
for(; r; r = r->chain){
```

```
    ⟨applyrules() skip this rule and continue if some conditions 78a⟩
    ⟨applyrules() infinite rule detection part1 86e⟩
    ⟨applyrules() when found a regular rule for target node, set flags 90b⟩

    ⟨applyrules() if no prerequisites in rule r 78d⟩
    else
        for(pre = r->prereqs; pre; pre = pre->next){
            // recursive call!
            arc = newarc(applyrules(pre->s, cnt), r, "", rmatch);
            // add_list(head, arc)
            lasta->next = arc;
            lasta = lasta->next;
    }
    ⟨applyrules() infinite rule detection part2 86f⟩
}
```

Uses `S_TARGET` 37b, `applyrules()` 77a, `newarc()` 41c, and `symlook()` 30a.

As I mentioned before, `applyrules()` simply calls itself recursively for each prerequisite of a matching rule and adds a new arc between the current node and the root of the subgraph returned by `applyrules()`.

Some tools such as `gcc -MM` (which can be used to generate a `.depend` file included from your `mkfile`) can generate rules without neither a recipe nor prerequisites (e.g., `foo.5:  #no deps`). `applyrules()` can simply skip those rules:

⟨`applyrules()` *skip this rule and continue if some conditions* 78a⟩≡ (77c)

```
if(empty_recipe(r) && empty_prereqs(r))
      continue; /* no effect; ignore */
```

Uses `empty_prereqs` 78c and `empty_recipe` 78b.

The code above relies on two macros that factorize the checks for empty recipe and empty prerequisites:

⟨*macro* `empty_recipe` 78b⟩≡ (182d)

```
#define empty_recipe(r) (!r->recipe || !*r->recipe)
```

⟨*macro* `empty_prereqs` 78c⟩≡ (182d)

```
#define empty_prereqs(r) (!r->prereqs || !r->prereqs->s || !*r->prereqs->s)
```

Some rules may have a recipe but no prerequisites, for instance, when the target is a virtual target (see Section 11.5.1). In those cases, `mk` still adds an arc to the node, but without a destination node. You will see later code using those "fake" arcs.

⟨`applyrules()` *if no prerequisites in rule r* 78d⟩≡ (77c)

```
// no prerequisites, a leaf, still create fake arc
if(empty_prereqs(r)) {
    arc = newarc((Node *)nil, r, "", rmatch);
    // add_list(head, arc)
    lasta->next = arc;
    lasta = lasta->next;
}
```

Uses `empty_prereqs` 78c and `newarc()` 41c.

6.3 Finding matching metarules

The other step to find the dependencies of a node is to look for metarules containing a target that matches the node. This time, `applyrules()`[77a] needs to go through all the meta rules stored in the global `metarules`[35e]:

⟨`applyrules()` *apply meta rules* 78e⟩≡ (77a)

```
for(r = metarules; r; r = r->next){
    ⟨applyrules() skip this meta rule and continue if some conditions 79d⟩
    ⟨applyrules() if regexp rule then continue if some conditions 130f⟩
```

```
    else {
        if(match(node->name, r->target, stem)) {
            ⟨applyrules() infinite rule detection part1 86e⟩

            ⟨applyrules() if no prerequisites in meta rule r 79e⟩
            else
                for(pre = r->prereqs; pre; pre = pre->next) {
                    ⟨applyrules() if regexp rule, adjust buf and rmatch 130g⟩
                    else
                        subst(stem, pre->s, buf, sizeof(buf));
                    // recursive call!
                    arc = newarc(applyrules(buf, cnt), r, stem, rmatch);
                    // add_list(head, arc)
                    lasta->next = arc;
                    lasta = lasta->next;
                }
             ⟨applyrules() infinite rule detection part2 86f⟩
        }
    }
}
```

Uses `applyrules()` 77a, `metarules` 35e, `newarc()` 41c, and `subst()` 80c.

The code above relies on the functions `match()`[80a] and `subst()`[80c], which I will describe fully in the next two sections. `match()` checks whether a string can match another string called the *template*. This template can contain a pattern character (e.g., `foo%.5`), as explained in Section 2.1.1. If the template matches the string, `match()` then stores the string matched by the pattern character, called the *stem*, in a buffer passed in its last parameter. Here is the stem buffer `applyrules()` passes to `match()`:

⟨applyrules *other locals* 79a⟩+≡ (77a) ◁77b 79c▷

```
  char stem[NAMEBLOCK];
```

Uses `NAMEBLOCK` 79b.

⟨*constant* NAMEBLOCK 79b⟩≡ (182d)

```
  #define NAMEBLOCK 1000
```

`subst()` then substitutes the stem in another template (e.g., `foo%.c`) and stores the result in another buffer passed as a parameter. Here is the output buffer `applyrules()` passes to `subst()`:

⟨applyrules *other locals* 79c⟩+≡ (77a) ◁79a 130d▷

```
  char buf[NAMEBLOCK];
```

Uses `NAMEBLOCK` 79b.

Just like for the simple rules, `mk` can ignore certain meta rules:

⟨applyrules() *skip this meta rule and continue if some conditions* 79d⟩≡ (78e) 142k▷

```
  if(empty_recipe(r) && empty_prereqs(r))
      continue; /* no effect; ignore */
```

Uses `empty_prereqs` 78c and `empty_recipe` 78b.

Some meta rules can also contain virtual targets:

⟨applyrules() *if no prerequisites in meta rule r* 79e⟩≡ (78e)

```
  if(empty_prereqs(r)) {
      arc = newarc((Node *)nil, r, stem, rmatch);
      // add_list(head, arc)
      lasta->next = arc;
      lasta = lasta->next;
  }
```

Uses `empty_prereqs` 78c and `newarc()` 41c.

The code above is almost identical to code in Section 6.2, except `applyrules()` passes here the `stem` buffer to `newarc()`[41c] instead of the empty string.

6.3.1 Matching a pattern: `match()`

`match()` below iterates over two string parameters (`name` and `template`) at the same time by incrementing both pointers until it finds the special pattern character (`'%'` or `'&'`) in the `template` parameter. Once it found the pattern character, it computes the length of the string matched by the pattern character and it makes sure the rest of the template also matches the end of the name. Finally it modifies the `stem` buffer passed from `applyrules()`[77a]. Figure 6.1 illustrates how `match()` works on a simple example.

⟨*function* `match` 80a⟩≡ (187b)

```
bool
match(char *name, char *template,    char *stem)
{
    Rune r;
    int n;

    // Before the pattern character
    while(*name && *template){
        n = chartorune(&r, template);
        if (PERCENT(r))
            break;
        while (n--)
            if(*name++ != *template++)
                return false;
    }

    // On pattern character
    if(!PERCENT(*template))
        return false;
    // how many characters % is matching
    n = strlen(name) - strlen(template+1);
    if (n < 0)
        return false;

    // After the pattern character
    if (strcmp(template+1, name+n))
        return false;

    strncpy(stem, name, n);
    stem[n] = '\0';

    ⟨match() if ampersand template 80b⟩

    return true;
}
```

As I mentioned in Section 3.3.3, `mk` supports two kinds of pattern characters: `'%'` and `'&'`. The `'&'` pattern can not match filenames containing a dot or a slash:

⟨`match()` *if ampersand template* 80b⟩≡ (80a)

```
if(*template == '&')
    return !charin(stem, "./");
```

6.3.2 Substituting the stem: `subst()`

Figure 6.2 illustrates how `subst()` works on a simple example.

⟨*function* `subst` 80c⟩≡ (187b)

```
void
subst(char *stem, char *template,   char *dest, int dlen)
{
```

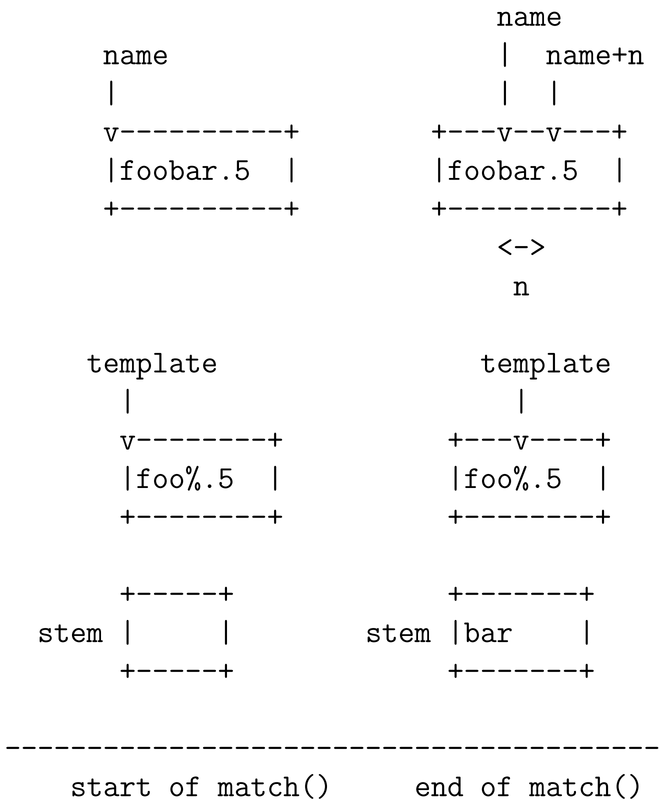


Figure 6.1: `match("foobar.5", "foo\%.5", ...)`.

```
    Rune r;
    char *s, *e;
    int n;

    e = dest + dlen - 1;
    while(*template){
        n = chartorune(&r, template);
        if (PERCENT(r)) {
            template += n;
            for (s = stem; *s; s++)
                if(dest < e)
                    *dest++ = *s;
        } else
            while (n--){
                if(dest < e)
                    *dest++ = *template;
                template++;
            }
    }
    *dest = '\0';
}
```

Uses `PERCENT` 35g.

6.4 Node cache

As I mentioned in Section 2.1.2, the graph of dependencies can be more than a simple tree; it can also be a direct acyclic graph (DAG). Thus, before creating a new node for a target, `applyrules()` makes sure the target

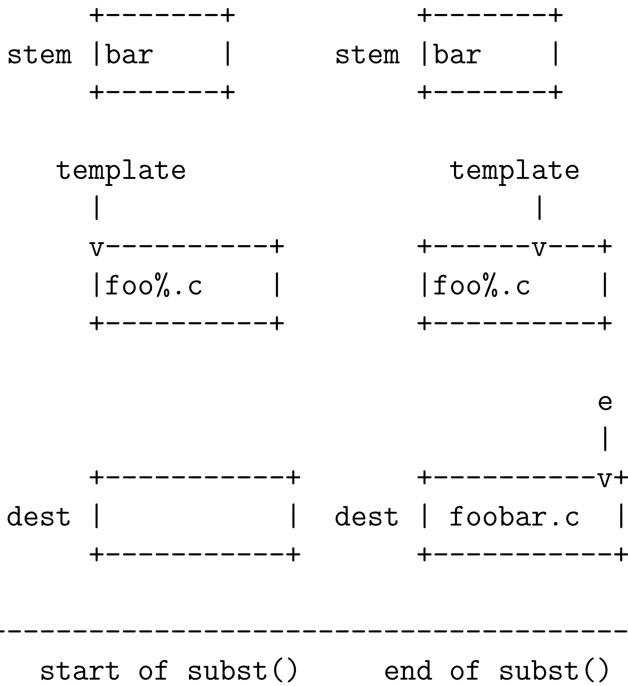


Figure 6.2: `subst("bar", "foo\%.c", ...)`.

node does not exist already. To do so, it relies on the new namespace below:

⟨`Sxxx` *cases* 82a⟩+≡ (29a) ◁74c 122b▷
```
  S_NODE,  /* target name -> node */
```

Each time `newnode()`[40a] creates a new node for a given file, it adds this node in the symbol table before returning it (e.g., to `applyrules()`).

⟨`newnode()` *update node cache* 82b⟩≡ (40a)
```
  symlook(name, S_NODE, (void *)node);
```
Uses `S_NODE` 82a and `symlook()` 30a.

Then, `applyrules()` can consult the symbol table and possibly returns a pointer to a previously created node instead of recomputing the subtree for this node:

⟨`applyrules` *check node cache if target is already there* 82c⟩≡ (77a)
```
  sym = symlook(target, S_NODE, nil);
  if(sym)
      return sym->u.ptr;
```
Uses `S_NODE` 82a and `symlook()` 30a.

Creating a single node per file is important not only to avoid unnecessary calls to `applyrules()` but also because once `mk` finished executing a recipe, `mk` must update the modification time of the target created by the recipe by modifying a single node.

6.5 `timeof()`

The last important function used to build the graph of dependencies is `timeof()`[82d] called from `newnode()`[40a] to label a node. Indeed, as I explained in Section 2.1.2, each node is labeled with the modification time of the file it represents.

⟨*function* `timeof` 82d⟩≡ (189c)
```
  /// newnode -> <> -> mkmtime -> (bulktime; dirstat (libc))
  ulong
```

```
timeof(char *name, bool force)
{
    ⟨timeof() locals 157b⟩

    ⟨timeof() if name archive member 150f⟩
    if(force)
        return mkmtime(name, true);
    // else
    ⟨timeof() if not force, use time cache 157c⟩
}
```

I will explain in Section 11.10.3 the `force` argument.

⟨*function* `mkmtime` 83a⟩≡ (199)

```
ulong
mkmtime(char *name, bool force)
{
    Dir *d;
    ulong t;
    ⟨mkmtime locals 157g⟩

    ⟨mkmtime() bulk dir optimisation 158a⟩
    d = dirstat(name);
    ⟨mkmtime() check if inexistent file 83b⟩
    t = d->mtime;
    free(d);

    return t;
}
```

Note that if the file does not exist, `mkmtime()`[193c] and `timeof()` return `0`:

⟨`mkmtime()` *check if inexistent file* 83b⟩≡ (83a)

```
if(d == nil)
    return 0;
```

6.6 Checking the graph and the rules

Once `applyrules()`[77a] returned the graph of dependencies, `graph()`[76] can check for special properties of the graph that are signs of mistakes in the `mkfile`:

⟨`graph()` *checking the graph* 83c⟩≡ (76)

```
cyclechk(root);

⟨graph() before ambiguous() 90c⟩
ambiguous(root);
```

Uses `ambiguous()` 87 and `cyclechk()` 84b.

There are a few mistakes `mk` can detect, as explained in the following sections.

6.6.1 Cycle detection

The first error `mk` can detect is the presence of a *cycle* in the graph, as shown in Figure 6.3. Here is an example of an `mkfile` that leads to the graph in Figure 6.3:

⟨*tests/mk/mkfile-with-cycle* 83d⟩≡

```
foo: foo.5 bar.5
    5l -o foo foo.5 bar.5
%.5: %.c
```

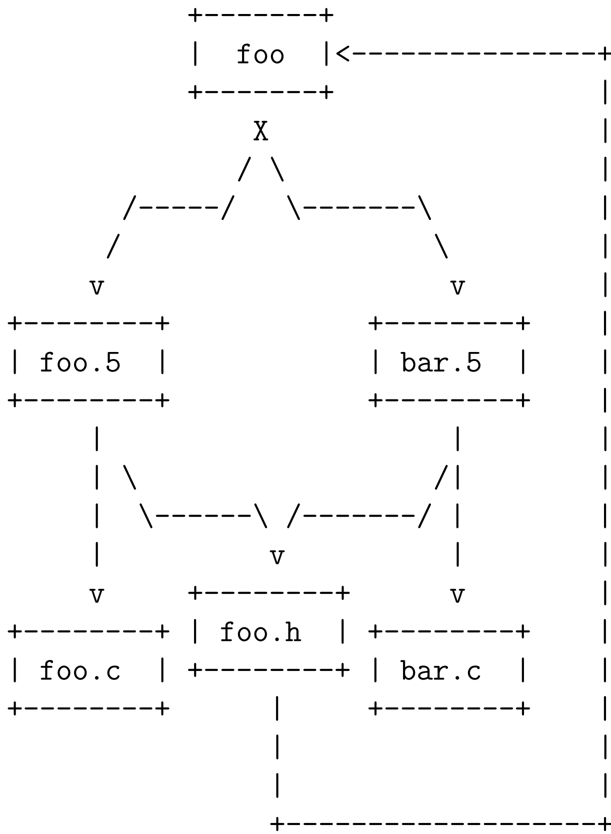


Figure 6.3: A cycle in a graph of dependencies.

```
    5c -c $stem
foo.5: foo.h
bar.5: foo.h

VERSION=2
foo.h: foo    # typo, should be foo.x
    cat foo.x | sed -e s/VERSION/$VERSION/ > foo.h
```

In the `mkfile` above, the user made a typo and added a dependency from `foo.h` to `foo` instead of `foo.x`. Here is the error message displayed by `mk` for this mkfile:

```
$ mk -f mkfile-with-cycle
mk: cycle in graph detected at target foo
```

To detect the mistake above, `mk` performs a DFS on the graph and marks nodes with the special flag below when it visits a node:

⟨`Node_flag` *cases* 84a⟩+≡ (40c) ◁40d 89e▷

```
  CYCLE      = 0x0002,
```

This flag is stored in a unique bit in the `Node.flags`[40b] field; it will not conflict with the other node flags that I introduced in Section 3.4.1 (e.g., `NOTMADE`[40d] is `0x0020`).

The function below assumes it is called with the `CYCLE` flag unset for every nodes, which is true because `newnode()`[40a] set `Node.flags` to `0`.

⟨*function* `cyclechk` 84b⟩≡ (191a)

```
  static void
  cyclechk(Node *n)
  {
```

```
    Arc *a;

    if((n->flags&CYCLE)){
        fprint(STDERR, "mk: cycle in graph detected at target %s\n", n->name);
        Exit();
    }
    n->flags |= CYCLE;
    for(a = n->arcs; a; a = a->next)
        if(a->n)
            cyclechk(a->n);
    n->flags &= ~CYCLE;
}
```
Uses `CYCLE` 84a and `cyclechk()` 84b.

Note that it is important for `cyclechk()`[84b] to remove the `CYCLE` flag at the very end before returning. Indeed, the graph of dependencies can be a DAG, in which case the same node may be referenced from multiple parents (e.g., `foo.h` in Figure 6.3). Without the last instruction in `cyclechk()`, `mk` would report another (this time wrong) cycle when visiting `foo.h` for the second time from `bar.5` instead of `foo.5`.

6.6.2 Infinite rule detection

The second mistake `mk` can detect is the presence of meta rules that `mk` could instantiate infinitely. Here is an example of an `mkfile` illustrating the issue:

⟨*tests/mk/mkfile-infinite* 85a⟩≡
```
all: foo bar

%: %.5
   5l $stem.5 -o $stem
%.5: %.c
 5c -c $stem.c
```

When building the graph for `foo`, `applyrules()`[77a] can instantiate first the meta rule `%: %.5` with `% = foo`. However, `applyrules()` when called recursively with `foo.5` as a target could again instantiate the meta rule `%: %.5`, this time with `% = foo.5` [1]. This process could go forever, which would stuck `mk`.

To avoid this infinite process, `mk` still allows the use of the meta rule above, but severely restricts its application. Indeed, `mk` allows to apply a specific rule only once during the building of a branch in the graph of dependencies. To do so, each rule gets first a unique *rule identifier* stored in `Rule.rule`:

⟨`Rule` *other fields* 85b⟩+≡ (34) ◁39b 128c▷
```
int   rule;  /* rule number */
```

⟨*global* `nrules` 85c⟩≡ (189a)
```
static int nrules = 0;
```
Uses `nrules-24` 85c.

⟨`addrule()` *set more fields* 85d⟩+≡ (36b) ◁51b 143d▷
```
r->rule = nrules++;
```
Uses `nrules-24` 85c.

Then, before calling `applyrules()`, `graph()`[76] initializes a map that will count how many times `applyrules()` used a rule:

⟨`graph()` *other locals* 85e⟩≡ (76)
```
// map<ruleid, int>
char *cnt;
```

[1] This is one of the motivations for the special pattern character `'&'`.

⟨graph() *set cnt for infinite rule detection* 86a⟩≡ (76)
```
cnt = rulecnt();
```
Uses rulecnt() 86b.

⟨*function* rulecnt 86b⟩≡ (189a)
```
char*
rulecnt(void)
{
    char *s;

    s = Malloc(nrules);
    memset(s, 0, nrules);
    return s;
}
```
Uses Malloc() 171a and nrules-24 85c.

⟨graph() *free cnt* 86c⟩≡ (76)
```
free(cnt);
```

graph() then passes this map as the second argument to applyrules() (see Section 6.1). Finally, applyrules() modifies cnt when it uses a rule before possibly calling itself recursively. applyrules() can then skip the rule if it already used the rule in a parent call to applyrules():

⟨*global* nreps 86d⟩≡ (191a)
```
int nreps = 1;
```
Uses nreps 86d.

⟨applyrules() *infinite rule detection part1* 86e⟩≡ (78e 77c)
```
if(cnt[r->rule] >= nreps)
    continue;

cnt[r->rule]++;
```
Uses nreps 86d.

Note that applyrules() must decrement the rule counter after the recursive call to itself. Indeed, it is ok to instantiate multiple times the same meta rule in independent branches of the graph of dependencies, for instance, the meta rule %.5: %.c with the foo.5 and bar.5 nodes in Figure 6.3.

⟨applyrules() *infinite rule detection part2* 86f⟩≡ (78e 77c)
```
cnt[r->rule]--;
```

6.6.3 Ambiguous rules detection

The third error mk can detect is the use of *ambiguous rules*. For example, in the mkfile below, if both foo.c and foo.h are more recent than foo.5, mk can not decide which recipe to run to update foo.5.

⟨*tests/mk/mkfile-ambiguous* 86g⟩≡
```
all: foo

foo: foo.5
    5l -o foo foo.5

foo.5: foo.c
    5c -c foo.c
foo.5: foo.h
    5c -g -c foo.c
```

Here is the error message displayed by `mk` for this mkfile:

```
$ mk -f mkfile-ambiguous
mk: ambiguous recipes for foo.5:
 foo.5 <-(mk-ambiguous-simple:7)- foo.c
 foo.5 <-(mk-ambiguous-simple:9)- foo.h
```

To check for ambiguities, `mk` goes through every nodes and checks whether a node has two arcs with different recipes. As I said in Section 2.1.2, `mk` allows to write multiple rules involving the same target as long as only one rule, the master rule, has a recipe.

⟨*function* `ambiguous` 87⟩≡ (191a)

```
static void
ambiguous(Node *n)
{
    Arc *a;
    Rule *master_rule = nil;
    Arc *master_arc = nil;
    bool error_reported = false;

    for(a = n->arcs; a; a = a->next){
        // recurse
        if(a->n)
            ambiguous(a->n);

        // arcs without any recipe do not generate ambiguity
        if(empty_recipe(a->r))
            continue;
        // else

        // first arc with a recipe (so no ambiguity)
        if(master_rule == nil) {
            master_rule = a->r;
            master_arc = a;
        }
        else{
            ⟨ambiguous() give priority to simple rules over meta rules 88d⟩
            if(master_rule->recipe != a->r->recipe){
                if(!error_reported){
                    fprint(STDERR, "mk: ambiguous recipes for %s:\n", n->name);
                    error_reported = true;
                    trace(n->name, master_arc);
                }
                trace(n->name, a);
            }
        }
    }
    if(error_reported)
        Exit();
    ⟨ambiguous() get rid of all skipped arcs 89b⟩
}
```

Uses `ambiguous()` 87, `empty_recipe` 78b, and `trace()` 88a.

Note that before reporting an ambiguity, the code above checks whether the recipes in the two different arcs are different (using a simple pointer comparison, not `strcmp()`). Having a node with multiple arcs does not necessarily mean ambiguity. For example, `foo` in Figure 6.3 has arcs to `foo.5` and `bar.5`. However those two arcs share the same recipe (the linking command), so there is no ambiguity in building `foo`. When `mk` builds the graph, it splits simple rules such as `foo: foo.5 bar.5 ...` in multiple arcs, but it remembers also that all those arcs come from the same rule.

`ambiguous()`[87] calls `trace()` below for each ambiguous arc:

⟨*function* `trace` 88a⟩≡ (191a)
```
static void
trace(char *s, Arc *a)
{
    fprint(STDERR, "\t%s", s);
    while(a){
        fprint(STDERR, " <-(%s:%d)- %s", a->r->file, a->r->line,
            a->n? a->n->name:"");
        ⟨trace() possibly continue if prereq is also a target 88b⟩
        else
            a = nil;
    }
    fprint(STDERR, "\n");
}
```

⟨`trace()` *possibly continue if prereq is also a target* 88b⟩≡ (88a)
```
if(a->n){
    for(a = a->n->arcs; a; a = a->next)
        if(*a->r->recipe)
            break;
}
```

Specialized versus generic rules

It is common to want to write a generic meta rule that can handle most files and some specialized rules for a few files. For example, only a few files may require to be compiled with special flags, as in the following example:

⟨*tests/mk/mk-generic-specialized* 88c⟩≡
```
foo: foo.5 bar.5 foobar.5

%.5: %.c
    5c -c $stem.c

foobar.5: foobar.c
    5c -c -D VERSION=1 foobar.c
```

Normally, `mk` should report an ambiguity for the `mkfile` above. Indeed, `foobar.5` matches the meta rule, which would lead to a second arc from `foobar.5` to `foobar.c` in the graph of dependencies, with the recipe of the meta rule. However, `mk` allows ambiguity between two rules when one of the two rules is a simple rule. *Simples rules have priority over meta rules*, thanks to the code below:

⟨`ambiguous()` *give priority to simple rules over meta rules* 88d⟩≡ (87)
```
if(master_rule->recipe != a->r->recipe){
    if((master_rule->attr&META) && !(a->r->attr&META)){
        master_rule = a->r;
        master_arc->remove = true;
        master_arc = a;
    } else if(!(master_rule->attr&META) && (a->r->attr&META)){
        a->remove = true;
        continue;
    }
}
```
Uses `META` 36a.

The code above not only adjusts the master rule to be the specialized rule, it also marks the arc containing the meta rule as "to-be-removed", thanks to the following field:

⟨`Arc` *other fields* 88e⟩+≡ (40f) ◁41b 129d▷
```
short remove;
```

⟨newarc() *set other fields* 89a⟩≡ (41c) 129e ▷
```
a->remove = false;
```

Adjusting the graph, togo()

ambiguous()[87], before returning, removes all the arcs marked as to-be-removed on the current node:

⟨ambiguous() *get rid of all skipped arcs* 89b⟩≡ (87)
```
togo(n);
```
Uses togo() 89c.

⟨*function* togo 89c⟩≡ (191a)
```
static void
togo(Node *node)
{
    Arc *a;
    Arc *preva = nil;

    /* delete them now */
    for(a = node->arcs; a; preva = a, a = a->next)
        if(a->remove){
            //remove_list(a, node->arcs)
            if(a == node->arcs)
                node->arcs = a->next;
            else {
                preva->next = a->next;
                a = preva;
            }
        }
}
```

Vacuous nodes removal

As we have just seen before, the use of a specialized simple rule and a generic meta rule can lead to unfortunate ambiguities. mk handles some ambiguities by giving priorities to specialized rules, which is convenient. The use of multiple meta rules can also lead to unfortunate ambiguities. For example, in the mkfile below, foo.5 could be generated either from a foo.c C file or from a foo.s assembly file.

⟨*tests/mk/mkfile-vacuous* 89d⟩≡
```
all: foo

foo: foo.5 bar.5
   5l foo.5 bar.5 -o foo
%.5: %.c
   5c -c $stem.c
%.5: %.s
   5a $stem.s
```

Again, mk should normally report an ambiguity with this mkfile because there will be two arcs from foo.5 with two different recipes in the graph of dependencies (one arc to foo.c and another one to foo.s). However, it is convenient to factorize rules for different programming languages with different meta rules. In practice, only one of the two meta rules above should apply for each source file in a project. The directory of the project above should contain either foo.c or foo.s, but not both.

This is why mk allows the use of multiple meta rules that can conflict with each other if it can detect that certain nodes instantiated from certain meta rules (e.g., foo.c, foo.s) are *vacuous* because they do not correspond to existing files. To detect and remove vacuous nodes, mk relies first on the following node flag:

⟨Node_flag *cases* 89e⟩+≡ (40c) ◁ 84a 90e ▷
```
PROBABLE   = 0x0100,
```

`mk` marks nodes as `PROBABLE` when they contain an existing file, which can be checked by looking at the modification time of the file in `newnode()`[40a] (remember that `timeof()`[82d] returns `0` for inexistent files):

⟨`newnode()` *adjust flags of node* 90a⟩≡ (40a)
```
if(node->time)
    node->flags = PROBABLE;
```
Uses `PROBABLE` 89e.

In the example above, `mk` would mark `foo.c` as `PROBABLE` but not `foo.s` if the directory contains only `foo.c`.

Moreover, `mk` marks also nodes mentioned as targets of simple rules as `PROBABLE`:

⟨`applyrules()` *when found a regular rule for target* `node`*, set flags* 90b⟩≡ (77c)
```
node->flags |= PROBABLE;
```
Uses `PROBABLE` 89e.

Finally, the root node is also marked as `PROBABLE` because it is not a node we want `mk` to remove, even if it does not correspond yet to an existing file:

⟨`graph()` *before* `ambiguous()` 90c⟩≡ (83c) 90d ▷
```
root->flags |= PROBABLE; /* make sure it doesn't get deleted */
```
Uses `PROBABLE` 89e.

Once `mk` created the nodes in the graph of dependencies, `graph()`[76] calls `vacuous()`[90g] to explore the graph to detect and remove vacuous nodes.

⟨`graph()` *before* `ambiguous()` 90d⟩+≡ (83c) ◁ 90c
```
vacuous(root);
```
Uses `vacuous()` 90g.

Note that `graph()` must call `vacuous()` before `ambiguous()`[87], to remove the vacuous nodes and arcs, otherwise `ambiguous()` would report ambiguities.

`vacuous()` relies on two extra node flags to operate:

⟨`Node_flag` *cases* 90e⟩+≡ (40c) ◁ 89e 90f ▷
```
VACUOUS     = 0x0200,
```

⟨`Node_flag` *cases* 90f⟩+≡ (40c) ◁ 90e 140e ▷
```
READY       = 0x0004,
```

`VACUOUS` marks vacuous nodes and `READY` marks nodes `vacuous()` already visited (the graph can be a DAG).

Here is finally the code of `vacuous()`:

⟨*function* `vacuous` 90g⟩≡ (191a)
```
static bool
vacuous(Node *node)
{
    Arc *a;
    bool vac = !(node->flags&PROBABLE);
    ⟨vacuous() other locals 91a⟩

    if(node->flags&READY)
        return node->flags&VACUOUS;
    node->flags |= READY;

    for(a = node->arcs; a; a = a->next)
        if(a->n && vacuous(a->n) && (a->r->attr&META))
            a->remove = true;
        else
            vac = false;
    ⟨vacuous possibly undelete some arcs 91b⟩

    togo(node);
```

```
    if(vac) {
        node->flags |= VACUOUS;
    }
    return vac;
}
```

Uses `META` 36a, `PROBABLE` 89e, `READY` 90f, `VACUOUS` 90e, `togo()` 89c, and `vacuous()` 90g.

Note that `vacuous()` removes only arcs derived from meta rules. In the example I mentioned before, `vacuous()` would mark `foo.c` as `PROBABLE` but not `foo.s`. Then, `vacuous()` would return true when exploring the `foo.s` node, because the node did not have the `PROBABLE` mark and the node does not have any children (so `vac` will remain true). Then, when `vacuous()` backtracks on the `foo.5` node, during the arc iteration, `vacuous()` will mark the arc to `foo.s` as to-be-removed.

⟨`vacuous()` *other locals* 91a⟩≡ (90g)

```
Arc *a2;
```

The arc loop above marks arcs for removal one at a time, but vacuousness is really a property of a *rule*, not of an individual arc. A single meta-rule can expand into several arcs (several prerequisites), and if even one of those prerequisites is non-vacuous — a real file the rule genuinely needs — then the rule applies and *all* of its arcs must stay, including ones that on their own looked vacuous. So the per-arc detection above is not wrong, just too fine-grained. This pass repairs the granularity: it scans the surviving arcs and, for any rule that still has a live arc (`a2->r == a->r`), un-removes the sibling arcs the first loop had flagged. Removal thus becomes all-or-nothing per rule, and `mk` drops an arc only when every arc that rule produced is vacuous.

⟨`vacuous` *possibly undelete some arcs* 91b⟩≡ (90g)

```
/* if a rule generated arcs that DON'T go; no others from that rule go */
for(a = node->arcs; a; a = a->next)
    if(!(a->remove))
        for(a2 = node->arcs; a2; a2 = a2->next)
            if((a2->remove) && (a2->r == a->r)){
                a2->remove = false;
            }
```

Chapter 7

Finding Outdated Files

The next component in the building pipeline is responsible for finding outdated files in the graph of dependencies built in Chapter 6. In this chapter, you will see the function containing the core algorithm behind `mk`. This function, appropriately named `mk()`[92], takes a target as a parameter, builds the graph of dependencies for this target (with `graph()`[76]), explores the graph to find outdated files, with a function called `work()`[95a], and schedules recipes to be executed to update those outdated files, with a function called `dorecipe()`[97e]. The following sections will explain the code of `mk()`, `work()`, and `dorecipe()`.

7.1 mk()

`mk()`[92] is arguably the most important function in `mk`. I outlined its core algorithm in Section 2.1.3 when I introduced the principles of a job scheduler, and in Section 2.5 when I presented the software architecture of `mk`. Figure 7.1 is a copy of Figure 2.6. The goal is just to remind you of the order in which `mk` visits the nodes of a graph during the DFS, and of the changes to the building status of nodes (`NOTMADE`[40d], `BEINGMADE`[40d], and `MADE`[40d]) during the DFS. In the scenario of Figure 7.1, the user modified `world.c`, which should trigger a compilation to regenerate `world.5` and a linking command to regenerate `hello`.

`mk()` operates in *waves*: each call to `work()`[95a] performs a DFS looking for outdated files. When it finds one whose prerequisites are all `MADE`, it schedules the recipe and marks the node `BEINGMADE`. When `work()` cannot schedule anything (because prerequisites are still `BEINGMADE`), `mk()` calls `waitup()`[107c] to wait for a running recipe to finish. The finished node transitions to `MADE`, and the next wave of `work()` can now schedule recipes that depended on it. This wave-based approach is what enables parallelism: `work()` can schedule multiple independent recipes in a single wave (up to `nproclimit`), and `waitup()` reaps whichever finishes first. The loop terminates when the root node is no longer `NOTMADE`—either everything is `MADE`, or a final `waitup()` drains the last running job.

Here is the code of `mk()`:

⟨*function* `mk` 92⟩≡ (191b)

```
/// main -> <> -> graph() ; clrmade(); work()
void
mk(char *target)
{
    Node *root;
    bool everdid = false;
    bool did = false;
    // enum<WaitupResult>
    int res;

    ⟨mk() initializations 102a⟩

    root = graph(target);
```

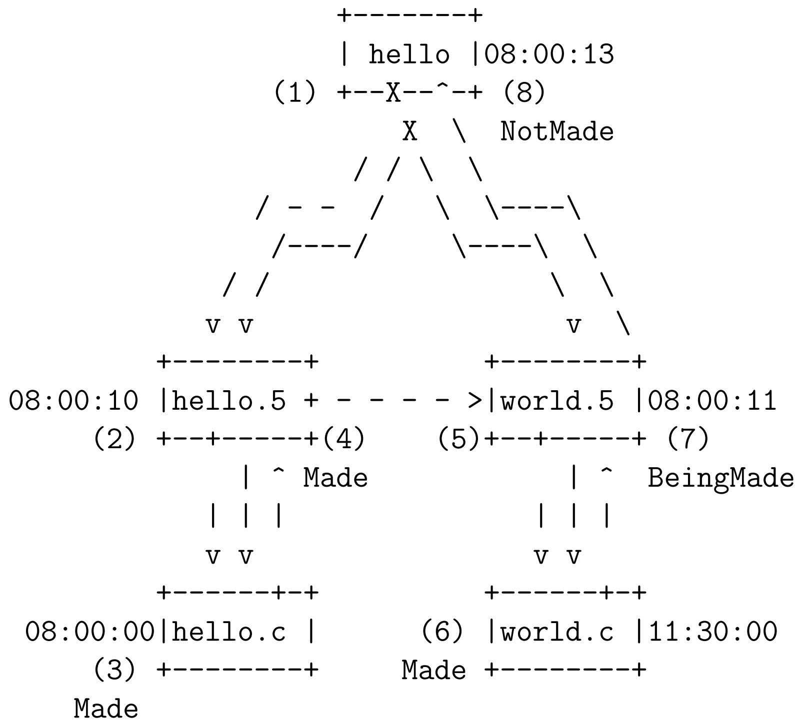


Figure 7.1: Graph of dependencies with building-status labels.

```
⟨mk() if DEBUG(D_GRAPH) 167a⟩
clrmade(root);

while(root->flags&NOTMADE){
    did = false;
    work(root, &did,   (Node *)nil, (Arc *)nil);
    if(did)
        everdid = true; /* found something to do */
    else {
        res = waitup(EMPTY_CHILDREN_IS_OK, (int *)nil);
        ⟨mk() if no child to waitup and root not MADE, possibly break 161h⟩
    }
}
if(root->flags&BEINGMADE)
    waitup(EMPTY_CHILDREN_IS_ERROR, (int *)nil);

⟨mk() before returning, more waitup() if there was an error 162b⟩
if(!everdid)
    Bprint(&bout, "mk: '%s' is up to date\n", root->name);
return;
}
```

Uses `BEINGMADE` 40d, `EMPTY_CHILDREN_IS_ERROR` 109b, `EMPTY_CHILDREN_IS_OK` 109b, `NOTMADE` 40d, `bout` 45b, `clrmade()` 94a, `graph()` 76, `waitup()` 107c, and `work()` 95a.

`mk()` operates in six steps:

1. `mk()` builds the graph of dependencies with `graph()`[76] (see Chapter 6).
2. `mk()` sets the building status of all nodes in the graph to `NOTMADE` with `clrmade()`[94a] (see Section 7.2).
3. `mk()` explores the graph of dependencies repeatedly with `work()` in successive waves to find outdated files (see Section 7.3). When `work()` finds up-to-date files, it marks them as `MADE`, (e.g., `hello.c` in Figure 7.1).

When it finds outdated files, `work()` schedules a recipe to be executed with `dorecipe()`[97e], and marks the node as `BEINGMADE`. For example, in Figure 7.1, `work()` discovered that `world.5` was older than `world.c` and so scheduled a recipe to update `world.5`; `work()` also marked `world.5` as `BEINGMADE`. `work()` also sets to true the `did` boolean passed by reference when it schedules a job.

4. Sometimes, `work()` does not schedule any new job and `did` stays false. One reason might be that there is nothing to build because everything is already up-to-date. Another reason is that `work()` may need to wait for some processes to finish. For example, in Figure 7.1, if a previous wave of `work()` scheduled `world.5` to be built (hence the `BEINGMADE` flag), another wave of `work()` might not be able to schedule anything because `hello` can be built only if all the object files are `MADE`.

   In that case, `mk()` calls `waitup()` (see Section 8.4) to wait for some user processes executing recipes to finish. `waitup()` will wait for a job to finish and will set as `MADE` the nodes that were the targets of the job. Thus, the next wave of `work()` will be able to progress.

5. At some point, `work()` will schedule a job for the root target in which case its building status will switch from `NOTMADE` to `BEINGMADE`. `mk()` will then wait one more time for the children process responsible for building the root target to finish.

6. `mk()` returns.

The code of `mk()` is small, but subtle. For example, the first call to `waitup()` in `work()` uses the flag `EMPTY_CHILDREN_IS_OK`[109b] to indicate that it is ok if `waitup()` does not find any children process to wait for. Indeed, everything was maybe already up-to-date, in which case `work()` will not schedule any job (`work()` will have marked the root node as `MADE`) and `did` will stay false. However, the second call to `waitup()` in `work()` uses the `EMPTY_CHILDREN_IS_ERROR`[109b] flag because `mk` knows at this point that there is still the process responsible to update the root target to wait for.

The last two parameters of `work()` contains the parent node and parent arc of the current node. Here in `mk()` we start from the root node, so both arguments are `nil`. You will see in Section 11.10.1 why `work()` needs such information for advanced features of `mk`.

7.2 Initializing nodes: `clrmade()`

`clrmade()` below simply marks all nodes as `NOTMADE`[40d]:

⟨*function* `clrmade` 94a⟩≡ (191b)
```
/// main -> mk -> <>
void
clrmade(Node *n)
{
    Arc *a;

    ⟨clrmade() n->flags pretend adjustments 154d⟩
    MADESET(n, NOTMADE);
    for(a = n->arcs; a; a = a->next)
        if(a->n)
            // recurse
            clrmade(a->n);
}
```
Uses `MADESET` 94b, `NOTMADE` 40d, and `clrmade()` 94a.

Note that `Node.flags`[40b] does not contain only the building status of a node. This is why the macro below makes sure it resets only the bits in `Node.flags` that contain the building status, and leaves unchanged the other bits.

⟨*function* `MADESET` 94b⟩≡ (182d)
```
#define MADESET(n,m) n->flags = (n->flags&~(NOTMADE|BEINGMADE|MADE))|(m)
```

7.3 Exploring the graph: `work()`

`work()` below performs a DFS on the graph while looking for outdated `NOTMADE`[40d] nodes:

⟨*function* `work` 95a⟩≡ (191b)
```
/// main -> mk -> <> -> outofdate(); dorecipe()
void
work(Node *node, bool *did,   Node *parent_node, Arc *parent_arc)
{
    ⟨work() locals 95c⟩

    ⟨work() debug 169a⟩
    if(node->flags&BEINGMADE)
        return;
    // else
    ⟨work() possibly unpretending node 155a⟩
    if(node->arcs == nil){
        ⟨work() no arcs, a leaf 95b⟩
    } else {
        ⟨work() some arcs, a node 96b⟩
    }
}
```
Uses `BEINGMADE` 40d.

The following two sections explain the code of `work()` when the node is a leaf and when it has children.

7.3.1 The leaf case

The leaf case is easy. When a node has no prerequisites and it contains an existing file, for instance, `hello.c` in Figure 7.1, then this file is already `MADE`[40d].

⟨`work()` *no arcs, a leaf* 95b⟩≡ (95a)
```
/* consider no prerequisite case */
if(node->time == 0){
    ⟨work() print error when inexistent file without prerequisites 95d⟩
} else
    MADESET(node, MADE);
```
Uses `MADE` 40d and `MADESET` 94b.

Otherwise, `mk` should report an error:

⟨`work()` *locals* 95c⟩≡ (95a) 95e▷
```
char cwd[256];
```

⟨`work()` *print error when inexistent file without prerequisites* 95d⟩≡ (95b)
```
if(getwd(cwd, sizeof cwd))
    fprint(STDERR, "mk: don't know how to make '%s' in directory %s\n", node->name, cwd);
else
    fprint(STDERR, "mk: don't know how to make '%s'\n", node->name);
⟨work() when inexistent target without prerequisites, if kflag 161e⟩
else
    Exit();
```

7.3.2 The parent case

The parent case is more complex. For the parent case, `work()`[95a] relies on the two important booleans below:

⟨`work()` *locals* 95e⟩+≡ (95a) ◁95c 96a▷
```
bool weoutofdate = false;
bool ready = true;
```

`weoutofdate` checks whether the current node is out-of-date with one of its children. `ready` checks whether the node is ready to be built because all its children are `MADE`[40d].

Here is finally the code of `work()` that handles nodes with children (and using the locals above):

⟨`work()` *locals* 96a⟩+≡ (95a) ◁95e 155c▷
```
Arc *a;
```

⟨`work()` *some arcs, a node* 96b⟩≡ (95a)
```
⟨work() adjust weoutofdate if aflag 159d⟩
/*
 *   now see if we are out of date or what
 */
for(a = node->arcs; a; a = a->next) {
    if(a->n){
        // recursive call! go in depth
        work(a->n, did,  node, a);

        if(a->n->flags&(NOTMADE|BEINGMADE))
            ready = false;
        if(outofdate(node, a, false)){
            weoutofdate = true;
            ⟨work() update ra when outofdate node with arc a 155d⟩
        }
    } else {
        if(node->time == 0){
            weoutofdate = true;
            ⟨work() update ra when no dest in arc and no src 155e⟩
        }
    }
}

if(!ready) /* can't do anything now */
    return;
if(!weoutofdate){
    MADESET(node, MADE);
    return;
}
⟨work() possibly pretending node 154e⟩
// else, out of date

dorecipe(node, did);
return;
```
Uses `BEINGMADE` 40d, `MADE` 40d, `MADESET` 94b, `NOTMADE` 40d, `dorecipe()` 97e, `outofdate()` 96c, and `work()` 95a.

There are a few important things to note about the code above. First, it is important for `work()` to call itself recursively before checking whether the current node is out-of-date with one of its children. Indeed, in Figure 7.1, the root node may appear to be up-to-date because all the object files are older than the executable. However, the root may still be out-of-date because deep in the graph a source file may be more recent than its object file. In that case, the object file is out-of-date and should be recompiled, which in turn makes the root node out-of-date. `work()` must first go in depth by calling itself recursively to perform a DFS.

Secondly, when `work()` finds out that the current node is not ready, `work()` should not return yet. Indeed, `work()` must continue to loop over the remaining arcs and continue to call itself recursively. That way, `work()` may find jobs to schedule in other branches.

The code to check whether a node is out-of-date simply compares the modification time of two nodes:

⟨*function* `outofdate` 96c⟩≡ (191b)
```
bool
outofdate(Node *node, Arc *arc, bool eval)
{
```

```
    ⟨outofdate() locals 144a⟩

    ⟨outofdate() if arc->prog 144b⟩
    else
     ⟨outofdate() if arc node is an archive member 153c⟩
     else
        /*
         * Treat equal times as out-of-date.
         * It's a race, and the safer option is to do
         * extra building rather than not enough.
         */
        return node->time <= arc->n->time;
}
```

Note that on recent machines, where compilation and linking can be extremely fast, it is not uncommon to generate object files and executables in the same second. Thus, it is important for the modification time of a file to be at a granularity finer than the second (ideally, nanosecond).

7.4 Scheduling recipes: `dorecipe()`

The job of `dorecipe()`[97e], called from `work()`[95a], is to construct a `Job`[41d] to run a recipe that will update a target `Node`[39d]. To do so, `dorecipe()` must first find the master rule of a node. Indeed, as I mentioned in Section 2.1.2, multiple rules can mention the same target, but only one of those rules can have a recipe, the master rule:

⟨`dorecipe()` *other locals* 97a⟩≡ (97e) 97b▷

```
  Rule *master_rule = nil;
  Arc *master_arc = nil;
```

Moreover, as I mentioned in Section 2.1.2, a rule may contain multiple targets. Thus, `dorecipe()` must also compute the set of all targets mentioned in the master rule:

⟨`dorecipe()` *other locals* 97b⟩+≡ (97e) ◁97a 97c▷

```
  // list<ref<Word> (last = last_alltargets, head elt to skip)
  Word alltargets;
```

It is also important for `dorecipe()` to compute not only the set of target names, but also to the set of target nodes. Indeed, once a job finished, `mk` must update the modification times of all those nodes in the graph of dependencies:

⟨`dorecipe()` *other locals* 97c⟩+≡ (97e) ◁97b 97d▷

```
  // list<ref<Node>> (next = Node.next)
  Node *nlist = node;
```

In practice, most rules have a single target so `alltargets` and `nlist` should contain only one element.

Finally, `dorecipe()` computes also the set of prerequisites:

⟨`dorecipe()` *other locals* 97d⟩+≡ (97e) ◁97c 98a▷

```
  // list<ref<Word>> (last = last_allprereqs, head elt to skip)
  Word allprereqs;
```

This set will be used when exporting the special variable `$prereqs` in `buildenv()`[117c].

Here is finally the code of `dorecipe()` leveraging the locals above:

⟨*function* `dorecipe` 97e⟩≡ (192a)

```
  % main -> mk -> work -> <> -> run(newjob())
  void
  dorecipe(Node *node, bool *did)
  {
      // iterators
```

```
    Arc *a;
    Node *n;
    Word *w;
    Symtab *s;
    ⟨dorecipe() other locals 97a⟩

    /*
     *   pick up the master rule
     */
    for(a = node->arcs; a; a = a->next)
        if(!empty_recipe(a->r)) {
            master_rule = a->r;
            master_arc = a;
        }

    ⟨dorecipe() if no recipe found 98b⟩
    // else

    /*
     *   build the node list
     */
    ⟨dorecipe() build lists of targets and node list 99b⟩

    /*
     *   gather the params for the job
     */
    allprereqs.next = newprereqs.next = nil;
    for(n = nlist; n; n = n->next){
        ⟨dorecipe() build lists of prerequisites 100⟩
        MADESET(n, BEINGMADE);
    }

    // run the job
    run(newjob(master_rule, nlist,
               master_arc->stem, master_arc->match,
               allprereqs.next, newprereqs.next,
               alltargets.next, oldtargets.next));
    *did = true; // finally
    return;
}
```

Uses `BEINGMADE` 40d, `MADESET` 94b, `empty_recipe` 78b, `newjob()` 42b, and `run()` 101a.

I described `newjob()`[42b] called above before. I will explain `run()`[101a], which adds the job in a job queue and possible schedules the job, in Chapter 8.

If `work()` found an outdated file but `dorecipe()` can not find any recipe to update the file, `mk` should report an error:

⟨dorecipe() *other locals* 98a⟩+≡ (97e) ◁97d 99a▷

```
char cwd[256];
```

⟨dorecipe() *if no recipe found* 98b⟩≡ (97e)

```
/*
 *   no recipe? go to buggery!
 */
if(master_rule == nil){
    ⟨dorecipe() when no recipe found, if virtual or norecipe node 140h⟩
    else {
        if(getwd(cwd, sizeof cwd))
            fprint(STDERR, "mk: no recipe to make '%s' in directory %s\n",
                   node->name, cwd);
```

```
        else
            fprint(STDERR, "mk: no recipe to make '%s'\n", node->name);
        Exit();
    }
}
```

7.4.1 Building the list of target nodes

To build the list of all targets, `dorecipe()`[97e] can rely on the `Rule.alltargets`[39b] field setup in `addrule()`[36b] (see Section 3.3.4):

⟨`dorecipe()` *other locals* 99a⟩+≡ (97e) ◁98a 147c▷

```
  Word *last_alltargets = &alltargets;
  char buf[BIGBLOCK];
```

Uses `BIGBLOCK` 182a.

⟨`dorecipe()` *build lists of targets and node list* 99b⟩≡ (97e)

```
  nlist->next = nil;
  alltargets.next = oldtargets.next = nil;
  ⟨dorecipe() if regexp rule 128f⟩
  else {
      for(w = master_rule->alltargets; w; w = w->next){
          if(master_rule->attr&META)
              subst(master_arc->stem, w->s, buf, sizeof(buf));
          else
              strecpy(buf, buf + sizeof buf - 1, w->s);

          //add_list(newword(buf), alltargets)
          last_alltargets->next = newword(buf);
          last_alltargets = last_alltargets->next;

          s = symlook(buf, S_NODE, nil);
          ⟨dorecipe() sanity check s 99c⟩
          n = s->u.ptr;

          ⟨dorecipe() update list of outdated targets 147d⟩

          // add_set(n, nlist)
          if(n == node)
              continue;
          n->next = nlist->next;
          nlist->next = n;
      }
  }
```

Uses `META` 36a, `S_NODE` 82a, `newword()` 32d, `subst()` 80c, and `symlook()` 30a.

Again, `mk` uses a pointer (`last_alltargets`) to point to the head of a list allocated in the stack (`alltargets`). This is the same C idiom I introduced in Section 6.1, which allows to add an element in a list without having to worry whether the list is empty.

`dorecipe()` stores the set of target names in the list `alltargets`. It also uses the node cache and the `S_NODE`[82a] namespace (see Section 6.4) to access the node of a target, and then chains together all the nodes with the `Node.next`[42a] field (see Section 3.5). `alltargets` will be used when exporting the special variable `$target` to the process executing the recipe. The node list will be used by `waitup()`[107c] to update the modification time and building status of those nodes in the graph.

⟨`dorecipe()` *sanity check s* 99c⟩≡ (99b)

```
  if(s == nil)
      continue; /* not a node we are interested in */
```

7.4.2 Building the list of prerequisites

To build the list of prerequisites, `dorecipe()`[97e] does not use the master rule but instead go through all the arcs. Indeed, many arcs do not contain a recipe but still contributes a dependency. This is also why the code below uses `addw()`[33b], which treats the list of words as a set and avoids adding duplicates:

```
⟨dorecipe() build lists of prerequisites 100⟩≡                                   (97e)
  for(a = n->arcs; a; a = a->next){
      if(a->n){
          addw(&allprereqs, a->n->name);

          if(outofdate(n, a, false)){
              ⟨dorecipe() when outofdate node, update list of newprereqs 148b⟩
              ⟨dorecipe() explain when found arc a making target n out of date 124c⟩
          }
      } else {
          ⟨dorecipe() explain when found target n with no prerequisite 124d⟩
      }
  }
```

Uses `addw()` 33b and `outofdate()` 96c.

Chapter 8

Scheduling Jobs

In this chapter, you will see the remaining important functions of `mk` that I introduced in Figure 2.7. I mentioned `run()` (called from `dorecipe()`[97e] before), which enqueues a job, and `waitup()` (called from `mk()`[92]), which waits for a job to finish. Both functions eventually call `sched()`, which takes a job from the job queue and schedules it for execution by a shell with `execsh()`. The following sections will explain the code of `run()`, `waitup()`, `sched()`, and `execsh()`.

8.1 Enqueuing jobs: `run()`

`run()` simply adds a job in the global job queue `jobs`[42c] I presented before and runs the scheduler:

⟨*function* `run` 101a⟩≡ (190)

```
/// main -> mk -> work -> dorecipe -> <> -> sched
void
run(Job *j)
{
    Job *jj;

    // enqueue(j, jobs)
    if(jobs){
        for(jj = jobs; jj->next; jj = jj->next)
            ;
        jj->next = j;
    } else
        jobs = j;
    j->next = nil;

    /* this code also in waitup after parse redirect */
    if(nrunning < nproclimit)
        sched();
}
```

Uses `jobs` 42c, `nproclimit-17` 101c, `nrunning-16` 101b, and `sched()` 103e.

`run()`[101a] relies on the two globals below:

⟨*global* `nrunning` 101b⟩≡ (190)

```
static int nrunning;
```

⟨*global* `nproclimit` 101c⟩≡ (190)

```
static int nproclimit;
```

`nrunning`[101b] counts the number of processes currently running a recipe, while `nproclimit`[101c] sets a limit on the number of processes to run at the same time. This last global usually corresponds to the number of processors on the machine. If there are no more free processors, then `run()` just enqueues the job. Hopefully at

some point mk will call waitup()[107c], which will wait for some process to finish, and which will call sched()[103e] to schedule one of the enqueued jobs.

You can modify nproclimit by modifying the $NPROC environment variable. Indeed, mk at startup time reads the environment (with readenv()[193c]) and looks for this variable to setup nproclimit with the code below:

⟨mk() *initializations* 102a⟩≡ (92) 148g ▷

```
nproc(); /* it can be updated dynamically */
```

Uses nproc() 102b.

⟨*function* nproc 102b⟩≡ (190)

```
void
nproc(void)
{
    Symtab *sym;
    Word *w;

    sym = symlook("NPROC", S_VAR, nil);
    if(sym) {
        w = sym->u.ptr;
        if (!empty_words(w))
            nproclimit = atoi(w->s);
    }
    if(nproclimit < 1)
        nproclimit = 1;
    ⟨nproc() if DEBUG(D_EXEC) 168f⟩

    ⟨nproc() grow nevents if necessary 103b⟩
}
```

Uses S_VAR 29a, empty_words 58b, nproclimit-17 101c, and symlook() 30a.

8.2 Scheduling jobs

Before showing the code of sched()[103e], I need to present an important data structure used by sched().

8.2.1 RunEvent and events

sched()[103e] takes a job from the job queue and creates a process to execute a shell that will interpret shell commands from a recipe. This same process will be waited for later by waitup()[107c]. However, waitup() needs to know which job corresponds to which process, so it can know which nodes in the graph of dependencies to update once a process finished. This is why sched(), in addition to executing new processes, needs also to remember the mapping between a process and a job. RunEvent below records a single mapping between a process id and a job:

⟨*struct* RunEvent 102c⟩≡ (190)

```
struct RunEvent {
    // option<Pid> (None = 0)
    int pid;

    // ref_own<Job>
    Job *job;
};
```

The list of mappings is stored in the following global:

⟨*global* events 102d⟩≡ (190)

```
// growing_array<Runevent> (allocated = nevents (== nproclimit))
static RunEvent *events;
```

⟨*global* nevents 103a⟩≡ (190)
```
static int nevents;
```

events[102d] is a growing array. The index of a mapping in this array is called a *slot*. The growing array is initialized in nproc()[102b]:

⟨nproc() *grow nevents if necessary* 103b⟩≡ (102b)
```
if(nproclimit > nevents){
    if(nevents)
        events = (RunEvent *)Realloc((char *)events,
                                     nproclimit*sizeof(RunEvent));
    else
        events = (RunEvent *)Malloc(nproclimit*sizeof(RunEvent));

    while(nevents < nproclimit)
        events[nevents++].pid = 0;
}
```
Uses Malloc() 171a, Realloc() 171b, events-14 102d, nevents-15 103a, and nproclimit-17 101c.

sched() relies on a few helper functions to operate on events. nextslot() below finds an available slot in events:

⟨*function* nextslot 103c⟩≡ (190)
```
/// sched -> <>
int
nextslot(void)
{
    int i;

    for(i = 0; i < nproclimit; i++)
        if(events[i].pid <= 0)
            return i;
    // else
    assert(/*out of slots!!*/ false);
    return 0; /* cyntax */
}
```
Uses events-14 102d and nproclimit-17 101c.

pidslot() below finds the slot of the mapping for a certain process id:

⟨*function* pidslot 103d⟩≡ (190)
```
/// waitup -> <>
int
pidslot(int pid)
{
    int i;

    for(i = 0; i < nevents; i++)
        if(events[i].pid == pid)
            return i;
    // else
    ⟨pidslot() if DEBUG(D_EXEC) 168e⟩
    return ERROR_NEG1;
}
```
Uses events-14 102d and nevents-15 103a.

8.2.2 sched()

Here is finally the code of sched():

⟨*function* sched 103e⟩≡ (190)

```
/// main -> mk -> work -> dorecipe -> run -> <>
static void
sched(void)
{
    Job *j;
    int slot;
    char *flags;
    ShellEnvVar *e;
    ⟨sched() other locals 120a⟩

    ⟨sched() return if no jobs 104⟩
    // else

    // j = pop(jobs)
    j = jobs;
    jobs = j->next;
    ⟨sched() if DEBUG(D_EXEC) 167e⟩

    slot = nextslot();
    events[slot].job = j;

    e = buildenv(j, slot);
    ⟨sched() print recipe command on stdout 120b⟩

    ⟨sched() if dry mode or touch mode, alternate to execsh 124h⟩
    else {
       ⟨sched() if DEBUG(D_EXEC) print recipe 168a⟩
        flags = "-e";
       ⟨sched() reset flags if NOMINUSE rule 142d⟩

        // launching the job!
        events[slot].pid = execsh(flags, j->r->recipe, nil, e);

        usage();
        nrunning++;
       ⟨sched() if DEBUG(D_EXEC) print pid 168b⟩
    }
}
```

Uses `buildenv()` 117c, `events-14` 102d, `jobs` 42c, `nextslot()` 103c, `nrunning-16` 101b, and `usage()` 126g.

The most important call in the code above is the call to `execsh()`[193c], which will create a new shell process that will interpret the commands from the recipe of a job (stored in `j->r>recipe`). This is why `sched()`[103e] above increments `nrunning`[101b] after the call to `execsh()`. `execsh()` will return the process id of this newly created process and `sched()` associates this id with the dequeued job in `events`[102d].

`execsh()` takes also as a first parameter a string containing a set of flags to pass to the shell as shell arguments. `mk` passes the `-e` flag so that every commands from the recipe returning an error will abort the whole execution of the recipe (see the SHELL book [Pad18]), a useful default.

The last argument to `execsh()` (`'e'`) contains the environment in which to execute the shell. This environment will contain the values for the special `mk` variables (e.g., `$target`, `$prereq`, `$stem`) and user variables. This environment is built by `buildenv()`[117c] called above, which I will present in Chapter 9. Thanks to this environment, the shell process executing the recipe will be able to get the values for the `mk` variables referenced in this recipe.

As you will see soon, `sched()` is also called from `waitup()`[107c], in which case the job queue may be empty. In that case, `sched()` simply returns.

⟨`sched()` *return if no jobs* 104⟩≡ (103e)

```
if(jobs == nil){
    usage();
```

```
    return;
}
```

Uses `jobs` 42c and `usage()` 126g.

8.3 Executing Jobs: `execsh()`

`execsh()` relies on the globals below to know which shell to execute. Under Plan 9, `mk` uses the shell `rc` (see the SHELL book [Pad18]):

⟨*global* `shell` 105a⟩≡ (199)

```
Shell* shell = &rc;
```

⟨*struct* `Shell` 105b⟩≡ (182d)

```
struct Shell {
    char* shell;
    char* shellname;
} Shell;
```

⟨*global* `rc` 105c⟩≡ (199)

```
Shell rc = {
  .shellname = "rc",
  .shell = "/bin/rc",
};
```

We do not want the shell to print a prompt before each command from the recipe, so `execsh()` adds the `-I` flag as a shell argument (in addition to `-e` added by `sched()`[103e]):

⟨*global* `shflags` 105d⟩≡ (186a)

```
char *shflags = "-I"; /* rc flag to force non-interactive mode */
```

Uses `shflags` 105d.

`execsh()` below essentially forks a shell interpreter, creates a pipe, and feeds this shell with commands from the recipe through this pipe (via another forked process):

⟨*function* `execsh` 105e⟩≡ (199)

```
/// main -> mk -> work -> dorecipe -> run -> sched -> <>
int
execsh(char *shargs, char *shinput, Bufblock *buf, ShellEnvVar *e)
{
    int pid1, pid2;
    fdt in[2]; // pipe descriptors
    int err;
    ⟨execsh() other locals 106a⟩

    ⟨execsh() if buf then create pipe to save output 132e⟩

    pid1 = rfork(RFPROC|RFFDG|RFENVG);
    ⟨execsh() sanity check pid1 rfork 106c⟩
    if(pid1 == 0){
        // child
        ⟨execsh() in child, if buf, close one side of pipe 132f⟩
        err = pipe(in);
        ⟨execsh() sanity check err pipe 107b⟩
        pid2 = fork();
        ⟨execsh() sanity check pid2 fork 107a⟩
        if(pid2 != 0){
            // parent of grandchild, the shell interpreter
            // input must come from the pipe
            dup(in[0], STDIN);
```

```
            ⟨execsh() in child, if buf, dup and close 132g⟩
            close(in[0]);
            close(in[1]);
            ⟨execsh() in child, export environment before exec 118a⟩
            if(shflags)
                execl(shell->shell, shell->shellname, shflags, shargs, nil);
            else
                execl(shell->shell, shell->shellname, shargs, nil);
            // should not be reached
            perror(shell->shell);
            _exits("exec");
        }
        // else, grandchild, feeding the shell with recipe, through a pipe
        ⟨execsh() in grandchild, if buf, close other side of pipe 132h⟩
        close(in[0]);
        // feed the shell
        ⟨execsh() in grandchild, write cmd in pipe 106b⟩
        close(in[1]); // will flush
        _exits(nil);
    }
    // else parent
    ⟨execsh() in parent, if buf, close other side of pipe and read output 133⟩
    return pid1;
}
```

Uses `shflags` 105d.

`execsh()` relies on the `rfork()`, `pipe()`, `dup()`, `close()`, and `_exits()` functions, which are syscalls to the kernel (see the KERNEL book [Pad14]), as well as on the `fork()`, `execl()`, and `perror()` functions from the C library (see the LIBCORE book [Pad16c]), which are thin wrappers around syscalls.

`execsh()` creates two processes: one for the shell, and one whose only job is to feed the shell through a pipe. By relying on this last process, `mk` can continue to schedule jobs without having to wait for the shell to finish reading the commands from the recipe.

Note that it is important for `execsh()` to execute the shell interpreter in the direct child, not the grandchild. Indeed, the parent process of the child, the `mk` process, knows only about the process id of the child. It is this process id that is stored later in `events`[102d] and looked for by `waitup()`. The parent process does not know about the process id of the grandchild. Moreover, this grandchild will terminate quickly, before the shell finishes executing the recipe.

Here is the code to feed the shell in the grandchild:

⟨execsh() *other locals* 106a⟩≡ (105e) 132d ▷

```
  char *endshinput;
```

⟨execsh() *in grandchild, write cmd in pipe* 106b⟩≡ (105e)

```
  endshinput = shinput + strlen(shinput);
  while(shinput < endshinput){
      n = write(in[1], shinput, endshinput - shinput);
      if(n < 0)
          break;
      shinput += n;
  }
```

⟨execsh() *sanity check pid1 rfork* 106c⟩≡ (105e)

```
  if(pid1 < 0){
      perror("mk rfork");
      Exit();
  }
```

⟨execsh() *sanity check pid2 fork* 107a⟩≡ (105e)
```
  if(pid2 < 0){
      perror("mk fork");
      Exit();
  }
```

⟨execsh() *sanity check err pipe* 107b⟩≡ (105e)
```
  if(err < 0){
      perror("pipe");
      Exit();
  }
```

8.4 Waiting for jobs to finish

The interface of `waitup()` is complex. I will focus first on the easy case where I do not care about the arguments passed to `waitup()`, and describe later the use of those arguments and the different edge cases of `waitup()`.

8.4.1 waitup()

Once `sched()`[103e] scheduled a job and modified `events`[102d], `mk()`[92] can call `waitup()` to wait for a job to finish. Indeed, `waitup()` can now rely on `events` to know which job is associated with the waited process. Then, `waitup()` can call `update()`[108b] to update the graph of dependencies and `sched()` to schedule more jobs:

⟨*function* waitup 107c⟩≡ (190)
```
  /// mk -> work; <> -> update; sched
  int
  waitup(int echildok, int *retstatus)
  {
      // child process
      int pid;
      // return string of child process
      char buf[ERRMAX];
      // index in events[]
      int slot;
      Job *j;
      Symtab *sym;
      Node *node;
      Word *w;
      bool fake = false;
      ⟨waitup() other locals 110a⟩

      ⟨waitup() if retstatus, check process list 146a⟩
  again:  /* rogue processes */

      pid = xwaitfor(buf);
      ⟨waitup() if no more children 109d⟩
      ⟨waitup() if DEBUG(D_EXEC) print pid 168c⟩
      ⟨waitup() if retstatus, check if matching pid 146b⟩

      slot = pidslot(pid);
      ⟨waitup() if slot not found, not a job pid, update process list 145d⟩

      j = events[slot].job;
      usage();
      nrunning--;
      // free events[slot]
      events[slot].pid = -1;
```

```
    ⟨waitup() if error in child process, possibly set fake or exit 110b⟩
    // else

    for(w = j->t; w; w = w->next){
        sym = symlook(w->s, S_NODE, nil);
        node = (Node*) sym->u.ptr;
        ⟨waitup() skip if node not found 110c⟩
        update(node, fake);
    }

    if(nrunning < nproclimit)
        sched();
    return JOB_ENDED;
}
```

Uses JOB_ENDED 109c, S_NODE 82a, events-14 102d, nproclimit-17 101c, nrunning-16 101b, pidslot() 103d, sched() 103e, symlook() 30a, update() 108b, and usage() 126g.

`waitup()`[107c] above relies on the helper function `xwaitfor()` below, which calls itself `wait()` from the C library (see the Libcore book [Pad16c]), which itself relies on the `await()` system call (see the Kernel book [Pad14]).

⟨*function* `xwaitfor`*(plan9.c)* 108a⟩≡ (199)

```
int
xwaitfor(char *msg)
{
    Waitmsg *w;
    int pid;

    // blocking call, wait for any children
    w = wait();
    // no more children
    if(w == nil)
        return ERROR_NEG1;
    strecpy(msg, msg+ERRMAX, w->msg);
    pid = w->pid;
    free(w);
    return pid;
}
```

`wait()` provides an easier interface than `await()` to wait for a child. Indeed, `wait()` returns a `Waitmsg` data structure (see the Libcore book [Pad16c]), which allows easy access to the pid and returned string of the child process.

8.4.2 update()

`update()`, called in `waitup()`[107c], marks the target nodes of a job as `MADE`[40d] and updates the modification time of those nodes.

⟨*function* `update` 108b⟩≡ (191b)

```
/// mk -> waitup -> <>
void
update(Node *node, bool fake)
{
    Arc *a;

    ⟨update() if fake 161g⟩
    else
       MADESET(node, MADE);
    ⟨update() debug 169b⟩
```

```
    ⟨update() if virtual node or inexistent file 141a⟩
    else {
        node->time = timeof(node->name, true);
        ⟨update() unpretend node 155f⟩
        ⟨update() set outofdate prereqs if arc prog 143j⟩
    }
}
```
Uses MADE 40d, MADESET 94b, and timeof() 82d.

I will present the use of the fake parameter later in Section 11.13.2.

8.4.3 waitup() edge cases

As I mentioned before, the interface of waitup()[107c] is complex:

⟨*signature waitup* 109a⟩≡
```
int waitup(int echildok, int *retstatus);
```

The first parameter of waitup(), of the type below, encodes whether it is ok or not for mk to have children to wait for.

⟨*type* WaitupParam 109b⟩≡ (182d)
```
enum WaitupParam {
  EMPTY_CHILDREN_IS_OK = 1,
  EMPTY_CHILDREN_IS_ERROR = -1,
  ⟨WaitupParam other cases 134b⟩
};
```

The return value of waitup(), of the type below, describes a few possible scenarios:

⟨*type* WaitupResult 109c⟩≡ (182d)
```
enum WaitupResult {
  JOB_ENDED = 0,
  EMPTY_CHILDREN = 1,
  NOT_A_JOB_PROCESS = -1
};
```

To understand NOT_A_JOB_PROCESS above, it is important to understand that under Plan 9, with the system call await() (called from wait()), you can not specify which child you are interested in to wait for. You can just wait for any children to finish. In fact, await() will also report children that already finished. This is partly the reason for the complexity of waitup(). Indeed, certain advanced features of mk (see Section 11.3, Section 11.2, and Section 11.5.8) will create processes that are not related to a job. Those processes will interfere with the call to wait(). The end of those processes must also be processed by waitup(). In fact, the second parameter of waitup() (retstatus) is used only by those advanced features and processes.

I can now describe the code to handle the edge cases of waitup(). As I showed in Section 7.1, mk()[92] calls waitup() in different contexts and the presence or not of a child can trigger an error or be perfectly ok depending on the context:

⟨waitup() *if no more children* 109d⟩≡ (107c)
```
if(pid == -1){
    if(echildok == EMPTY_CHILDREN_IS_OK)
        return EMPTY_CHILDREN;
    else {
        fprint(STDERR, "mk: (waitup %d) ", echildok);
        perror("mk wait");
        Exit();
    }
}
```
Uses EMPTY_CHILDREN 109c and EMPTY_CHILDREN_IS_OK 109b.

If a child returns an error string, for example by doing `exits("error")` instead of `exits(nil)`, then this error string will be captured by `wait()` and stored in the local buffer `buf` of `waitup()`, hence the condition below:

⟨`waitup()` *other locals* 110a⟩≡ (107c) 120c ▷
```
Bufblock *bp;
```

⟨`waitup()` *if error in child process, possibly set fake or exit* 110b⟩≡ (107c)
```
if(buf[0]){
    bp = newbuf();
    ⟨waitup() if error in child process, print recipe in bp 120d⟩
    fprint(STDERR, "mk: %s: exit status=%s", bp->start, buf);
    freebuf(bp);
    ⟨waitup() when error in child process, delete if DELETE node 141g⟩
    fprint(STDERR, "\n");

    ⟨waitup() when error in child process, if kflag 161f⟩
    else {
        jobs = nil;
        Exit();
    }
}
```
Uses `freebuf()` 172e, `jobs` 42c, and `newbuf()` 172d.

⟨`waitup()` *skip if node not found* 110c⟩≡ (107c)
```
if(sym == nil)
    continue; /* not interested in this node */
```

8.5 Process lifecycle: `Exit()`

When one of the children of `mk` returns an error, `mk` can not just exit. For example, if a C file in a project contains a syntax error, then `5c` run from `mk` will exit with an error. We do not want however `mk` to exit immediately. Indeed, there may still be other jobs running in parallel, for example, compiling other C files in the same project. If `mk` was exiting, those other jobs may get a signal that abruptly interrupts them. In those cases, the files generated by those jobs (e.g., object files for a compiler or assembler) may become corrupted. Thus, is is important before exiting to let those other jobs finish quietly. This is why `mk` calls `Exit()` below instead of calling directly `exits()`:

⟨*function* `Exit` 110d⟩≡ (199)
```
void
Exit(void)
{
    while(waitpid() >= 0)
        ;
    exits("error");
}
```

8.6 Notes (signals) management

Pressing `Control-C` on a build that is compiling ten files in parallel is the dangerous case. Under Plan 9, the *interrupt note* is delivered to all processes in the same *note group* (see the KERNEL book [Pad14]); `mk` and its forked shells all receive it together. The naive response—let `mk exit` right away—is wrong for the same reason we don't want `mk` to exit right away if one shell command returns an error like in the previous section. The fix in `killchildren()`[111e] below is to clear the job queue (so no new jobs start), keep `waitup()`[107c] alive long

enough for all running children to finish their current operation, and only then `Exit()`[193c]. This is the same reason ordinary error handling routes through `Exit()` instead of `exits()` directly.

⟨`main()` *initializations before building* 111a⟩≡ (45a) 114a ▷
```
catchnotes();
```

⟨*function* `catchnotes` 111b⟩≡ (199)
```
/// main -> <>
void
catchnotes()
{
    atnotify(notifyf, 1);
}
```

⟨*function* `notifyf` 111c⟩≡ (199)
```
int
notifyf(void *a, char *msg)
{
    ⟨notifyf() sanity check not too many notes 111d⟩
    if(strcmp(msg, "interrupt")!=0 && strcmp(msg, "hangup")!=0)
        return 0;
    killchildren(msg);
    return -1;
}
```
Uses `killchildren()` 111e.

⟨`notifyf()` *sanity check not too many notes* 111d⟩≡ (111c)
```
static int nnote;

USED(a);
if(++nnote > 100){ /* until andrew fixes his program */
    fprint(STDERR, "mk: too many notes\n");
    notify(0);
    abort();
}
```

⟨*function* `killchildren` 111e⟩≡ (190)
```
void
killchildren(char *msg)
{
    ⟨killchildren() locals 146c⟩

    jobs = nil;  /* make sure no more get scheduled */
    kflag = true; /* to make sure waitup doesn't exit */

    ⟨killchildren() expunge not-job processes 146d⟩

    while(waitup(EMPTY_CHILDREN_IS_OK, (int *)nil) == JOB_ENDED)
        ;
    Bprint(&bout, "mk: %s\n", msg);
    Exit();
}
```
Uses `EMPTY_CHILDREN_IS_OK` 109b, `JOB_ENDED` 109c, `bout` 45b, `jobs` 42c, `kflag` 161a, and `waitup()` 107c.

Chapter 9

The Shell Environment

In this chapter, you will see the functions to initialize, import, adjust, and export the environment to the shell processes launched from mk. Indeed, it is through the environment that mk communicates to the shell interpreter the values of mk's special variables (e.g., $target, $stem) or user variables (e.g., $CFLAGS) used in the recipes.

Why route variables through the environment rather than splicing them into the recipe text? Because the recipe text is opaque to mk: a recipe like 5c $CFLAGS -c $stem.c is handed to the shell verbatim and the shell expands $CFLAGS and $stem using its own variable lookup. By writing those values into /env/ before the exec, mk makes them available to the shell without ever touching the recipe string. This is the "minimal extension" philosophy at work: mk does not reimplement variable expansion, it just feeds the shell from the side. This is also why mk's recipes do not need the $$i double-dollar escape Make requires—there is no distinction between Make variables and shell variables to maintain, because there is only one expander.

9.1 ShellEnvVar and shellenv

The symbol table of mk (hash[29b]) contains already in the S_VAR[29a] namespace the values of user variables, as well as the values of the variables in the environment of mk itself. mk also stores the set of special variables in the symbol table in the S_INTERNAL[31c] namespace (but without any value). However, mk uses another data structure to communicate the value of all those variables to the shell. The structure below maps a variable name to a list of words.

⟨*struct* Envy 112a⟩≡ (182d)

```
struct ShellEnvVar
{
    // ref<string>, the key
    char   *name;

    // list<ref_own<Word>>, the value
    Word   *values;
};
```

All the variables and their values are stored in the following global:

⟨*global* shellenv 112b⟩≡ (187c)

```
// growing_array<ShellEnvVar> (endmarker = (nil,nil), used = nextv, allocated = envsize)
ShellEnvVar *shellenv;
```

The size of this array is stored in a static local variable in envinsert() below. However, you can iterate over shellenv without having to know the value of this static variable. Indeed, mk uses a special marker, (nil, nil), for the last ShellEnvVar entry in shellenv.

Here is the function to add an entry in shellenv:

⟨*global* nextv 112c⟩≡ (187c)

```
// idx for next free entry in shellenv array
static int nextv;
```

⟨*function* `envinsert` 113a⟩≡ (187c)

```
/// envupd | ecopy | execinit -> <>
static void
envinsert(char *name, Word *value)
{
    ⟨envinsert() locals 113b⟩

    ⟨envinsert() grow array if necessary 113c⟩
    shellenv[nextv].name = name;
    shellenv[nextv].values = value;
    nextv++;
}
```

Uses `nextv-7` 112c and `shellenv` 112b.

⟨`envinsert()` *locals* 113b⟩≡ (113a)

```
static int envsize = 0;
```

⟨`envinsert()` *grow array if necessary* 113c⟩≡ (113a)

```
if (nextv >= envsize) {
    envsize += ENVQUANTA;
    shellenv = (ShellEnvVar *) Realloc((char *) shellenv,
                                       envsize*sizeof(ShellEnvVar));
}
```

Uses `ENVQUANTA-6` 113d, `Realloc()` 171b, `nextv-7` 112c, and `shellenv` 112b.

⟨*constant* `ENVQUANTA` 113d⟩≡ (187c)

```
#define ENVQUANTA 10
```

The execution of each recipe requires a specific shell environment. Indeed, the values for `$target`, `$stem`, and other special variables are different for each rule. However, the values of the user variables are always the same. To avoid allocating a new shell environment for each execution, `mk` reuses `shellenv` for all executions, but relies on the function below to adjust the values of a few variables:

⟨*function* `envupd` 113e⟩≡ (187c)

```
static void
envupd(char *name, Word *value)
{
    ShellEnvVar *e;

    for(e = shellenv; e->name; e++)
        if(strcmp(name, e->name) == 0){
            freewords(e->values);
            e->values = value;
            return;
        }
    ⟨envupd() if variable not found 113f⟩
}
```

Uses `freewords()` 33a and `shellenv` 112b.

⟨`envupd()` *if variable not found* 113f⟩≡ (113e)

```
// else
e->name = name;
e->values = value;
envinsert(nil,nil);
```

Uses `envinsert()` 113a.

9.2 Initializing the shell environment: initenv()

To initialize `shellenv`[112b], `main()`[45a] calls `initenv()`[114b] before calling `mk()`[92]:

⟨`main()` *initializations before building* 114a⟩+≡ (45a) ◁111a 125c▷
```
  initenv();
```
Uses `initenv()` 114b.

⟨*function* `initenv` 114b⟩≡ (187c)
```
  /// main -> parse; mk; <>
  void
  initenv(void)
  {
      char **p;

      nextv = 0; // reset shellenv

      // internal mk variables
      for(p = specialvars; *p; p++)
          envinsert(*p, stow(""));

      // user variables in mkfiles, or mk process environment
      symtraverse(S_VAR, ecopy);

      // end marker
      envinsert(nil, nil);
  }
```
Uses `S_VAR` 29a, `ecopy()` 114c, `envinsert()` 113a, `nextv-7` 112c, `specialvars-8` 31d, `stow()` 60b, and `symtraverse()` 31a.

I described `symtraverse()`[31a] before. It allows to iterate over a namespace while applying a function, here `ecopy()`:

⟨*function* `ecopy` 114c⟩≡ (187c)
```
  static void
  ecopy(Symtab *s)
  {
      char **p;

      ⟨ecopy() return and do not copy if S_NOEXPORT symbol 147b⟩
      ⟨ecopy() return and do not copy if conflict with mk internal variable 114d⟩
       // else
       envinsert(s->name, s->u.ptr);
  }
```
Uses `envinsert()` 113a.

Note that `initenv()` calls `envinsert()`[113a] to create first the entries for the special `mk` variables. It is those variables that `mk` needs to adjust for each shell execution. By adding those entries first in `shellenv`, `envupd()`[113e] will be slightly faster because `envupd()` tries to find the variable to update by starting from the start of the `shellenv` array.

⟨`ecopy()` *return and do not copy if conflict with mk internal variable* 114d⟩≡ (114c)
```
  for(p = specialvars; *p; p++)
      if(strcmp(*p, s->name) == 0)
          return;
```
Uses `specialvars-8` 31d.

9.3 Importing the environment: `readenv()`

`initenv()`[114b] iterates over the `S_VAR`[29a] namespace with `symtraverse()`[31a] to add in `shellenv`[112b] the user variables, for instance, a variable `$CFLAGS` defined in the `mkfile`. In fact, the `S_VAR` namespace contains also the variables from the environment of `mk` itself. Indeed `main()`[45a] calls `inithash()`[32a] to initialize the symbol table, and `inithash()` calls `readenv()` below to populate the symbol table with variables from the environment (e.g., `$HOME`, `$objtype`, `$CC`).

Under Plan 9, the environment variables of a process are accessible through the filesystem under `/env/` (see the KERNEL book [Pad14]). `readenv()` below simply iterates over all the files under `/env/`.

⟨*function* `readenv` *(plan9.c)* 115a⟩≡ (199)

```
/// main -> inithash -> <>
void
readenv(void)
{
    fdt envdir;
    fdt envfile;
    Dir *e;
    int i, n, len, len2;
    char *p;
    char name[1024];
    Word *w;

    rfork(RFENVG); /*  use copy of the current environment variables */

    envdir = open("/env", OREAD);
    ⟨readenv() sanity check envdir 116b⟩
    while((n = dirread(envdir, &e)) > 0){
        for(i = 0; i < n; i++){
            len = e[i].length;
            ⟨readenv() skip some names 115b⟩

            snprint(name, sizeof name, "/env/%s", e[i].name);
            envfile = open(name, OREAD);
            ⟨readenv() sanity check envfile 116c⟩
            p = Malloc(len+1);
            len2 = read(envfile, p, len);
            ⟨readenv() sanity check len2 116d⟩
            close(envfile);
            ⟨readenv() add null terminator character at end of p 117b⟩
            w = encodenulls(p, len);
            free(p);
            p = strdup(e[i].name);

            // populating symbol table
            setvar(p, (void *) w);
        }
        free(e);
    }
    close(envdir);
}
```

Uses `Malloc()` 171a and `setvar()` 31b.

`readenv()`[193c] skips entries under `/env/` that would lead to empty variables or variables that would conflict with one of `mk`'s special variables:

⟨`readenv()` *skip some names* 115b⟩≡ (115a)

```
/* don't import funny names, NULL values,
 * or internal mk variables
```

```
  */
 if(len <= 0
    || *shname(e[i].name) != '\0'
    || symlook(e[i].name, S_INTERNAL, nil))
     continue;
```

Uses S_INTERNAL 31c, shname() 116a, and symlook() 30a.

⟨*function* shname 116a⟩≡ (186c)

```
 char *
 shname(char *a)
 {
     Rune r;
     int n;

     while (*a) {
         n = chartorune(&r, a);
         if (!WORDCHR(r))
             break;
         a += n;
     }
     return a;
 }
```

Uses WORDCHR 64e.

⟨readenv() *sanity check envdir* 116b⟩≡ (115a)

```
 if(envdir < 0)
     return;
```

⟨readenv() *sanity check envfile* 116c⟩≡ (115a)

```
 if(envfile < 0)
     continue;
```

⟨readenv() *sanity check len2* 116d⟩≡ (115a)

```
 if(len2 != len){
     perror(name);
     close(envfile);
     continue;
 }
```

Under Plan 9, environment variables can contain a list of words, just like mk's variables (and rc's variables). The format of this list uses the null character not to mark the end of a string but as a word separator. An alternative would be to use the space character to separate words. However, because under Plan 9 certain filenames can contain spaces (but not null characters), and because certain environment variables reference a list of filenames (e.g., $PATH), it is more convenient to use the null character as a separator. This avoids the need to escape space characters (and later to parse escaped characters).

The UNIX world eventually discovered the same trick, but only as a bolted-on option in individual tools rather than as a native data format. The canonical example is `find -print0` piped into `xargs -0`, which exchange filenames separated by null bytes precisely so that names containing spaces or newlines survive intact; the GNU coreutils sprouted a parallel family of `-z`/`-0` flags for the same reason (`sort -z`, `grep -z`, `uniq -z`, `perl -0`, `read -d ''` in `bash`). The contrast is not that Plan 9 tools never face this problem — a Plan 9 tool that serialized a filename list into a pipe would need exactly the same null-separator convention at both ends. The difference is that the environment is a structured store rather than a byte stream, so Plan 9 can make null-separated lists the native, built-in format at the `/env` level; the escaping question simply never arises for environment variables, even if it still would for an ad-hoc pipe.

The function below, called from `readenv()`, recognizes the null character as a word separator and splits the string `s` in a list of words. Note that you must also pass the length of the string as an argument to `encodenulls()` because you can not rely anymore on the null character to mark the end of the string.

⟨*function* `encodenulls` 117a⟩≡ (199)
```
/// readenv -> <>
/* break string of values into words at 01's or nulls*/
static Word *
encodenulls(char *s, int n)
{
    Word *head, *lastw;
    char *cp;

    head = lastw = nil;
    while (n-- > 0) {
        for (cp = s; *cp && *cp != '\0'; cp++)
                n--;
        *cp = '\0';

        // add_list(newword(s), head)
        if (lastw) {
            lastw->next = newword(s);
            lastw = lastw->next;
        } else
            head = lastw = newword(s);

        s = cp+1;
    }
    if (!head)
        head = newword("");
    return head;
}
```

⟨`readenv()` *add null terminator character at end of* `p` 117b⟩≡ (115a)
```
if (p[len-1] == '\0')
    len--;
else
    p[len] = '\0';
```

9.4 Adjusting the shell environment: `buildenv()`

Once `shellenv`[112b] has been initialized, `sched()`[103e] can call `buildenv()` to adjust the environment with the specific values of `mk`'s special variable for a specific job:

⟨*function* `buildenv` 117c⟩≡ (187c)
```
/// main -> mk -> work -> dorecipe -> run -> sched -> <>
ShellEnvVar*
buildenv(Job *j, int slot)
{
    ⟨buildenv() locals 130b⟩

    // main variables
    envupd("target", wdup(j->t));
    ⟨buildenv() if regexp rule 129a⟩
    else
        envupd("stem", newword(j->stem));
    envupd("prereq", wdup(j->p));
```

```
    // advanced variables
    ⟨buildenv() envupd some variables 130c⟩

    return shellenv;
}
```
Uses `envupd()` 113e, `newword()` 32d, `shellenv` 112b, and `wdup()` 33c.

Note that as I mentioned in Section 9.1, `buildenv()`[117c] above reuses `shellenv`; `buildenv()` does not allocate each time a new environment. Note also that some rules do not have prerequisites, or a stem, in which case `buildenv()` will store the empty list for those entries in `shellenv`.

9.5 Exporting the shell environment: `exportenv()`

Once `sched()`[103e] updated `shellenv`[112b] with `buildenv()`[117c] and called `execsh()`[193c] with this environment, `execsh()` forks a shell interpreter and calls `exportenv()`[193c] in the child process to modify its own environment:

⟨`execsh()` *in child, export environment before exec* 118a⟩≡ (105e)
```
if (e)
    exportenv(e);
```

Under Plan 9, a process can modify its environment by writing in files under `/env/`. `exportenv()` below iterates over the entries in `shellenv` (bound to the `e` parameter), and writes into files under `/env/`:

⟨*function* `exportenv` *(plan9.c)* 118b⟩≡ (199)
```
/// execsh -> <>
/* as well as 01's, change blanks to nulls, so that rc will
 * treat the words as separate arguments
 */
void
exportenv(ShellEnvVar *e)
{
    int n;
    fdt f;
    bool first;
    Word *w;
    char name[256];
    ⟨exportenv() other locals 119b⟩

    for(;e->name; e++){
        ⟨exportenv() skip entry if not a user variable and no value 119c⟩
        // else
        snprint(name, sizeof name, "/env/%s", e->name);
        ⟨exportenv() if existing symbol but no value, remove from env 119d⟩
        // else
        f = create(name, OWRITE, 0666L);
        ⟨exportenv() sanity check f 119e⟩
        first = true;
        for (w = e->values; w; w = w->next) {
            n = strlen(w->s);
            if (n) {
                ⟨exportenv() write null separator 119a⟩
                if (write(f, w->s, n) != n)
                    perror(name);
            }
        }
        close(f);
    }
}
```

As I mentioned in Section 9.3, the files under /env/ use the null character as a word separator:

⟨exportenv() *write null separator* 119a⟩≡ (118b)
```
if(first)
    first = false;
else{
    if (write (f, "\000", 1) != 1)
        perror(name);
}
```

There are a few situations where it is useless to write in /env/. Indeed, certain entries in shellenv might not contain any value because the variable is undefined for a job, for instance, $stem in a non-meta rule has no value (it is just the empty word). exportenv() can skip those entries:

⟨exportenv() *other locals* 119b⟩≡ (118b)
```
Symtab *sym;
bool hasvalue;
```

⟨exportenv() *skip entry if not a user variable and no value* 119c⟩≡ (118b)
```
hasvalue = !empty_words(e->values);
sym = symlook(e->name, S_VAR, nil);
if(sym == nil && !hasvalue) /* non-existant null symbol */
    continue;
```
Uses S_VAR 29a, empty_words 58b, and symlook() 30a.

However, if you defined a variable but assigned it the empty list, exportenv() will delete this environment variable:

⟨exportenv() *if existing symbol but no value, remove from env* 119d⟩≡ (118b)
```
if (sym != nil && !hasvalue) { /* Remove from environment */
    /* we could remove it from the symbol table
     * too, but we're in the child copy, and it
     * would still remain in the parent's table.
     */
    remove(name);
    freewords(e->values);
    e->values = nil;  /* memory leak */
    continue;
}
```
Uses freewords() 33a.

⟨exportenv() *sanity check f* 119e⟩≡ (118b)
```
if(f < 0) {
    fprint(STDERR, "can't create %s, f=%d\n", name, f);
    perror(name);
    continue;
}
```

Chapter 10

Debugging and Profiling Support

A build tool spends most of its life doing the right thing silently, but when a recipe fails or a user wonders why a target is being rebuilt, `mk` needs to explain itself. This chapter gathers the facilities that let it do so: echoing the commands it runs (`shprint()`[121b]), explaining its rebuild decisions (`mk -e`), previewing work without doing it (`mk -n` and `mk -w`), and reporting how well it kept the processors busy (`mk -u`). Accurate error reporting is cheap here because the groundwork was laid during parsing: every `Rule`[34] remembers the `file` and `line` where it was defined (Chapter 5). When a build fails or the dependency graph is dumped, `mk` can point at the exact `mkfile` location that introduced an arc rather than leaving the user to guess which rule was at fault.

10.1 Printing jobs: `shprint()`

When a recipe runs or fails, `mk` wants to show the user the command that actually executed. The subtlety is that `mk` does *not* expand the variable references in a recipe (e.g., `$CFLAGS`) before running it: the recipe text is handed to the shell verbatim, and the shell interprets the variables, drawing their values from the environment `mk` exported (see `exportenv()`[193c]). The `shprint()`[121b] function re-expands the recipe against `mk`'s own variables purely for display, so the echoed line shows concrete values instead of the `$CFLAGS` template; references `mk` does not know about are copied through untouched for the shell to resolve.

⟨`sched()` *other locals* 120a⟩≡ (103e) 124g▷

```
  Bufblock *buf;
```

⟨`sched()` *print recipe command on stdout* 120b⟩≡ (103e)

```
  buf = newbuf();
  shprint(j->r->recipe, e, buf);
  if(!tflag && (nflag || !(j->r->attr&QUIET)))
      Bwrite(&bout, buf->start, (long)strlen(buf->start));
  freebuf(buf);
```

Uses `QUIET` 141i, `bout` 45b, `freebuf()` 172e, `newbuf()` 172d, `nflag` 124e, `shprint()` 121b, and `tflag` 156a.

⟨`waitup()` *other locals* 120c⟩+≡ (107c) ◁110a 141f▷

```
  ShellEnvVar *e;
```

⟨`waitup()` *if error in child process, print recipe in* `bp` 120d⟩≡ (110b)

```
  e = buildenv(j, slot);
  shprint(j->r->recipe, e, bp);
  front(bp->start);
```

Uses `buildenv()` 117c, `front()` 121a, and `shprint()` 121b.

`front()`[121a] abbreviates a long command line in place to its first few words (then `...` and the final word) so an error report shows the head of a recipe rather than dumping a huge fully-expanded command.

⟨*function* `front` 121a⟩≡ (188b)
```
void
front(char *s)
{
    char *t, *q;
    int i, j;
    char *flds[512];

    q = strdup(s);
    i = getfields(q, flds, nelem(flds), 0, " \t\n");
    if(i > 5){
        flds[4] = flds[i-1];
        flds[3] = "...";
        i = 5;
    }
    t = s;
    for(j = 0; j < i; j++){
        for(s = flds[j]; *s; *t++ = *s++);
        *t++ = ' ';
    }
    *t = 0;
    free(q);
}
```

We can finally show the code of `shprint()`:

⟨*function* `shprint` 121b⟩≡ (188b)
```
/// sched | update -> <>
void
shprint(char *s, ShellEnvVar *env, Bufblock *buf)
{
    Rune r;
    int n;

    while(*s) {
        n = chartorune(&r, s);
        if (r == '$')
            s = vexpand(s, env, buf);
        else {
            rinsert(buf, r);
            s += n;
            s = copyq(s, r, buf); /*handle quoted strings*/
        }
    }
    insert(buf, 0);
}
```
Uses `copyq()` 123b, `insert()` 173a, `rinsert()` 173b, and `vexpand()` 121c.

10.1.1 Expanding and printing variables

`vexpand()`[121c] handles a single `$` reference for `shprint()`[121b]: it parses the variable name (either `$name` or `$...`), substitutes its value via `mygetenv()`[122a], and returns a pointer just past the reference so the caller can resume scanning.

⟨*function* `vexpand` 121c⟩≡ (188b)
```
static char*
vexpand(char *w, ShellEnvVar *env, Bufblock *buf)
```

```
{
    char *s;
    char *p, *q;
    char carry;

    assert(/*vexpand no $*/ *w == '$');
    p = w+1; /* skip dollar sign */
    if(*p == '{') {
        p++;
        q = utfrune(p, '}');
        if (!q)
            q = strchr(p, 0);
    } else
        q = shname(p);

    carry = *q;
    *q = '\0';
    s = mygetenv(p, env);
    *q = carry;

    if (carry == '}')
        q++;

    if (s) {
        bufcpy(buf, s, strlen(s));
        free(s);
    } else   /* copy name intact*/
        bufcpy(buf, w, q-w);

    return q;
}
```

Uses `bufcpy()` 173c, `mygetenv()` 122a, and `shname()` 116a.

`mygetenv()` resolves a variable's value, but only for variables `mk` itself set (`S_WESET`[122b]) or internal ones (`S_INTERNAL`); it returns nil for anything else, so unknown references are left intact for the shell to interpret.

⟨*function* `mygetenv` 122a⟩≡ (188b)

```
static char*
mygetenv(char *name, ShellEnvVar *env)
{
    if (!env)
        return nil;
    if (!symlook(name, S_WESET, nil) &&
        !symlook(name, S_INTERNAL, nil))
        return nil;
    // else

    /* only resolve internal variables and variables we've set */
    for(; env->name; env++){
        if (strcmp(env->name, name) == 0)
            return wtos(env->values, ' ');
    }
    return nil;
}
```

Uses `S_INTERNAL` 31c, `S_WESET` 122b, `symlook()` 30a, and `wtos()` 33d.

⟨Sxxx *cases* 122b⟩+≡ (29a) ◁82a 143i▷

```
S_WESET, /* variable; we set in the mkfile */
```

⟨parse() *when parsing variable definitions, extra setting* 123a⟩≡ (73c)
```
symlook(head->s, S_WESET, (void *)"");
```
Uses S_WESET 122b and symlook() 30a.

10.1.2 Printing quoted strings

⟨*function* copyq 123b⟩≡ (186a)
```
/*
 * check for quoted strings.  backquotes are handled here; single quotes above.
 * s points to char after opening quote, q.
 */
char *
copyq(char *s, Rune q, Bufblock *buf)
{
    if(q == '\'')    /* copy quoted string */
        return copysingle(s, buf);

    if(q != '`')    /* not quoted */
        return s;
    // else

    while(*s){    /* copy backquoted string */
        s += chartorune(&q, s);
        rinsert(buf, q);
        if(q == '}')
            break;
        if(q == '\'')
            s = copysingle(s, buf); /* copy quoted string */
    }
    return s;
}
```
Uses copysingle() 123c and rinsert() 173b.

⟨*function* copysingle 123c⟩≡ (186a)
```
/*
 * copy a single-quoted string; s points to char after opening quote
 */
static char *
copysingle(char *s, Bufblock *buf)
{
    Rune r;

    while(*s){
        s += chartorune(&r, s);
        rinsert(buf, r);
        if(r == '\'')
            break;
    }
    return s;
}
```
Uses rinsert() 173b.

10.2 Explain mode: mk -e

When a build does more (or less) than expected, the question is always *why* did mk decide a target was out of date. The -e flag answers it: for each rebuild, mk prints the offending prerequisite alongside both timestamps

(e.g. `target(time) < prereq(time)`), or notes that a target had no prerequisites at all. It is a window into the outdated-file comparison of Chapter 7, costing just two `fprint` calls guarded by a flag.

⟨*global* `explain` 124a⟩≡ (184)
```
bool explain = false;
```
Uses `explain` 124a.

⟨`main()` *-xxx switch cases* 124b⟩+≡ (46b) ◁49f 124f▷
```
case 'e':
    explain = true;
    break;
```
Uses `explain` 124a.

⟨`dorecipe()` *explain when found arc* `a` *making target* `n` *out of date* 124c⟩≡ (100)
```
if(explain)
    fprint(STDOUT, "%s(%ld) < %s(%ld)\n",
        n->name, n->time, a->n->name, a->n->time);
```
Uses `explain` 124a.

⟨`dorecipe()` *explain when found target* `n` *with no prerequisite* 124d⟩≡ (100)
```
if(explain)
    fprint(STDOUT, "%s has no prerequisites\n", n->name);
```
Uses `explain` 124a.

10.3 Dry mode: `mk -n`

Dry mode prints the recipes `mk` *would* run without actually running them, so you can preview the consequences of a build before committing to it. The implementation is a small detour in `sched()`[103e]: instead of handing the job to `execsh()`[193c], `mk` still echoes the recipe but then simply stamps each target as `MADE` with the current time, letting the rest of the graph walk proceed as if the work had succeeded.

⟨*global* `nflag` 124e⟩≡ (184)
```
bool nflag = false;
```
Uses `nflag` 124e.

⟨`main()` *-xxx switch cases* 124f⟩+≡ (46b) ◁124b 125b▷
```
case 'n':
    nflag = true;
    break;
```
Uses `nflag` 124e.

⟨`sched()` *other locals* 124g⟩+≡ (103e) ◁120a
```
Node *n;
```

⟨`sched()` *if dry mode or touch mode, alternate to execsh* 124h⟩≡ (103e)
```
if(nflag || tflag){
    for(n = j->n; n; n = n->next){
        ⟨sched() if touch mode 156d⟩
        n->time = time((long *)nil);
        MADESET(n, MADE);
    }
}
```
Uses `MADE` 40d, `MADESET` 94b, `nflag` 124e, and `tflag` 156a.

10.4 What-if mode: `mk -w <file>`

What-if mode answers a hypothetical: *if* I were to touch this file, what would need rebuilding? Rather than actually modifying the file, `mk` pokes its time cache, forcing the named file's mtime to "now" so the outdated-file comparison treats everything downstream of it as stale. Paired with dry mode (typically `mk -n -w prog.h`) it lets you see the blast radius of a change before making it.

⟨`main()` *locals* 125a⟩+≡ (45a) ◁49e 149f▷
```
  Bufblock *whatif = nil;
```

⟨`main()` *-xxx switch cases* 125b⟩+≡ (46b) ◁124f 126b▷
```
  case 'w':
      if(whatif == nil)
          whatif = newbuf();
      else
          insert(whatif, ' ');
      if(argv[0][2])
          bufcpy(whatif, &argv[0][2], strlen(&argv[0][2]));
      else {
          if(*++argv == '\0')
              badusage();
          bufcpy(whatif, &argv[0][0], strlen(&argv[0][0]));
      }
      break;
```
Uses `badusage()` 46c, `bufcpy()` 173c, `insert()` 173a, and `newbuf()` 172d.

⟨`main()` *initializations before building* 125c⟩+≡ (45a) ◁114a 149g▷
```
  if(whatif){
      insert(whatif, '\0');
      timeinit(whatif->start);
      freebuf(whatif);
  }
```
Uses `freebuf()` 172e, `insert()` 173a, and `timeinit()` 125d.

`timeinit()`[125d] implements what-if mode: it splits the `-w` argument into individual filenames and stamps each one's cached mtime (`S_TIME`[157a], see Section 11.10.3) to the current time, so the outdated-file check later treats everything downstream of those files as stale.

⟨*function* `timeinit` 125d⟩≡ (189c)
```
  void
  timeinit(char *s)
  {
      ulong t;
      char *cp;
      Rune r;
      int c, n;

      t = time(nil);
      while (*s) {
          cp = s;
          do{
              n = chartorune(&r, s);
              if (r == ' ' || r == ',' || r == '\n')
                  break;
              s += n;
          } while(*s);
          c = *s;
          *s = '\0';

          symlook(strdup(cp), S_TIME, (void *)t)->u.value = t;
```

```
        if (c)
            *s++ = c;
        while(*s){
            n = chartorune(&r, s);
            if(r != ' ' && r != ',' && r != '\n')
                break;
            s += n;
        }
    }
}
```
Uses S_TIME 157a and symlook() 30a.

10.5 Processor utilization: `mk -u`

The whole point of `mk`'s parallel scheduler is to keep the machine busy, so `-u` measures how well it succeeds. The idea is a histogram indexed by *degree of parallelism*: `usage()`[126g] is called whenever `nrunning` changes and charges the elapsed time to the `tslot` bucket for the number of jobs that were running. At the end, `prusage()`[127b] prints how many seconds were spent with zero jobs running, one job, two jobs, and so on — making it easy to see whether the build was actually parallel or mostly serialized on a critical path.

⟨*global* `uflag` 126a⟩≡ (193b)
```
bool uflag = false;
```
Uses uflag 126a.

⟨`main()` *-xxx switch cases* 126b⟩+≡ (46b) ◁125b 154b▷
```
case 'u':
    uflag = true;
    break;
```
Uses uflag 126a.

⟨*global* `tslot` 126c⟩≡ (190)
```
// map<nrunning, int>
static ulong tslot[1000];
```

⟨*global* `tick` 126d⟩≡ (190)
```
static ulong tick;
```

⟨`main()` *setup profiling* 126e⟩≡ (46a) 170b▷
```
usage();
```
Uses usage() 126g.

⟨`main()` *profile initializations* 126f⟩≡ (46a)
```
usage();
```
Uses usage() 126g.

⟨*function* `usage` 126g⟩≡ (190)
```
void
usage(void)
{
    ulong t;

    t = time(nil);
    if(tick)
        tslot[nrunning] += t - tick;
    tick = t;
}
```
Uses nrunning-16 101b, tick-21 126d, and tslot-20 126c.

⟨`main()` *print profiling stats if uflag* 127a⟩≡ (45a)

```
if(uflag)
    prusage();
```

Uses `prusage()` 127b and `uflag` 126a.

⟨*function* `prusage` 127b⟩≡ (190)

```
void
prusage(void)
{
    int i;

    usage();
    for(i = 0; i <= nevents; i++)
        fprint(STDOUT, "%d: %lud\n", i, tslot[i]);
}
```

Uses `nevents-15` 103a, `tslot-20` 126c, and `usage()` 126g.

Chapter 11

Advanced Features

The earlier chapters covered the core pipeline of `mk`: parsing rules, building the dependency graph, finding outdated targets, and scheduling jobs. This chapter collects the more advanced `mkfile` features layered on top of that core — the ones a simple `mkfile` usually does not need but that make `mk` expressive: regular-expression meta-rules (a more powerful alternative to `%`-patterns), shell-command expansion via backquotes, dynamically generated mkfiles (`<|`), substitution variables, and the various rule and variable attributes.

11.1 Regular-expression rules: `:R:`

A `%`-style meta rule has a single stem: the one piece of the target matched by `%`. The `:R:` attribute generalizes this by letting the target be a full regular expression, compiled with `regcomp()` from `libregexp` (see the LIBCORE book [Pad16c]) and matched with `regexec()`. The parenthesized subexpressions then become `$stem1` through `$stem9` (with `$stem0` the whole match), usable in both the prerequisites and the recipe. This buys two things a `%`-rule cannot express: several independent stems — a target can capture, say, a basename and an object-type suffix as separate `$stem1` and `$stem2` — and the full power of alternation and character classes.

⟨`Rule_attr` *cases* 128a⟩≡ (36a) 140c ▷
```
  REGEXP = 0x0020,
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 128b⟩≡ (70e) 140d ▷
```
  case 'R':
      *attr |= REGEXP;
      break;
```

⟨`Rule` *other fields* 128c⟩+≡ (34) ◁ 85b 143b ▷
```
  Reprog  *pat;  /* reg exp goo */
```

⟨*global* `patrule` 128d⟩≡ (184)
```
  Rule *patrule;
```

⟨`addrule()` *if REGEXP attribute* 128e⟩≡ (37a)
```
  if(attr&REGEXP){
      patrule = r;
      r->pat = regcomp(target);
  }
```
Uses `REGEXP` 128a and `patrule` 128d.

⟨`dorecipe()` *if regexp rule* 128f⟩≡ (99b)
```
  if(master_rule->attr&REGEXP){
      last_oldtargets->next = newword(node->name);
      last_alltargets->next = newword(node->name);
  }
```
Uses `REGEXP` 128a and `newword()` 32d.

⟨buildenv() *if regexp rule* 129a⟩≡ (117c)
```
  if(j->r->attr&REGEXP)
      envupd("stem", newword(""));
```
Uses REGEXP 128a, envupd() 113e, and newword() 32d.

⟨*function* regerror 129b⟩≡ (189a)
```
  //@Scheck: not dead, called via regcomp() when have regexp syntax error
  void regerror(char *s)
  {
      if(patrule)
          fprint(STDERR, "mk: %s:%d: regular expression error; %s\n",
              patrule->file, patrule->line, s);
      else
          fprint(STDERR, "mk: %s:%d: regular expression error; %s\n",
              infile, mkinline, s);
      Exit();
  }
```
Uses infile 51a, mkinline 51c, and patrule 128d.

⟨*constant* NREGEXP 129c⟩≡ (182d)
```
  #define  NREGEXP  10
```

⟨Arc *other fields* 129d⟩+≡ (40f) ◁88e 143e▷
```
  char  *match[NREGEXP];
```
Uses NREGEXP 129c.

⟨newarc() *set other fields* 129e⟩+≡ (41c) ◁89a 143f▷
```
  rcopy(a->match, match, NREGEXP);
```
Uses NREGEXP 129c.

⟨*function* rcopy 129f⟩≡ (199)
```
  void
  rcopy(char **to, Resub *match, int n)
  {
      int c;
      char *p;

      *to = match->s.sp;  /* stem0 matches complete target */
      for(to++, match++; --n > 0; to++, match++){
          if(match->s.sp && match->e.ep){
              p = match->e.ep;
              c = *p;
              *p = 0;
              *to = strdup(match->s.sp);
              *p = c;
          }
          else
              *to = 0;
      }
  }
```

⟨Job *other fields* 129g⟩≡ (41d) 147f▷
```
  char  **match;
```

⟨specialvars *other array elements* 130a⟩≡ (31d) 147e ▷
```
"stem0",  /* must be in order from here */
"stem1",
"stem2",
"stem3",
"stem4",
"stem5",
"stem6",
"stem7",
"stem8",
"stem9",
```

⟨buildenv() *locals* 130b⟩≡ (117c) 149b ▷
```
char **p;
int i;
```

⟨buildenv() *envupd some variables* 130c⟩≡ (117c) 147h ▷
```
/* update stem0 -> stem9 */
for(p = specialvars; *p; p++)
    if(strcmp(*p, "stem0") == 0)
        break;
for(i = 0; *p; i++, p++){
    if((j->r->attr&REGEXP) && j->match[i])
        envupd(*p, newword(j->match[i]));
    else
        envupd(*p, newword(""));
}
```
Uses REGEXP 128a, envupd() 113e, newword() 32d, and specialvars-8 31d.

⟨applyrules *other locals* 130d⟩+≡ (77a) ◁ 79c
```
Resub rmatch[NREGEXP];
```
Uses NREGEXP 129c.

⟨applyrules *other initializations* 130e⟩≡ (77a)
```
memset((char*)rmatch, 0, sizeof(rmatch));
```

⟨applyrules() *if regexp rule then continue if some conditions* 130f⟩≡ (78e)
```
if(r->attr&REGEXP){
    stem[0] = '\0';
    memset((char*)rmatch, 0, sizeof(rmatch));
    patrule = r;
    if(regexec(r->pat, node->name, rmatch, NREGEXP) == 0)
        continue;
}
```
Uses NREGEXP 129c, REGEXP 128a, and patrule 128d.

⟨applyrules() *if regexp rule, adjust buf and rmatch* 130g⟩≡ (78e)
```
if(r->attr&REGEXP)
    regsub(pre->s, buf, sizeof(buf), rmatch, NREGEXP);
```
Uses NREGEXP 129c and REGEXP 128a.

11.2 Shell-command expansion: `<cmd>`

Backquote expansion is the `mkfile` equivalent of the Bourne shell (`sh`) `` `ls *.c` `` (or in `rc` `` `{ls *.c} ``, see the SHELL book [Pad18]): it runs a command at parse time and substitutes its standard output back into the line being assembled. The most common use is generating a list of source files (`` OFILES=`ls *.c | sed ``

's/.c/.o/'), which would otherwise require either listing every file by hand or hoping a meta rule covers them all.

mk inherits its backquote syntax from sh: `...`, terminated by a closing backquote. That form is notoriously hard to nest — a backquote inside a backquote must be backslash-escaped, and the escaping compounds at every level — which is why bash and the POSIX shells later added the $(...) form ($(ls $(dirname x)) nests cleanly), and why rc switched to `{...}, where the braces delimit the command unambiguously. mk does not understand $(...), but bquote()[131b] does accept the rc brace form as well: if the character after the backquote is a { it reads up to the matching }, otherwise it reads sh-style up to the next backquote.

The implementation in bquote() is subtle because the same Bufblock[171c] holds the command text on the way out and the command's output on the way back: bquote() remembers the start position before invoking execsh()[193c] and rewinds buf->current to that point so the output overwrites the command in place. The grandchild-pipe trick from Section 8.3 is reused here, this time with the parent capturing the pipe's read end into buf instead of discarding it.

11.2.1 Parsing backquotes: bquote()

⟨assline() *switch character cases* 131a⟩+≡ (53b) ◁56a

```
case '`':
    if (bquote(bp, buf) == ERROR_0)
        Exit();
    break;
```

Uses bquote() 131b.

⟨*function* bquote 131b⟩≡ (189b)

```
/*
 * assemble a back-quoted shell command into a buffer
 */
static error0
bquote(Biobuf *bp, Bufblock *buf)
{
    int line;
    int c, term;
    int start;

    line = mkinline;
    ⟨bquote() skip spaces 132b⟩
    if(c == '{'){
        term = '}';  /* rc style */
        ⟨bquote() skip spaces 132b⟩
    } else
        term = '`';  /* sh style */

    start = buf->current - buf->start;
    for(;c > 0; c = nextrune(bp, false)){
        if(c == term){
            insert(buf, '\n');
            insert(buf, '\0');
            // prepare to overwrite the command string with its output
            buf->current = buf->start + start;

            ⟨bquote() execute shell command in buf 132c⟩
            return OK_1;
        }
        if(c == '\n')
            break; // go to error
        ⟨bquote() handle quotes and escape characters 132a⟩
        rinsert(buf, c);
```

```
        }
        SYNERR(line);
        fprint(STDERR, "missing closing %c after `\n", term);
        return ERROR_0;
    }
```

Uses SYNERR 51d, insert() 173a, mkinline 51c, nextrune() 54a, and rinsert() 173b.

⟨bquote() *handle quotes and escape characters* 132a⟩≡ (131b)

```
    if(c == '\'' || c == '"' || c == '\\'){
        insert(buf, c);
        if(!escapetoken(bp, buf, true, c))
            return ERROR_0;
        continue;
    }
```

Uses escapetoken() 56b and insert() 173a.

⟨bquote() *skip spaces* 132b⟩≡ (131)

```
    while((c = Bgetrune(bp)) == ' ' || c == '\t')
            ;
```

11.2.2 Adjusting execsh()

⟨bquote() *execute shell command in* buf 132c⟩≡ (131b)

```
    initenv();
    // running the command, passing a buf argument
    execsh(nil, buf->current, buf, shellenv);
```

Uses initenv() 114b and shellenv 112b.

⟨execsh() *other locals* 132d⟩+≡ (105e) ◁106a 132i▷

```
    fdt out[2];
```

⟨execsh() *if buf then create pipe to save output* 132e⟩≡ (105e)

```
    if(buf && pipe(out) < 0){
        perror("pipe");
        Exit();
    }
```

⟨execsh() *in child, if buf, close one side of pipe* 132f⟩≡ (105e)

```
    if(buf)
        close(out[0]);
```

⟨execsh() *in child, if buf, dup and close* 132g⟩≡ (105e)

```
    if(buf){
        // output now goes in the pipe
        dup(out[1], STDOUT);
        close(out[1]);
    }
```

⟨execsh() *in grandchild, if buf, close other side of pipe* 132h⟩≡ (105e)

```
    if(buf)
        close(out[1]);
```

⟨execsh() *other locals* 132i⟩+≡ (105e) ◁132d

```
    int tot, n;
```

⟨execsh() *in parent, if buf, close other side of pipe and read output* 133⟩≡ (105e)

```
  if(buf){
      close(out[1]);
      tot = 0;
      for(;;){
          if (buf->current >= buf->end)
              growbuf(buf);
          n = read(out[0], buf->current, buf->end-buf->current);
          if(n <= 0)
              break;
          buf->current += n;
          tot += n;
      }
      if (tot && buf->current[-1] == '\n')
          buf->current--;
      close(out[0]);
  }
```

Uses growbuf() 173d.

11.3 Dynamic mkfile: <| <prog>

GNU Make has ifdef/ifeq/else for conditional sections. mk does not, and on purpose. Instead of growing the DSL with a preprocessor, mk reuses the file-inclusion mechanism but lets the included "file" be the output of a program: <|prog runs prog, pipes its stdout into parse()[51e], and the result is a fragment of mkfile computed at run time. This is how the Plan 9 kernel's mkfiles pull configuration data from a separate config program, and how a build can branch on environment without adding control flow to the mkfile grammar.

For example, suppose the compiler flags should depend on the target architecture. In GNU Make you reach for a conditional baked into the makefile language:

```
ifeq ($(OBJTYPE),arm)
CFLAGS=-mfpu=neon
else
CFLAGS=-O2
endif
```

mk has no such construct. Instead you move the decision into a program and include its output:

```
<|flags

prog: $OFILES
        $CC $CFLAGS -o prog $OFILES
```

where flags is any script that writes mkfile text to stdout:

```
#!/bin/rc
if(~ $objtype arm)
        echo CFLAGS=-mfpu=neon
if not
        echo CFLAGS=-O2
```

mk runs flags, parses the single line it prints (say CFLAGS=-mfpu=neon) exactly as if it had been typed in the mkfile, and the branching lives in a language already designed for it — the shell — instead of in a bolted-on mkfile preprocessor.

The implementation reuses `parse()` reentrantly: the recursive `parse()` call on the pipe descriptor is the same one that handles a normal `<file` include—one of several places where keeping the parser in a single function pays off.

⟨`rhead()` *adjust sep if dynamic mkfile* `<|` 134a⟩≡ (57b)
```
if(sep == '<' && *p == '|'){
    sep = '|';
    p++;
}
```

⟨`WaitupParam` *other cases* 134b⟩≡ (109b) 162a ▷
```
EMPTY_CHILDREN_IS_ERROR3 = -3,
```

⟨`parse()` *other locals* 134c⟩+≡ (51e) ◁ 73b 143c ▷
```
int pid;
```

⟨`parse()` *switch rhead cases* 134d⟩+≡ (51e) ◁ 73c
```
case '|':
    p = wtos(tail, ' ');
    ⟨parse() sanity check p for include program name 134e⟩

    initenv();
    pid = pipecmd(p, shellenv, &newfd);
    ⟨parse() sanity check newfd 134f⟩
    else
        // recursive call
        parse(p, newfd, 0);

    while(waitup(EMPTY_CHILDREN_IS_ERROR3, &pid) >= 0)
        ;
    ⟨parse() sanity check pid after waitup 134g⟩
    break;
```
Uses `EMPTY_CHILDREN_IS_ERROR3` 134b, `initenv()` 114b, `parse()` 51e, `shellenv` 112b, `waitup()` 107c, and `wtos()` 33d.

⟨`parse()` *sanity check* `p` *for include program name* 134e⟩≡ (134d)
```
if(*p == '\0'){
    SYNERR(-1);
    fprint(STDERR, "missing include program name\n");
    Exit();
}
```
Uses `SYNERR` 51d.

⟨`parse()` *sanity check* `newfd` 134f⟩≡ (134d)
```
if(newfd < 0){
    fprint(STDERR, "warning: skipping missing program file: ");
    perror(p);
}
```

⟨`parse()` *sanity check* `pid` *after waitup* 134g⟩≡ (134d)
```
if(pid != 0){
    fprint(STDERR, "bad include program status\n");
    Exit();
}
```

⟨*function* pipecmd 135a⟩≡ (199)

```
int
pipecmd(char *cmd, ShellEnvVar *e, int *fd)
{
    int pid;
    fdt pfd[2];

    if(DEBUG(D_EXEC))
        fprint(STDOUT, "pipecmd='%s'\n", cmd);/**/

    if(fd && pipe(pfd) < 0){
        perror("pipe");
        Exit();
    }
    pid = rfork(RFPROC|RFFDG|RFENVG);
    if(pid < 0){
        perror("mk fork");
        Exit();
    }
    if(pid == 0){
        if(fd){
            close(pfd[0]);
            dup(pfd[1], 1);
            close(pfd[1]);
        }
        if(e)
            exportenv(e);
        if(shflags)
            execl(shell->shell, shell->shellname, shflags, "-c", cmd, nil);
        else
            execl(shell->shell, shell->shellname, "-c", cmd, nil);
        perror(shell->shell);
        _exits("exec");
    }
    if(fd){
        close(pfd[1]);
        *fd = pfd[0];
    }
    return pid;
}
```

Uses `DEBUG` 165c, `D_EXEC` 165a, and `shflags` 105d.

11.4 Substitution variables: `$<name>:<pattern>=<subst>`

This is `mk`'s answer to a real problem: given a list of source files, derive the corresponding list of object files without listing both. The syntax `$OFILES:%.c=%.o` takes the value of `$OFILES`, matches each word against `%.c`, and rewrites the `%` into `.o`. Compared to GNU Make's `$(patsubst \%.c,\%.o,$(OFILES))`, it is shorter and reuses the same `%` pattern character that meta rules use, so there is one matching mechanism to learn instead of two. The catch is that the parser must recognize `$ ...` as atomic—the `:` inside would otherwise look like the rule separator `rhead()`[57b] is hunting for, hence the `vargen` flag in `charin()`[58c] that suppresses separator detection between `${` and `}`.

11.4.1 Parsing adjustments

⟨`charin()` *switch rune cases* 135b⟩+≡ (58c) ◁59a

```
case '$':
```

```
        if(*(cp+1) == '{')
            vargen = true;
        break;
    case '}':
        if(vargen)
            vargen = false;
        else
           // same as default: case
           if(utfrune(pat, r))
              return cp;
        break;
```

⟨charin() *sanity check vargen* 136a⟩≡ (58c)

```
  if(vargen){
      SYNERR(-1);
      fprint(STDERR, "missing closing } in pattern generator\n");
  }
```

Uses SYNERR 51d.

⟨varsub() *if variable starts with open brace* 136b⟩≡ (63d)

```
  if(**s == '{')  /* either ${name} or ${name: A%B==C%D}*/
      return expandvar(s);
```

Uses expandvar() 136c.

⟨*function* expandvar 136c⟩≡ (192b)

```
  static Word*
  expandvar(char **s)
  {
      Word *w;
      Bufblock *buf;
      Symtab *sym;
      char *cp, *begin, *end;

      begin = *s;
      (*s)++;      /* skip the '{' */
      buf = varname(s);
      if (buf == nil)
          return nil;
      cp = *s;
      if (*cp == '}') {    /* ${name} variant*/ //$
          (*s)++;     /* skip the '}' */
          w = varmatch(buf->start);
          freebuf(buf);
          return w;
      }


      if (*cp != ':') {
          SYNERR(-1);
          fprint(STDERR, "bad variable name <%s>\n", buf->start);
          freebuf(buf);
          return nil;
      }
      cp++;
      end = charin(cp , "}");
      if(end == nil){
          SYNERR(-1);
          fprint(STDERR, "missing '}': %s\n", begin);
          Exit();
      }
```

```
        *end = '\0';
        *s = end+1;

        sym = symlook(buf->start, S_VAR, nil);
        if(sym == nil || sym->u.value == 0)
            w = newword(buf->start);
        else
            w = subsub(sym->u.ptr, cp, end);
        freebuf(buf);
        return w;
    }
```

Uses SYNERR 51d, S_VAR 29a, charin() 58c, freebuf() 172e, newword() 32d, symlook() 30a, and varname() 64d.

11.4.2 Substitutions: subsub()

⟨*function* subsub 137⟩≡ (192b)

```
    static Word*
    subsub(Word *v, char *s, char *end)
    {
        int nmid;
        Word *head, *tail, *w, *h;
        Word *a, *b, *c, *d;
        Bufblock *buf;
        char *cp, *enda;

        a = extractpat(s, &cp, "=%&", end);
        b = c = d = nil;
        if(PERCENT(*cp))
            b = extractpat(cp+1, &cp, "=", end);
        if(*cp == '=')
            c = extractpat(cp+1, &cp, "&%", end);
        if(PERCENT(*cp))
            d = stow(cp+1);
        else if(*cp)
            d = stow(cp);

        head = tail = nil;
        buf = newbuf();
        for(; v; v = v->next){
            h = w = nil;
            if(submatch(v->s, a, b, &nmid, &enda)){
                /* enda points to end of A match in source;
                 * nmid = number of chars between end of A and start of B
                 */
                if(c){
                    h = w = wdup(c);
                    while(w->next)
                        w = w->next;
                }
                if(PERCENT(*cp) && nmid > 0){
                    if(w){
                        bufcpy(buf, w->s, strlen(w->s));
                        bufcpy(buf, enda, nmid);
                        insert(buf, '\0');
                        free(w->s);
                        w->s = strdup(buf->start);
                    } else {
                        bufcpy(buf, enda, nmid);
                        insert(buf, '\0');
```

```
                h = w = newword(buf->start);
            }
            buf->current = buf->start;
        }
        if(d && *d->s){
            if(w){

                bufcpy(buf, w->s, strlen(w->s));
                bufcpy(buf, d->s, strlen(d->s));
                insert(buf, '\0');
                free(w->s);
                w->s = strdup(buf->start);
                w->next = wdup(d->next);
                while(w->next)
                    w = w->next;
                buf->current = buf->start;
            } else
                h = w = wdup(d);
        }
    }
    if(w == nil)
        h = w = newword(v->s);

    if(head == nil)
        head = h;
    else
        tail->next = h;
    tail = w;
}
freebuf(buf);
freewords(a);
freewords(b);
freewords(c);
freewords(d);
return head;
}
```

⟨*function* extractpat 138⟩≡ (192b)

```
static Word*
extractpat(char *s, char **r, char *term, char *end)
{
    int save;
    char *cp;
    Word *w;

    cp = charin(s, term);
    if(cp){
        *r = cp;
        if(cp == s)
            return nil;
        save = *cp;
        *cp = '\0';
        w = stow(s);
        *cp = save;
    } else {
        *r = end;
        w = stow(s);
    }
    return w;
}
```

Uses charin() 58c and stow() 60b.

⟨*function* submatch 139⟩≡ (192b)
```
static bool
submatch(char *s, Word *a, Word *b, int *nmid, char **enda)
{
    Word *w;
    int n;
    char *end;

    n = 0;
    for(w = a; w; w = w->next){
        n = strlen(w->s);
        if(strncmp(s, w->s, n) == 0)
            break;
    }
    if(a && w == nil)  /*  a == NULL matches everything*/
        return false;

    *enda = s+n;  /* pointer to end a A part match */
    *nmid = strlen(s)-n; /* size of remainder of source */
    end = *enda+*nmid;
    for(w = b; w; w = w->next){
        n = strlen(w->s);
        if(strcmp(w->s, end-n) == 0){
            *nmid -= n;
            break;
        }
    }
    if(b && w == nil)  /* b == NULL matches everything */
        return false;
    return true;
}
```

11.5 Rule attributes

A *rule attribute* is a single-letter flag attached to a rule between two colons, as in `clean:V:` for a virtual target. I described it briefly in Section 5.2.2. Attributes let the user override individual aspects of `mk`'s default behavior on a per-rule basis without bloating the syntax:

- `V` virtual: do not check for a file with this name
- `Q` quiet: do not echo the recipe
- `D` delete the target if the recipe fails
- `E` execute the recipe shell without `-e`
- `N` no warning if the recipe is empty
- `P` use a custom program to compare timestamps
- `R` the target is a regular expression

The implementation pattern is the same for all of them: `rhead()`[57b] sets a bit in `Rule.attr` when it sees the letter, `attribute()`[140b] propagates the bit from rules onto the nodes that use them (so `work()`[95a], `dorecipe()`[97e],

and `waitup()`[107c] can consult the node directly without re-walking arcs), and the relevant code path checks the flag.

⟨`graph()` *propagate attributes* 140a⟩≡ (76)
```
  // propagate attributes in rules to their node
  attribute(root);
```
Uses `attribute()` 140b.

⟨*function* `attribute` 140b⟩≡ (191a)
```
  static void
  attribute(Node *n)
  {
      Arc *a;

      for(a = n->arcs; a; a = a->next){
          ⟨attribute() propagate rule attribute to node cases 140f⟩
          // recurse
          if(a->n)
              attribute(a->n);
      }
      ⟨attribute() if virtual node 140g⟩
  }
```
Uses `attribute()` 140b.

11.5.1 Virtual targets: `:V:`

⟨`Rule_attr` *cases* 140c⟩+≡ (36a) ◁128a 141b▷
```
  VIR    = 0x0010,
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 140d⟩+≡ (70e) ◁128b 141c▷
```
  case 'V':
      *attr |= VIR;
      break;
```

⟨`Node_flag` *cases* 140e⟩+≡ (40c) ◁90f 141d▷
```
  VIRTUAL    = 0x0001,
```

⟨`attribute()` *propagate rule attribute to node cases* 140f⟩≡ (140b) 141e▷
```
  if(a->r->attr&VIR)
      n->flags |= VIRTUAL;
```
Uses `VIR` 140c and `VIRTUAL` 140e.

⟨`attribute()` *if virtual node* 140g⟩≡ (140b)
```
  if(n->flags&VIRTUAL)
      n->time = 0;
```
Uses `VIRTUAL` 140e.

⟨`dorecipe()` *when no recipe found, if virtual or norecipe node* 140h⟩≡ (98b)
```
  if((node->flags&VIRTUAL) || (node->flags&NORECIPE)){
      ⟨dorecipe() when no recipe found, if archive name 150e⟩
      else
          update(node, false);
      ⟨dorecipe() when no recipe found, if tflag 156c⟩
      //bugfix:
      return;
  }
```
Uses `NORECIPE` 142g, `VIRTUAL` 140e, and `update()` 108b.

⟨update() *if virtual node or inexistent file* 141a⟩≡ (108b)
```
if((node->flags&VIRTUAL) || (access(node->name, AEXIST) != OK_0)){
    node->time = 1;
    for(a = node->arcs; a; a = a->next)
        if(a->n && outofdate(node, a, true))
            node->time = a->n->time;
}
```
Uses VIRTUAL 140e and outofdate() 96c.

11.5.2 Deleting a target when the recipe returns an error: :D:

⟨Rule_attr *cases* 141b⟩+≡ (36a) ◁140c 141i▷
```
DEL    = 0x0080,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 141c⟩+≡ (70e) ◁140d 142a▷
```
case 'D':
    *attr |= DEL;
    break;
```

⟨Node_flag *cases* 141d⟩+≡ (40c) ◁140e 142b▷
```
DELETE     = 0x0800,
```

⟨attribute() *propagate rule attribute to node cases* 141e⟩+≡ (140b) ◁140f 142h▷
```
if(a->r->attr&DEL)
    n->flags |= DELETE;
```
Uses DEL 141b and DELETE 141d.

⟨waitup() *other locals* 141f⟩+≡ (107c) ◁120c 145e▷
```
Node *n;
bool done;
```

⟨waitup() *when error in child process, delete if DELETE node* 141g⟩≡ (110b)
```
for(n = j->n, done = false; n; n = n->next)
    if(n->flags&DELETE){
        if(!done) {
            fprint(STDERR, ", deleting");
            done = true;
        }
        fprint(STDERR, " '%s'", n->name);
        delete(n->name);
    }
```
Uses DELETE 141d and delete() 141h.

⟨*function* delete 141h⟩≡ (189c)
```
void
delete(char *name)
{
    if(utfrune(name, '(') == nil) {  /* file */
        if(remove(name) < 0)
            perror(name);
    } else
        fprint(STDERR, "hoon off; mk can't delete archive members\n");
}
```

11.5.3 Quiet mode (not printing the recipe): :Q:

⟨Rule_attr *cases* 141i⟩+≡ (36a) ◁141b 142e▷
```
QUIET  = 0x0008,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 142a⟩+≡ (70e) ◁141c 142c▷
```
case 'Q':
    *attr |= QUIET;
    break;
```

11.5.4 Running a shell script without -e: :E:

⟨Node_flag *cases* 142b⟩+≡ (40c) ◁141d 142g▷
```
NOMINUSE   = 0x1000,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 142c⟩+≡ (70e) ◁142a 142f▷
```
case 'E':
    *attr |= NOMINUSE;
    break;
```

⟨sched() *reset flags if* NOMINUSE *rule* 142d⟩≡ (103e)
```
if (j->r->attr&NOMINUSE)
    flags = nil;
```
Uses NOMINUSE 142b.

11.5.5 Disabling the no-recipe warning, :N:

⟨Rule_attr *cases* 142e⟩+≡ (36a) ◁141i 142i▷
```
NOREC  = 0x0040,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 142f⟩+≡ (70e) ◁142c 142j▷
```
case 'N':
    *attr |= NOREC;
    break;
```

⟨Node_flag *cases* 142g⟩+≡ (40c) ◁142b 154c▷
```
NORECIPE   = 0x0400,
```

⟨attribute() *propagate rule attribute to node cases* 142h⟩+≡ (140b) ◁141e
```
if(a->r->attr&NOREC)
    n->flags |= NORECIPE;
```
Uses NOREC 142e and NORECIPE 142g.

11.5.6 Forbidding metarules to match virtual targets: :n:

⟨Rule_attr *cases* 142i⟩+≡ (36a) ◁142e
```
NOVIRT = 0x0100,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 142j⟩+≡ (70e) ◁142f 143a▷
```
case 'n':
    *attr |= NOVIRT;
    break;
```

⟨applyrules() *skip this meta rule and continue if some conditions* 142k⟩+≡ (78e) ◁79d
```
if ((r->attr&NOVIRT) && lasta != &head && (lasta->r->attr&VIR))
    continue;
```
Uses NOVIRT 142i and VIR 140c.

11.5.7 Interactive recipes: `:I:`

⟨`rhead()` *when parsing rule attributes, switch rune cases* 143a⟩+≡ (70e) ◁142j 143h▷
```
//PAD: this is an extension in omk that I ignore here
case 'I':
    break;
```

11.5.8 Custom-dependency comparison program: `:P:`

Every attribute so far flips a boolean. `:P:` is different: the characters after the `P` up to the closing colon are a program name, and `mk` runs that program to decide whether a target is out of date instead of comparing modification times. It is invoked as `rc -c prog 'target' 'prereq'` and must exit with a null (success) status exactly when `target` is up to date with respect to `prereq`. This is `mk`'s hook for content-based change detection: the classic use is a generated header such as yacc's `y.tab.h`, rewritten on every parser build but usually with byte-for-byte identical contents. A rule `x.tab.h:Pcmp -s:  y.tab.h` runs `cmp -s`, so `x.tab.h` counts as up to date whenever the bytes match, sparing every file that includes it from a needless recompile. It anticipates, in one optional attribute, the content-hashing strategy that later build systems like Bazel made their default.

⟨`Rule` *other fields* 143b⟩+≡ (34) ◁128c
```
char  *prog;  /* to use in out of date */
```

⟨`parse()` *other locals* 143c⟩+≡ (51e) ◁134c
```
char *prog;
```

⟨`addrule()` *set more fields* 143d⟩+≡ (36b) ◁85d
```
r->prog = prog;
```

⟨`Arc` *other fields* 143e⟩+≡ (40f) ◁129d
```
char  *prog;
```

⟨`newarc()` *set other fields* 143f⟩+≡ (41c) ◁129e
```
a->prog = r->prog;
```

⟨`rhead()` *other locals* 143g⟩+≡ (57b) ◁70d
```
char *pp;
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 143h⟩+≡ (70e) ◁143a
```
case 'P':
    pp = utfrune(p, ':');
    if (pp == nil || *pp == 0)
        goto eos;
    *pp = 0;
    *prog = strdup(p);
    *pp = ':';
    p = pp;
    break;
```

⟨`Sxxx` *cases* 143i⟩+≡ (29a) ◁122b 147a▷
```
S_OUTOFDATE, /* n1\377n2 -> 2(outofdate) or 1(not outofdate) */
```

⟨`update()` *set outofdate prereqs if arc prog* 143j⟩≡ (108b)
```
for(a = node->arcs; a; a = a->next)
    if(a->prog)
        outofdate(node, a, true);
```
Uses `outofdate()` 96c.

⟨outofdate() *locals* 144a⟩≡ (96c)
```
  char buf[3*NAMEBLOCK];
  char *str = nil;
  Symtab *sym;
  int ret;
```
Uses NAMEBLOCK 79b.

⟨outofdate() *if* arc->prog 144b⟩≡ (96c)
```
  if(arc->prog){
      snprint(buf, sizeof buf, "%s%c%s", node->name, 0377,
          arc->n->name);
      sym = symlook(buf, S_OUTOFDATE, nil);
      if(sym == nil || eval){
          if(sym == nil)
              str = strdup(buf);
          ret = pcmp(arc->prog, node->name, arc->n->name);
          if(sym)
              sym->u.value = ret;
          else
              symlook(str, S_OUTOFDATE, (void *)ret);
      } else
          ret = sym->u.value;
      return (ret-1);
  }
```
Uses S_OUTOFDATE 143i, pcmp() 144c, and symlook() 30a.

⟨*function* pcmp 144c⟩≡ (191b)
```
  static int
  pcmp(char *prog, char *p, char *q)
  {
      char buf[3*NAMEBLOCK];
      int pid;

      Bflush(&bout);
      snprint(buf, sizeof buf, "%s '%s' '%s'\n", prog, p, q);
      pid = pipecmd(buf, nil, nil);
      while(waitup(EMPTY_CHILDREN_IS_ERROR3, &pid) >= 0)
          ;
      return (pid? 2:1);
  }
```
Uses EMPTY_CHILDREN_IS_ERROR3 134b, NAMEBLOCK 79b, bout 45b, and waitup() 107c.

Process

⟨*struct* Process 144d⟩≡ (190)
```
  struct Process {
      int pid;
      int status;

      // Extra
      // double_list<ref_own<Process> backward, forward
      Process *b, *f;
  };
```

⟨*global* phead 144e⟩≡ (190)
```
  // double_list<ref_own<Process> (next = Process.f)
  static Process *phead;
```

⟨*global* pfree 145a⟩≡ (190)
```
// double_list<ref_own<Process> (next = Process.f)
static Process *pfree;
```

⟨*function* pnew 145b⟩≡ (190)
```
static void
pnew(int pid, int status)
{
    Process *p;

    // p = pop_list(pfree)
    if(pfree){
        p = pfree;
        pfree = p->f;
    } else
        p = (Process *)Malloc(sizeof(Process));

    p->pid = pid;
    p->status = status;

    // add_list(p, phead)
    p->f = phead;
    phead = p;
    if(p->f)
        p->f->b = p;
    p->b = nil;
}
```
Uses Malloc() 171a, pfree-19 145a, and phead-18 144e.

⟨*function* pdelete 145c⟩≡ (190)
```
static void
pdelete(Process *p)
{
    // remove_double_list(p, phead, pfree)
    if(p->f)
        p->f->b = p->b;
    if(p->b)
        p->b->f = p->f;
    else
        phead = p->f;
    p->f = pfree;
    pfree = p;
}
```
Uses pfree-19 145a and phead-18 144e.

waitup() adjustments

⟨waitup() *if slot not found, not a job pid, update process list* 145d⟩≡ (107c)
```
if(slot < 0){
   ⟨waitup() if DEBUG(D_EXEC) and slot ¡ 0 168d⟩
    pnew(pid, buf[0]? 1:0);
    goto again;
}
```
Uses pnew() 145b.

⟨waitup() *other locals* 145e⟩+≡ (107c) ◁141f
```
Process *p;
```

⟨`waitup()` *if retstatus, check process list* 146a⟩≡ (107c)
```
/* first check against the process list */
if(retstatus)
    for(p = phead; p; p = p->f)
        if(p->pid == *retstatus){
            *retstatus = p->status;
            pdelete(p);
            return -1;
        }
```
Uses `pdelete()` 145c and `phead-18` 144e.

⟨`waitup()` *if retstatus, check if matching pid* 146b⟩≡ (107c)
```
if(retstatus && pid == *retstatus){
    *retstatus = buf[0]? 1:0;
    return -1;
}
```

`killchildren()` adjustments

⟨`killchildren()` *locals* 146c⟩≡ (111e)
```
Process *p;
```

⟨`killchildren()` *expunge not-job processes* 146d⟩≡ (111e)
```
for(p = phead; p; p = p->f)
    expunge(p->pid, msg);
```
Uses `phead-18` 144e.

⟨*function* `expunge` 146e⟩≡ (199)
```
void
expunge(int pid, char *msg)
{
    postnote(PNPROC, pid, msg);
}
```

11.6 Variable attributes

Just as a rule carries attributes between its `:` colons, a variable definition can carry an attribute. There is only one, `U`, covered next.

11.6.1 Private variables: `=U=`

Marking a variable with the `U` attribute (written `=U=`) makes it private to the `mkfile`: it can still be used in rules and other variable definitions, but `mk` will not export it into the environment of the shell processes that run recipes. By default every `mk` variable becomes an environment variable for recipes; `U` (for unexport) opts out, by joining the `S_NOEXPORT`[147a] set that `ecopy()`[114c] consults when assembling each recipe's environment. It is the way to keep `mkfile` bookkeeping variables from leaking into — or colliding with — the environment a recipe actually sees.

⟨`rhead()` *when parsing variable attributes, switch rune cases* 146f⟩≡ (74e)
```
case 'U':
    *attr = 1;
    break;
```

⟨`parse()` *when parsing variable definitions, if variable with attr* 146g⟩≡ (73c)
```
if(attr)
    symlook(head->s, S_NOEXPORT, (void *)"");
```
Uses `S_NOEXPORT` 147a and `symlook()` 30a.

⟨Sxxx *cases* 147a⟩+≡ (29a) ◁143i 150d▷
```
  S_NOEXPORT, /* var -> noexport */ // set of noexport variables
```

⟨ecopy() *return and do not copy if* S_NOEXPORT *symbol* 147b⟩≡ (114c)
```
  if(symlook(s->name, S_NOEXPORT, nil))
      return;
```
Uses S_NOEXPORT 147a and symlook() 30a.

11.7 Advanced mk variables

Beyond the `$stem` of a meta rule, `mk` installs a handful of automatic variables into every recipe's environment, each set by `buildenv()`[117c] just before the recipe runs. The next subsections cover them: the rule's targets and prerequisites (in two flavours each), and a few process-level values like `$pid` and `$nproc`.

11.7.1 $target versus $alltarget

`$alltarget` is every target named on the left-hand side of the rule; `$target` is only the subset actually out of date and being rebuilt. For an ordinary single-target rule they are equal. The distinction matters for a multi-target virtual rule used to dispatch a recursive build, such as `all install clean nuke:V:` whose recipe loops over subdirectories running `mk $target`. Running `mk clean` then sets `$target` to just `clean` — the one target that triggered the rule — so each subdirectory does the single action asked, whereas `$alltarget` would expand to the whole `all install clean nuke` list and try to do everything.

⟨dorecipe() *other locals* 147c⟩+≡ (97e) ◁99a 148a▷
```
  Word oldtargets;
  Word *last_oldtargets = &oldtargets;
```

⟨dorecipe() *update list of outdated targets* 147d⟩≡ (99b)
```
  if(!aflag && n->time) {
      for(a = n->arcs; a; a = a->next)
          if(a->n && outofdate(n, a, false))
              break;
      // no out of date arc, node does not need to be regenerated
      if(a == nil)
          continue;
      // else, find an outdated arc for node of target
  }
  last_oldtargets->next = newword(buf);
  last_oldtargets = last_oldtargets->next;
```
Uses aflag 159b, newword() 32d, and outofdate() 96c.

⟨specialvars *other array elements* 147e⟩+≡ (31d) ◁130a 148c▷
```
  "alltarget",
```

⟨Job *other fields* 147f⟩+≡ (41d) ◁129g 148d▷
```
  Word  *at; /* all targets */
```

⟨newjob() *setting other fields* 147g⟩≡ (42b) 148e▷
```
  j->at = alltargets;
```

⟨buildenv() *envupd some variables* 147h⟩+≡ (117c) ◁130c 148f▷
```
  envupd("alltarget", wdup(j->at));
```
Uses envupd() 113e and wdup() 33c.

11.7.2 $prereq versus $newprereq

`$prereq` is the full prerequisite list; `$newprereq` is only those prerequisites that were actually out of date (newer than the target). The distinction earns its keep when updating an archive: given a rule `libc.a: printf.o read.o write.o` where only `read.o` has changed, `$newprereq` is just `read.o`, so the recipe can `ar` in the one modified member instead of rearchiving all three (see also Section 11.9 for more on the use of archives and `mk`).

⟨dorecipe() *other locals* 148a⟩+≡ (97e) ◁147c
```
Word newprereqs;
```

⟨dorecipe() *when outofdate node, update list of newprereqs* 148b⟩≡ (100)
```
addw(&newprereqs, a->n->name);
```
Uses addw() 33b.

⟨specialvars *other array elements* 148c⟩+≡ (31d) ◁147e 149a▷
```
"newprereq",
```

⟨Job *other fields* 148d⟩+≡ (41d) ◁147f
```
Word  *np; /* new prerequisites */
```

⟨newjob() *setting other fields* 148e⟩+≡ (42b) ◁147g
```
j->np = newprereqs;
```

⟨buildenv() *envupd some variables* 148f⟩+≡ (117c) ◁147h 149c▷
```
envupd("newprereq", wdup(j->np));
```
Uses envupd() 113e and wdup() 33c.

11.7.3 $NREP

`$NREP` caps how many times `mk` will re-apply a single meta or regexp rule while chaining rules to reach a target — the guard against arbitrarily deep rule application. It is read dynamically by `nrep()`[148h] at the start of each `mk()` pass (so an `mkfile` can change it on the fly), and floored at 1.

⟨mk() *initializations* 148g⟩+≡ (92) ◁102a 161d▷
```
nrep();  /* it can be updated dynamically */
```

⟨*function* nrep 148h⟩≡ (191a)
```
void
nrep(void)
{
    Symtab *sym;
    Word *w;

    sym = symlook("NREP", S_VAR, nil);
    if(sym){
        w = sym->u.ptr;
        if (w && w->s && *w->s)
            nreps = atoi(w->s);
    }
    if(nreps < 1)
        nreps = 1;
    ⟨nrep() if DEBUG(D_GRAPH) 167d⟩
}
```

11.7.4 $pid

`$pid` is the process id of the `mk` master process (note that `buildenv()`[117c] runs there, not in the `execsh()` child). It is handed to recipes that want a unique token — for instance to name a temporary file or coordinate between rules.

⟨`specialvars` *other array elements* 149a⟩+≡ (31d) ◁148c 149d▷
```
  "pid",
```

⟨`buildenv()` *locals* 149b⟩+≡ (117c) ◁130b 150b▷
```
  char buf[256];
```

⟨`buildenv()` *envupd some variables* 149c⟩+≡ (117c) ◁148f 149e▷
```
  snprint(buf, sizeof buf, "%d", getpid());
  envupd("pid", newword(buf));
```
Uses `envupd()` 113e and `newword()` 32d.

11.7.5 $nproc

`$nproc` is the scheduler slot number assigned to this job, a small integer below the parallelism limit. A recipe running in parallel can use it to pick a per-slot scratch file or output area without colliding with the other jobs `mk` launched concurrently.

⟨`specialvars` *other array elements* 149d⟩+≡ (31d) ◁149a 150a▷
```
  "nproc",
```

⟨`buildenv()` *envupd some variables* 149e⟩+≡ (117c) ◁149c 150c▷
```
  snprint(buf, sizeof buf, "%d", slot);
  envupd("nproc", newword(buf));
```
Uses `envupd()` 113e and `newword()` 32d.

11.8 Shell choice: $MKSHELL

By default `mk` runs recipes with `rc`, but a build may prefer `sh`. Setting the `MKSHELL` variable to a shell's path overrides `main()`[45a]'s default, so the same `mk` binary can drive either shell — useful when porting `mkfiles` to UNIX, where `rc` is usually not installed.

⟨`main()` *locals* 149f⟩+≡ (45a) ◁125a 160a▷
```
  Symtab* sym;
```

⟨`main()` *initializations before building* 149g⟩+≡ (45a) ◁125c
```
  //pad-ext: MKSHELL environment var to specify the path to rc
  sym = symlook("MKSHELL", S_VAR, 0);
  if(sym != nil) {
    w = (Word*) sym->u.value;
    if(w != nil && w->s != nil) {
      shell->shell = w->s;
    }
  }
```
Uses `S_VAR` 29a and `symlook()` 30a.

11.9 Dealing with archives (libraries)

Plan 9, like UNIX, packs object files into `ar` archives (`.a` libraries), and `mk` can name an individual member as a target with the `lib.a(member.o)` syntax. Such a target is called an *aggregate*: ts timestamp is not the archive's own mtime but the member's time recorded inside the `.a` header. `atimeof()`[150g] reads those per-member times (caching the whole archive's table under `S_AGG`[150d] so each archive is scanned only once), and `atouch()`[151] writes them back, letting `mk` rebuild only the members whose sources changed rather than the entire library.

⟨`specialvars` *other array elements* 150a⟩+≡ (31d) ◁149d
```
  "newmember",
```

⟨`buildenv()` *locals* 150b⟩+≡ (117c) ◁149b
```
  Word *w, *v, **l;
  char *cp, *qp;
```

⟨`buildenv()` *envupd some variables* 150c⟩+≡ (117c) ◁149e
```
  // newmember
  l = &v;
  v = w = wdup(j->np);
  while(w){
      cp = strchr(w->s, '(');
      if(cp){
          qp = strchr(cp+1, ')');
          if(qp){
              *qp = 0;
              strcpy(w->s, cp+1);
              l = &w->next;
              w = w->next;
              continue;
          }
      }
      *l = w->next;
      free(w->s);
      free(w);
      w = *l;
  }
  envupd("newmember", v);
```
Uses `envupd()` 113e and `wdup()` 33c.

⟨`Sxxx` *cases* 150d⟩+≡ (29a) ◁147a 152b▷
```
  S_AGG,  /* aggregate -> time */
```

⟨`dorecipe()` *when no recipe found, if archive name* 150e⟩≡ (140h)
```
  if(strchr(node->name, '(') && node->time == 0)
      MADESET(node, MADE);
```
Uses `MADE` 40d and `MADESET` 94b.

⟨`timeof()` *if name archive member* 150f⟩≡ (82d)
```
  if(utfrune(name, '('))
      return atimeof(force, name);  /* archive */
```
Uses `atimeof()` 150g.

⟨*function* `atimeof` 150g⟩≡ (187a)
```
  ulong
  atimeof(int force, char *name)
  {
      Symtab *sym;
      ulong t;
      char *archive, *member, buf[512];
```

```
    archive = split(name, &member);
    if(archive == nil)
        Exit();

    t = mkmtime(archive, true);
    sym = symlook(archive, S_AGG, nil);
    if(sym){
        if(force || t > sym->u.value){
            atimes(archive);
            sym->u.value = t;
        }
    }
    else{
        atimes(archive);
        /* mark the aggegate as having been done */
        symlook(strdup(archive), S_AGG, "")->u.value = t;
    }
        /* truncate long member name to sizeof of name field in archive header */
    snprint(buf, sizeof(buf), "%s(%.*s)", archive, utfnlen(member, SARNAME), member);
    sym = symlook(buf, S_TIME, nil);
    if (sym)
        return sym->u.value;
    return 0;
}
```

Uses S_AGG 150d, S_TIME 157a, atimes() 152a, split() 153b, and symlook() 30a.

⟨*function* atouch 151⟩≡ (187a)

```
void
atouch(char *name)
{
    char *archive, *member;
    int fd, i;
    struct ar_hdr h;
    long t;

    archive = split(name, &member);
    if(archive == nil)
        Exit();

    fd = open(archive, ORDWR);
    if(fd < 0){
        fd = create(archive, OWRITE, 0666);
        if(fd < 0){
            perror(archive);
            Exit();
        }
        write(fd, ARMAG, SARMAG);
    }
    if(symlook(name, S_TIME, nil)){
        /* hoon off and change it in situ */
        seek(fd, SARMAG, 0);
        while(read(fd, (char *)&h, sizeof(h)) == sizeof(h)){
            for(i = SARNAME-1; i > 0 && h.name[i] == ' '; i--)
                    ;
            h.name[i+1] = 0;
            if(strcmp(member, h.name) == 0){
                t = SARNAME-sizeof(h); /* ughgghh */
                seek(fd, t, 1);
                fprint(fd, "%-12ld", time(nil));
```

```
                break;
            }
            t = atol(h.size);
            if(t&01) t++;
            seek(fd, t, 1);
        }
    }
    close(fd);
}
```

Uses S_TIME 157a, split() 153b, and symlook() 30a.

⟨*function* atimes 152a⟩≡ (187a)

```
static void
atimes(char *ar)
{
    struct ar_hdr h;
    ulong at, t;
    int fd, i;
    char buf[BIGBLOCK];
    Dir *d;

    fd = open(ar, OREAD);
    if(fd < 0)
        return;

    if(read(fd, buf, SARMAG) != SARMAG){
        close(fd);
        return;
    }
    if((d = dirfstat(fd)) == nil){
        close(fd);
        return;
    }
    at = d->mtime;
    free(d);
    while(read(fd, (char *)&h, SAR_HDR) == SAR_HDR){
        t = strtoul(h.date, nil, 0);
        if(t >= at) /* new things in old archives confuses mk */
            t = at-1;
        if(t == 0) /* as it sometimes happens; thanks ken */
            t = 1;
        for(i = sizeof(h.name)-1; i > 0 && h.name[i] == ' '; i--)
            ;
        if(h.name[i] == '/')  /* system V bug */
            i--;
        h.name[i+1]=0;  /* can stomp on date field */
        snprint(buf, sizeof buf, "%s(%s)", ar, h.name);
        symlook(strdup(buf), S_TIME, (void*)t)->u.value = t;
        t = atol(h.size);
        if(t&01) t++;
        seek(fd, t, 1);
    }
    close(fd);
}
```

Uses BIGBLOCK 182a, S_TIME 157a, and symlook() 30a.

⟨Sxxx *cases* 152b⟩+≡ (29a) ◁150d 157a▷

```
S_BITCH, /* bitched about aggregate not there */
```

⟨*function* type 153a⟩≡ (187a)
```
static int
type(char *file)
{
    int fd;
    char buf[SARMAG];

    fd = open(file, OREAD);
    if(fd < 0){
        if(symlook(file, S_BITCH, nil) == nil){
            Bprint(&bout, "%s doesn't exist: assuming it will be an archive\n", file);
            symlook(file, S_BITCH, (void *)file);
        }
        return 1;
    }
    if(read(fd, buf, SARMAG) != SARMAG){
        close(fd);
        return 0;
    }
    close(fd);
    return strncmp(ARMAG, buf, SARMAG) == 0;
}
```
Uses S_BITCH 152b, bout 45b, and symlook() 30a.

⟨*function* split 153b⟩≡ (187a)
```
static char*
split(char *name, char **member)
{
    char *p, *q;

    p = strdup(name);
    q = utfrune(p, '(');
    if(q){
        *q++ = 0;
        if(member)
            *member = q;
        q = utfrune(q, ')');
        if (q)
            *q = 0;
        if(type(p))
            return p;
        free(p);
        fprint(STDERR, "mk: '%s' is not an archive\n", name);
    }
    return nil;
}
```
Uses type() 153a.

⟨outofdate() *if arc node is an archive member* 153c⟩≡ (96c)
```
if(strchr(arc->n->name, '(') && arc->n->time == 0)
   /* missing archive member */
   return true;
```

11.10 Optimizations

The features in this section do not change what mk builds, only how quickly it decides what to build, or how little disk it needs along the way: pretending that deleted intermediate files are still present (mk -I), stamping

targets instead of rebuilding them (`mk -t`), and two layers of caching over the costly step of asking the filesystem for modification times.

11.10.1 Missing-intermediates optimization: `mk -I`

This optimization has a historical motivation. On 1970s and 1980s machines disk space was scarce, and users would routinely `rm *.o` after building a library to reclaim space. The next `mk` run would then see the `.o` files missing and rebuild the entire library, even though nothing in the source had changed and the `.a` file was newer than every `.c`. The fix is for `mk` to pretend a missing intermediate file is still present and up-to-date, provided the next level up in the graph is also up-to-date, so the library is not pointlessly rebuilt. The flags `CANPRETEND` and `PRETENDING` track this two-step state machine: a node "can pretend" if its name does not look like an archive member, and it is "pretending" once `work()`[95a] has actually elided it. The catch is that if anything below a pretending node turns out to be modified after all, `mk` must unpretend the node and treat it as missing again, hence the backtracking pass at the end of the `work()` arc loop.

⟨*global* `iflag` 154a⟩≡ (184)
```
bool iflag = false;
```
Uses `iflag` 154a.

⟨`main()` *-xxx switch cases* 154b⟩+≡ (46b) ◁126b 156b▷
```
case 'i':
    iflag = true;
    break;
```
Uses `iflag` 154a.

⟨`Node_flag` *cases* 154c⟩+≡ (40c) ◁142g
```
CANPRETEND = 0x0008,
PRETENDING = 0x0010,
```

⟨`clrmade() n->flags` *pretend adjustments* 154d⟩≡ (94a)
```
n->flags &= ~(CANPRETEND|PRETENDING);
if(strchr(n->name, '(') == nil || n->time)
    n->flags |= CANPRETEND;
```
Uses `CANPRETEND` 154c and `PRETENDING` 154c.

⟨`work()` *possibly pretending node* 154e⟩≡ (96b)
```
/*
 *   can we pretend to be made?
 */
if((!iflag) && (node->time == 0)
        && (node->flags&(PRETENDING|CANPRETEND))
        && parent_node && ra->n && !outofdate(parent_node, ra, false)){
    node->flags &= ~CANPRETEND;
    MADESET(node, MADE);
    if(explain && ((node->flags&PRETENDING) == 0))
        fprint(STDOUT, "pretending %s has time %lud\n", node->name, node->time);
    node->flags |= PRETENDING;
    return;
}
/*
 *   node is out of date and we REALLY do have to do something.
 *   quickly rescan for pretenders
 */
for(a = node->arcs; a; a = a->next)
    if(a->n && (a->n->flags&PRETENDING)){
        if(explain)
            Bprint(&bout, "unpretending %s because of %s because of %s\n",
```

```
            a->n->name, node->name,
            ra->n? ra->n->name : "rule with no prerequisites");

        unpretend(a->n);
        work(a->n, did, node, a);
        ready = false;
    }
if(!ready) { /* try later unless nothing has happened for -k's sake */
    work(node, did, parent_node, parent_arc);
    return;
}
```

Uses CANPRETEND 154c, MADE 40d, MADESET 94b, PRETENDING 154c, bout 45b, explain 124a, iflag 154a, outofdate() 96c, unpretend() 155b, and work() 95a.

⟨work() *possibly unpretending node* 155a⟩≡ (95a)

```
if((node->flags&MADE) && (node->flags&PRETENDING) && parent_node
    && outofdate(parent_node, parent_arc, false)){
    if(explain)
        fprint(STDOUT, "unpretending %s(%lud) because %s is out of date(%lud)\n",
            node->name, node->time, parent_node->name, parent_node->time);
    unpretend(node);
}
/*
 *   have a look if we are pretending in case
 *   someone has been unpretended out from underneath us
 */
if(node->flags&MADE){
    if(node->flags&PRETENDING){
        node->time = 0;
    }else
        return;
}
```

Uses MADE 40d, PRETENDING 154c, explain 124a, outofdate() 96c, and unpretend() 155b.

⟨*function* unpretend 155b⟩≡ (191b)

```
static void
unpretend(Node *n)
{
    MADESET(n, NOTMADE);
    n->flags &= ~(CANPRETEND|PRETENDING);
    n->time = 0;
}
```

Uses CANPRETEND 154c, MADESET 94b, NOTMADE 40d, and PRETENDING 154c.

⟨work() *locals* 155c⟩+≡ (95a) ◁96a

```
Arc *ra = nil;
```

⟨work() *update* ra *when outofdate* node *with arc* a 155d⟩≡ (96b)

```
if((ra == nil) || (ra->n == nil) || (ra->n->time < a->n->time))
    ra = a;
```

⟨work() *update* ra *when no dest in arc and no src* 155e⟩≡ (96b)

```
if(ra == nil)
    ra = a;
```

⟨update() *unpretend node* 155f⟩≡ (108b)

```
node->flags &= ~(CANPRETEND|PRETENDING);
```

Uses CANPRETEND 154c and PRETENDING 154c.

11.10.2 Touching-mode optimization: `mk -t`

`mk -t` ("touch") marks targets up to date without running their recipes, by bumping each target's mtime to now (`touch()`, or `atouch()`[151] for archive members). It is the escape hatch for when you know a target is already current — say after editing only a comment — and want to stop `mk` from rebuilding the world, at the risk of lying to the dependency graph if you are wrong.

⟨*global* `tflag` 156a⟩≡ (184)
```
bool tflag = false;
```
Uses `tflag` 156a.

⟨`main()` *-xxx switch cases* 156b⟩+≡ (46b) ◁154b 159c▷
```
case 't':
    tflag = true;
    break;
```
Uses `tflag` 156a.

⟨`dorecipe()` *when no recipe found, if tflag* 156c⟩≡ (140h)
```
if(tflag){
    if(!(node->flags&VIRTUAL))
        touch(node->name);
    else if(explain)
        Bprint(&bout, "no touch of virtual '%s'\n", node->name);
}
```
Uses `VIRTUAL` 140e, `bout` 45b, `explain` 124a, `tflag` 156a, and `touch()` 156e.

⟨`sched()` *if touch mode* 156d⟩≡ (124h)
```
if(tflag){
    if(!(n->flags&VIRTUAL))
        touch(n->name);
    else if(explain)
        Bprint(&bout, "no touch of virtual '%s'\n", n->name);
}
```
Uses `VIRTUAL` 140e, `bout` 45b, `explain` 124a, `tflag` 156a, and `touch()` 156e.

⟨*function* `touch` 156e⟩≡ (189c)
```
void
touch(char *name)
{
    Bprint(&bout, "touch(%s)\n", name);
    if(nflag)
        return;

    if(utfrune(name, '('))
        atouch(name);  /* archive */
    else
     if(chgtime(name) < 0) {
        perror(name);
        Exit();
    }
}
```
Uses `atouch()` 151, `bout` 45b, and `nflag` 124e.

⟨*function* `chgtime` 156f⟩≡ (199)
```
int
chgtime(char *name)
{
    Dir sbuf;
```

```
    if(access(name, AEXIST) >= 0) {
        nulldir(&sbuf);
        sbuf.mtime = time((long *)nil);
        return dirwstat(name, &sbuf);
    }
    return close(create(name, OWRITE, 0666));
}
```

11.10.3 Time cache

Evaluating the graph means asking "how old is this file?" over and over, often for the same file. The time cache makes that cheap: the first `timeof()`[82d] for a path consults the filesystem once and stores the answer in the `S_TIME`[157a] symbol table, and every later query for the same name returns the cached value. The `force` flag bypasses the cache when `mk` must see a fresh time — notably right after a recipe has regenerated the file.

⟨Sxxx *cases* 157a⟩+≡ (29a) ◁152b 158d▷

```
S_TIME,  /* file -> time */
```

⟨timeof() *locals* 157b⟩≡ (82d) 157d▷

```
ulong t;
```

⟨timeof() *if not force, use time cache* 157c⟩≡ (82d)

```
⟨timeof() check time cache 157e⟩
t = mkmtime(name, false);
⟨timeof() update time cache 157f⟩
return t;
```

⟨timeof() *locals* 157d⟩+≡ (82d) ◁157b

```
Symtab *sym;
```

⟨timeof() *check time cache* 157e⟩≡ (157c)

```
sym = symlook(name, S_TIME, nil);
if (sym)
    return sym->u.value;  /* uggh */
```

Uses S_TIME 157a and symlook() 30a.

⟨timeof() *update time cache* 157f⟩≡ (157c)

```
if(t == 0)
    return 0;
symlook(name, S_TIME, (void*)t);  /* install time in cache */
```

Uses S_TIME 157a and symlook() 30a.

11.10.4 Bulk time optimisation

The time cache above saves repeated lookups of the same file; the bulk optimisation goes further by prefilling it. Rather than stat-ing files one at a time, `mk` reads a whole directory in a single pass and installs an `S_TIME`[157a] entry for every name it finds, turning N system calls into one on directories where `mk` ends up needing most of the entries anyway.

⟨mkmtime *locals* 157g⟩≡ (83a) 158b▷

```
//char *s, *ss;
//char carry;
//Symtab *sym;
```

⟨mkmtime() *bulk dir optimisation* 158a⟩≡ (83a)

```
  ⟨mkmtime() cleanup name 158c⟩
  USED(force);
  //TODO    s = utfrrune(name, '/');
  //TODO    if(s == name)
  //TODO        s++;
  //TODO    if(s){
  //TODO        ss = name;
  //TODO        carry = *s;
  //TODO        *s = '\0';
  //TODO    }else{
  //TODO        ss = nil;
  //TODO        carry = '\0';
  //TODO    }
  //TODO    if(carry)
  //TODO        *s = carry;
  //TODO
  //TODO bulkmtime(ss);
  //TODO if(!force){
  //TODO     sym = symlook(name, S_TIME, 0);
  //TODO     if(sym)
  //TODO         return sym->u.value;
  //TODO     return 0;
  //TODO }
```

⟨mkmtime *locals* 158b⟩+≡ (83a) ◁157g

```
  char buf[4096];
```

⟨mkmtime() *cleanup name* 158c⟩≡ (158a)

```
  strecpy(buf, buf + sizeof buf - 1, name);
  cleanname(buf);
  name = buf;
```

⟨Sxxx *cases* 158d⟩+≡ (29a) ◁157a

```
  S_BULKED, /* we have bulked this dir */
```

⟨*function* bulkmtime 158e⟩≡ (199)

```
  void
  bulkmtime(char *dir)
  {
      char buf[4096];
      char *ss, *s, *sym;

      if(dir){
          sym = dir;
          s = dir;
          if(strcmp(dir, "/") == 0)
              strecpy(buf, buf + sizeof buf - 1, dir);
          else
              snprint(buf, sizeof buf, "%s/", dir);
      }else{
          s = ".";
          sym = "";
          buf[0] = 0;
      }
      if(symlook(sym, S_BULKED, 0))
          return;
      // else
      ss = strdup(sym);
      symlook(ss, S_BULKED, (void*)ss);
```

```
        dirtime(s, buf);
    }
```
Uses `S_BULKED` 158d, `dirtime()` 159a, and `symlook()` 30a.

⟨*function* `dirtime` 159a⟩≡ (199)
```
    void
    dirtime(char *dir, char *path)
    {
        int i, fd, n;
        ulong mtime;
        Dir *d;
        char buf[4096];

        fd = open(dir, OREAD);
        if(fd >= 0){
            while((n = dirread(fd, &d)) > 0){
                for(i=0; i<n; i++){
                    mtime = d[i].mtime;
                    /* defensive driving: this does happen */
                    if(mtime == 0)
                        mtime = 1;
                    snprint(buf, sizeof buf, "%s%s", path,
                        d[i].name);
                    if(symlook(buf, S_TIME, 0) == nil)
                        symlook(strdup(buf), S_TIME,
                            (void*)mtime)->u.value = mtime;
                }
                free(d);
            }
            close(fd);
        }
    }
```
Uses `S_TIME` 157a and `symlook()` 30a.

11.11 Recompiling everything: `mk -a`

`mk -a` forces a rebuild of everything regardless of timestamps, by making every node test as out of date (`weoutofdate`). It also turns on `-i` so that missing intermediates are rebuilt too rather than pretended present — a clean "from scratch" build without the bluntness of `rm`-ing the tree by hand.

⟨*global* `aflag` 159b⟩≡ (184)
```
    bool aflag = false;
```
Uses `aflag` 159b.

⟨`main()` *-xxx switch cases* 159c⟩+≡ (46b) ◁156b 161b▷
```
    case 'a':
        aflag = true;
        iflag = true;
        break;
```
Uses `aflag` 159b and `iflag` 154a.

⟨`work()` *adjust weoutofdate if aflag* 159d⟩≡ (96b)
```
    if(aflag)
        weoutofdate = true;
```
Uses `aflag` 159b.

11.12 Recursive `mk`: `$MKFLAGS` and `$MKARGS`

Recursive `mk` is the pattern of one `mkfile`'s recipe calling `mk` in a subdirectory—common in Plan 9, where the top-level `/sys/src/mkfile` descends into `cmd/`, `lib/`, and so on. The challenge is that the inner `mk` should inherit the original command-line flags and variable assignments (not the targets, since those were already consumed by the outer `mk`). Consider cross-compiling with `mk objtype=arm`: that `objtype=arm` assignment must reach every subdirectory build, or the inner `mk`s would quietly compile for the host architecture instead. `$MKFLAGS` and `$MKARGS` solve this: `main()`[45a] captures both before nulling out the `var=value` entries in `argv`, and exports them through the symbol table. A child `mkfile` can then recursively invoke `mk $MKFLAGS $MKARGS target` and the inner build inherits the same `objtype=arm`, `-k`, or `-n` settings the user originally typed. Whether to build a large tree this way at all is a long-running debate. Peter Miller's "Recursive Make Considered Harmful" [Mil97] argues that per-directory invocations leave each `mk` with only a partial dependency graph, inviting both redundant work and missed rebuilds, and advocates a single monolithic makefile instead; the later "Non-recursive Make Considered Harmful" [MMJM16] retorts that the monolithic approach does not scale either, and reframes the problem around build systems as reusable libraries. Plan 9 stays with recursion, trading a globally-perfect graph for the simplicity of one self-contained `mkfile` per directory.

⟨`main()` *locals* 160a⟩+≡ (45a) ◁149f 165d▷
```
  Bufblock *buf = newbuf();
```
Uses `newbuf()` 172d.

⟨`main()` *add* `argv[0]` *in* `buf` 160b⟩≡ (48d 46b)
```
  bufcpy(buf, argv[0], strlen(argv[0]));
  insert(buf, ' ');
```
Uses `bufcpy()` 173c and `insert()` 173a.

⟨`main()` *add* `argv[i]` *in* `buf` 160c⟩≡ (47b)
```
  bufcpy(buf, argv[i], strlen(argv[i]));
  insert(buf, ' ');
```
Uses `bufcpy()` 173c and `insert()` 173a.

⟨`main()` *set variables for recursive mk* 160d⟩≡ (46a)
```
  ⟨main() set MKFLAGS variable 160e⟩
  ⟨main() set MKARGS variable 160f⟩
```

⟨`main()` *set MKFLAGS variable* 160e⟩≡ (160d)
```
  if (buf->current != buf->start) {
      buf->current--;
      insert(buf, '\0');
  }
  symlook("MKFLAGS", S_VAR, (void*) stow(buf->start));
```
Uses `S_VAR` 29a, `insert()` 173a, `stow()` 60b, and `symlook()` 30a.

⟨`main()` *set MKARGS variable* 160f⟩≡ (160d)
```
  buf->current = buf->start;
  for(i = 0; argv[i]; i++){
      if(*argv[i] == '\0')
          continue;
      if(i)
          insert(buf, ' ');
      bufcpy(buf, argv[i], strlen(argv[i]));
  }
  insert(buf, '\0');
  symlook("MKARGS", S_VAR, (void *) stow(buf->start));

  freebuf(buf);
```
Uses `S_VAR` 29a, `bufcpy()` 173c, `freebuf()` 172e, `insert()` 173a, `stow()` 60b, and `symlook()` 30a.

11.13 Keep-going mode: `mk -k`

By default `mk` stops at the first failed recipe. `mk -k` ("keep going") instead presses on, building every target it still can and only skipping those downstream of the failure — the same idea as `make -k`. It is handy for a full-tree build where you want to see all the breakage in one run rather than fixing and restarting repeatedly. The `runerrs` counter records how many recipes failed, so `mk` can still exit nonzero at the end.

11.13.1 `kflag` and `runerrs`

⟨*global* `kflag` 161a⟩≡ (184)

```
bool kflag = false;
```

Uses `kflag` 161a.

⟨`main()` *-xxx switch cases* 161b⟩+≡ (46b) ◁159c 165e▷

```
case 'k':
    kflag = true;
    break;
```

Uses `kflag` 161a.

⟨*global* `runerrs` 161c⟩≡ (191b)

```
int runerrs;
```

⟨`mk()` *initializations* 161d⟩+≡ (92) ◁148g

```
runerrs = 0;
```

Uses `runerrs` 161c.

11.13.2 Adjusting `mk()`, `work()`, and `waitup()`

⟨`work()` *when inexistent target without prerequisites, if kflag* 161e⟩≡ (95d)

```
if(kflag){
    node->flags |= BEINGMADE;
    runerrs++;
}
```

Uses `BEINGMADE` 40d, `kflag` 161a, and `runerrs` 161c.

⟨`waitup()` *when error in child process, if kflag* 161f⟩≡ (110b)

```
if(kflag){
    runerrs++;
    fake = true;
}
```

Uses `kflag` 161a and `runerrs` 161c.

⟨`update()` *if fake* 161g⟩≡ (108b)

```
if(fake)
    MADESET(node, BEINGMADE);
```

Uses `BEINGMADE` 40d and `MADESET` 94b.

⟨`mk()` *if no child to waitup and root not MADE, possibly break* 161h⟩≡ (92)

```
if(res > 0){
    if(root->flags&(NOTMADE|BEINGMADE)){
        assert(/*must be run errors*/ runerrs);
        break; /* nothing more waiting */
    }
}
```

Uses `BEINGMADE` 40d, `NOTMADE` 40d, and `runerrs` 161c.

⟨`WaitupParam` *other cases* 162a⟩+≡ (109b) ◁134b

```
  EMPTY_CHILDREN_IS_ERROR2 = -2,
```

⟨`mk()` *before returning, more* `waitup()` *if there was an error* 162b⟩≡ (92)

```
  while(jobs)
      waitup(EMPTY_CHILDREN_IS_ERROR2, (int *)nil);
  assert(/*target didnt get done*/ runerrs || (root->flags&MADE));
```

Uses `EMPTY_CHILDREN_IS_ERROR2` 162a, `MADE` 40d, `jobs` 42c, `runerrs` 161c, and `waitup()` 107c.

Chapter 12

Conclusion

You now know how the Plan 9 build system `mk` works, to the smallest details, and more generally how many build systems work. You have followed a command like `mk all` from the parsing of the `mkfile`, through the construction of the dependency graph, the depth-first walk that compares modification times, and finally the scheduling of shell jobs to rebuild what is outdated.

12.1 Patterns and techniques

These techniques apply far beyond build systems:

- *Dependency DAG*: the same structure drives spreadsheet recalculation (cells depending on other cells), package managers (libraries depending on other libraries), CI/CD pipelines (stages depending on earlier stages), and reactive UI frameworks (computed values depending on observable state). Any time outputs depend on inputs, a DAG is the natural model.

- *Three-state traversal*: the `NOTMADE` / `BEINGMADE` / `MADE` state machine enables wavefront parallelism—nodes whose prerequisites are all `MADE` can execute concurrently. The same technique appears in garbage collectors (white/grey/black marking) and topological sort algorithms wherever work items have dependencies.

- *Declarative specification, imperative execution*: the user writes *what* depends on *what*; `mk` figures out *how* and *when*. This separation is the core idea behind SQL, Terraform, and constraint solvers: declare the goal, let the engine plan the execution.

12.2 Connections to other books

Because `mk` relies heavily on the shell `rc` to execute recipes, the SHELL book [Pad18] is the next logical step after this book. Moreover, many features of `mk` are inspired by features from the shell; the ability to call programs easily, file redirection (`<`), pipes (`<|`), and the use of variables (`$xxx`) are all inherited from `rc`.

Here are a few other connections:

- The KERNEL book [Pad14] explains the system calls behind file modification times (`stat()`), process management (`rfork()`, `exec()`, `wait()`), and file operations—all of which `mk` relies on to decide what to rebuild and how.

- The LIBCORE book [Pad16c] describes the many core libraries used by `mk`: Buffered I/O with `libbio`, regular expressions with `libregexp`, or Unicode and UTF-8 with `libc`.

- The COMPILER book [Pad16b] and ASSEMBLER book [Pad15a] describe the tools that `mk` orchestrates. A typical `mkfile` describes how to turn `.c` files into `.o` files (via the compiler) and `.o` files into executables (via the linker).

12.3 Beyond `mk`

Build systems have evolved considerably since `mk`. Here are some of the features found in modern tools:

- *Dependency management:* Tools like Cargo, Maven, or Yarn fused the build system with the package manager to automatically fetch and compile external libraries (and their transitive dependencies). This is one of the biggest usability improvements in modern development—specifying a library name and having the entire dependency tree resolved, downloaded, and compiled is transformative for developer productivity.
- *Content-based rebuilds:* `mk` uses file modification times to decide what is out of date. Build systems like Bazel and `redo` use content fingerprints (hashes) instead, which is more reliable: renaming a file, copying it, or checking it out from version control does not trigger spurious rebuilds.
- *Distributed and cached builds:* Bazel, Buck, and similar tools can distribute compilation across many machines and cache build artifacts remotely, so that a rebuild can reuse results from a previous build on a different machine. For large codebases (millions of lines), this reduces build times from hours to minutes.
- *Hermetic builds:* Bazel aims for fully reproducible builds by sandboxing each action: tools and inputs are declared explicitly, and the build system ensures that no undeclared dependencies leak in. This contrasts with `mk` (and GNU `make`), where recipes can access anything on the filesystem.

Despite these additions, the fundamental algorithm remains the same: build a dependency graph, find what is out of date, and rebuild in parallel. `mk` implements this core cleanly in about 4500 lines of C—a good foundation for understanding what all the larger systems are doing underneath.

Appendix A

Debugging

`mk` supports three debugging flags (`mk -dxxx`) that dump internal state at different stages of the pipeline: `D_PARSE` traces the parser, `D_GRAPH` dumps the dependency graph, and `D_EXEC` traces job execution.

⟨*enum* Dxxx 165a⟩≡ (182d)

```
enum Dxxx {
    // for rules
    D_PARSE =  0x01,
    // for node and arcs
    D_GRAPH =  0x02,
    // for jobs
    D_EXEC  =  0x04,

    // tracing some calls
    D_TRACE  =  0x08,
};
```

⟨*global* debug 165b⟩≡ (184)

```
// bitset<enum<dxxx>>
int debug;
```

⟨*function* DEBUG 165c⟩≡ (182d)

```
#define DEBUG(x) (debug&(x))
```

⟨main() *locals* 165d⟩+≡ (45a) ◁160a

```
char *s;
```

⟨main() *-xxx switch cases* 165e⟩+≡ (46b) ◁161b

```
case 'd':
    if(*(s = &argv[0][2]))
        while(*s)
         switch(*s++) {
         case 'p': debug |= D_PARSE; break;
         case 'g': debug |= D_GRAPH; break;
         case 'e': debug |= D_EXEC; break;
        }
    else
        debug = 0xFFFF; // D_PARSE | D_GRAPH | D_EXEC
    break;
```

Uses D_EXEC 165a, D_GRAPH 165a, D_PARSE 165a, and debug 165b.

A.1 Dumping the rules: mk -dp

⟨main() *if* DEBUG(D_PARSE) 166a⟩≡ (48e)
```
if(DEBUG(D_PARSE)){
     dumpw("default targets", target1);
     dumpr("rules", rules);
     dumpr("metarules", metarules);
     dumpv("variables");
}
```
Uses DEBUG 165c, D_PARSE 165a, dumpr() 166c, dumpv() 166d, dumpw() 166b, metarules 35e, rules 35b, and target1 49b.

⟨*dumper dumpw* 166b⟩≡ (193a)
```
void
dumpw(char *s, Word *w)
{
     Bprint(&bout, "%s", s);
     for(; w; w = w->next)
          Bprint(&bout, " '%s'", w->s);
     Bputc(&bout, '\n');
}
```
Uses bout 45b.

⟨*dumper dumpr* 166c⟩≡ (193a)
```
void
dumpr(char *s, Rule *r)
{
     Bprint(&bout, "%s: start=%p\n", s, r);
     for(; r; r = r->next){
          Bprint(&bout, "\tRule %p: %s:%d attr=%x next=%p chain=%p alltarget='%s'",
               r, r->file, r->line, r->attr, r->next, r->chain, wtos(r->alltargets, ' '));
          if(r->prog)
               Bprint(&bout, " prog='%s'", r->prog);
          Bprint(&bout, "\n\ttarget=%s: %s\n", r->target, wtos(r->prereqs,' '));
          Bprint(&bout, "\trecipe@%p='%s'\n", r->recipe, r->recipe);
     }
}
```
Uses bout 45b and wtos() 33d.

⟨*dumper dumpv* 166d⟩≡ (193a)
```
void
dumpv(char *s)
{
     Bprint(&bout, "%s:\n", s);
     symtraverse(S_VAR, &print1);
}
```
Uses S_VAR 29a, bout 45b, print1() 166e, and symtraverse() 31a.

⟨*function* print1 166e⟩≡ (193a)
```
static void
print1(Symtab *s)
{
     Word *w;

     Bprint(&bout, "\t%s=", s->name);
     for (w = s->u.ptr; w; w = w->next)
          Bprint(&bout, "'%s'", w->s);
     Bprint(&bout, "\n");
}
```
Uses bout 45b.

A.2 Dumping the graph: mk -dg

⟨mk() *if* DEBUG(D_GRAPH) 167a⟩≡ (92)

```
if(DEBUG(D_GRAPH)){
	dumpn("new target\n", root);
	Bflush(&bout);
}
```

Uses DEBUG 165c, D_GRAPH 165a, bout 45b, and dumpn() 167b.

⟨*dumper dumpn* 167b⟩≡ (193a)

```
void
dumpn(char *s, Node *n)
{
	char buf[1024];
	Arc *a;

	Bprint(&bout, "%s%s@%p: time=%ld flags=0x%x next=%p\n",
		s, n->name, n, n->time, n->flags, n->next);
	for(a = n->arcs; a; a = a->next){
		snprint(buf, sizeof buf, "%s   ", (*s == ' ')? s:"");
		dumpa(buf, a);
	}
}
```

Uses bout 45b and dumpa() 167c.

⟨*dumper dumpa* 167c⟩≡ (193a)

```
void
dumpa(char *s, Arc *a)
{
	char buf[1024];

	Bprint(&bout, "%sArc@%p: n=%p r=%p flag=0x%x stem='%s'",
		s, a, a->n, a->r, a->remove, a->stem);
	if(a->prog)
		Bprint(&bout, " prog='%s'", a->prog);
	Bprint(&bout, "\n");

	if(a->n){
		snprint(buf, sizeof(buf), "%s    ", (*s == ' ')? s:"");
		dumpn(buf, a->n);
	}
}
```

Uses bout 45b and dumpn() 167b.

⟨nrep() *if* DEBUG(D_GRAPH) 167d⟩≡ (148h)

```
if(DEBUG(D_GRAPH))
	Bprint(&bout, "nreps = %d\n", nreps);
```

A.3 Tracing jobs: mk -de

⟨sched() *if* DEBUG(D_EXEC) 167e⟩≡ (103e)

```
if(DEBUG(D_EXEC))
	fprint(STDOUT, "firing up job for target %s\n", wtos(j->t, ' '));
```

Uses DEBUG 165c, D_EXEC 165a, and wtos() 33d.

⟨sched() *if* DEBUG(D_EXEC) *print recipe* 168a⟩≡ (103e)
```
  if(DEBUG(D_EXEC))
      fprint(STDOUT, "recipe='%s'\n", j->r->recipe);
  Bflush(&bout);
```
Uses DEBUG 165c, D_EXEC 165a, and bout 45b.

⟨sched() *if* DEBUG(D_EXEC) *print pid* 168b⟩≡ (103e)
```
  if(DEBUG(D_EXEC))
      fprint(STDOUT, "pid for target %s = %d\n", wtos(j->t, ' '), events[slot].pid);
```
Uses DEBUG 165c, D_EXEC 165a, events-14 102d, and wtos() 33d.

⟨waitup() *if* DEBUG(D_EXEC) *print pid* 168c⟩≡ (107c)
```
  if(DEBUG(D_EXEC))
      fprint(STDOUT, "waitup got pid=%d, status='%s'\n", pid, buf);
```
Uses DEBUG 165c and D_EXEC 165a.

⟨waitup() *if* DEBUG(D_EXEC) *and slot ¡ 0* 168d⟩≡ (145d)
```
   if(DEBUG(D_EXEC))
       fprint(STDERR, "mk: wait returned unexpected process %d\n", pid);
```
Uses DEBUG 165c and D_EXEC 165a.

⟨pidslot() *if* DEBUG(D_EXEC) 168e⟩≡ (103d)
```
  if(DEBUG(D_EXEC))
      fprint(STDERR, "mk: wait returned unexpected process %d\n", pid);
```
Uses DEBUG 165c and D_EXEC 165a.

⟨nproc() *if* DEBUG(D_EXEC) 168f⟩≡ (102b)
```
  if(DEBUG(D_EXEC))
      fprint(STDERR, "nprocs = %d\n", nproclimit);
```
Uses DEBUG 165c, D_EXEC 165a, and nproclimit-17 101c.

⟨*dumper dumpj* 168g⟩≡ (193a)
```
  void
  dumpj(char *s, Job *j, int all)
  {
      Bprint(&bout, "%s\n", s);
      while(j){
          Bprint(&bout, "job@%p: r=%p n=%p stem='%s'\n",
              j, j->r, j->n, j->stem);
          Bprint(&bout, "\ttarget='%s' alltarget='%s' prereq='%s' nprereq='%s'\n",
              wtos(j->t, ' '), wtos(j->at, ' '), wtos(j->p, ' '), wtos(j->np, ' '));
          j = all? j->next : nil;
      }
  }
```
Uses bout 45b and wtos() 33d.

A.4 Tracing function calls: mk -dt

⟨applyrules *debug* 168h⟩≡ (77a)
```
  if(DEBUG(D_TRACE))
      print("applyrules(%lux='%s')\n", target, target);
```
Uses DEBUG 165c and D_TRACE 165a.

⟨newnode() *debug* 168i⟩≡ (40a)
```
  if(DEBUG(D_TRACE))
      print("newnode(%s), time = %d\n", name, node->time);
```
Uses DEBUG 165c and D_TRACE 165a.

⟨`work()` *debug* 169a⟩≡ (95a)
```
  if(DEBUG(D_TRACE))
      print("work(%s) flags=0x%x time=%lud\n", node->name, node->flags, node->time);
```
Uses `DEBUG` 165c and `D_TRACE` 165a.

⟨`update()` *debug* 169b⟩≡ (108b)
```
  if(DEBUG(D_TRACE))
      print("update(): node %s time=%lud flags=0x%x\n", node->name, node->time, node->flags);
```
Uses `DEBUG` 165c and `D_TRACE` 165a.

Appendix B

Profiling

mk includes optional profiling support, compiled in when the PROF preprocessor symbol is defined. This simply enables prof(1)-style instrumentation on mk itself (see the PROFILER book [Pad26]), which is useful for profiling the build system rather than the programs being built.

⟨*global* buf 170a⟩≡ (193b)
```
short buf[10000];
```

⟨main() *setup profiling* 170b⟩+≡ (46a) ◁126e
```
#ifdef PROF
    {
        extern int etext();
        monitor(main, etext, buf, sizeof buf, 300);
    }
#endif
```

⟨*function* symstat 170c⟩≡ (185c)
```
void
symstat(void)
{
    Symtab **s, *ss;
    int n;
    int l[1000];

    memset((char *)l, 0, sizeof(l));
    for(s = hash; s < &hash[NHASH]; s++){
        for(ss = *s, n = 0; ss; ss = ss->next)
            n++;
        l[n]++;
    }
    for(n = 0; n < 1000; n++)
        if(l[n])
            Bprint(&bout, "%d of length %d\n", l[n], n);
}
```
Uses NHASH-1 29c, bout 45b, and hash-3 29b.

Appendix C

Utilities

This appendix collects the utility code used throughout `mk` such as memory allocation wrappers or string buffers (`Bufblock`) word-list manipulation.

C.1 Memory management

⟨*function* `Malloc` 171a⟩≡ (185a)

```
void*
Malloc(int n)
{
    void *s;

    s = malloc(n);
    if(!s) {
        fprint(STDERR, "mk: cannot alloc %d bytes\n", n);
        Exit();
    }
    return s;
}
```

Note the use of `Exit()`[193c] for proper error management in the context of `mk` to wait for children before exiting (See Section 8.5).

⟨*function* `Realloc` 171b⟩≡ (185a)

```
void *
Realloc(void *s, int n)
{
    if(s)
        s = realloc(s, n);
    else
        s = malloc(n);
    if(!s) {
        fprint(STDERR, "mk: cannot alloc %d bytes\n", n);
        Exit();
    }
    return s;
}
```

C.2 Buffer management

⟨*struct* `Bufblock` 171c⟩≡ (182d)

```
struct Bufblock
{
```

```
    char   *start;
    char   *end;

    // between start and end
    char   *current;

    // Extra
    ⟨Bufblock extra fields 172a⟩
};
```

⟨Bufblock *extra fields* 172a⟩≡ (171c)

```
struct Bufblock *next;
```

Uses Bufblock 171c.

⟨*global* freelist 172b⟩≡ (185b)

```
static Bufblock *freelist;
```

⟨*constant* QUANTA 172c⟩≡ (185b)

```
#define QUANTA 4096
```

⟨*constructor* newbuf 172d⟩≡ (185b)

```
Bufblock *
newbuf(void)
{
    Bufblock *p;

    if (freelist) {
        p = freelist;
        freelist = freelist->next;
    } else {
        p = (Bufblock *) Malloc(sizeof(Bufblock));
        p->start = Malloc(QUANTA*sizeof(char));
        p->end = p->start+QUANTA;
    }
    p->current = p->start;
    *p->start = '\0';
    p->next = nil;

    return p;
}
```

Uses Malloc() 171a, QUANTA-12 172c, and freelist-11 172b.

⟨*destructor* freebuf 172e⟩≡ (185b)

```
void
freebuf(Bufblock *p)
{
    p->next = freelist;
    freelist = p;
}
```

Uses freelist-11 172b.

⟨*macro* isempty 172f⟩≡ (180)

```
#define isempty(buf) (buf->current == buf->start)
```

⟨*macro* resetbuf 172g⟩≡ (180)

```
#define resetbuf(buf) do { buf->current = buf->start; } while(0)
```

⟨*macro* bufcontent 172h⟩≡ (180)

```
#define bufcontent(buf) buf->start
```

⟨*function* insert 173a⟩≡ (185b)
```
void
insert(Bufblock *buf, int c)
{

    if (buf->current >= buf->end)
        growbuf(buf);
    *buf->current++ = c;
}
```
Uses growbuf() 173d.

⟨*function* rinsert 173b⟩≡ (185b)
```
void
rinsert(Bufblock *buf, Rune r)
{
    int n;

    n = runelen(r);
    if (buf->current+n > buf->end)
        growbuf(buf);
    runetochar(buf->current, &r);
    buf->current += n;
}
```
Uses growbuf() 173d.

⟨*function* bufcpy 173c⟩≡ (185b)
```
void
bufcpy(Bufblock *buf, char *cp, int n)
{

    while (n--)
        insert(buf, *cp++);
}
```
Uses insert() 173a.

⟨*function* growbuf 173d⟩≡ (185b)
```
void
growbuf(Bufblock *p)
{
    int n;
    Bufblock *f;
    char *cp;

    n = p->end-p->start+QUANTA;
        /* search the free list for a big buffer */
    for (f = freelist; f; f = f->next) {
        if (f->end-f->start >= n) {
            memcpy(f->start, p->start, p->end-p->start);
            cp = f->start;
            f->start = p->start;
            p->start = cp;
            cp = f->end;
            f->end = p->end;
            p->end = cp;
            f->current = f->start;
            break;
        }
    }
    if (!f) {  /* not found - grow it */
```

```
        p->start = Realloc(p->start, n);
        p->end = p->start+n;
    }
    p->current = p->start+n-QUANTA;
}
```

Uses `QUANTA-12` 172c, `Realloc()` 171b, and `freelist-11` 172b.

C.3 File management

⟨*function* `maketmp` 174⟩≡ (199)

```
char*
maketmp(void)
{
    static char temp[] = "/tmp/mkargXXXXXX";

    mktemp(temp);
    return temp;
}
```

Appendix D

Examples of `mkfiles`

Having seen every feature in isolation, this chapter shows them working together in real `mkfiles`. The first is `mk`'s own `mkfile` — the build system describing how to build itself. The rest are the small library of shared `mkfile` fragments that the whole of Principia Softwarica (and Plan 9 before it) is built from: per-architecture configuration like `$objtype/mkfile`, and reusable templates such as `mkone` (build one command), `mklib` (build a library), and `mkdirs` (recurse into subdirectories). A concrete `mkfile` then reduces to a few variable definitions — `TARG`, `OFILES`, `HFILES` — followed by a single `<` include that pulls in one of these templates, which is where the inclusion and recursive-`mk` machinery of the previous chapter finally pays off.

D.1 The `mkfile` of `mk`

`mk`'s own `mkfile` is a good first example because it is almost all declaration and almost no rules. It sets `TOP` to the source root, and then its real content is sandwiched between two includes: at the top `<$TOP/mkfiles/$objtype/mkfil` pulls in the compiler and linker for the current architecture, and at the bottom `<$TOP/mkfiles/mkone` pulls in the build rules for a single program. In between, the `mkfile` states only what is specific to `mk`: the target name (`TARG=mk`), the object files (`OFILES`), and the headers (`HFILES`). Both included fragments are defined in the next section — here we see only the client side of the inclusion mechanism. (Note `Plan9.$O` in the object list, and the absence of `Posix.$O`: this build links the Plan 9 host backend.)

⟨*mk/mkfile* 175⟩≡

```
TOP=../..
<$TOP/mkfiles/$objtype/mkfile

TARG=mk

OFILES=\
    globals.$O\
    utils.$O\
    dumpers.$O\
    archive.$O\
    bufblock.$O\
    env.$O\
    file.$O\
    graph.$O\
    lex.$O\
    main.$O\
    match.$O\
    mk.$O\
    parse.$O\
    rc.$O\
    recipe.$O\
    rule.$O\
```

```
    run.$O\
    shprint.$O\
    symtab.$O\
    var.$O\
    varsub.$O\
    word.$O\
    Plan9.$O\

# no Posix.$O here

HFILES=fns.h mk.h

<$TOP/mkfiles/mkone
```

D.2 The mkfiles of Principia Softwarica

D.2.1 $objtype/mkfile for the ARM

Each supported architecture has a one-screen `mkfile` that includes the shared prototype and then just names its toolchain; for the ARM that is the kencc `5`-series tools (`5c` compiler, `5a` assembler, `5l` linker, with `$O` set to `5`), following Plan 9's one-character-per-architecture naming convention.

⟨*mkfiles/arm/mkfile* 176a⟩≡
```
<$TOP/mkfiles/mkfile.proto

CC=5c
LD=5l
O=5
AS=5a
```

D.2.2 mkfiles/mkfile.proto

`mkfile.proto` holds everything the per-architecture mkfiles share, so it is the first thing each of them includes. It fixes the include and library paths (parameterized by `$objtype` so one set of definitions serves every CPU), the default `lex` and `yacc`, and the list of `CPUS` to build for. Its last act is to clear `TARG`, `OFILES`, `HFILES`, and `YFILES`: under recursive `mk` these arrive preset from the parent build, which is never what a child directory wants, so the prototype resets them to empty before the leaf `mkfile` fills them in.

⟨*mkfiles/mkfile.proto* 176b⟩≡
```
ROOT=$TOP/ROOT

#
# common mkfile parameters shared by all architectures
#

OS=58
CPUS=386 arm

CFLAGS=-FTVw -I$TOP/include/arch/$objtype -I$TOP/include/ALL
LDFLAGS=-L$ROOT/arch/$objtype/lib

LEX=lex
YACC=iyacc

# recursive mk will have these set from the parent
# this is never what we want.  clear them
```

```
#pad: if you remove those settings, you will get an error
# in windows/plumb: "don't know how to make '/386/include/u.h^A...."

TARG=
OFILES=
HFILES=
YFILES=
```

D.2.3 mkfiles/mkone

`mkone` is the template for a directory that builds a single program: it supplies the `%.$O` compile rules and the `all`, `install`, `clean`, and `nuke` targets, so a leaf `mkfile` only has to declare its `TARG` and `OFILES` and include it.

⟨*mkfiles/mkone* 177⟩≡

```
# -*- sh -*-
YFLAGS=-d

BIN=$ROOT/arch/$objtype/bin
all:V:  $O.out


$O.out: $OFILES $LIB
        $LD $LDFLAGS -o $target $prereq


%.$O:   $HFILES         # don't combine with following %.$O rules

%.$O:   %.c
        $CC $CFLAGS $stem.c

%.$O:   %.s
        $AS $AFLAGS $stem.s


y.tab.h y.tab.c:        $YFILES
        $YACC $YFLAGS $prereq

lex.yy.c:       $LFILES
        $LEX $LFLAGS $prereq


install:V:      $BIN/$TARG

$BIN/$TARG:     $O.out
        cp $prereq $target

installall:V:
        for(objtype in $CPUS)
                mk install

uninstall:V:
        rm -f $BIN/$TARG

allall:V:
        for(objtype in $CPUS)
                mk all

clean:V:
        rm -f *.[$OS] [$OS].out y.tab.? lex.yy.c y.debug y.output $TARG $CLEANFILES
```

```
nuke:V:
	rm -f *.[$OS] [$OS].out y.tab.? lex.yy.c y.debug y.output *.acid $TARG $CLEANFILES



safeinstall:V: $O.out
	test -e $BIN/$TARG && mv $BIN/$TARG $BIN/_$TARG
	cp $prereq $BIN/$TARG

safeinstallall:V:
	for (objtype in $CPUS)
		mk safeinstall


%.acid: %.$O $HFILES
	$CC $CFLAGS -a $stem.c >$target

#pad: generate warnings with mk-rc.byte
#%.man: $MAN/$stem
#	cp $stem.man $MAN/$stem

man:V:	$TARG.man
```

D.2.4 mkfiles/mklib

`mklib` is the same idea for a directory that builds a library: instead of linking an executable, its top target is `$LIB`, assembled from the object files with `iar`, while the per-file compile rules and housekeeping targets stay essentially those of `mkone`.

⟨*mkfiles/mklib* 178⟩≡

```
YFLAGS=-d

all:V:	$LIB

$LIB:	$OFILES
	iar vu $LIB $prereq


%.$O:	$HFILES		# don't combine with following %.$O rules

%.$O:	%.c
	$CC $CFLAGS $CFLAGS_EXTRA $stem.c

%.$O:	%.s
	$AS $AFLAGS $stem.s


y.tab.h y.tab.c:	$YFILES
	$YACC $YFLAGS $prereq

lex.yy.c:	$LFILES
	$LEX $LFLAGS $prereq


install:VQ: $LIB

uninstall:VQ:
	echo nothing to uninstall
```

```
installall:V:
	for (objtype in $CPUS)
		mk install

%.all:V:
	for (objtype in $CPUS)
		mk $stem


clean:V:
	rm -f *.[$OS] y.tab.? y.output y.error $CLEANFILES

nuke:V:
	rm -f *.[$OS] y.tab.? y.output y.error $CLEANFILES $LIB
```

D.2.5 mkfiles/mkdirs

`mkdirs` is for a directory that has no source of its own, only subdirectories: every target simply loops over `$DIRS` and runs `mk` in each, using `rc`'s `@{...}` subshell per entry so the `cd` is undone automatically without an explicit pushd/popd.

⟨*mkfiles/mkdirs* 179⟩≡

```
# by using '@' below, rc will create a subprocess for each entry,
# which avoids the need for a pushd/popd in other shells.

all:VQ:
	for (i in $DIRS) @{
	    echo $i
		cd $i
		mk $MKFLAGS $target
	}

install uninstall clean nuke:QV:
	for (i in $DIRS) @{
	    echo $i
		cd $i
		mk $MKFLAGS $target
	}
```

Appendix E

Extra Code

This appendix collects the code chunks the preceding chapters referred to but did not show inline: header boilerplate, forward declarations, and the small uninteresting helpers whose definitions would only have interrupted the narrative. Nothing here is new — it is the remainder of `mk`'s source, included so that the literate program remains, as it must, the whole program and not just its interesting parts.

E.1 mk/

E.1.1 mk/fns.h

⟨mk/fns.h 180⟩≡

```
// Constructors/destructors for core data structures

// bufblock.c
Bufblock* newbuf(void);
void freebuf(Bufblock*);
void growbuf(Bufblock *);
void bufcpy(Bufblock *, char *, int);
void insert(Bufblock *, int);
void rinsert(Bufblock *, Rune);
⟨macro isempty 172f⟩
⟨macro resetbuf 172g⟩
⟨macro bufcontent 172h⟩

// words.c
Word* newword(char*);
void freewords(Word*);
Word* wdup(Word*);
char* wtos(Word*, int);
void addw(Word*, char*);

// symtab.c
Symtab* symlook(char*, int, void*);
void symtraverse(int, void(*)(Symtab*));
void symstat(void);

// var.c
void setvar(char*, void*);
char* shname(char*);

// rule.c
void addrule(char*, Word*, char*, Word*, int, int, char*);
```

```
void addrules(Word*, Word*, char*, int, int, char*);
char* rulecnt(void);



// env.c
void inithash(void);
void initenv(void);
ShellEnvVar* buildenv(Job*, int);
void exportenv(ShellEnvVar *e);


// lex.c
bool  assline(Biobuf *, Bufblock *);
int  nextrune(Biobuf*, bool);

// parse.c
void parse(char*, fdt, bool);

// varsub.c
Word* stow(char*);



// graph.c
Node* graph(char*);
void nrep(void);

// file.c
ulong timeof(char*, bool);
void timeinit(char*);
void touch(char*);
ulong mkmtime(char*, bool);
void delete(char*);

// match.c
bool  match(char*, char*, char*);
void subst(char*, char*, char*, int);



// mk.c
void mk(char*);
bool    outofdate(Node*, Arc*, bool);
void update(Node*, bool);

// recipe.c
void  dorecipe(Node*, bool*);

// run.c
void run(Job*);
int  waitup(int, int*);
void nproc(void);
//
void prusage(void);
void usage(void);
//
int  execsh(char*, char*, Bufblock*, ShellEnvVar*);
int  pipecmd(char*, ShellEnvVar*, int*);
void catchnotes(void);
```

```
void Exit(void);

// shprint.c
void shprint(char*, ShellEnvVar*, Bufblock*);
void front(char*);



// rc.c
char*  charin(char *, char *);
char* copyq(char*, Rune, Bufblock*);
error0 escapetoken(Biobuf*, Bufblock*, bool, int);
char* expandquote(char*, Rune, Bufblock*);


// archive.c
ulong atimeof(int,char*);
void atouch(char*);


// utils.c
void* Malloc(int);
void* Realloc(void*, int);
char* maketmp(void);

// Dumpers
void dumpv(char*);
void dumpw(char*, Word*);
void dumpr(char*, Rule*);
void dumpn(char*, Node*);
void dumpj(char*, Job*, int);
```

E.1.2 mk/mk.h

⟨*constant* BIGBLOCK 182a⟩≡ (182d)

```
#define BIGBLOCK 20000
```

⟨*function* RERR 182b⟩≡ (182d)

```
//#define RERR(r)  (fprint(STDERR, "mk: %s:%d: rule error; ", (r)->file, (r)->line))
```

⟨*function* SEP 182c⟩≡ (182d)

```
//#define SEP(c)  (((c)==' ')||((c)=='\t')||((c)=='\n'))
```

⟨mk/mk.h 182d⟩≡

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

extern Biobuf bout;

typedef struct Symtab Symtab;
typedef struct Word Word;
typedef struct Rule Rule;
typedef struct Node Node;
typedef struct Arc Arc;
typedef struct ShellEnvVar ShellEnvVar;
typedef struct Job Job;
```

```
typedef struct Bufblock Bufblock;
typedef struct Shell Shell;

⟨struct Bufblock 171c⟩

⟨struct Word 32b⟩

// used by main and parse.c
extern Word *target1;


⟨struct Envy 112a⟩

extern ShellEnvVar *shellenv;

⟨struct Rule 34⟩

extern Rule *rules, *metarules, *patrule;

⟨enum Rule_attr 36a⟩

⟨macro empty_recipe 78b⟩
⟨macro empty_prereqs 78c⟩
⟨macro empty_words 58b⟩

⟨constant NREGEXP 129c⟩

⟨struct Arc 40f⟩

⟨struct Node 39d⟩

⟨enum Node_flag 40c⟩
⟨function MADESET 94b⟩

⟨struct Job 41d⟩

extern Job *jobs;

⟨struct Symtab 28⟩

⟨enum Namespace 29a⟩

⟨type WaitupParam 109b⟩
⟨type WaitupResult 109c⟩

extern int debug;
extern bool nflag, tflag, iflag, kflag, aflag;
extern int mkinline;
extern char *infile;
extern bool explain;
extern int runerrs;

⟨function SYNERR 51d⟩
⟨function RERR 182b⟩
⟨constant NAMEBLOCK 79b⟩
⟨constant BIGBLOCK 182a⟩

⟨function SEP 182c⟩
⟨function WORDCHR 64e⟩
```

```
⟨enum Dxxx 165a⟩
⟨function DEBUG 165c⟩

⟨function PERCENT 35g⟩

//pad: Shell below allows to change the shell used by mk at runtime.
// Thus, mk of goken can be used to compile goken itself and fork-plan9,
// which have different requirements. I did that in xix/mk first
// and apparently 9-cc-colombier did something similar but only
// with MKSHELL defined in the mkfile itself (with some pushshell()/popshell())
⟨struct Shell 105b⟩
typedef struct Shell Shell;

//TODO: later
//    char *shflags;
//
//    int IWS;
//    char* termchars;
//
//    // methods
//    char* (*charin)(char *cp, char *pat);
//    char* (*expandquote)(char *s, Rune r, Bufblock *b);
//    int (*escapetoken)(Biobuf *bp, Bufblock *buf, int preserve, int esc);
//    char* (*copyq)(char *s, Rune q, Bufblock *buf);

//old:
//extern char *termchars;
//extern int IWS;
//extern char  *shell;
//extern char  *shellname;
//extern char  *shflags;
//extern Shell sh;
//extern Shell rc;
// either sh or rc
//extern Shell *shell;

// right now always rc, but can be configured with MKSHELL to use a different
// path than /bin/rc (e.g., /opt/plan9/bin/rc when under Linux or even goken/bin/rc/)
extern Shell *shell;

extern char *termchars;
extern char  *shflags;

#include "fns.h"
```

Uses `Arc` 40f, `Bufblock` 171c, `Job` 41d, `Node` 39d, `Rule` 34, `Shell` 105b, `ShellEnvVar` 112a, `Symtab` 28, and `Word` 32b.

E.1.3 mk/globals.c

⟨mk/globals.c 184⟩≡

```
#include "mk.h"

// used by DEBUG which is used by many files
⟨global debug 165b⟩

⟨global infile 51a⟩
⟨global mkinline 51c⟩

⟨global rules 35b⟩
⟨global metarules 35e⟩
```

```
⟨global patrule 128d⟩

// was in main.c, but used also by parse.c
⟨global target1 49b⟩

⟨global nflag 124e⟩
⟨global tflag 156a⟩
⟨global iflag 154a⟩
⟨global kflag 161a⟩
⟨global aflag 159b⟩

⟨global explain 124a⟩

⟨global jobs 42c⟩

⟨global bout 45b⟩
```

E.1.4 mk/utils.c

```
⟨mk/utils.c 185a⟩≡
  #include "mk.h"

  ⟨function Malloc 171a⟩

  ⟨function Realloc 171b⟩

  // maketmp() is back in Plan9.c
```

E.1.5 mk/bufblock.c

```
⟨mk/bufblock.c 185b⟩≡
  #include "mk.h"

  ⟨global freelist 172b⟩
  ⟨constant QUANTA 172c⟩

  ⟨constructor newbuf 172d⟩

  ⟨destructor freebuf 172e⟩

  ⟨function growbuf 173d⟩

  ⟨function bufcpy 173c⟩

  ⟨function insert 173a⟩

  ⟨function rinsert 173b⟩
```

E.1.6 mk/symtab.c

```
⟨mk/symtab.c 185c⟩≡
  #include "mk.h"

  ⟨constant NHASH 29c⟩
  ⟨constant HASHMUL 30d⟩
  ⟨global hash 29b⟩
```

```
⟨function symlook 30a⟩

⟨function symtraverse 31a⟩

⟨function symstat 170c⟩
```

E.1.7 mk/rc.c

```
⟨mk/rc.c 186a⟩≡
  #include "mk.h"

  ⟨global termchars 75a⟩
  ⟨global shflags 105d⟩

  /*
   * This file contains functions that depend on rc's syntax.  Most
   * of the routines extract strings observing rc's escape conventions
   */

  ⟨function squote 59b⟩

  ⟨function charin 58c⟩

  ⟨function expandquote(rc.c) 62c⟩

  ⟨function escapetoken(rc.c) 56b⟩

  ⟨function copysingle 123c⟩

  ⟨function copyq 123b⟩
```

E.1.8 mk/word.c

```
⟨mk/word.c 186b⟩≡
  #include "mk.h"

  ⟨constructor newword 32d⟩


  ⟨function wtos 33d⟩

  ⟨function wdup 33c⟩

  ⟨destructor freewords 33a⟩

  // was in recipe.c before
  ⟨function addw 33b⟩
```

E.1.9 mk/var.c

```
⟨mk/var.c 186c⟩≡
  #include "mk.h"

  ⟨function setvar 31b⟩

  ⟨function shname 116a⟩
```

E.1.10 mk/archive.c

```
⟨mk/archive.c 187a⟩≡
  #include "mk.h"
  #include <ar.h>

  static void atimes(char *);
  static char *split(char*, char**);

  ⟨function atimeof 150g⟩

  ⟨function atouch 151⟩

  ⟨function atimes 152a⟩

  ⟨function type 153a⟩

  ⟨function split 153b⟩
```

E.1.11 mk/match.c

```
⟨mk/match.c 187b⟩≡
  #include "mk.h"

  ⟨function match 80a⟩

  ⟨function subst 80c⟩
```

E.1.12 mk/env.c

```
⟨mk/env.c 187c⟩≡
  #include "mk.h"

  ⟨constant ENVQUANTA 113d⟩

  ⟨global shellenv 112b⟩
  ⟨global nextv 112c⟩

  ⟨global specialvars 31d⟩

  // encodenulls() is back in Plan9.c
  // readenv() is back in Plan9.c
  extern void readenv(void);
  // exportenv() is back in plan9.c

  ⟨function inithash 32a⟩

  ⟨function envinsert 113a⟩

  ⟨function envupd 113e⟩

  ⟨function ecopy 114c⟩

  ⟨function initenv 114b⟩

  ⟨function buildenv 117c⟩
```

E.1.13 mk/parse.c

```
⟨mk/parse.c 188a⟩≡
  #include        "mk.h"

  void    ipop(void);
  void    ipush(void);
  static int   rhead(char *, Word **, Word **, int *, char **);
  static char* rbody(Biobuf*);

  ⟨function parse 51e⟩

  ⟨function addrules 39a⟩

  ⟨function rhead 57b⟩

  ⟨function rbody 69a⟩

  ⟨struct input 72c⟩
  ⟨global inputs 72d⟩

  ⟨function ipush 72f⟩

  ⟨function ipop 73a⟩
```

E.1.14 mk/shprint.c

```
⟨mk/shprint.c 188b⟩≡
  #include "mk.h"

  static char *vexpand(char*, ShellEnvVar*, Bufblock*);
  static char *shquote(char*, Rune, Bufblock*);
  static char *shbquote(char*, Bufblock*);

  ⟨function shprint 121b⟩

  ⟨function mygetenv 122a⟩

  ⟨function vexpand 121c⟩

  ⟨function front 121a⟩
```

E.1.15 mk/job.c

```
⟨mk/job.c 188c⟩≡
  #include "mk.h"
```

E.1.16 mk/arc.c

```
⟨mk/arc.c 188d⟩≡
  #include "mk.h"
```

E.1.17 mk/rule.c

⟨mk/rule.c 189a⟩≡
```
#include "mk.h"

⟨global lr 35d⟩
⟨global lmr 35f⟩

⟨global nrules 85c⟩

static int rcmp(Rule *r, char *target, Word *tail);


⟨function addrule 36b⟩


⟨function rcmp 38b⟩

⟨function rulecnt 86b⟩

⟨function regerror 129b⟩
```

E.1.18 mk/lex.c

⟨mk/lex.c 189b⟩≡
```
#include "mk.h"

static int bquote(Biobuf*, Bufblock*);

⟨function assline 53b⟩

⟨function bquote 131b⟩

⟨function nextrune 54a⟩
```

E.1.19 mk/file.c

⟨mk/file.c 189c⟩≡
```
#include "mk.h"

// chgtime() is back in Plan9.c
extern int chgtime(char *name);
// dirtime() is back in Plan9.c
// bulkmtime() is back in Plan9.c
// mkmtime is back in Plan9.c

/* table-driven version in bootes dump of 12/31/96 */

⟨function timeof 82d⟩

⟨function touch 156e⟩

⟨function delete 141h⟩

⟨function timeinit 125d⟩
```

E.1.20 mk/run.c

```
⟨mk/run.c 190⟩≡
  #include "mk.h"

  typedef struct RunEvent RunEvent;
  typedef struct Process Process;

  int nextslot(void);
  int pidslot(int);
  void killchildren(char *msg);

  static void sched(void);

  static void pnew(int, int);
  static void pdelete(Process *);

  ⟨struct RunEvent 102c⟩

  ⟨global events 102d⟩
  ⟨global nevents 103a⟩
  ⟨global nrunning 101b⟩
  ⟨global nproclimit 101c⟩

  ⟨struct Process 144d⟩
  ⟨global phead 144e⟩
  ⟨global pfree 145a⟩

  ⟨function run 101a⟩

  // shell is back in Plan9.c
  // shellname is back in Plan9.c

  ⟨function sched 103e⟩

  // execsh() is back in Plan9.c
  // xwaitfor() is back in Plan9.c
  extern int xwaitfor(char *msg);

  ⟨function waitup 107c⟩

  ⟨function nproc 102b⟩

  ⟨function nextslot 103c⟩

  ⟨function pidslot 103d⟩


  ⟨function pnew 145b⟩

  ⟨function pdelete 145c⟩

  // Exit() is back in Plan9.c
  // notifyf() is back in Plan9.c
  // catchnotes() is back in Plan9.c
  // expunge() is back in Plan9.c
  extern void expunge(int pid, char *msg);

  ⟨function killchildren 111e⟩
```

```
⟨global tslot 126c⟩
⟨global tick 126d⟩

⟨function usage 126g⟩

⟨function prusage 127b⟩

// pipecmd() is back in Plan9.c
```
Uses Process 144d and RunEvent 102c.

E.1.21 mk/graph.c

⟨mk/graph.c 191a⟩≡
```
#include "mk.h"

static Node *applyrules(char *, char *);
static void togo(Node *);
static bool vacuous(Node *);
Arc* newarc(Node *n, Rule *r, char *stem, Resub *match);

static Node *newnode(char *);
static void trace(char *, Arc *);
static void cyclechk(Node *);
static void ambiguous(Node *);
static void attribute(Node *);

⟨global nreps 86d⟩

⟨function graph 76⟩

⟨function applyrules 77a⟩

⟨function nrep 148h⟩

⟨function togo 89c⟩

⟨function vacuous 90g⟩

⟨constructor newnode 40a⟩

// rcopy() is back in Plan9.c
extern void rcopy(char **to, Resub *match, int n);

⟨constructor newarc 41c⟩


⟨function trace 88a⟩

⟨function cyclechk 84b⟩

⟨function ambiguous 87⟩

⟨function attribute 140b⟩
```

E.1.22 mk/mk.c

⟨mk/mk.c 191b⟩≡
```
#include "mk.h"
```

```
void clrmade(Node*);
void work(Node*, bool*, Node*, Arc*);

⟨global runerrs 161c⟩

⟨function mk 92⟩

⟨function clrmade 94a⟩

⟨function unpretend 155b⟩

⟨function work 95a⟩

⟨function update 108b⟩

⟨function pcmp 144c⟩

⟨function outofdate 96c⟩
```

E.1.23 mk/recipe.c

⟨mk/recipe.c 192a⟩≡

```
#include "mk.h"

⟨constructor newjob 42b⟩

⟨function dorecipe 97e⟩
```

E.1.24 mk/varsub.c

⟨mk/varsub.c 192b⟩≡

```
#include "mk.h"

static Word  *subsub(Word*, char*, char*);
static Word  *expandvar(char**);
static Bufblock *varname(char**);
static Word  *extractpat(char*, char**, char*, char*);
static bool  submatch(char*, Word*, Word*, int*, char**);
static Word  *varmatch(char *);

⟨function varsub 63d⟩

⟨function varname 64d⟩

⟨function varmatch 65b⟩

⟨function expandvar 136c⟩

⟨function extractpat 138⟩

⟨function subsub 137⟩

⟨function submatch 139⟩


⟨function nextword 61⟩
```

```
⟨function stow 60b⟩
```

E.1.25 mk/dumpers.c

```
⟨mk/dumpers.c 193a⟩≡
  #include "mk.h"

  void dumpa(char*, Arc*);

  ⟨dumper dumpn 167b⟩

  ⟨dumper dumpa 167c⟩

  ⟨dumper dumpj 168g⟩

  ⟨function print1 166e⟩

  ⟨dumper dumpv 166d⟩

  ⟨dumper dumpr 166c⟩

  ⟨dumper dumpw 166b⟩
```

E.1.26 mk/main.c

```
⟨mk/main.c 193b⟩≡
  #include "mk.h"

  ⟨constant MKFILE 48f⟩

  // see also globals.c

  ⟨global uflag 126a⟩

  void badusage(void);

  #ifdef PROF
  ⟨global buf 170a⟩
  #endif

  ⟨function main 45a⟩

  ⟨function badusage 46c⟩
```

E.1.27 mk/Posix.c

```
⟨mk/Posix.c 193c⟩≡

  // to avoid conflict for wait(), waitpid() signatures
  #define NOPLAN9DEFINES
  #include        "mk.h"

  // the unix includes
  #include        <dirent.h>
```

```
#include        <signal.h>
#include        <sys/wait.h>
#include        <utime.h>
#include        <stdio.h>

typedef struct ShellEnvVar Envy;
int     IWS = '\1'; /* inter-word separator in env - not used in plan 9 */

//old:
//char  *shell =        "/bin/sh";
//char  *shellname =    "sh";

// I still want to default to rc in Unix especially in goken context
Shell rc = {
  .shellname = "rc",
  .shell = "/bin/rc",
};

Shell* shell = &rc;

// see man page environ(7)
extern char **environ;

void
readenv(void)
{
  char **p, *s;
  Word *w;

  for(p = environ; *p; p++){
    s = shname(*p);
    if(*s == '=') {
      *s = 0;
      w = newword(s+1);
    } else
      w = newword("");
    if (symlook(*p, S_INTERNAL, 0))
      continue;
    s = strdup(*p);
    setvar(s, (void *)w);
    //symlook(s, S_EXPORTED, (void*)"")->value = (void*)"";
  }
}

/*
 *      done on child side of fork, so parent's env is not affected
 *      and we don't care about freeing memory because we're going
 *      to exec immediately after this.
 */
void
exportenv(Envy *e)
{
  int i;
  char **p;
  char *values;

  p = 0;
  for(i = 0; e->name; e++, i++) {
    p = (char**) Realloc(p, (i+2)*sizeof(char*));
    if (e->values)
```

```
      values = wtos(e->values, IWS);
    else
      values = "";
    p[i] = malloc(strlen(e->name) + strlen(values) + 2);
    sprint(p[i], "%s=%s", e->name,  values);
  }
  p[i] = 0;
  environ = p;
}

int
xwaitfor(char *msg)
{
  int status;
  int pid;

  *msg = 0;
  pid = wait(&status);
  if(pid > 0) {
    if(status&0x7f) {
      if(status&0x80)
 snprint(msg, ERRMAX, "signal %d, core dumped", status&0x7f);
      else
 snprint(msg, ERRMAX, "signal %d", status&0x7f);
    } else if(status&0xff00)
      snprint(msg, ERRMAX, "exit(%d)", (status>>8)&0xff);
  }
  return pid;
}

void
expunge(int pid, char *msg)
{
  if(strcmp(msg, "interrupt"))
    kill(pid, SIGINT);
  else
    kill(pid, SIGHUP);
}

int
execsh(char *args, char *cmd, Bufblock *buf, Envy *e)
{
  char *p;
  int tot, n, pid, in[2], out[2];

  if(DEBUG(D_EXEC))
    fprint(1, "execsh='%s'\n", cmd);/**/

  if(buf && pipe(out) < 0){
    perror("pipe");
    Exit();
  }
  pid = fork();
  if(pid < 0){
    perror("mk fork");
    Exit();
  }
  if(pid == 0){
    if(buf)
      close(out[0]);
```

```
    if(pipe(in) < 0){
      perror("pipe");
      Exit();
    }
    pid = fork();
    if(pid < 0){
      perror("mk fork");
      Exit();
    }
    if(pid != 0){
      dup2(in[0], 0);
      if(buf){
 dup2(out[1], 1);
 close(out[1]);
      }
      close(in[0]);
      close(in[1]);
      if (e)
 exportenv(e);
      if(shflags)
 // to debug mk/rc you can add "-r", "-s", "-x", "-v" after shflags below
 execl(shell->shell, shell->shellname, shflags, args, nil);
      else
 execl(shell->shell, shell->shellname, args, nil);
      perror(shell->shell);
      _exits("exec");
    }
    close(out[1]);
    close(in[0]);
    if(DEBUG(D_EXEC))
      fprint(1, "starting: %s\n", cmd);
    p = cmd+strlen(cmd);
    while(cmd < p){
      n = write(in[1], cmd, p-cmd);
      if(n < 0)
 break;
      cmd += n;
    }
    close(in[1]);
    _exits(0);
  }
  if(buf){
    close(out[1]);
    tot = 0;
    for(;;){
      if (buf->current >= buf->end)
 growbuf(buf);
      n = read(out[0], buf->current, buf->end-buf->current);
      if(n <= 0)
 break;
      buf->current += n;
      tot += n;
    }
    if (tot && buf->current[-1] == '\n')
      buf->current--;
    close(out[0]);
  }
  return pid;
}
```

```
int
pipecmd(char *cmd, Envy *e, int *fd)
{
  int pid, pfd[2];

  if(DEBUG(D_EXEC))
    fprint(1, "pipecmd='%s'\n", cmd);/**/

  if(fd && pipe(pfd) < 0){
    perror("pipe");
    Exit();
  }
  pid = fork();
  if(pid < 0){
    perror("mk fork");
    Exit();
  }
  if(pid == 0){
    if(fd){
      close(pfd[0]);
      dup2(pfd[1], 1);
      close(pfd[1]);
    }
    if(e)
      exportenv(e);
    if(shflags)
      execl(shell->shell, shell->shellname, shflags, "-c", cmd, nil);
    else
      execl(shell->shell, shell->shellname, "-c", cmd, nil);
    perror(shell->shell);
    _exits("exec");
  }
  if(fd){
    close(pfd[1]);
    *fd = pfd[0];
  }
  return pid;
}

void
Exit(void)
{
  while(wait(0) >= 0)
    ;
  exits("error");
}

static  struct
{
  int    sig;
  char   *msg;
}       sigmsgs[] =
  {
    SIGALRM,       "alarm",
    SIGFPE,                "sys: fp: fptrap",
    SIGPIPE,       "sys: write on closed pipe",
    SIGILL,                "sys: trap: illegal instruction",
    SIGSEGV,       "sys: segmentation violation",
    0,             0
  };
```

```
static void
notifyf(int sig)
{
  int i;

  for(i = 0; sigmsgs[i].msg; i++)
    if(sigmsgs[i].sig == sig)
      killchildren(sigmsgs[i].msg);

  /* should never happen */
  signal(sig, SIG_DFL);
  kill(getpid(), sig);
}

void
catchnotes()
{
  int i;

  for(i = 0; sigmsgs[i].msg; i++)
    signal(sigmsgs[i].sig, notifyf);
}

char*
maketmp(void)
{
  static char temp[L_tmpnam];

  return tmpnam(temp);
}

int
chgtime(char *name)
{
  Dir *sbuf;
  struct utimbuf u;

  if((sbuf = dirstat(name)) != nil) {
    u.actime = sbuf->atime;
    free(sbuf);
    u.modtime = time(0);
    return utime(name, &u);
  }
  return close(p9create(name, OWRITE, 0666));
}

void
rcopy(char **to, Resub *match, int n)
{
  int c;
  char *p;

  *to = match->s.sp;             /* stem0 matches complete target */
  for(to++, match++; --n > 0; to++, match++){
    if(match->s.sp && match->e.ep){
      p = match->e.ep;
      c = *p;
      *p = 0;
      *to = strdup(match->s.sp);
```

```
      *p = c;
    }
    else
      *to = 0;
  }
}

ulong
mkmtime(char *name, bool _force)
{
  Dir *buf;
  ulong t;

  buf = dirstat(name);
  if(buf == nil)
    return 0;
  t = buf->mtime;
  free(buf);
  return t;
}


char *stab;

char *
membername(char *s, int fd, char *sz)
{
  long t;
  char *p, *q;

  if(s[0] == '/' && s[1] == '\0'){       /* long file name string table */
    t = atol(sz);
    if(t&01) t++;
    stab = malloc(t);
    if(read(fd, stab, t) != t)
      {}
    return nil;
  }
  else if(s[0] == '/' && stab != nil)    {       /* index into string table */
    p = stab+atol(s+1);
    q = strchr(p, '/');
    if (q)
      *q = 0;                             /* terminate string here */
    return p;
  }else
    return s;
}
```

Uses DEBUG 165c, D_EXEC 165a, IWS 193c, Realloc() 171b, S_INTERNAL 31c, ShellEnvVar 112a, __anon_struct_1 193c, growbuf() 173d, killchildren() 111e, newword() 32d, setvar() 31b, shflags 105d, shname() 116a, sigmsgs-5 193c, stab 193c, symlook() 30a, and wtos() 33d.

E.1.28 mk/Plan9.c

⟨mk/Plan9.c 199⟩≡
```
#include "mk.h"

// could be in utils.c
⟨function maketmp 174⟩
```

```
// could be in env.c
⟨function encodenulls 117a⟩
⟨function readenv(plan9.c) 115a⟩
⟨function exportenv(plan9.c) 118b⟩

// could be in file.c
⟨function chgtime 156f⟩
⟨function dirtime 159a⟩
⟨function bulkmtime 158e⟩
⟨function mkmtime 83a⟩

// could be in run.c
⟨global rc 105c⟩

⟨global shell 105a⟩

// could be in run.c
⟨function execsh 105e⟩
⟨function xwaitfor(plan9.c) 108a⟩

// could be in run.c
⟨function Exit 110d⟩

// back in run.c
extern void killchildren(char *msg);

⟨function notifyf 111c⟩
⟨function catchnotes 111b⟩
⟨function expunge 146e⟩

// could be in run.c
⟨function pipecmd 135a⟩

// could be in graph.c
⟨function rcopy 129f⟩
```

Glossary

```
LOC = Lines Of Code
DSL = Domain Specific Language
IDE = Integrated Developement Environment
DAG = Directed Acyclic Graph
DFS = Depth-First Search
```

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.

addrule(): 36b, 39a
addrules(): 39a, 50, 67c
addw(): 33b, 100, 148b
aflag: 147d, 159b, 159b, 159c, 159d
ambiguous(): 83c, 87, 87
applyrules(): 76, 77a, 77c, 78e
Arc: 40f, 41a, 182d
Arc.match: 129d
Arc.n: 40f
Arc.next: 41a
Arc.prog: 143e
Arc.r: 40f
Arc.remove: 88e
Arc.stem: 41b
Arc (typedef): 182d
assline(): 51e, 53b
atimeof(): 150f, 150g
atimes(): 150g, 152a
atouch(): 151, 156e
attribute(): 140a, 140b, 140b
badusage(): 46b, 46c, 48d, 125b
BEINGMADE: 40d, 92, 95a, 96b, 97e, 161e, 161g, 161h
BIGBLOCK: 99a, 152a, 182a
bout: 45b, 45c, 92, 111e, 120b, 144c, 153a, 154e, 156c, 156d, 156e, 166b, 166c, 166d, 166e, 167a, 167b, 167c, 168a, 168g, 170c
bquote(): 131a, 131b
Bufblock: 171c, 172a, 182d
Bufblock.current: 171c
Bufblock.end: 171c
Bufblock.next: 172a
Bufblock.start: 171c
Bufblock (typedef): 182d
bufcontent: 53b, 172h
bufcpy(): 65c, 65d, 67a, 121c, 125b, 160b, 160c, 160f, 173c
buildenv(): 103e, 117c, 120d
bulkmtime(): 158e
CANPRETEND: 154c, 154d, 154e, 155b, 155f
catchnotes(): 193c

charin(): 37a, 58c, 67d, 136c, 138
chgtime(): 193c
clrmade(): 92, 94a, 94a
copyq(): 121b, 123b
copysingle(): 123b, 123c
CYCLE: 84a, 84b
cyclechk(): 83c, 84b, 84b
DEBUG: 135a, 165c, 166a, 167a, 167e, 168a, 168b, 168c, 168d, 168e, 168f, 168h, 168i, 169a, 169b, 193c
debug: 165b, 165e
DEL: 141b, 141e
DELETE: 141d, 141e, 141g
delete(): 141g, 141h
dirtime(): 158e, 159a
dorecipe(): 96b, 97e
dumpa(): 167b, 167c
dumpj(): 168g
dumpn(): 167a, 167b, 167c
dumpr(): 166a, 166c
dumpv(): 166a, 166d
dumpw(): 166a, 166b
Dxxx: 165a
D_EXEC: 135a, 165a, 165e, 167e, 168a, 168b, 168c, 168d, 168e, 168f, 193c
D_GRAPH: 165a, 165e, 167a
D_PARSE: 165a, 165e, 166a
D_TRACE: 165a, 168h, 168i, 169a, 169b
ecopy(): 114b, 114c
EMPTY_CHILDREN_IS_ERROR2: 162a, 162b
EMPTY_CHILDREN_IS_ERROR3: 134b, 134d, 144c
EMPTY_CHILDREN_IS_ERROR: 92, 109b
EMPTY_CHILDREN_IS_OK: 92, 109b, 109d, 111e
EMPTY_CHILDREN: 109c, 109d
empty_prereqs: 78a, 78c, 78d, 79d, 79e
empty_recipe: 78a, 78b, 79d, 87, 97e
empty_words: 58b, 102b, 119c
envinsert(): 113a, 113f, 114b, 114c
ENVQUANTA-6: 113c, 113d
envupd(): 113e, 117c, 129a, 130c, 147h, 148f, 149c, 149e, 150c
Envy (typedef): 193c
escapetoken(): 56a, 56b, 132a
events-14: 102d, 103b, 103c, 103d, 103e, 107c, 168b
execsh(): 193c
Exit(): 193c
expandquote(): 62b, 62c
expandvar(): 136b, 136c
explain: 124a, 124a, 124b, 124c, 124d, 154e, 155a, 156c, 156d
exportenv(): 193c
expunge(): 193c
extractpat(): 138
freebuf(): 33d, 51e, 61, 63d, 65a, 69a, 110b, 120b, 125c, 136c, 160f, 172e

freelist-11: 172b, 172d, 172e, 173d
freewords(): 33a, 113e, 119d
front(): 120d, 121a
graph(): 76, 92
growbuf(): 133, 173a, 173b, 173d, 193c
hash-3: 29b, 30a, 30e, 31a, 170c
HASHMUL-2: 30c, 30d
iflag: 154a, 154a, 154b, 154e, 159c
infile: 51a, 51b, 51e, 72f, 73a, 129b
initenv(): 114a, 114b, 132c, 134d
inithash(): 32a, 46a
Input: 72c, 72d, 72e, 72f, 73a
Input.file: 72c
Input.line: 72c
Input.next: 72e
inputs-13: 72d, 72d, 72f, 73a
insert(): 33d, 53b, 55c, 61, 64d, 65c, 65d, 67a, 69a, 121b, 125b, 125c, 131b, 132a, 160b, 160c, 160e, 160f, 173a, 173c
ipop(): 72b, 73a
ipush(): 72a, 72f
isempty: 53b, 55c, 61, 65a, 65c, 172f
IWS: 193c, 193c
Job: 41d, 42d, 182d
Job.at: 147f
Job.match: 129g
Job.n: 41d
Job.next: 42d
Job.np: 148d
Job.p: 41d
Job.r: 41d
Job.stem: 41d
Job.t: 41d
Job (typedef): 182d
jobs: 42c, 101a, 103e, 104, 110b, 111e, 162b
JOB_ENDED: 107c, 109c, 111e
kflag: 111e, 161a, 161a, 161b, 161e, 161f
killchildren(): 111c, 111e, 193c
lmr-23: 35f, 37a
lr-22: 35d, 36c
MADE: 40d, 95b, 96b, 108b, 124h, 150e, 154e, 155a, 162b
MADESET: 94a, 94b, 95b, 96b, 97e, 108b, 124h, 150e, 154e, 155b, 161g
main-10(): 45a
maketmp(): 193c
Malloc(): 30e, 32d, 36b, 40a, 41c, 42b, 72f, 86b, 103b, 115a, 145b, 171a, 172d
membername(): 193c
META: 36a, 37a, 88d, 90g, 99b
metarules: 35e, 37a, 78e, 166a
mk(): 49d, 49g, 50, 92
MKFILE-9: 48e, 48f

mkinline: 50, 51c, 51e, 54b, 54c, 55b, 56b, 69b, 72f, 73a, 129b, 131b
mkmtime(): 193c
mygetenv(): 121c, 122a
NAMEBLOCK: 79a, 79b, 79c, 144a, 144c
Namespace: 29a
nevents-15: 103a, 103b, 103d, 127b
newarc(): 41c, 77c, 78d, 78e, 79e
newbuf(): 33d, 51e, 61, 64d, 69a, 110b, 120b, 125b, 160a, 172d
newjob(): 42b, 97e
newnode(): 40a, 77a
newword(): 32d, 33b, 33c, 50, 60c, 61, 99b, 117c, 128f, 129a, 130c, 136c, 147d, 149c, 149e, 193c
nextrune(): 53b, 54a, 56b, 131b
nextslot(): 103c, 103e
nextv-7: 112c, 113a, 113c, 114b
nextword(): 60b, 61
nflag: 120b, 124e, 124e, 124f, 124h, 156e
NHASH-1: 29b, 29c, 30c, 31a, 170c
Node: 39d, 40f, 42a, 182d
Node.arcs: 40e
Node.flags: 40b
Node.name: 39d
Node.next: 42a
Node.time: 39d
Node (typedef): 182d
Node_flag: 40c
NOMINUSE: 142b, 142d
NOPLAN9DEFINES-4: 193c
NOREC: 142e, 142h
NORECIPE: 140h, 142g, 142h
notifyf(): 193c
NOTMADE: 40d, 92, 94a, 96b, 155b, 161h
NOT_A_JOB_PROCESS: 109c
NOVIRT: 142i, 142k
nproc(): 102a, 102b
nproclimit-17: 101a, 101c, 102b, 103b, 103c, 107c, 168f
NREGEXP: 129c, 129d, 129e, 130d, 130f, 130g
nreps: 86d, 86d, 86e
nrules-24: 85c, 85c, 85d, 86b
nrunning-16: 101a, 101b, 103e, 107c, 126g
outofdate(): 96b, 96c, 100, 141a, 143j, 147d, 154e, 155a
parse(): 47b, 48e, 51e, 71b, 134d
patrule: 128d, 128e, 129b, 130f
pcmp(): 144b, 144c
pdelete(): 145c, 146a
PERCENT: 35g, 80c
pfree-19: 145a, 145b, 145c
phead-18: 144e, 145b, 145c, 146a, 146d
pidslot(): 103d, 107c
pipecmd(): 193c

pnew(): 145b, 145d
PRETENDING: 154c, 154d, 154e, 155a, 155b, 155f
print1(): 166d, 166e
PROBABLE: 89e, 90a, 90b, 90c, 90g
Process: 144d, 190
Process.b: 144d
Process.f: 144d
Process.pid: 144d
Process.status: 144d
Process (typedef): 190
prusage(): 127a, 127b
QUANTA-12: 172c, 172d, 173d
QUIET: 120b, 141i
rbody(): 67c, 69a
rc: 193c
rcopy(): 193c
readenv(): 193c
READY: 90f, 90g
Realloc(): 103b, 113c, 171b, 173d, 193c
regerror(): 129b
REGEXP: 37a, 67d, 128a, 128e, 128f, 129a, 130c, 130f, 130g
resetbuf: 53b, 65c, 65d, 172g
rinsert(): 53b, 56a, 56b, 61, 62c, 64d, 69a, 69c, 121b, 123b, 123c, 131b, 173b
Rule: 34, 35c, 37c, 182d
Rule.alltargets: 39b
Rule.attr: 35h
Rule.chain: 37c
Rule.file: 35a
Rule.line: 35a
Rule.next: 35c
Rule.pat: 128c
Rule.prereqs: 34
Rule.prog: 143b
Rule.recipe: 34
Rule.rule: 85b
Rule.target: 34
Rule (typedef): 182d
rulecnt(): 86a, 86b
rules: 35b, 36c, 166a
Rule_attr: 36a
run(): 97e, 101a
runerrs: 161c, 161d, 161e, 161f, 161h, 162b
RunEvent: 102c, 190
RunEvent.job: 102c
RunEvent.pid: 102c
RunEvent (typedef): 190
sched(): 101a, 103e, 107c
setvar(): 31b, 73c, 115a, 193c
Shell: 105b, 182d

shell: 193c
Shell.shell: 105b
Shell.shellname: 105b
Shell (typedef): 182d
shellenv: 112b, 113a, 113c, 113e, 117c, 132c, 134d
ShellEnvVar: 112a, 182d, 193c
ShellEnvVar.name: 112a
ShellEnvVar.values: 112a
ShellEnvVar (typedef): 182d
shflags: 105d, 105d, 105e, 135a, 193c
shname(): 115b, 116a, 121c, 193c
shprint(): 120b, 120d, 121b
sigmsgs-5: 193c, 193c
specialvars-8: 31d, 32a, 114b, 114d, 130c
split(): 150g, 151, 153b
squote(): 59a, 59b
stab: 193c, 193c
stow(): 60b, 114b, 138, 160e, 160f
submatch(): 139
subst(): 78e, 80c, 99b
symlook(): 30a, 31b, 32a, 37e, 38a, 74b, 77c, 82b, 82c, 99b, 102b, 107c, 115b, 119c, 122a, 123a, 125d, 136c, 144b, 146g, 147b, 149g, 150g, 151, 152a, 153a, 157e, 157f, 158e, 159a, 160e, 160f, 193c
symstat(): 170c
Symtab: 28, 29d, 182d
Symtab.name: 28
Symtab.next: 29d
Symtab.space: 28
Symtab.u: 28
Symtab (typedef): 182d
symtraverse(): 31a, 114b, 166d
SYNERR: 51d, 53a, 56b, 59b, 65a, 71c, 73d, 131b, 134e, 136a, 136c
S_AGG: 150d, 150g
S_BITCH: 152b, 153a
S_BULKED: 158d, 158e
S_INTERNAL: 31c, 32a, 115b, 122a, 193c
S_NODE: 82a, 82b, 82c, 99b, 107c
S_NOEXPORT: 146g, 147a, 147b
S_OUTOFDATE: 143i, 144b
S_OVERRIDE: 74b, 74c
S_TARGET: 37b, 37e, 38a, 77c
S_TIME: 125d, 150g, 151, 152a, 157a, 157e, 157f, 159a
S_VAR: 29a, 31b, 102b, 114b, 119c, 136c, 149g, 160e, 160f, 166d
S_WESET: 122a, 122b, 123a
target1: 49b, 49d, 67d, 166a
termchars: 75a, 75a
tflag: 120b, 124h, 156a, 156a, 156b, 156c, 156d
tick-21: 126d, 126g
timeinit(): 125c, 125d
timeof(): 40a, 82d, 108b

togo(): 89b, 89c, 90g
touch(): 156c, 156d, 156e
trace(): 87, 88a
tslot-20: 126c, 126g, 127b
type(): 153a, 153b
uflag: 126a, 126a, 126b, 127a
unpretend(): 154e, 155a, 155b
update(): 107c, 108b, 140h
usage(): 103e, 104, 107c, 126e, 126f, 126g, 127b
VACUOUS: 90e, 90g
vacuous(): 90d, 90g, 90g
varname(): 63d, 64d, 136c
varsub(): 63c, 63d
vexpand(): 121b, 121c
VIR: 50, 140c, 140f, 142k
VIRTUAL: 140e, 140f, 140g, 140h, 141a, 156c, 156d
waitup(): 92, 107c, 111e, 134d, 144c, 162b
WaitupParam: 109b
WaitupResult: 109c
wdup(): 33c, 67d, 117c, 147h, 148f, 150c
Word: 32b, 32c, 182d
Word.next: 32c
Word.s: 32b
Word (typedef): 182d
WORDCHR: 64d, 64e, 116a
work(): 92, 95a, 96b, 154e
wtos(): 33d, 71b, 122a, 134d, 166c, 167e, 168b, 168g, 193c
xwaitfor(): 193c
__anon_struct_1.msg: 193c
__anon_struct_1.sig: 193c
__anon_struct_1: 193c, 193c
__anon_struct_2.ptr: 28
__anon_struct_2.value: 28
__anon_struct_2: 28, 28

Bibliography

[dB88] Adam de Boor. Pmake — a tutorial, 1988. Available at https://www.freebsd.org/doc/en_US.ISO8859-1/books/pmake/. cited page(s) 9

[Fel79] Stuart I. Feldman. Make — a program for maintaining computer programs. In *Software Practice & Experience*, pages 255–266, 1979. Also available at builders/docs/make.pdf. cited page(s) 7, 8

[HF95] Andrew G. Hume and Bob Flandrena. Maintaining files on plan 9 with mk. Technical report, Bell Labs, 1995. Also available at builders/docs/mk.pdf. cited page(s) 12

[HL02] Erik Hatcher and Steve Loughran. *Java Development with Ant*. Manning Publications, 2002. cited page(s) 9

[Hum87] Andrew G. Hume. Mk: A successor to make. In *USENIX Summer Conference*, 1987. Also available at builders/docs/mk_make_successor.pdf. cited page(s) 8, 12

[Joh79] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual Vol 2b*, 1979. Also available at generators/docs/yacc.pdf. cited page(s) 15, 21

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

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 12

[LS79] M. E. Lesk and E. Schmidt. Lex — a lexical analyzer generator. In *Unix Programmer's Manual Vol 2b*, 1979. Also available at generators/docs/lex.pdf. cited page(s) 15

[Mec04] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly, 2004. Available at http://www.oreilly.com/openbook/make3/book/. cited page(s) 9, 12

[Mil97] Peter Miller. Recursive make considered harmful. In *AUUG Conference Proceedings*, 1997. The classic argument that splitting a build across per-directory makefiles gives each `make` only a partial dependency graph, causing both needless and missed rebuilds; advocates a single non-recursive makefile. Available at https://aegis.sourceforge.net/auug97.pdf. cited page(s) 160

[MMJM16] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. Non-recursive make considered harmful: Build systems at scale. In *Haskell Symposium*, 2016. The deliberate riposte; argues the monolithic alternative does not scale either and reframes builds as reusable libraries (leading to Shake). Available at https://doi.org/10.1145/2976002.2976011. cited page(s) 160

[OL96] Andy Oram and Mike Loukides. *Programming with GNU Software*. O'Reilly, 1996. cited page(s) 12

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

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 106, 108, 110, 115, 163

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 163

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 11

[Pad16a] Yoann Padioleau. *Principia Softwarica: (OCaml)Lex and (OCaml)Yacc*. 2016. cited page(s) 15

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 11, 52, 54, 163

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 15, 30, 46, 47, 52, 59, 106, 108, 128, 163

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 9, 13, 15, 56, 104, 105, 130, 163

[Pad26] Yoann Padioleau. *Principia Softwarica: The The Plan 9 Profilers*. 2026. cited page(s) 170

[SMS16] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make*. Free Software Foundation, 2016. Available at https://www.gnu.org/software/make/manual/. cited page(s) 12

[WMW20] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google*. O'Reilly, 2020. Its "Build Systems and Build Philosophy" chapter describes Google's Blaze/Bazel. Available at https://abseil.io/resources/swe-book. cited page(s) 10