Principia Softwarica: The Build System `mk`

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Andrew Hume

April 28, 2026

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 . . . 11
- 1.5 Requirements . . . 11
- 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 . . . 23
- 2.2 `mk` command-line interface . . . 25
- 2.3 `hello.mk` . . . 25
- 2.4 Code organization . . . 26
- 2.5 Software architecture . . . 26
- 2.6 Book structure . . . 29

3 Core Data Structures **30**
- 3.1 Symbol table . . . 30
  - 3.1.1 `Symtab` . . . 30
  - 3.1.2 `hash` . . . 31
  - 3.1.3 Namespaces . . . 33
- 3.2 `Words` . . . 34
- 3.3 Rules . . . 36
  - 3.3.1 `Rule` . . . 36
  - 3.3.2 Simple `rules` . . . 37
  - 3.3.3 `metarules` . . . 37
  - 3.3.4 Adding rules . . . 38
- 3.4 Graph . . . 41
  - 3.4.1 `Node` . . . 42
  - 3.4.2 `Arc` . . . 43
- 3.5 `Jobs` . . . 44

4 `main()` 47
4.1 `main()` skeleton . . . 47
4.2 `mk -`*flag* arguments processing . . . 48
4.3 `mk` *var*=*values* arguments processing . . . 49
4.4 `mk` remaining arguments processing . . . 50
4.5 Using the `mkfile` or `mk -f`*file* . . . 51
4.6 Building the target(s) . . . 51
4.6.1 Default target: `target1` . . . 51
4.6.2 Building Sequentiallly . . . 52
4.6.3 Building in Parallel . . . 52

5 Parsing the `mkfile` 54
5.1 `parse()` . . . 54
5.1.1 Assembling a line: `assline()` . . . 57
5.1.2 Parsing the head of a line: `rhead()` . . . 61
5.1.3 Splitting a string in words: `stow()` . . . 64
5.2 Parsing Rules: *target*:*prereqs* . . . 71
5.2.1 Parsing the recipe: `rbody()` . . . 73
5.2.2 Parsing rule attributes . . . 74
5.3 Parsing Included files: `<`*file* . . . 75
5.4 Parsing Variable definitions: *var*=*values* . . . 78
5.4.1 Overriding variable definitions . . . 78
5.4.2 Parsing variable attributes . . . 79

6 Building the Graph of Dependencies 80
6.1 `graph()` and `applyrules()` . . . 80
6.2 Finding the simple rule(s) for a target . . . 82
6.3 Finding matching metarules . . . 83
6.3.1 Matching a pattern: `match()` . . . 85
6.3.2 Substituting the stem: `subst()` . . . 86
6.4 Node cache . . . 87
6.5 `timeof()` . . . 88
6.6 Checking the graph and the rules . . . 88
6.6.1 Cycle detection . . . 89
6.6.2 Infinite rule detection . . . 90
6.6.3 Ambiguous rules detection . . . 92

7 Finding Outdated Files 98
7.1 Mtime vs content-hash change detection . . . 98
7.2 `mk()` . . . 99
7.3 Initializing nodes: `clrmade()` . . . 101
7.4 Exploring the graph: `work()` . . . 101
7.4.1 The leaf case . . . 102
7.4.2 The parent case . . . 102
7.5 Scheduling recipes: `dorecipe()` . . . 104
7.5.1 Building the list of target nodes . . . 106
7.5.2 Building the list of prerequisites . . . 106

8 Scheduling Jobs **108**
8.1 Enqueuing jobs: `run()` . . . 108
8.2 Scheduling jobs . . . 109
8.2.1 `RunEvent` and `events` . . . 109
8.2.2 `sched()` . . . 111
8.3 Executing Jobs: `execsh()` . . . 112
8.4 Waiting for jobs to finish . . . 115
8.4.1 `waitup()` . . . 115
8.4.2 `update()` . . . 116
8.4.3 `waitup()` edge cases . . . 117
8.5 Process management, `Exit()` . . . 118
8.6 Notes (signals) management . . . 119

9 The Shell Environment **120**
9.1 `Shellenv` and `shellenv` . . . 120
9.2 Initializing the shell environment: `initenv()` . . . 122
9.3 Importing the environment: `readenv()` . . . 122
9.4 Adjusting the shell environment: `buildenv()` . . . 125
9.5 Exporting the shell environment: `exportenv()` . . . 125

10 Debugging and Profiling Support TODO **128**
10.1 Printing jobs: `shprint()` . . . 128
10.1.1 Expanding and printing variables . . . 129
10.1.2 Printing quoted strings . . . 130
10.2 Explain mode: `mk -e` . . . 131
10.3 Dry mode: `mk -n` . . . 131
10.4 What-if mode: `mk -w`*file* . . . 132
10.5 Processor utilization: `mk -u` . . . 133

11 Advanced Features TODO **135**
11.1 Regular-expression rules: `:R:` . . . 135
11.2 Shell-command expansion: ‘*cmd*‘ . . . 137
11.2.1 Parsing backquotes: `bquote()` . . . 137
11.2.2 Adjusting `execsh()` . . . 138
11.3 Dynamic `mkfile`: `<|`*prog* . . . 139
11.4 Substitution variables: `$`*name*`:`*pattern*`=`*subst* . . . 141
11.4.1 Parsing adjustments . . . 141
11.4.2 Substitutions: `subsub()` . . . 143
11.5 Rule attributes . . . 145
11.5.1 Virtual target: `:V:` . . . 146
11.5.2 Deleting a target when the recipe returns an error: `:D:` . . . 146
11.5.3 Not printing the recipe (quiet mode): `:Q:` . . . 147
11.5.4 Running a shell script without `-e`: `:E:` . . . 147
11.5.5 Disabling the no-recipe warning, `:N:` . . . 148
11.5.6 Forbidding metarules to match virtual targets: `:n:` . . . 148
11.5.7 Custom-dependency comparison program: `:P:` . . . 148
11.6 Variable attributes . . . 151
11.6.1 Private variables: `=U=` . . . 151
11.7 Advanced `mk` variables . . . 152
11.7.1 `$target` versus `$alltargets` . . . 152

11.7.2 $prereq versus $newprereq . . . 152
11.7.3 $NREP . . . 153
11.7.4 $pid . . . 153
11.7.5 $nproc . . . 153
11.8 Dealing with archives (libraries) . . . 154
11.9 Optimizations . . . 157
11.9.1 Missing-intermediates optimization: mk -I . . . 157
11.9.2 Touching-mode optimization: mk -t . . . 159
11.9.3 Time cache . . . 160
11.9.4 Bulk time optimisation . . . 161
11.10 Recompiling everything: mk -a . . . 163
11.11 Recursive mk . . . 163
11.12 mk -k . . . 164
11.12.1 kflag and runerrs . . . 164
11.12.2 Adjusting mk(), work(), and waitup() . . . 165

12 Conclusion **166**
12.1 Patterns and techniques . . . 166
12.2 Connections to other books . . . 166
12.3 Beyond mk . . . 167

A Debugging **168**
A.1 Dumping the rules: mk -dp . . . 169
A.2 Dumping the graph: mk -dg . . . 170
A.3 Tracing jobs: mk -de . . . 170
A.4 Tracing function calls: mk -dt . . . 171

B Profiling **173**

C Utilities **174**
C.1 Memory management . . . 174
C.2 Buffer management . . . 174
C.3 File management . . . 177

D Examples of mkfiles TODO **178**
D.1 The mkfile of mk . . . 178
D.2 The mkfiles of Plan 9 . . . 178
D.2.1 /$objtype/mkfile for the ARM . . . 178
D.2.2 /sys/src/mkfile.proto . . . 178
D.2.3 /sys/src/cmd/mkone . . . 178
D.2.4 /sys/src/cmd/mklib . . . 178

E Extra Code **179**
E.1 mk/ . . . 179
E.1.1 mk/fns.h . . . 179
E.1.2 mk/mk.h . . . 181
E.1.3 mk/globals.c . . . 183
E.1.4 mk/utils.c . . . 184
E.1.5 mk/bufblock.c . . . 184
E.1.6 mk/symtab.c . . . 184
E.1.7 mk/rc.c . . . 185

E.1.8 mk/word.c . . . 185
E.1.9 mk/var.c . . . 185
E.1.10 mk/archive.c . . . 185
E.1.11 mk/match.c . . . 186
E.1.12 mk/env.c . . . 186
E.1.13 mk/parse.c . . . 186
E.1.14 mk/shprint.c . . . 187
E.1.15 mk/job.c . . . 187
E.1.16 mk/arc.c . . . 187
E.1.17 mk/rule.c . . . 187
E.1.18 mk/lex.c . . . 188
E.1.19 mk/file.c . . . 188
E.1.20 mk/run.c . . . 188
E.1.21 mk/graph.c . . . 190
E.1.22 mk/mk.c . . . 190
E.1.23 mk/recipe.c . . . 191
E.1.24 mk/varsub.c . . . 191
E.1.25 mk/dumpers.c . . . 192
E.1.26 mk/main.c . . . 192
E.1.27 mk/Posix.c . . . 192
E.1.28 mk/Plan9.c . . . 198

Glossary **200**

Index **201**

References **207**

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, package, 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.

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 5500 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.

The interesting design lesson in `mk` is that "simpler and more powerful" are not in tension here. Most of `mk`'s improvements over Make come from removing things rather than adding them: there is no `$(VAR)` vs. `$VAR` distinction because `mk` reuses the shell's variable syntax verbatim; there is no `=` vs. `:=` vs. `?=` because `mk`'s variables are constants; there is no `$$i` double-dollar escape because recipes are passed to the shell uninterpreted. Each removal eliminates an entire class of confusion. The power comes from a single architectural decision (build the full dependency graph statically before doing any work) which in turn enables parallel execution and exhaustive cycle and ambiguity checks.

`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`) 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

[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/.

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.

- 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). 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 variables such as `$target` or `$prereq`.

- Ant [HL02][8] (for Another Neat Tool) is 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 almost 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. 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/

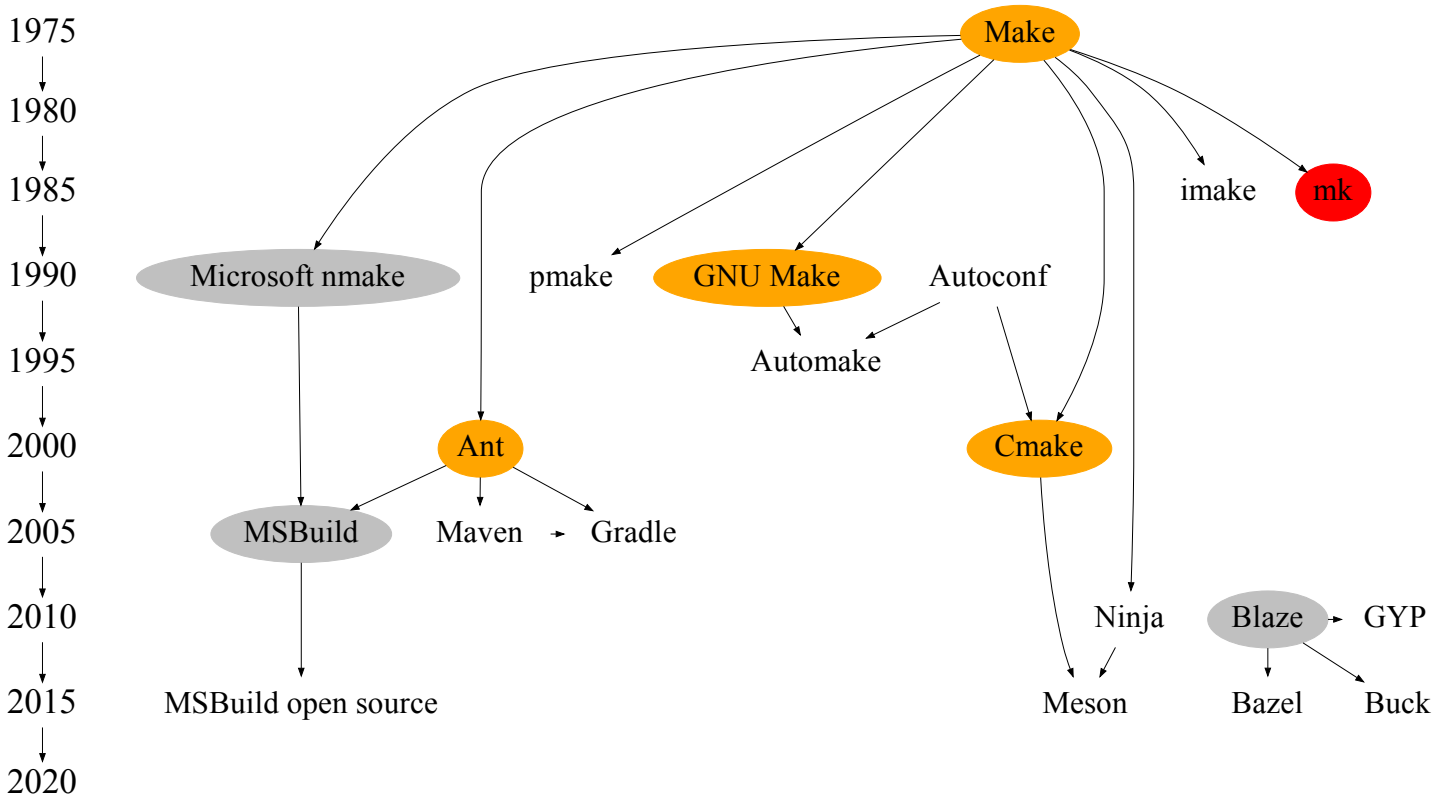


Figure 1.1: Build systems timeline

- CMake[9] is a cross-platform build system used in many large open source projects (e.g., LLVM). It was designed, like Ant, to overcome the lack of portability of `Makefile`s. Unlike Ant, CMake acts as a frontend to Make (and other build systems). Indeed, CMake is a *meta build system*. Instead of writing `Makefile`s, the user of CMake creates `CMakeLists.txt` files containing builtin (portable) commands to compile source code. CMake then generates from those configuration files regular `Makefile`s 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.

Note that the lack of portability of `Makefile`s 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[10] for Microsoft Windows, and macports[11] or Homebrew[12] for macOS), `Makefile`s 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 (5500 LOC).

[9] https://cmake.org/
[10] https://www.cygwin.com/
[11] https://www.macports.org/
[12] http://brew.sh/

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). You can also use `mk` on Linux, macOS, or Windows through plan9port[13] or Goken9cc[14], where it is compiled natively using gcc or clang. Once installed you can test `mk` under Plan 9 with the following commands:

```
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.

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.

[13] https://9fans.github.io/plan9port/
[14] https://github.com/aryx/goken9cc

1.6 About this document

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

1.7 Copyright

1.8 Acknowledgments

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

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.

The progression from shell script to build system is not just a matter of convenience—it illustrates a fundamental shift from imperative to declarative thinking. A shell script says "do these steps in this order." A build system says "here are the relationships between files; figure out what needs updating." The key insight is that by describing what depends on what rather than what to do, the system can automatically determine the minimal set of actions needed after any change.

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.

The next sentence is the thesis statement of the entire book. Each of its three key words—"concisely," "efficiently," and "dependencies"—maps directly to a major component of `mk`: the DSL (conciseness), the job scheduler (efficiency), and the dependency graph (the central data structure). The rest of this chapter unpacks each word in turn.

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

[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.

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.

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 author's observation here is subtle and important: the shell is already half the solution. It provides concise syntax for launching programs, piping, and redirection—exactly what recipes need. So `mk`'s DSL does not reinvent command execution; it only adds the missing piece: a way to declare which files depend on which other files. This "minimal extension" philosophy—add only what the shell cannot express—is a recurring design principle throughout `mk`.

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 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 rule semantics look deceptively simple: compare modification times. But as the author's TODO notes hint, the full story is more subtle. The comparison is not just between a target and its immediate prerequisites—`mk` must first recursively ensure that the prerequisites themselves are up-to-date. This is why a DFS is needed (Section 2.1.2): you must reach the leaves and work your way back up, because a target can appear "up-to-date" with its direct prerequisites while a transitive dependency deep in the graph has changed.

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 *embedded* inside `mk`'s DSL. This is similar to other UNIX DSLs, for example Lex [LS79] and Yacc [Joh79]. 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*, 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*, 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 most variables from the environment, so you can also use variables such as `$HOME` in your `mkfile`.

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

The distinction below—that `mk` variables are really constants—has deep implications for the implementation. Because `mk` must know all variable values before building the dependency graph, it expands variables in targets and prerequisites at parse time. Recipe variables, however, are expanded later by the shell (via environment export), which is why `$target` and `$stem` work in recipes without `mk` doing string substitution itself. This two-phase expansion is the source of much confusion in Make (with its `=` vs. `:=` distinction); `mk` sidesteps the problem by making all definitions single-assignment.

Note that the term "variable" in the context of `mk` is slightly misleading. Indeed, in `mk`, variables are constants. Variable definitions are more binding definitions. Indeed, `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 files. Those files 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, `$objtype` is a special Plan 9 environment variable containing the type of architecture of the current machine (e.g., `arm`, `386`). You can then define for each architecture a specific `mkfile` with variables such as `$CC` or `$LD` containing the name of the native compiler and linker. Appendix D.2.1 presents such an `mkfile` for the ARM architecture. It is good practice to include `/$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. 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.

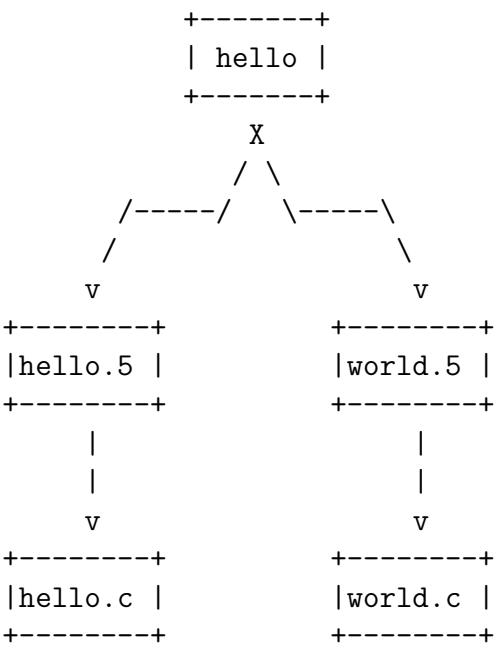


Figure 2.1: Graph of dependencies for `hello`.

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 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

The choice of DFS over BFS is not arbitrary. A breadth-first 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.

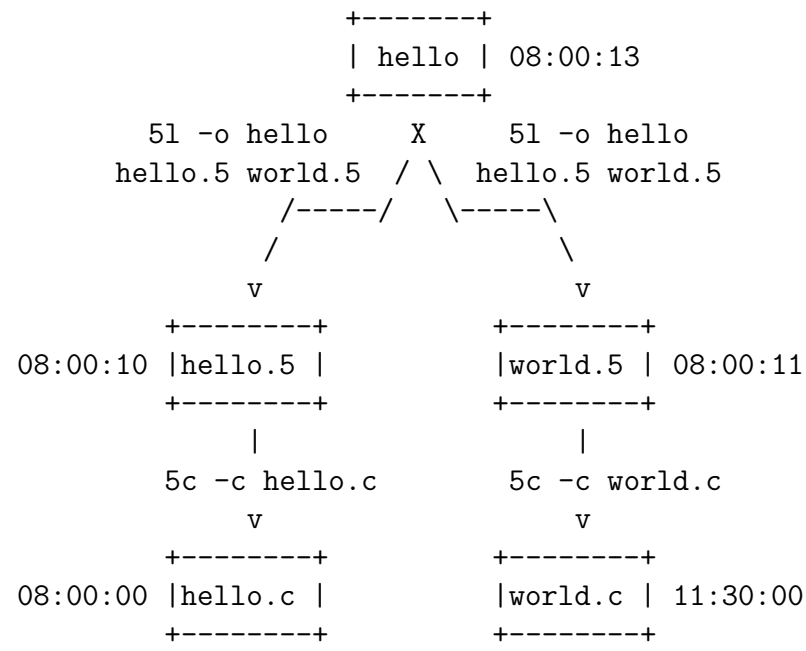


Figure 2.2: Graph of dependencies for `hello`, with labels.

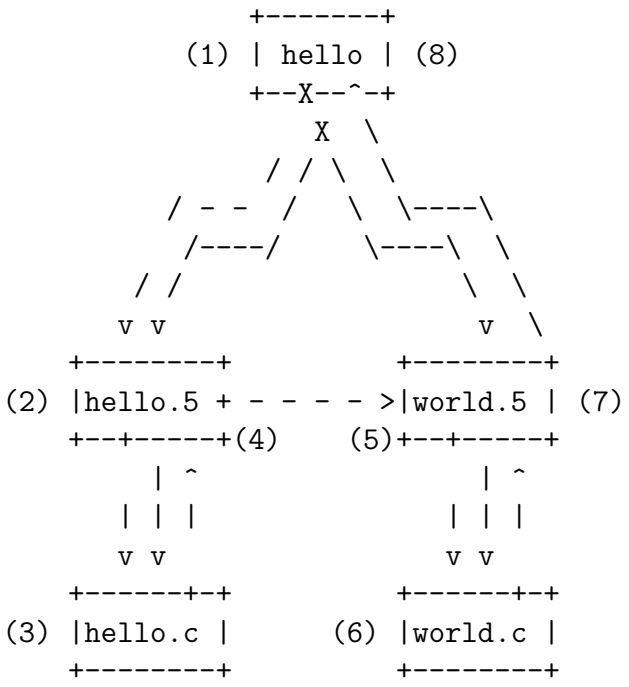


Figure 2.3: Depth-first search traversal of the graph dependencies for `hello`.

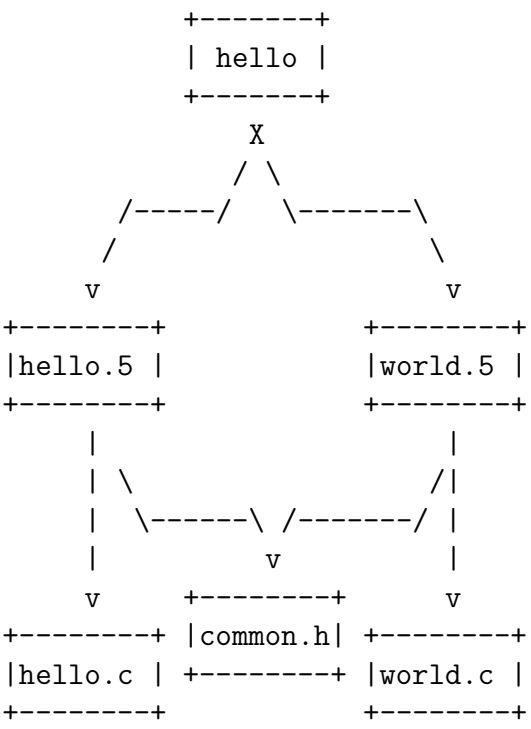


Figure 2.4: Graph of dependencies for `hello` with `common.h`.

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 2.2, 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.

A direct acyclic graph

The jump from tree to DAG is where build systems earn their keep. In a tree, each file appears exactly once, so the DFS visits every node once. In a DAG, a shared file like `common.h` is reachable from multiple branches. A naive DFS would visit it multiple times, potentially triggering redundant rebuilds. This is why `mk` must track which nodes have already been visited—the `MADE`/`BeingMade` status labels introduced later in Section 2.1.3 serve double duty as both build-progress indicators and visited-node markers. In the previous examples, the graph was simply a tree. However, 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].

There are multiple ways to add the dependency to `common.h` from `hello.5` (and `world.5`) in the `mkfile`s 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
   ```

   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.

Approach (3) is elegant because it cleanly separates what to rebuild (the master rule with a recipe) from when to rebuild (additional recipe-less rules that contribute extra prerequisites). This separation is what makes automatic dependency generation practical: tools like `gcc -MM` can emit simple "target: header" lines without needing to know the compilation recipe, and `mk` merges them with the master rule at graph-construction time.

`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`.

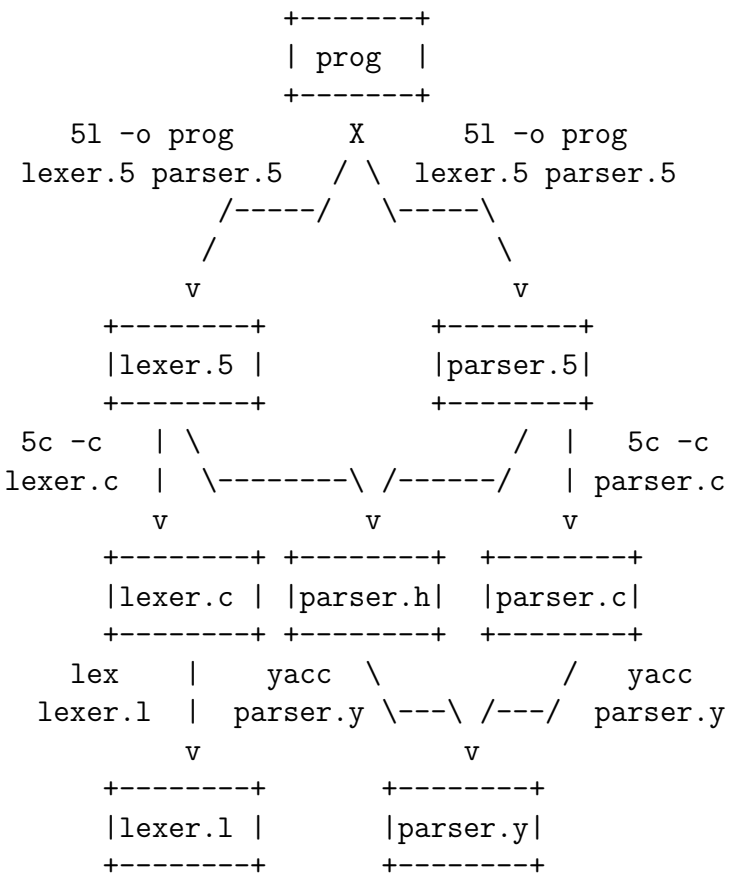


Figure 2.5: Graph with many-to-one dependencies.

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
```

The many-to-one problem is one of the trickiest aspects of build system design. A single command (`yacc`) produces multiple outputs, but the graph models dependencies per-file. If each output has its own rule, the build system has no way to know that both outputs come from the same invocation, so it runs the command twice. The solution—a single rule with multiple targets—requires `mk` to treat all targets of a rule as a unit: when any target is out of date, run the recipe once and update the modification times of all targets.

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.

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.

The job scheduler is the third pillar of `mk`, corresponding to the word "efficiently" in the thesis statement. Incrementality (only rebuilding what changed) is one dimension of efficiency; parallelism is another. The key insight is that the dependency graph already contains all the information needed for both: timestamps tell you what to rebuild, and the graph structure tells you what can run concurrently—two recipes are independent if neither target is an ancestor of the other.

[4] Section 6.6.1 presents the code to check for cycles in the graph of dependencies.
[5] Section 6.6.3 presents the code to check for the presence of ambiguities.

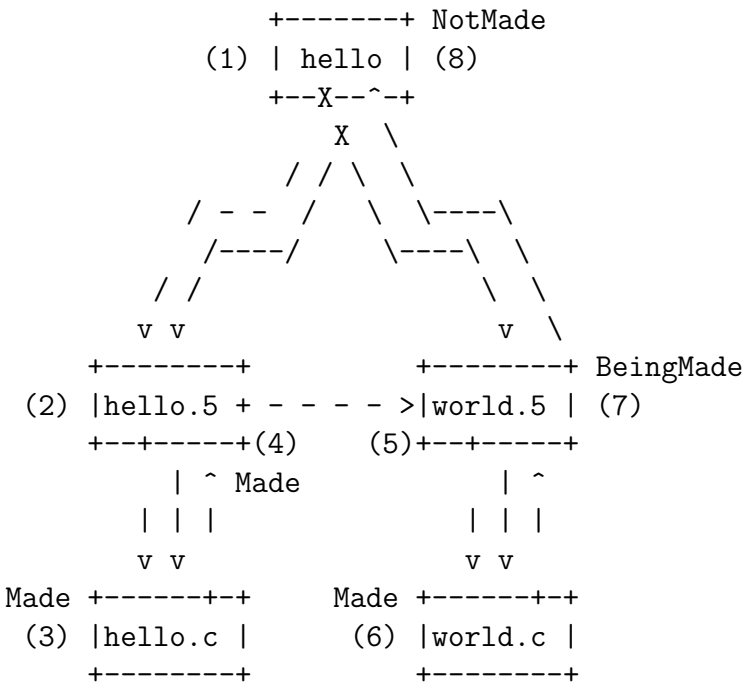


Figure 2.6: Graph of dependencies with building-status labels.

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 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.

The three-state machine below (`NotMade` → `BeingMade` → `Made`) is the key mechanism that ties the DFS to the job scheduler. Without `BeingMade`, the DFS would have to block until each recipe finishes—no parallelism. With it, the DFS can "fire and forget": mark a node `BeingMade`, enqueue its recipe, and continue exploring other branches. When the DFS encounters a `BeingMade` prerequisite, it knows to leave the parent as `NotMade`—the parent will be reconsidered in the next wave, after the prerequisite's job completes and transitions the node to `Made`.

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 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`. 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`.

The wave model is what trades latency for parallelism. A synchronous design (run the DFS once, blocking on each recipe in turn) is simpler but only ever uses one processor. The wave model lets `mk` launch every independent recipe in the current frontier, then wait for any one to finish, then launch whatever became eligible—which on a four-core machine can run four compiles concurrently with no extra bookkeeping in the rules. The cost of each wave is small: the DFS skips already-`Made` nodes and only re-examines the `BeingMade` frontier, so most of the work is done in the first wave.

`mk` will run the DFS in a loop multiple times until the root node is marked as `Made`. During each of those loops, 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 file 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`:

```
$ 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* 25⟩≡

```
OBJS=hello.5 world.5
CFLAGS=
LDFLAGS=

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

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

[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).

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 `mkfile`s, 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()`[55]) to extract the rules and meta rules in the file
2. *Build* the graph of dependencies (via `graph()`[80]) for a specific target given the rules extracted previously
3. *Find* outdated files in the graph (via `work()`[101c])
4. *Schedule* jobs (via `sched()`[111c]) that will run the shell recipes (via `execsh()`[192c]) to update the outdated files

Starting from the top of Figure 2.7, the function `main()`[47b], 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()`[57b]), which is then split in words, which are then analyzed to populate important globals such as `rules`[37b] and `metarules`[37e], which contain the lists of extracted rules. `parse()` also sets the global `target1`[51f] 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()`[99] (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.

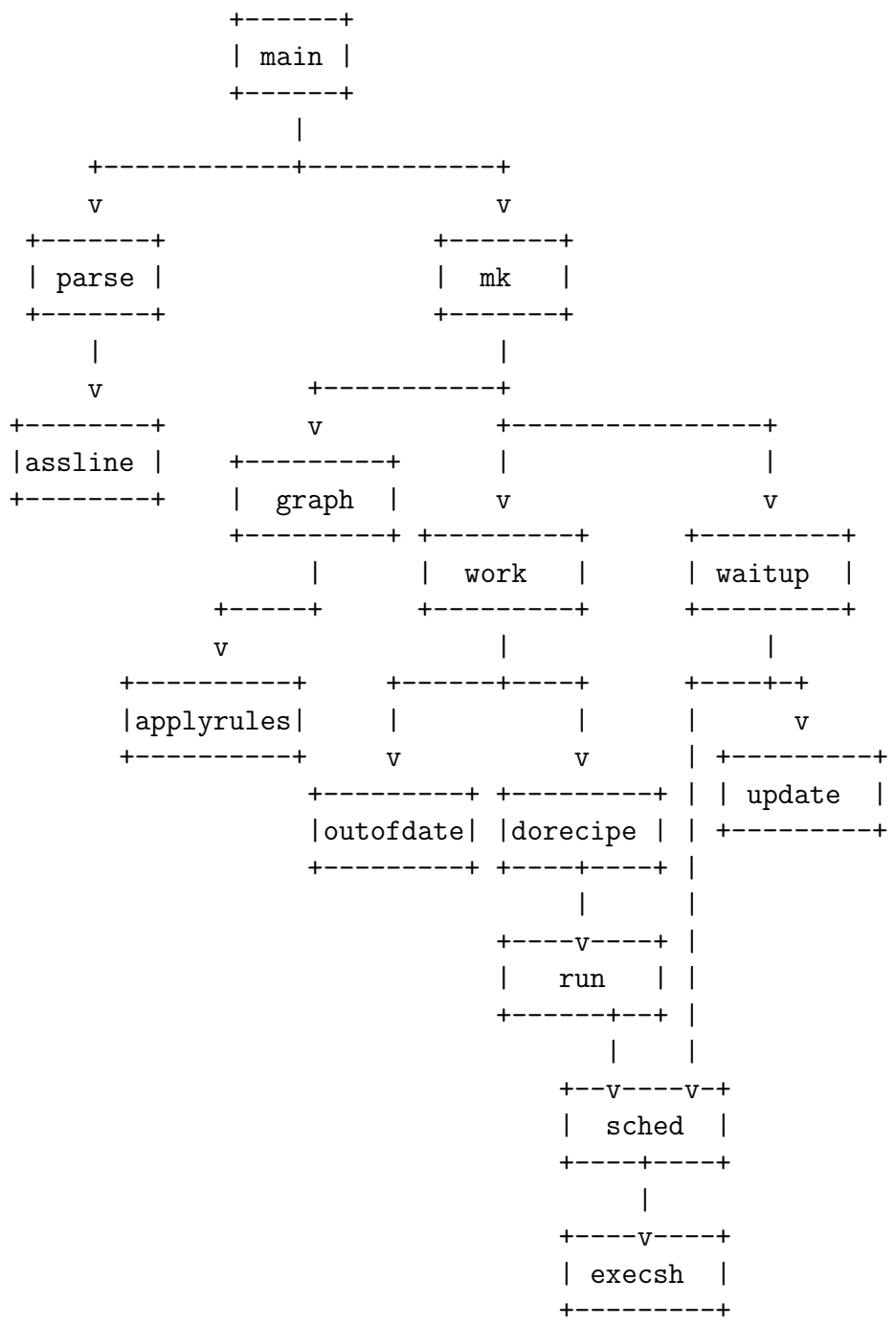


Figure 2.7: Control flow diagram of `mk`.

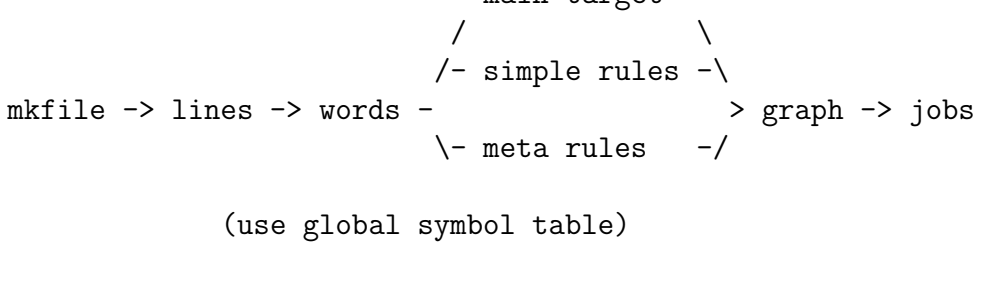


Figure 2.8: Data flow diagram of `mk`.

Function	Ch.	File	Entities	LOC
data structures and constants	3	mk.h	Symtab[30] Word[34b] Rule[36] Node[42a] Arc[43b] Job[44a]	375
symbol table and cache	3	symtab.c	hash[31b] symlook()[32a] symtraverse()[33a]	89
variables	3	var.c	setvar()[33b]	28
list of strings (words)	3	word.c	newword()[34d] wtos()[35d]	90
globals	3	globals.c	rules[37b] metarules[37e] jobs[45a]	63
adding rules	3	rule.c	addrule()[38b]	163
function prototypes	3	fns.h		140
entry point	4	main.c	main()[47b]	331
lexer	5	lex.c	assline()[57b] nextrune()[58a]	160
parser	5	parse.c	parse()[55] rhead()[61b] rbody()[73c]	418
escaping methods for rc	5	rc.c	escapetoken()[60b] charin()[62b] squote()[63b]	211
parsing and expanding variables	5	varsub.c	stow()[64b] varsub()[67d] varname()[68d]	439
building and checking the graph	6	graph.c	graph()[80] applyrules()[81] cyclechk()[89c]	455
pattern matching and substituting	6	match.c	match()[85a] subst()[86]	71
file and time management	6	file.c	timeof()[88a]	104
finding outdated files in the graph	7	mk.c	mk()[99] work()[101c] outofdate()[103b] update()[116b]	343
constructing a job	7	recipe.c	dorecipe()[104e]	197
scheduling jobs	8	run.c	run()[108a] RunEvent[109d] sched()[111c] waitup()[115c]	455
shell environment	9	env.c	buildenv()[125b] shellenv[120b] envinsert()[121a]	218
printing shell commands	10	shprint.c	shprint()[129a]	109
handling archives (libraries)	11	archive.c	atimeof()[154g]	178
dumpers	A	dumpers.c	dumpv()[169d] dumpr()[169c] dumpn()[170b]	105
memory management	C	utils.c	Malloc()[174a] Realloc()[174b]	34
string buffer management	C	bufblock.c	newbuf()[175d] insert()[176a]	107
Total				5445

Table 2.1: Chapters and associated mk source files.

graph() works by first creating a node for the target parameter, called the root of the graph, and by then calling applyrules()[81] 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()[104e] 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()[108a] (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[42e]). 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()`[115c] (at the right in Figure 2.7) to wait for those jobs to finish. Once a job is finished, `waitup()` calls `update()`[116b] 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()`[47b] 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

Before diving into the algorithms, it is worth understanding the data structures that `mk` operates on. A build system is fundamentally a graph problem, and the data structures below reflect that: rules describe edges, nodes represent files, and a symbol table maps names to values.

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`[181d] 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` 30⟩≡ (181d)

```
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* 31d⟩
};

Uses __anon_struct_2 30.

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 31a⟩≡ (181d)
```
enum Namespace {
    S_VAR, /* variable -> value */ // value is a list of words
    ⟨Sxxx cases 33c⟩
};
```

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 for a key 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 31b⟩≡ (184c)
```
// hash<(string * enum<Namespace>), 'a> (next = Symtab.next in bucket)
static Symtab *hash[NHASH];
```

Uses NHASH-1 31c.

⟨*constant* NHASH 31c⟩≡ (184c)
```
#define NHASH 4099
```

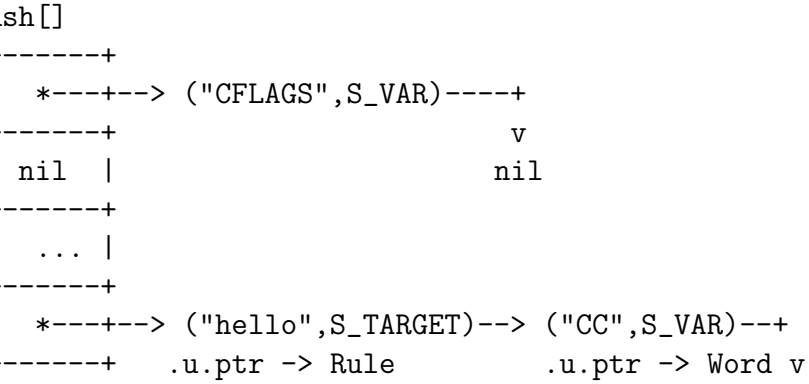
One way to implement a list of something in C is to embed in this something a next field pointing to the next element in the list:

⟨Symtab *extra fields* 31d⟩≡ (30)
```
// list<ref_own<Symtab>> (head = hash)
struct Symtab *next;
```

Uses Symtab 30.

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

A single bucket array can hold entries from all namespaces at once, because the namespace is folded into the hash key. 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:



```
|  ... |                                          nil
+------+
```

`symlook()`[32a] does linear chain walking, so collisions stay cheap as long as buckets are short. The chosen size `NHASH=4099` is prime to spread keys evenly, and the multiplier `HASHMUL=79` is folded byte by byte into the running hash. The lazy-create path (when `install` is non-`nil`) means that callers can use `symlook()` both as a lookup and as a defaulting insert in one call—a common idiom that avoids the lookup-then-insert race that would be necessary in a threaded implementation.

The main interface to the symbol table is the function `symlook()` below, which internally uses the global `hash`[31b]. `symlook()` takes a symbol name and a namespace, forming a full key, and returns the `Symtab`[30] in the symbol table `hash` associated with this key.

⟨*function* `symlook` 32a⟩≡ (184c)
```
Symtab*
symlook(char *sym, int space, void *install)
{
    Symtab *s;
    long h;
    ⟨symlook() other locals 32b⟩

    ⟨symlook() compute hash value h of sym 32c⟩

    // 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 32e⟩
}
```
Uses `hash-3` 31b.

⟨`symlook()` *other locals* 32b⟩≡ (32a)
```
char *p;
```

⟨`symlook()` *compute hash value* `h` *of* `sym` 32c⟩≡ (32a)
```
//h = hash(sym, space)
for(p = sym, h = space; *p; h += *p++)
    h *= HASHMUL;
if(h < 0)
    h = ~h;
h %= NHASH;
```
Uses `HASHMUL-2` 32d and `NHASH-1` 31c.

⟨*constant* `HASHMUL` 32d⟩≡ (184c)
```
#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* 32e⟩≡ (32a)
```
if(install == nil)
    return nil;

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() 174a and hash-3 31b.

Malloc()[174a], 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 33a⟩≡ (184c)
```
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 31c and hash-3 31b.

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

⟨*function* setvar 33b⟩≡ (185c)
```
void
setvar(char *name, void *value)
{
    symlook(name, S_VAR, value)->u.ptr = value;
}
```
Uses S_VAR 31a and symlook() 32a.

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[31a] for the variables set by the user (and environment), and S_INTERNAL for the special (internal) variables set by mk.

⟨Sxxx *cases* 33c⟩≡ (31a) 39b▷
```
S_INTERNAL, /* an internal mk variable (e.g., stem, target) */
```

Thus, to get the value of the special variable stem, call the symlook()[32a] function with the pair ("stem", S_INTERNAL).

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

⟨*global* specialvars 33d⟩≡ (186b)
```
static char *specialvars[] =
{
    "target",
    "prereq",
    "stem",

    ⟨specialvars other array elements 136g⟩
    0,
};
```

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

`specialvars`[33d] is used to initialize entries in the symbol table in `inithash()` (called from `main()`[47b]):

⟨*function* `inithash` 34a⟩≡ (186b)
```
void
inithash(void)
{
    char **p;

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

    readenv();    /* o.s. dependent */
}
```
Uses `S_INTERNAL` 33c, `specialvars-8` 33d, and `symlook()` 32a.

`readenv()`[192c] 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 lists 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` 34b⟩≡ (181d)
```
struct Word
{
    // ref_own<string>
    char   *s;

    // Extra
    ⟨Word extra fields 34c⟩
};
```

⟨`Word` *extra fields* 34c⟩≡ (34b)
```
// list<ref_own<Word>>
struct Word  *next;
```
Uses `Word` 34b.

`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` 34d⟩≡ (185b)
```
Word*
newword(char *s)
{
    Word *w;

    w = (Word *)Malloc(sizeof(Word));
    w->s = strdup(s);
    w->next = nil;
    return w;
}
```
Uses `Malloc()` 174a.

`freewords()` frees a list of words:

⟨*destructor* `freewords` 35a⟩≡ (185b)
```
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):

⟨*function* `addw` 35b⟩≡ (185b)
```
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()` 34d.

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

⟨*function* `wdup` 35c⟩≡ (185b)
```
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()` 34d.

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

⟨*function* `wtos` 35d⟩≡ (185b)
```
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()` 175e, `insert()` 176a, and `newbuf()` 175d.

`wtos()`[35d] relies on the `Bufblock`[174c] data structure described in Appendix C.2. `Bufblock` is an implementation of a *string buffer*. It implements efficiently string concatenation to avoid quadratic complexity when concatenating a set of strings together. The code for `Bufblock` is in `mk/bufblock.c`[184b], 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 concepts in a build system, and so the most important data structures 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` 36⟩≡ (181d)

```
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 37h⟩
    ⟨Rule debug fields 37a⟩

    // Extra
    ⟨Rule extra fields 37c⟩
};
```

`Rule.prereqs` contains a list of words, hence the use of a pointer to a `Word`[34b]. `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* 37a⟩≡ (36)
```
char*   file;  /* source file */
short   line;  /* source line */
```

Those fields will be useful when reporting errors in the `mkfile` to the user.

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` 37b⟩≡ (183)
```
// 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()`[38b] function).

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

⟨`Rule` *extra fields* 37c⟩≡ (36) 39c▷
```
// list<ref_own<Rule>> (head = rules | metarules)
struct Rule *next;
```
Uses `Rule` 36.

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

⟨*global* `lr` 37d⟩≡ (187d)
```
// 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` 37e⟩≡ (183)
```
// 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` 37f⟩≡ (187d)
```
// 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` 37g⟩≡ (181d)
```
#define PERCENT(ch) (((ch) == '%') || ((ch) == '&'))
```

The difference between those two special characters is explained in the manual page of `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`[37b], a meta rule contains also the `META` *rule attribute* in `Rule.attr`:

⟨`Rule` *other fields* 37h⟩≡ (36) 41d▷
```
// bitset<Rule_attr>
short   attr;  /* attributes */
```

⟨*enum* `Rule_attr` 38a⟩≡ (181d)
```
enum Rule_attr {
    META   = 0x0001,
    ⟨Rule_attr cases 135a⟩
};
```

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` 38b⟩≡ (187d)
```
void
addrule(char *target, Word *prereqs, char *recipe,
        Word *alltargets, int attr, int hline, char *prog)
{
    Rule *r = nil;
    ⟨addrule() other locals 39d⟩

    ⟨addrule() find if rule already exists, set reuse, update r 40c⟩

    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 41a⟩

    ⟨addrule() indexing r by target in S_TARGET 40a⟩

    ⟨addrule() if meta rule 39a⟩
    else {
        ⟨addrule() if simple rule 38c⟩
    }
}
```
Uses `Malloc()` 174a.

If the rule is a simple rule, `mk` populates `rules`[37b]:

⟨`addrule()` *if simple rule* 38c⟩≡ (38b)
```
⟨addrule() return if reuse, to not add the rule in a list 41b⟩
// else

// add_list(r, rules, lr)
if(rules == nil)
    rules = lr = r;
else {
    lr->next = r;
    lr = r;
```

```
}
```
Uses lr-23 37d and rules 37b.

If the rule is a meta rule, mk populates metarules[37e]. 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* 39a⟩≡ (38b)
```
if(charin(target, "%&") || (attr&REGEXP)){
    r->attr |= META;
    ⟨addrule() return if reuse, to not add the rule in a list 41b⟩
    // else
    ⟨addrule() if REGEXP attribute 135e⟩

    // add_list(r, metarules, lmr)
    if(metarules == nil)
        metarules = lmr = r;
    else {
        lmr->next = r;
        lmr = r;
    }
}
```
Uses META 38a, REGEXP 135a, charin() 62b, lmr-24 37f, and metarules 37e.

I will describe the function charin()[62b] and the rule attribute REGEXP[135a] 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

The Rule.chain field and the Rule.next field serve different purposes and are easy to confuse. Rule.next links all rules in the mkfile in order of appearance (a flat list). Rule.chain links only the rules that share the same target, forming a per-target chain stored in the S_TARGET namespace of the symbol table. When applyrules()[81] looks up a target, it follows the chain to find all rules for that target; typically only one (the master rule) has a recipe, while the others add extra prerequisites (e.g., from \texttt.depend files).

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* 39b⟩+≡ (31a) ◁33c 79a▷
```
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[36]:

⟨Rule *extra fields* 39c⟩+≡ (36) ◁37c
```
// list<ref<Rule>> (head = symlook(x, S_TARGET))
struct Rule *chain;  /* hashed per target */
```
Uses Rule 36.

Here is the code to update the symbol table in ¡[addrule()]¿:

⟨addrule() *other locals* 39d⟩≡ (38b) 40b▷
```
Symtab *sym;
Rule *rr;
```

⟨addrule() *indexing* r *by target in* S_TARGET 40a⟩≡ (38b)
```
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 39b and symlook() 32a.

Remember that the last parameter of `symlook()`[32a], 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() *other locals* 40b⟩+≡ (38b) ◁39d
```
bool reuse;
```

⟨addrule() *find if rule already exists, set* reuse*, update* r 40c⟩≡ (38b)
```
reuse = false;
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 39b and symlook() 32a.

Note that the code above relies on the indexing of rules in `S_TARGET`[39b] from the previous section.

⟨*function* rcmp 40d⟩≡ (187d)
```
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()`[38b] overwrites (reuses) a previous rule, the `Rule.next`[37c] field of this rule should not be modified. Otherwise, `Rule.next` should be set to `nil`:

⟨`addrule()` *set more fields* 41a⟩≡ (38b) 41e▷
```
if(!reuse){
    r->next = nil;
}
```

⟨`addrule()` *return if reuse, to not add the rule in a list* 41b⟩≡ (39a 38c)
```
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` 41c⟩≡ (186c)
```
void
addrules(Word *targets, Word *prereqs, char *recipe,
         int attr, int hline, char *prog)
{
    Word *w;

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

    ⟨addrules() set target1 73b⟩
    for(w = targets; w; w = w->next)
        addrule(w->s, prereqs, recipe, targets, attr, hline, prog);
}
```
Uses `addrule()` 38b.

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()`[41c] passes also the set of targets in the fourth argument to `addrule()`[38b]. This argument is then stored in a special field in the rule:

⟨`Rule` *other fields* 41d⟩+≡ (36) ◁37h 91a▷
```
// ref<list<ref_own<string>>
Word   *alltargets; /* all the targets */
```

⟨`addrule()` *set more fields* 41e⟩+≡ (38b) ◁41a 54b▷
```
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` 42a⟩≡ (181d)
```
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 43a⟩
    ⟨Node other fields 42c⟩

    // Extra
    ⟨Node extra fields 44b⟩
};
```

The function below constructs a new node:

⟨*constructor* `newnode` 42b⟩≡ (190a)
```
static Node*
newnode(char *name)
{
    Node *node;

    node = (Node *)Malloc(sizeof(Node));
    ⟨newnode() update node cache 87b⟩

    node->name = name;
    // call to timeof()!
    node->time = timeof(name, false);
    node->flags = 0;
    ⟨newnode() adjust flags of node 95c⟩

    node->arcs = nil;
    node->next = nil;
    ⟨newnode() debug 171i⟩
    return node;
}
```

Uses `Malloc()` 174a and `timeof()` 88a.

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

⟨`Node` *other fields* 42c⟩≡ (42a)
```
// bitset<enum<Node_flag>>
ushort  flags;
```

⟨*enum* `Node_flag` 42d⟩≡ (181d)
```
enum Node_flag {
    ⟨Node_flag cases 42e⟩
};
```

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* 42e⟩≡ (42d) 89b ▷
```
NOTMADE    = 0x0020,
BEINGMADE  = 0x0040,
MADE       = 0x0080,
```

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

3.4.2 Arc

A `Node`[42a] contains also a set of arcs where each arc contains a pointer to another node (a prerequisite):

⟨`Node` *arcs field* 43a⟩≡ (42a)
```
// list<ref_own<Arc>> (next = Arc.next)
Arc  *arcs;
```

⟨*struct* `Arc` 43b⟩≡ (181d)
```
struct Arc
{
    // option<ref<Node>>, the other node in the arc (the dependency)
    struct Node *n;
    // ref<Rule>, to generate the target node from the dependent node
    Rule *r;

    ⟨Arc other fields 43d⟩

    //Extra
    ⟨Arc extra fields 43c⟩
};
```
Uses `Node` 42a.

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* 43c⟩≡ (43b)
```
// list<ref_own<arc> (head = Node.arcs)
struct Arc *next;
```
Uses `Arc` 43b.

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* 43d⟩≡ (43b) 94c▷
```
// 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` 43e⟩≡ (190a)
```
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 94d⟩
    return a;
}
```

Uses Malloc() 174a.

3.5 Jobs

A Job[44a] captures everything needed to run a recipe and update the graph afterward. It carries more than just the recipe: the list of target nodes (n) is needed so that after the recipe completes, mk can mark all those nodes MADE[42e] and update their modification times. The word lists (t for targets, p for prerequisites) are needed to set the special shell variables \$target, \$prereq, and \$stem before invoking the recipe. The separation between the node list and the word lists exists because the same information is needed in two forms: nodes for graph updates, words for the shell environment.

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 44a⟩≡ (181d)

```
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 136f⟩

    // Extra
    ⟨Job extra fields 45b⟩
};
```

The list of target nodes of a job are chained together through an extra field in Node[42a]:

⟨Node *extra fields* 44b⟩≡ (42a)

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

Uses Node 42a.

The function below constructs a new job:

⟨*constructor* newjob 44c⟩≡ (191a)

```
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 152h⟩

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

Uses Malloc() 174a.

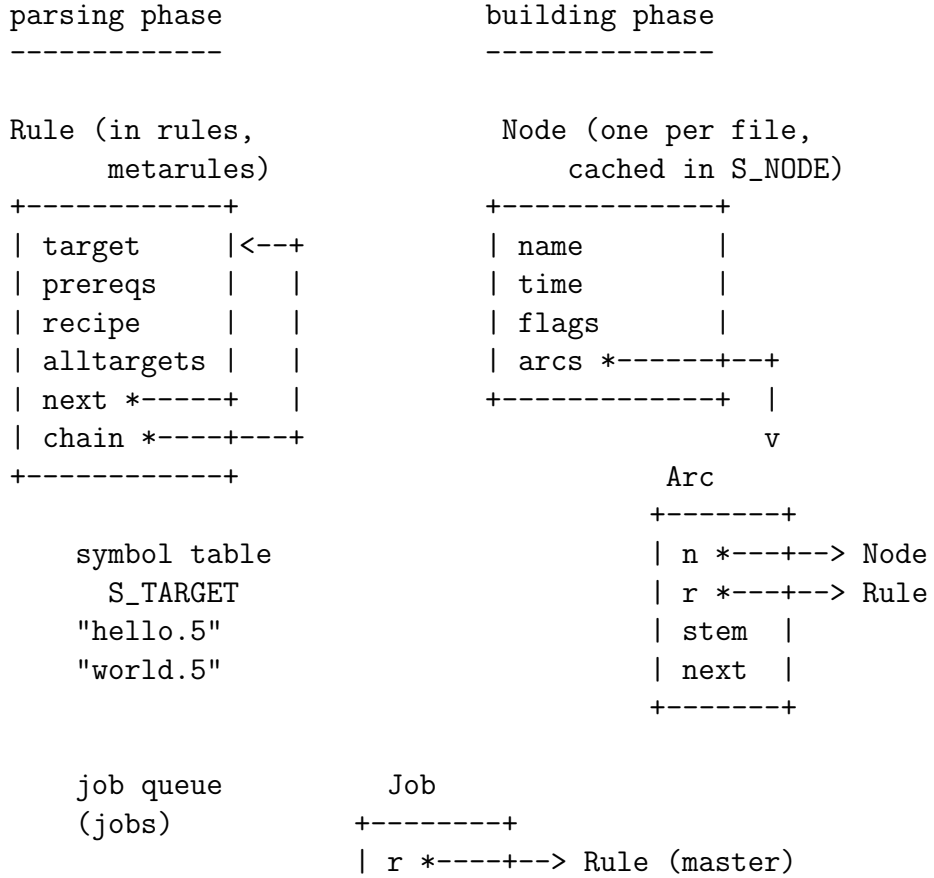
This job can then be added to the job queue, which is stored in the global jobs:

⟨*global* jobs 45a⟩≡ (183)
```
// list<ref_own<jobs>> (next = Job.next)
Job *jobs;
```

⟨Job *extra fields* 45b⟩≡ (44a)
```
// list<ref_own<Job>> (head = jobs)
struct Job *next;
```

Uses Job 44a.

It is worth pausing here to see how the core data structures fit together before main()[47b] starts using them. The diagram below shows the static side (Rule[36]s in rules[37b]/metarules[37e], indexed by S_TARGET[39b] in the symbol table) versus the dynamic side (Nodes and Arc[43b]s built once a target is requested, with Jobs carved out of that graph for the scheduler):



```
| n *----+--> Node list
| t,p,stem|    (targets, prereqs)
+--------+
```

The split is deliberate. `Rule`s describe the `mkfile` verbatim and never change after parsing. `Node`s and `Arc`s are an instantiation of those rules against the actual 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, but owns nothing itself.

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.

main()[47b] is mostly a sequencer. It does almost no real work itself; it threads four phases together: initialize globals, parse the mkfile (filling rules[37b], metarules[37e], and the symbol table), build and walk the graph for each target with mk()[99], and optionally print profiling stats before exiting. The interesting subtlety is that command-line arguments are processed in three passes (flags, then var=value assignments, then the remaining target names), with $MKFLAGS and $MKARGS saved before the var=value entries are nulled out so recursive mk calls can inherit the original invocation.

4.1 main() skeleton

⟨*global* version 47a⟩≡ (192b)
```
static char *version = "@(#)mk general release 4 (plan 9)";
```
Uses version-10 47a.

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

⟨*function* main 47b⟩≡ (192b)
```
void
main(int argc, char **argv)
{
    ⟨main() locals 49b⟩
    Symtab* sym;

    // Initializing

    ⟨main() initializations 48b⟩

    // Parsing the mkfile

    ⟨main() parsing mkfile, call parse() 51c⟩

    // Building the graph, finding out-of-date files

    ⟨main() initializations before building 119a⟩

    //pad-ext: MKSHELL environment var to specify the path to rc
    //LATER: allow also to change the shell from rc to sh (or something else)
    sym = symlook("MKSHELL", S_VAR, 0);
    if(sym != nil) {
      w = (Word*) sym->u.value;
      if(w != nil && w->s != nil) {
```

```
            shell->shell = w->s;
          }
        }

        ⟨main() setting the targets, call mk() 51e⟩

        // Reporting (optional)

        ⟨main() print profiling stats if uflag 134b⟩

        // Exiting

        exits(nil);
    }
```

Uses S_VAR 31a and symlook() 32a.

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 48a⟩≡                                                    (183)
  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 48b⟩≡                                  (47b) 48c▷
  Binit(&bout, STDOUT, OWRITE);
```

Uses bout 48a.

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 48c⟩+≡                                 (47b) ◁48b
  ⟨main() argv processing part 1, -xxx 48d⟩
  ⟨main() setup profiling 133e⟩
  inithash();
  ⟨main() argv processing part 2, xxx=yyy 49c⟩
  ⟨main() set variables for recursive mk 164b⟩
  ⟨main() argv processing part 3, skip xxx=yyy 50d⟩
  ⟨main() profile initializations 133f⟩
```

Uses inithash() 34a.

inithash()[34a] 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 48d⟩≡                            (48c)
  USED(argc);
  for(argv++; *argv && (**argv == '-'); argv++)
  {
      ⟨main() add argv[0] in buf 163e⟩

      switch(argv[0][1]) {
```

```
        ⟨main() -xxx switch cases 51b⟩
        default:
            badusage();
        }
    }
```

Uses badusage() 49a.

⟨*function* badusage 49a⟩≡ (192b)

```
    void
    badusage(void)
    {

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

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.

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()[55] 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* 49b⟩≡ (47b) 51a ▷

```
    char *temp = nil;
    fdt tfd = -1;
    Biobuf tb;
    int i;
```

⟨main() *argv processing part 2, xxx=yyy* 49c⟩≡ (48c)

```
    for(i = 0; argv[i]; i++)
      if(utfrune(argv[i], '=')){
        ⟨main() add argv[i] in buf 164a⟩

        ⟨main() create temporary file if not exist yet and set tb 50a⟩
        Bprint(&tb, "%s\n", argv[i]);
        ⟨main() mark argv[i] for skipping 50e⟩
      }

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

Uses parse() 55.

`utfrune()`, called above, is a function looking for a certain character in a string. However, the character and the string use a particular format. Indeed, `utfrune()` looks for a rune in a sequence of UTF-8 encoded characters. In Plan 9, a *rune* is the term used to represent a Unicode character. There are multiple ways to encode a Unicode character (a rune) in a sequence of bytes. Plan 9 uses the UTF-8[1] encoding, hence the use of the `utfrune()` function above. Indeed, Plan 9 supports filenames using Unicode characters, as well as command-line arguments using Unicode characters, as long as they are encoded with the UTF-8 format (see the LIBCORE book [Pad16c] for more information on Unicode and `utfrune()`).

The last argument to `parse()` above is a boolean indicating whether `parse()` should accept definitions overwriting previous definitions. Obviously, in this case `main()`[47b] passes `true` to `parse()` in the code above.

⟨`main()` *create temporary file if not exist yet and set* `tb` 50a⟩≡ (49c)
```
if(tfd < 0){
    temp = maketmp();
    ⟨main() when creating temporary file, sanity check temp 50b⟩
    tfd = create(temp, ORDWR, 0600);
    ⟨main() when creating temporary file, sanity check tfd 50c⟩
    Binit(&tb, tfd, OWRITE);
}
```

The code above omits the error management code shown below:

⟨`main()` *when creating temporary file, sanity check temp* 50b⟩≡ (50a)
```
if(temp == nil) {
    perror("temp file");
    Exit();
}
```

⟨`main()` *when creating temporary file, sanity check tfd* 50c⟩≡ (50a)
```
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* 50d⟩≡ (48c)
```
/* 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* 50e⟩≡ (49c)
```
/*
 *   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.

[1] The popular UTF-8 encoding was actually designed by Rob Pike and Ken Thompson, two of the designers of Plan 9. See http://doc.cat-v.org/bell_labs/utf-8_history for the history of UTF-8.

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`:

⟨`main()` *locals* 51a⟩+≡ (47b) ◁49b 52a▷
```
char *f = nil;
```

⟨`main()` *-xxx switch cases* 51b⟩≡ (48d) 52d▷
```
case 'f':
    if(*++argv == nil)
        badusage();
    f = *argv;
    ⟨main() add argv[0] in buf 163e⟩
    break;
```
Uses `badusage()` 49a.

The `parse()`[55] function, called below, will process the `mkfile` (or another file if `-f` was used) and modify `rules`[37b], `metarules`[37e], 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()` 51c⟩≡ (47b)
```
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) 169a⟩
```
Uses `MKFILE-9` 51d and `parse()` 55.

⟨*constant* `MKFILE` 51d⟩≡ (192b)
```
#define MKFILE  "mkfile"
```

4.6 Building the target(s)

Once `parse()`[55] processed the `mkfile` and modified some globals, `mk` is ready to build a target by calling `mk`[99]. 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()` 51e⟩≡ (47b)
```
if(*argv == nil){
    ⟨main() when no target arguments 52b⟩
} else {
    ⟨main() if sequential mode and target arguments given 52e⟩
    else {
        ⟨main() parallel mode and target arguments given 52f⟩
    }
}
```

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` 51f⟩≡ (183)
```
Word *target1;
```

Section 5.2 contains the code in `parse()`[55] 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()`[99]:

⟨`main()` *locals* 52a⟩+≡ (47b) ◁51a 52c▷
```
Word *w;
```

⟨`main()` *when no target arguments* 52b⟩≡ (51e)
```
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()` 99 and `target1` 51f.

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* 52c⟩+≡ (47b) ◁52a 132d▷
```
bool sflag = false;
```

⟨`main()` *-xxx switch cases* 52d⟩+≡ (48d) ◁51b 131c▷
```
case 's':
    sflag = true;
    break;
```

⟨`main()` *if sequential mode and target arguments given* 52e⟩≡ (51e)
```
if(sflag){
    for(; *argv; argv++)
        if(**argv)
            mk(*argv);
}
```
Uses `mk()` 99.

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()`[99] with this arbitrary string, which will trigger the DFS to build its prerequisites in parallel.

⟨`main()` *parallel mode and target arguments given* 52f⟩≡ (51e)
```
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 146a, addrules() 41c, mk() 99, mkinline 54c, and newword() 34d.

You can see in the code above a few calls to functions I described in Chapter 3, for instance, `newword()`[34d] and `addrules()`[41c]. I will describe `mk()`, called above, the most important function of `mk`, in Chapter 7.

The `VIR`[146a] 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`.

Chapter 5

Parsing the `mkfile`

Now that you have seen `main()`[47b], I can explain the different components in the building pipeline, starting in this chapter with the parsing functions.

I mentioned before `parse()`[55], which takes a path to an `mkfile` as a parameter, parses this `mkfile` to identify rules, meta rules, and definitions, and stores those entities in different globals (`rules`[37b], `metarules`[37e], and the symbol table `hash`[31b]). `parse()` is a complex function that relies on many other functions to scan a file, identify rules, expand variables, process included files, or define variables, as explained in the following sections.

Why a hand-written line-oriented parser instead of a yacc grammar like Unix Make used? Three reasons. First, recipes are opaque strings handed verbatim to the shell—they have their own escaping and tokenization that does not fit a uniform grammar. Second, `mk`'s syntax is context-sensitive at the line level: a `'#'` starts a comment in normal text but is just a character inside a quoted filename, and the `':'` `'='` `'<'` separators have meaning only in a rule head, not inside variable values. Third, the line is the natural unit for variable expansion, which must happen at parse time so the dependency graph can be built statically (see Chapter 6). The result is a parser organized around two layers: `assline()`[57b] assembles a logical line from the input (handling escaped newlines, comments, and quoted regions), and `rhead()`[61b] then dispatches on the separator character to populate rules, definitions, or inclusions.

5.1 `parse()`

Before showing the code of `parse()`[55], 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` 54a⟩≡ (183)
```
char *infile;
```

This global is used in `addrule()`[38b]:

⟨`addrule()` *set more fields* 54b⟩+≡ (38b) ◁41e 91c▷
```
r->file = infile;
```
Uses `infile` 54a.

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

⟨*global* `mkinline` 54c⟩≡ (183)
```
int mkinline;
```

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

⟨*function* `SYNERR` 54d⟩≡ (181d)
```
#define SYNERR(l) (fprint(STDERR, "mk: %s:%d: syntax error; ", \
                          infile, ((l)>=0)? (l) : mkinline))
```

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

⟨*function* `parse` 55⟩≡ (186c)

```
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 71b⟩

    ⟨parse() sanity check fd 56⟩
    ⟨parse() start, push 76e⟩

    // 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 57a⟩
        }
    }
    close(fd);
    freebuf(buf);
    ⟨parse() end, pop 76f⟩
}
```

Uses `assline()` 57b, `freebuf()` 175e, `infile` 54a, `mkinline` 54c, and `newbuf()` 175d.

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()`[57b]). 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()`[61b] 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`[37b] and `metarules`[37e], or the symbol table. I will describe in the next sections the cases of the `switch` above.

To answer the author's thinking note above, here is the text-to-`Word`[34b]-list pipeline that each meaningful line of the `mkfile` traverses. I use the line `OFILES=‘{echo foo.$O bar.$O} $EXTRA` as a running example, assuming `O=5` and `EXTRA=baz.5` are already in the symbol table:

```
bytes on disk
    |  Bgetrune (libbio, UTF-8 decoding)
    v
runes, escaped-\n-and-# stripped     <-- nextrune()
    |
    v
logical line (one Bufblock)          <-- assline()
    "OFILES=‘{echo foo.$O bar.$O} $EXTRA"
    |
    v                                <-- rhead()
head = "OFILES"     sep = '='    tail = "‘{echo foo.$O bar.$O} $EXTRA"
    |                                  |
    |                                  | stow() --> nextword() loop
    |                                  v
    |                          case '‘':  bquote() forks a shell,
    |                          captures   "foo.5 bar.5"
    |                          case '$':  varsub() looks up EXTRA,
    |                          returns    {baz.5}
    |                                  |
    |                                  v
    |                          Word list: [foo.5] -> [bar.5] -> [baz.5]
    |                                  |
    +----------------------------------+
                       |
                       v
              symbol table: S_VAR["OFILES"] = {foo.5, bar.5, baz.5}
```

Two observations are worth making about this pipeline before I dive into the individual stages. First, the split between the line layer (everything above `rhead()`) and the word layer (everything below) is what lets `mk` reuse `assline()` / `nextrune()`[58a] unchanged for rules, variable definitions, and includes: only the per-line action in `parse()` differs. Second, variable and backquote expansion happens inside `nextword()`[65], not as a separate preprocessing pass, because a single `'$'` can expand to several words—so the expander must be able to push multiple `Word`s back into the caller's list, which is easier if expansion lives inside the word splitter than if it tries to rewrite the raw line.

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

⟨`parse()` *sanity check fd* 56⟩≡ (55)

```
if(fd < 0){
    perror(f);
    Exit();
}
```

⟨parse() *switch rhead cases* 57a⟩≡ (55) 73a▷
```
default:
    SYNERR(hline);
    fprint(STDERR, "expected one of :<=\n");
    Exit();
    break;
```
Uses SYNERR 54d.

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 57b⟩≡ (188a)
```
/*
 * Assemble a line skipping blank lines, comments, and eliding
 * escaped newlines
 */
bool
assline(Biobuf *bp, Bufblock *buf)
{
    int c;
    ⟨assline() other locals 59a⟩

    resetbuf(buf);
    while ((c = nextrune(bp, true)) >= 0){
        switch(c)
        {
        case '\n':
            if (!isempty(buf)) {
                insert(buf, '\0');
                return true;
            }
            break;  /* skip empty lines */
        ⟨assline() switch character cases 59b⟩
        default:
            rinsert(buf, c);
            break;
        }
    }
eof:
    insert(buf, '\0');
    return *(bufcontent(buf)) != '\0';
}
```
Uses bufcontent 175h, insert() 176a, isempty 175f, nextrune() 58a, resetbuf 175g, and rinsert() 176b.

insert()[176a], rinsert()[176b] (for "Rune insert"), resetbuf()[175g], bufcontent()[175h], and isempty()[175f], called above, are all functions (or macros) operating on a string buffer and are described in Appendix C.2.

The code of assline()[57b] may look trivial, but as its (inappropriate) name suggests, assline() does not just read a line: it assembles a line. Indeed, if the current line is a comment, assline() will skip the line to return the next meaningful line to parse()[55]. Indeed, 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 '#'. Moreover, assline() handles also blank lines, escaped newlines, and certain quoted characters, as explained in the following sections.

[1] This function has a confusing name.

Escaped newline, `nextrune()`

`assline()`[57b] 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` 58a⟩≡ (188a)
```
/*
 * 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 58c⟩
        ⟨nextrune() handle mkinline 58b⟩
        return c;
    }
}
```

⟨`nextrune()` *handle mkinline* 58b⟩≡ (58a)
```
if (c == '\n')
    mkinline++;
```
Uses `mkinline` 54c.

`nextrune()`[58a] 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* 58c⟩≡ (58a)
```
if (c == '\\') {
    if (Bgetrune(bp) == '\n') {
        // an escaped newline!
        mkinline++;
        if (elide)
            continue;
        // else
        return ' ';
    }
    // else, it was just \
    Bungetrune(bp);
}
```
Uses `mkinline` 54c.

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. This is one of the reasons `mk` uses the `libbio` library instead of the reading functions of the C library.

Comments

As I mentioned before, `assline()`[57b] recognizes and skips comments:

⟨`assline()` *other locals* 59a⟩≡ (57b)
```
int prevc;
```

⟨`assline()` *switch character cases* 59b⟩≡ (57b) 60a▷
```
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 59e⟩
    ⟨assline() when processing comments, if not only comment on the line 59c⟩
    // else, skip lines with only a comment
    break;
```
Uses `mkinline` 54c.

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* 59c⟩≡ (59b)
```
if (!isempty(buf)) {
    insert(buf, '\0');
    return true;
}
```
Uses `insert()` 176a and `isempty` 175f.

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

⟨*example of escaped newline* 59d⟩≡
```
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* 59e⟩≡ (59b)
```
if (prevc == '\\')
    break;  /* propagate escaped newlines??*/
```

Quoted characters

Assembling a line sounds like an easy ask, but as you have just seen `assline()`[57b] is not trivial: it must handle escaped newlines (through `nextrune()`[58a]), 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. 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* 60a⟩+≡ (57b) ◁59b 137g▷

```
case '\'':
case '"':
case '\\':
    rinsert(buf, c);
    if (escapetoken(bp, buf, true, c) == ERROR_0)
        Exit();
    break;
```

Uses `escapetoken()` 60b and `rinsert()` 176b.

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`. 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` 60b⟩≡ (185a)

```
/*
 * 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;

    while((c = nextrune(bp, false)) > 0){
        if(c == '\''){
            if(preserve)
                rinsert(buf, c);
            ⟨escapetoken() return, unless double quote 61a⟩
        }
        // else
        rinsert(buf, c);
    }
    // must have reached EOF
    SYNERR(line);
    fprint(STDERR, "missing closing %c\n", esc);
    return ERROR_0;
}
```

Uses `SYNERR` 54d, `mkinline` 54c, `nextrune()` 58a, and `rinsert()` 176b.

Of course, since `"'"` is now a special character reserved to quote special characters, how do you quote `'` itself? A common technique found in most 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 double quotes:

⟨escapetoken() *return, unless double quote* 61a⟩≡ (60b)
```
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()[55] assembled a line, it can analyze the line with rhead() 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 61b⟩≡ (186c)
```
static int
rhead(char *line, Word **h, Word **t,    int *attr, char **prog)
{
    char *p;
    int sep; // one of : = <
    ⟨rhead() other locals 75c⟩

    p = charin(line, ":=<");
    if(p == nil)
        return '?';

    sep = *p;
    *p++ = '\0';
    ⟨rhead() adjust sep if dynamic mkfile <| 140a⟩
    ⟨rhead() adjust attr and prog 75b⟩

    // potentially expand variables in head
    *h = stow(line);
    ⟨rhead() sanity check h 61c⟩
    // potentially expand variables in tail
    *t = stow(p);

    return sep;
}
```

rhead()[61b] relies on the function charin()[62b] 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()[64b] (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.

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* 61c⟩≡ (61b)
```
if(empty_words(*h) && sep != '<' && sep != '|') {
    SYNERR(mkinline-1);
```

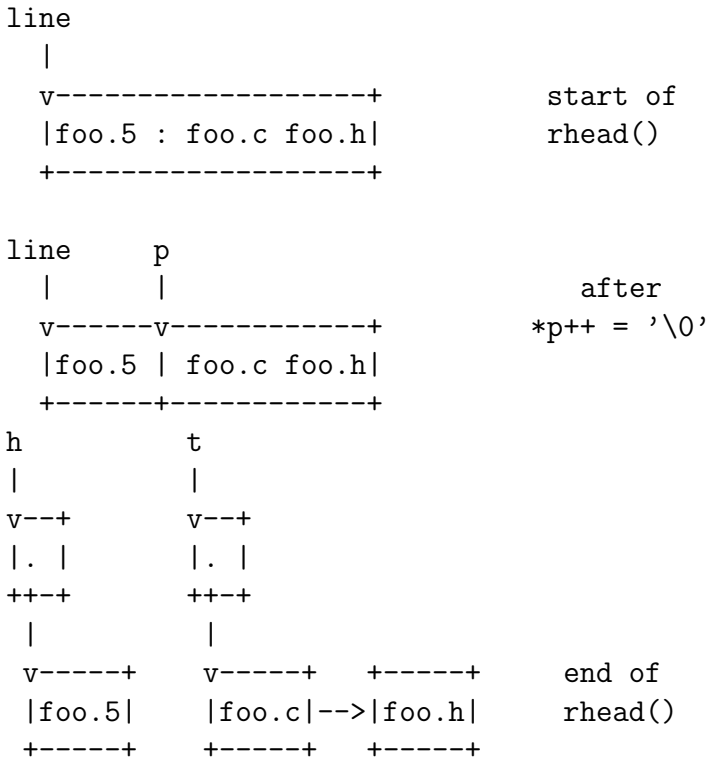


Figure 5.1: Evolution of local variables in `rhead()`.

```
    fprint(STDERR,
           "no var (or target) on left side of assignment (or rule)\n");
    Exit();
}
```

⟨*macro* `empty_words` 62a⟩≡ (181d)

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

Finding special characters, `charin()`

I mentioned the functions `charin()` a few times before. `addrule()`[38b] called `charin()` to check whether the target contains a special pattern character. `rhead()`[61b] called `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` 62b⟩≡ (185a)

```
/*
 * 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 63a⟩
        default:
            if(utfrune(pat, r) && !vargen)
                return cp;
            break;
        }
        cp += n;
    }
    ⟨charin() sanity check vargen 142a⟩
    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()`[58a] 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. 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()`[57b] needs special code to handle quoted strings (because they may contain a `'#'` that needs to be treated differently), `charin()`[62b] needs also special code to handle quoted strings, because they may contain one of the special characters `rhead()`[61b] is looking for (`':'`, `'='`, or `'<'`).

⟨`charin()` *switch rune cases* 63a⟩≡ (62b) 141b▷

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

Uses `squote()` 63b.

⟨*function* `squote` 63b⟩≡ (185a)

```
/*
 * 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 double quote 64a⟩
        }
        cp += n;
    }
    SYNERR(-1);  /* should never occur */
    fprint(STDERR, "missing closing '\n");
    return nil;
}
```

Uses `SYNERR` 54d.

⟨squote() *return, unless double quote* 64a⟩≡ (63b)
```
n += chartorune(&r, cp+n);
if(r != '\'')
    return cp;
// else, double '', continue while loop
```

5.1.3 Splitting a string in words: `stow()`

`stow()`[64b] and `nextword()`[65] do more than simple splitting on whitespace: they also expand variables (`\$FOO`) and backtick commands (`‘\cmd\`) inline. This is why `nextword()` returns a list of words rather than a single word: expanding a variable like `\$OFILES` can produce multiple words. The `restart` label in `nextword()` handles re-processing after expansion—a variable's value might itself contain variables that need further expansion. Crucially, variable expansion inside recipes is deferred: the recipe text is stored as a raw string and only expanded when the recipe is actually executed. This is necessary because special variables like `\$stem`, `\$target`, and `\$prereq` are only known at execution time, not at parse time.

After `rhead()`[61b] found the position of the special character in the line assembled by `assline()`[57b], `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` 64b⟩≡ (191b)
```
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 64c⟩
    return head;
}
```
Uses `nextword()` 65.

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* 64c⟩≡ (64b)
```
if (!head)
```

```
    head = newword("");
```
Uses newword() 34d.

An empty word is a list containing one word where the string is just the null character (see the code of newword()[34d]).

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 65⟩≡ (191b)
```
/*
 * 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 67b⟩

    buf = newbuf();

restart:
    head = lastw = nil;
    empty = true;
    ⟨nextword() skipping leading white space 66a⟩

    while(*cp){
        cp += chartorune(&r, cp);
        switch(r)
        {
        case ' ':
        case '\t':
        case '\n':
            goto out;
        ⟨nextword() switch rune cases 66b⟩
        default:
            empty = false;
            rinsert(buf, r);
            break;
        }
    }
out:
    *s = cp;
    if(!isempty(buf)){
        ⟨nextword() when buffer not empty, if there was already an head 71a⟩
         else {
            insert(buf, '\0');
```

```
            head = newword(buf->start);
        }
    }
    freebuf(buf);
    return head;
}
```

Uses freebuf() 175e, insert() 176a, isempty 175f, newbuf() 175d, newword() 34d, and rinsert() 176b.

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()[65].

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

⟨nextword() *skipping leading white space* 66a⟩≡ (65)

```
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()[65] must handle quoted strings. As opposed to assline()[57b], which inputs a quoted string, or charin()[62b], which skips over a quoted string, nextword() expands a quoted string and stores the expansion in the buffer b. Indeed, stow()[64b] and nextword() are the last steps in the parsing phase; there is no need to keep the quotes (or double quotes inside those quotes) anymore.

⟨nextword() *switch rune cases* 66b⟩≡ (65) 67c▷

```
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() 66c.

⟨*function* expandquote 66c⟩≡ (185a)

```
/*
 * 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 != '\'') {
        rinsert(buf, r);
        return s;
    }

    while(*s){
        s += chartorune(&r, s);
        if(r == '\'') {
            ⟨expandquote() return, unless double quote 67a⟩
        }
```

```
        rinsert(buf, r);
    }
    return nil;
}
```
Uses rinsert() 176b.

⟨expandquote() *return, unless double quote* 67a⟩≡ (66c)
```
if(*s == '\'')
    s++; // skip one of the double quotes
else
    return s;
```

Expanding variables

The expansion of variables in rules, variable definitions, and inclusions is done at parsing-time by `nextword()`[65], 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* 67b⟩≡ (65)
```
// list<ref_own<Word>
Word *w;
```

This is why `nextword()` returns a list of words, and why `stow()`[64b] 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()`[67d] 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* 67c⟩+≡ (65) ◁66b
```
case '$':
    w = varsub(&cp);
    ⟨nextword() when in variable case, if w is nil 68b⟩
    empty = false;

    ⟨nextword() when in variable case, if non-space chars before var 69c⟩
    ⟨nextword() when in variable case, if head is not empty 69d⟩
    else
        head = lastw = w;

    while(lastw->next)
        lastw = lastw->next;
    break;
```
Uses varsub() 67d.

`varsub()` in turn relies on `varname()`[68d] to extract the name of the variable from the string pointed by `s` (`varname()` also increments `s` by side effect), and `varmatch()`[69b] to grab the value of the variable from the symbol table:

⟨*function* varsub 67d⟩≡ (191b)
```
Word*
varsub(char **s)
{
    Bufblock *buf;
    Word *w;

    ⟨varsub() if variable starts with open brace 142b⟩
    // else

    buf = varname(s);
    ⟨varsub() sanity check buf 68a⟩
```

```
    w = varmatch(buf->start);

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

Uses freebuf() 175e and varname() 68d.

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

⟨varsub() *sanity check buf* 68a⟩≡ (67d)

```
if(buf == nil)
    return nil;
```

⟨nextword() *when in variable case, if w is nil* 68b⟩≡ (67c)

```
if(w == nil){
    ⟨nextword() when in variable case, if w is nil and no char before 68c⟩
    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* 68c⟩≡ (68b)

```
if(empty)
    goto restart;
```

varname() below returns a buffer containing the name of a variable in s, and increments s to point after the variable name:

⟨*function* varname 68d⟩≡ (191b)

```
/*
 * 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 69a⟩
    *s = cp;
    insert(buf, '\0');
    return buf;
}
```

Uses WORDCHR 68e, insert() 176a, newbuf() 175d, and rinsert() 176b.

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

⟨*function* WORDCHR 68e⟩≡ (181d)

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

⟨`varname()` *sanity check buf* 69a⟩≡ (68d)
```
if (isempty(buf)){
    SYNERR(-1);
    fprint(STDERR, "missing variable name <%s>\n", *s);
    freebuf(buf);
    return nil;
}
```
Uses `SYNERR` 54d, `freebuf()` 175e, and `isempty` 175f.

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

⟨*function* `varmatch` 69b⟩≡ (191b)
```
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, as at 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* 69c⟩≡ (67c)
```
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()` 176c, `insert()` 176a, `isempty` 175f, and `resetbuf` 175g.

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* 69d⟩≡ (67c)
```
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);
```

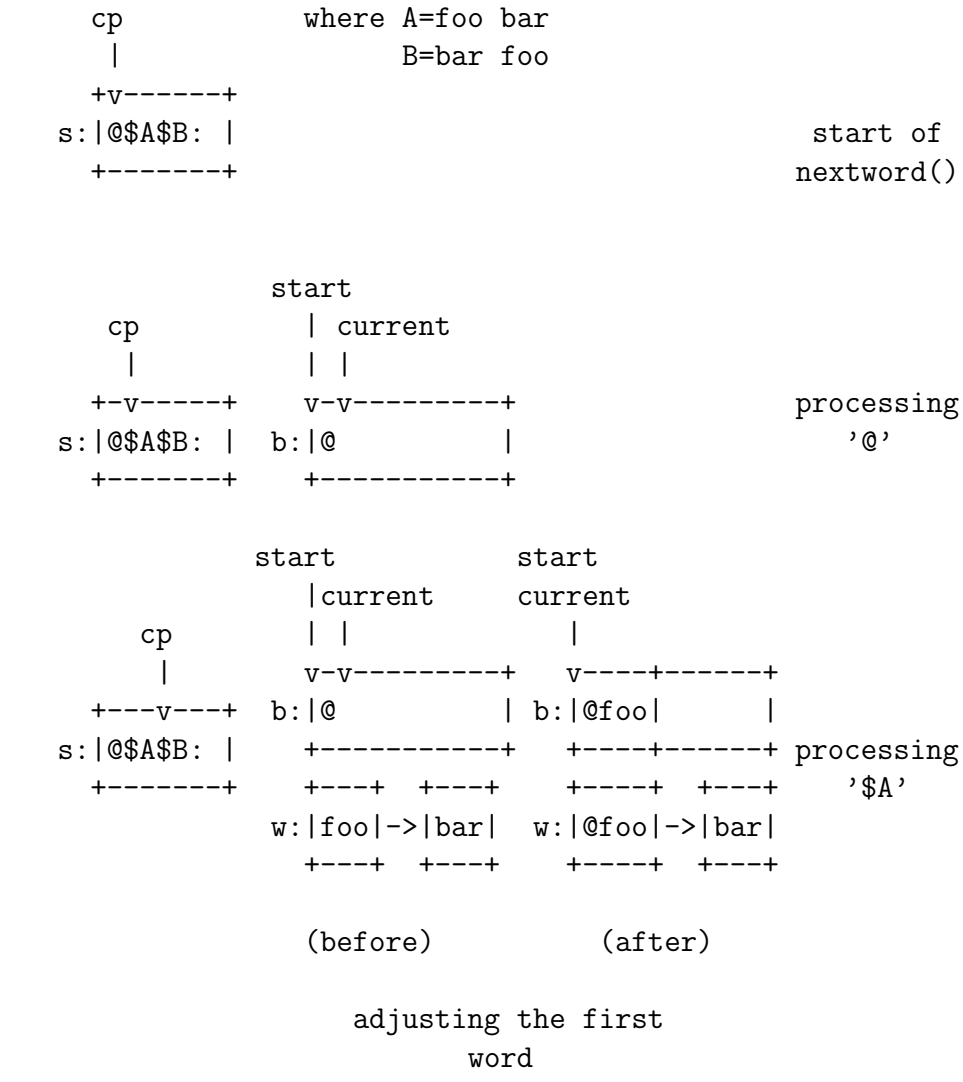


Figure 5.2: `nextword()` edge cases (part 1).

```
    resetbuf(buf);
}
```

Uses `bufcpy()` 176c, `insert()` 176a, and `resetbuf` 175g.

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* 71a⟩≡ (65)

```
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()` 176c and `insert()` 176a.

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()`[55] to process the rules, definitions, and inclusions. Those actions are based on the information returned by `rhead()`[61b]: 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 `':'`.

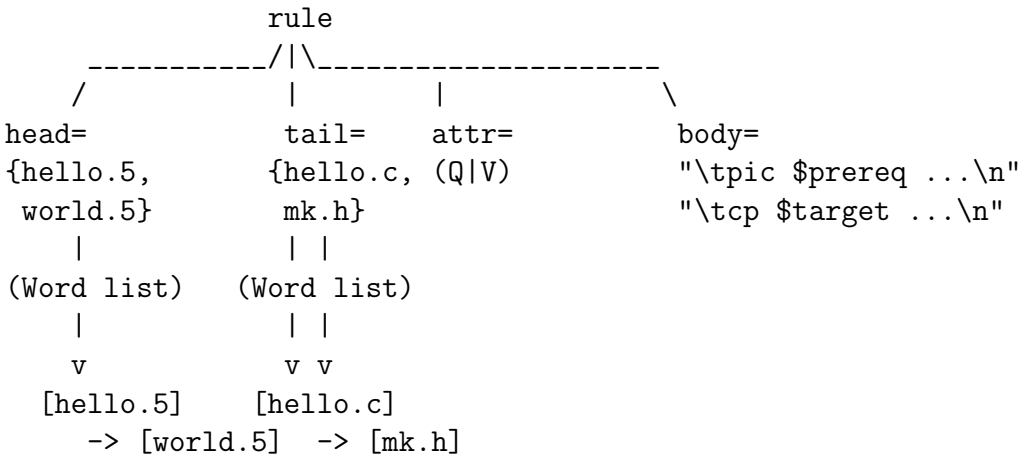
⟨`parse()` *other locals* 71b⟩≡ (55) 75a ▷

```
char *body;
```

Before showing the code, I find it useful to look at what the rule action receives after `rhead()` and `rbody()`[73c] are done. For the input

```
hello.5 world.5:QV: hello.c mk.h
    pic $prereq | tbl | troff -ms
    cp $target /tmp
```

the parse action sees a flat structure that I can draw as a pseudo-parse-tree (each field already a list of `Word`[34b]s, not a raw string):



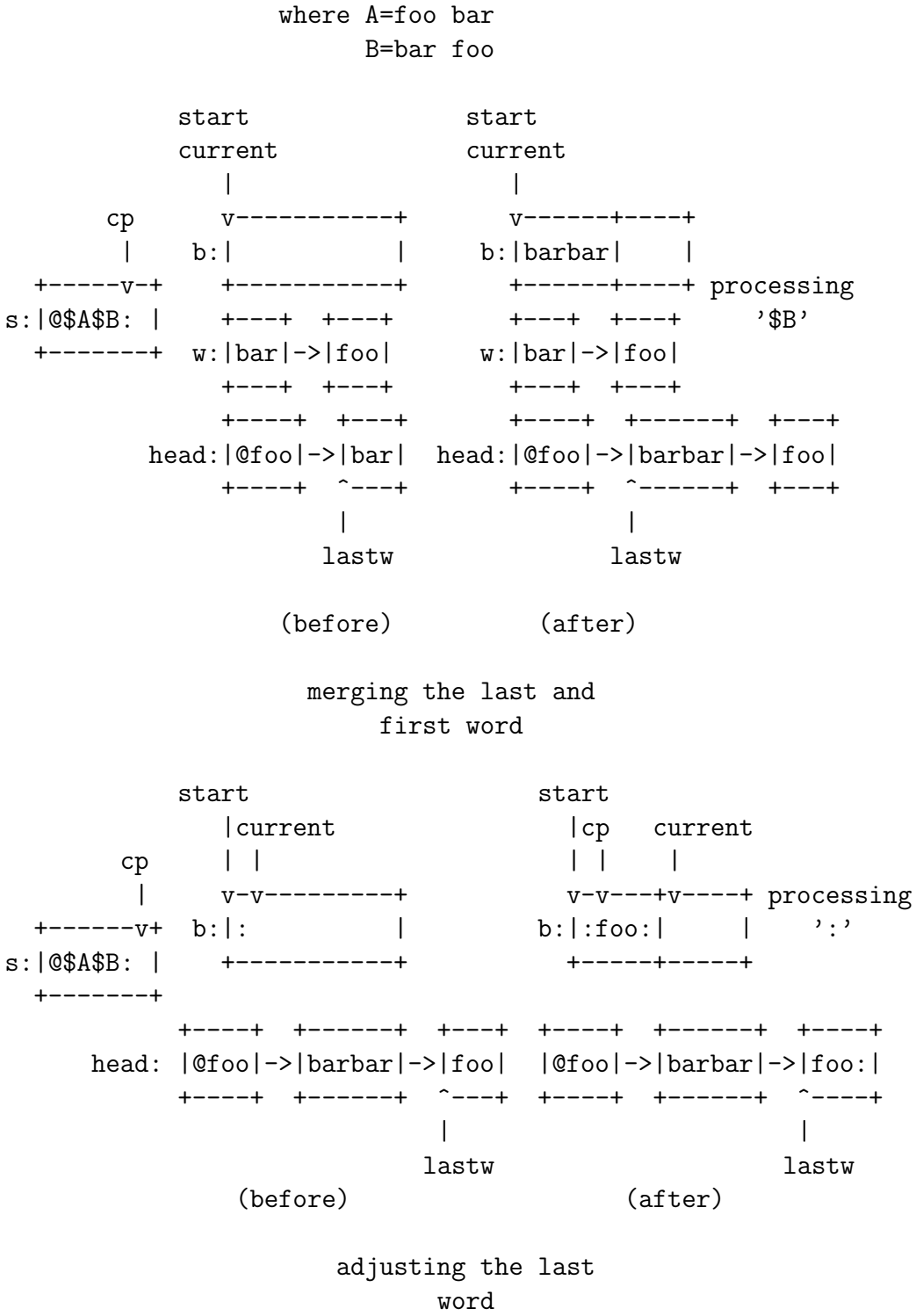


Figure 5.3: `nextword()` edge cases (part 2).

Three things to note. The targets and prerequisites are already expanded `Word` lists—variables, quotes, and backquotes have been resolved by `nextword()`[65] by the time `addrules()`[41c] runs, which is why the dependency graph can be built statically. The `attr` bitset comes from the characters between the two `':'`, here `Q` (quiet) and `V` (virtual), and will be propagated into every `Rule`[36] created from this line. The body, in contrast, is a raw string with `$prereq` and `$target` still visible; those will be substituted at recipe execution time from the shell environment built by `buildenv()`[125b], not here.

⟨`parse()` *switch rhead cases* 73a⟩+≡ (55) ◁57a 76a▷
```
case ':':
    body = rbody(&in);
    addrules(head, tail, body,   attr, hline, prog);
    break;
```
Uses `addrules()` 41c and `rbody()` 73c.

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()`[38b] for each target.

I mentioned also in Section 4.6.1 that `mk`, during parsing, sets the global `target1`[51f] to contain the first target found in the mkfile. Here is finally the code that sets `target1`:

⟨`addrules()` *set* `target1` 73b⟩≡ (41c)
```
/* 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` 135a, `charin()` 62b, `target1` 51f, and `wdup()` 35c.

Note the use of `charin()`[62b] 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`[135a] used above, see Section 11.1.

5.2.1 Parsing the recipe: `rbody()`

`rbody()`, like `assline()`[57b], 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. `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` 73c⟩≡ (186c)
```
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 74b⟩
            else
              if (r != ' ' && r != '\t') {
                Bungetrune(in);
                break;
            }
        } else
            // not in first column
            rinsert(buf, r);

        lastr = r;
        ⟨rbody() handle mkinline 74a⟩
    }

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

    return p;
}
```

Uses freebuf() 175e, insert() 176a, newbuf() 175d, and rinsert() 176b.

⟨rbody() *handle mkinline* 74a⟩≡ (73c)

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

Uses mkinline 54c.

If rbody()[73c] find 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* 74b⟩≡ (73c)

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

Uses rinsert() 176b.

If a rule has no recipe, rbody() returns the empty string to its caller parse()[55]. 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* 74c⟩≡

```
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`[37h], but before they are stored in a local variable in `parse()`[55]:

⟨`parse()` *other locals* 75a⟩+≡ (55) ◁71b 75e▷
```
// bitset<Rule_attr>
int attr;
```

This variable is passed by address to `rhead()`[61b] (see the call to `rhead()` in `parse()`). `rhead()` then initializes this variable and adjusts it depending on the separator:

⟨`rhead()` *adjust* `attr` *and* `prog` 75b⟩≡ (61b)
```
*attr = 0; // Nothing
*prog = nil;
// variable attributes
⟨rhead() if sep is = 79c⟩
// rule attributes
⟨rhead() if sep is : 75d⟩
```

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

⟨`rhead()` *other locals* 75c⟩≡ (61b) 148n▷
```
Rune r;
int n;
```

⟨`rhead()` *if sep is* : 75d⟩≡ (75b)
```
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 135b⟩
        //PAD: this is an extension in mk-in-ocaml that I ignore here
        case 'I':
            break;
        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()`[61b] is `'<'`.

A file inclusion in an `mkfile` will result in the opening of a new file, hence the following additional variables in `parse()`[55]:

⟨`parse()` *other locals* 75e⟩+≡ (55) ◁75a 78a▷
```
char *p;
fdt newfd;
```

Here is the code using `newfd`:

⟨`parse()` *switch rhead cases* 76a⟩+≡ (55) ◁73a 78b▷
```
case '<':
    p = wtos(tail, ' ');
    ⟨parse() when parsing included file, sanity check p 76b⟩
    newfd = open(p, OREAD);
    ⟨parse() when parsing included file, sanity check newfd 76c⟩
    else
        // recurse
        parse(p, newfd, false);
    break;
```
Uses `parse()` 55 and `wtos()` 35d.

⟨`parse()` *when parsing included file, sanity check p* 76b⟩≡ (76a)
```
if(*p == '\0'){
    SYNERR(-1);
    fprint(STDERR, "missing include file name\n");
    Exit();
}
```
Uses `SYNERR` 54d.

⟨`parse()` *when parsing included file, sanity check newfd* 76c⟩≡ (76a)
```
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 `'<'` (`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()`[64b], which performs quote and variable expansions. Thus, you can also use variables in the filename to include, as in

⟨*tests/mkfile/mkincludearc* 76d⟩≡
```
</$objtype/mkfile
```

I described `wtos()`[35d] 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`[54a] and `mkinline`[54c] must be saved before the call and restored after, hence the following calls in `parse()`:

⟨`parse()` *start, push* 76e⟩≡ (55)
```
ipush();
```
Uses `ipush()` 77c.

⟨`parse()` *end, pop* 76f⟩≡ (55)
```
ipop();
```
Uses `ipop()` 77d.

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

⟨*struct* `input` 76g⟩≡ (186c)
```
struct Input
{
    char *file;
    int line;

    // Extra
    ⟨Input extra fields 77b⟩
};
```

⟨*global* inputs 77a⟩≡ (186c)
```
// list<ref_own<Input>> (next = Input.next)
static struct Input *inputs = nil;
```
Uses Input 76g and inputs-14 77a.

⟨Input *extra fields* 77b⟩≡ (76g)
```
// list<ref_own<Input>> (head = inputs)
struct Input *next;
```
Uses Input 76g.

⟨*function* ipush 77c⟩≡ (186c)
```
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 76g, Malloc() 174a, infile 54a, inputs-14 77a, and mkinline 54c.

⟨*function* ipop 77d⟩≡ (186c)
```
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 76g, infile 54a, inputs-14 77a, and mkinline 54c.

5.4 Parsing Variable definitions: *var=values*

The final action of `parse()`[55] manages variable definitions, when the special character returned by `rhead()`[61b] is `'='`.

⟨`parse()` *other locals* 78a⟩+≡ (55) ◁75e 140c▷
```
bool set = true;
```

⟨`parse()` *switch rhead cases* 78b⟩+≡ (55) ◁76a 140d▷
```
case '=':
    ⟨parse() when parsing variable definitions, sanity check head 78c⟩
    ⟨parse() when parsing variable definitions, override handling 78e⟩
    if(set){
        setvar(head->s, (void *) tail);
        ⟨parse() when parsing variable definitions, extra setting 130c⟩
    }
    ⟨parse() when parsing variable definitions, if variable with attr 152a⟩
    break;
```
Uses `setvar()` 33b.

The code above relies mainly on the function `setvar()`[33b] to add an entry in the `S_VAR`[31a] 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* 78c⟩≡ (78b)
```
if(head->next){
    SYNERR(-1);
    fprint(STDERR, "multiple vars on left side of assignment\n");
    Exit();
}
```
Uses `SYNERR` 54d.

Note also that both `head` and `tail` used above are the results of calls to `stow()`[64b] 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* 78d⟩≡
```
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()`[55] 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* 78e⟩≡ (78b)
```
if(symlook(head->s, S_OVERRIDE, nil)){
    set = varoverride;
} else {
    set = true;
    if(varoverride)
        symlook(head->s, S_OVERRIDE, (void *)"");
}
```
Uses `S_OVERRIDE` 79a and `symlook()` 32a.

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* 79a⟩+≡ (31a) ◁39b 87a▷
```
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* 79b⟩≡
```
MYVAR=U= foo.5 bar.5 # a private variable
```

Here is the code in `rhead()`[61b] to extract variable attributes:

⟨`rhead()` *if sep is* `=` 79c⟩≡ (75b)
```
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 151h⟩
            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` 79d⟩≡ (185a)
```
char *termchars = "=' \t"; /*used in parse.c to isolate assignment attribute*/
```
Uses `termchars` 79d.

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* 79e⟩≡
```
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()`[80], 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`[37b] and `metarules`[37e] populated by `parse()`[55] in Section 5.2. I will also describe in this chapter the code to check for 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()`[81], which is the function containing the logic to build the graph of dependencies.

⟨*function* `graph` 80⟩≡ (190a)

```
Node*
graph(char *target)
{
    Node *root;
    ⟨graph() other locals 91d⟩

    ⟨graph() set cnt for infinite rule detection 91e⟩
    root = applyrules(target, cnt);
    ⟨graph() free cnt 91g⟩

    ⟨graph() checking the graph 88d⟩
    ⟨graph() propagate attributes 145b⟩

    return root;
}
```

Uses `applyrules()` 81.

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()`[42b]), then finds the rules or meta rules with matching targets, and finally applies 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` 81⟩≡ (190a)

```
static Node*
applyrules(char *target, char *cnt)
{
    Node *node;
    // list<ref<Arc> (next = Arc.next, last = lasta)
    Arc head;
    // ref<Arc>
    Arc *lasta = &head;
    ⟨applyrules other locals 82a⟩

    ⟨applyrules debug 171h⟩
    ⟨applyrules check node cache if target is already there 87c⟩
    // else

    target = strdup(target);
    node = newnode(target); // calls timeof() internally
    head.next = nil;
    ⟨applyrules other initializations 137d⟩

    // apply regular rules with target as a head (modifies lasta)
    ⟨applyrules() apply regular rules 82b⟩

    // apply meta rules (modifies lasta)
    ⟨applyrules() apply meta rules 83e⟩

    node->arcs = head.next;

    return node;
}
```

Uses `newnode()` 42b.

The code above relies on a local variable `head` containing an `Arc`[43b] 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()`[38b] with the list of rules, or in Section 4.6.3 with a list of words).

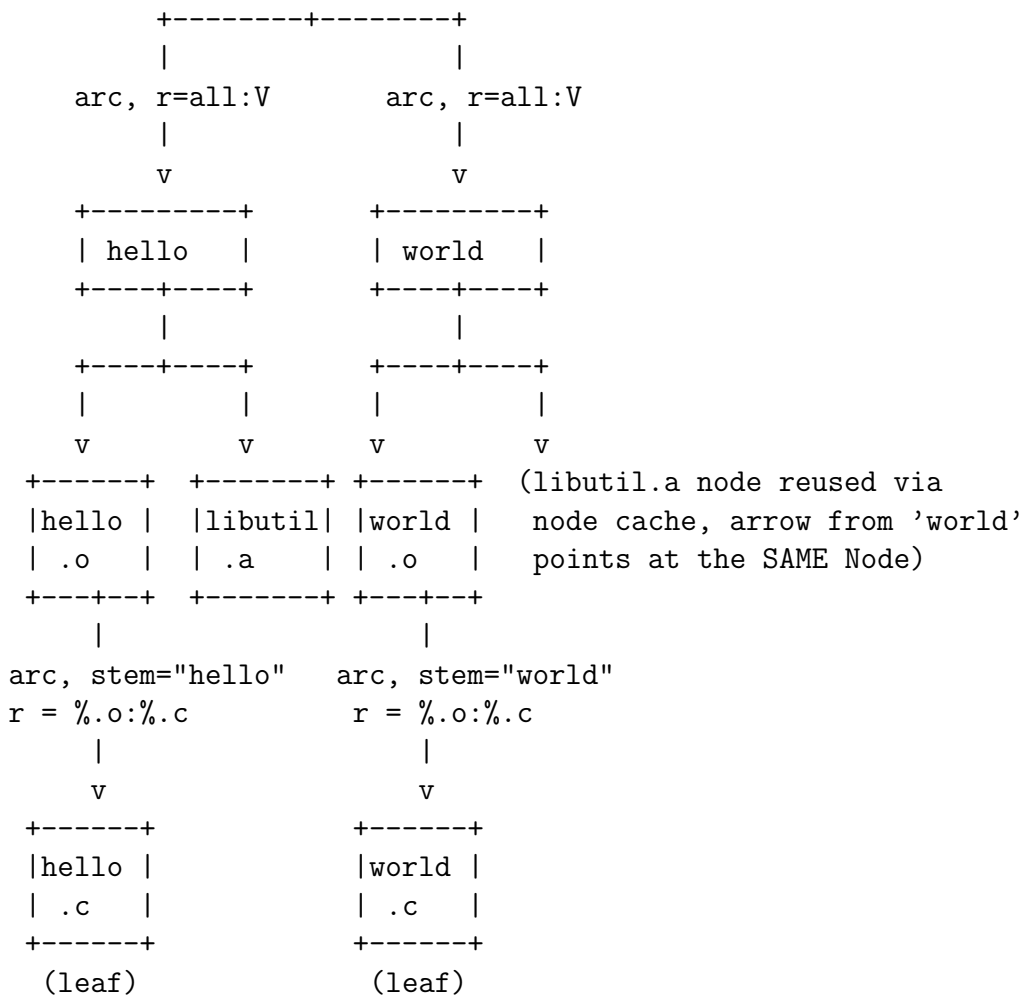
I will explain the code to apply rules and meta rules in the next two sections.

To make the recursive structure of `applyrules()` concrete, here is the graph it builds for a tiny three-target project. The `mkfile` is

```
all:V: hello world
hello: hello.o libutil.a
world: world.o libutil.a
%.o: %.c
    cc -c $stem.c
```

and the user runs `mk all`. `applyrules()` is called on `all`, recurses on `hello` and `world`, which each recurse on their `.o` targets (matched through the meta rule `%.o: %.c`) and on `libutil.a`. The node cache ensures that `libutil.a` gets a single `Node`[42a] shared by both parents, turning the tree into a proper DAG:

```
                +-----+
                | all |  (virtual target, Rule.attr has VIR)
                +--+--+
                   |
```



The details worth pausing on: each `Arc` carries its own `stem` because the same meta rule `%.o: %.c` is instantiated twice, once with `stem="hello"` and once with `stem="world"`, and the shell will need the right value when the recipe runs. The `libutil.a` node has two incoming arcs but still only one `Node` in memory—without the node cache `cc -c` would run for it twice. The `all` node has a `VIR` attribute and no recipe; its two arcs exist only to pull the real targets into the DAG, as I explained in Section 11.5.1. Finally, the leaves `hello.c` and `world.c` have no matching rule and no prerequisites—`applyrules()` stops there and `timeof()`[88a] reads their on-disk modification time into `Node.time`[42a].

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()`[81] can rely on the symbol table and the `S_TARGET`[39b] namespace to quickly access all the rules mentioning a specific target (as explained in Section 3.3.4):

⟨`applyrules` *other locals* 82a⟩≡ (81) 84a▷
```
Symtab *sym;
Rule *r;
Word *pre;
Arc *arc;
```

⟨`applyrules()` *apply regular rules* 82b⟩≡ (81)
```
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 83a⟩
```

```
    ⟨applyrules() infinite rule detection part1 91i⟩
    ⟨applyrules() when found a regular rule for target node, set flags 95d⟩

    ⟨applyrules() if no prerequisites in rule r 83d⟩
    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 92a⟩
}
```

Uses S_TARGET 39b, applyrules() 81, newarc() 43e, and symlook() 32a.

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 83a⟩≡ (82b)

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

Uses empty_prereqs 83c and empty_recipe 83b.

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

⟨macro empty_recipe 83b⟩≡ (181d)

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

⟨macro empty_prereqs 83c⟩≡ (181d)

```
#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 83d⟩≡ (82b)

```
// 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 83c and newarc() 43e.

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()`[81] needs to go through all the meta rules stored in the global `metarules`[37e]:

⟨applyrules() apply meta rules 83e⟩≡ (81)

```
for(r = metarules; r; r = r->next){
    ⟨applyrules() skip this meta rule and continue if some conditions 84d⟩
    ⟨applyrules() if regexp rule then continue if some conditions 137e⟩
    else {
```

```
        if(match(node->name, r->target, stem)) {
            ⟨applyrules() infinite rule detection part1 91i⟩

            ⟨applyrules() if no prerequisites in meta rule r 84e⟩
            else
                for(pre = r->prereqs; pre; pre = pre->next) {
                    ⟨applyrules() if regexp rule, adjust buf and rmatch 137f⟩
                    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 92a⟩
        }
    }
}
```

Uses applyrules() 81, metarules 37e, newarc() 43e, and subst() 86.

The code above relies on the functions match()[85a] and subst()[86], 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* 84a⟩+≡ (81) ◁82a 84c▷

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

Uses NAMEBLOCK 84b.

⟨*constant* NAMEBLOCK 84b⟩≡ (181d)

```
#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* 84c⟩+≡ (81) ◁84a 137c▷

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

Uses NAMEBLOCK 84b.

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

⟨applyrules() *skip this meta rule and continue if some conditions* 84d⟩≡ (83e) 148h▷

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

Uses empty_prereqs 83c and empty_recipe 83b.

Some meta rules can also contain virtual targets:

⟨applyrules() *if no prerequisites in meta rule r* 84e⟩≡ (83e)

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

Uses empty_prereqs 83c and newarc() 43e.

The code above is almost identical to code in Section 6.2, except applyrules() passes here the stem buffer to newarc()[43e] instead of the empty string.

6.3.1 Matching a pattern: `match()`

`match()` is the `mk` equivalent of one half of `fnmatch()`: it tests whether `name` fits a `template` containing exactly one wildcard, and if so it captures the wildcard text as the stem. There is no backtracking because there is at most one `%` in the template, so the algorithm is just: walk both strings until you hit `%` in the template, anchor the rest of the template at the end of the name (using the length difference to compute where the stem ends), and copy the stem out. The pair of `match()` (read) and `subst()`[86] (write) is how a meta rule like `%.5: %.c` becomes a concrete arc: `match("hello.5", "%.5", stem)` sets `stem="hello"`, and then `subst("hello", "%.c", buf)` writes `buf="hello.c"`.

`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()`[81]. Figure 6.1 illustrates how `match()` works on a simple example.

⟨*function* `match` 85a⟩≡ (186a)

```
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 85b⟩

    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* 85b⟩≡ (85a)

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

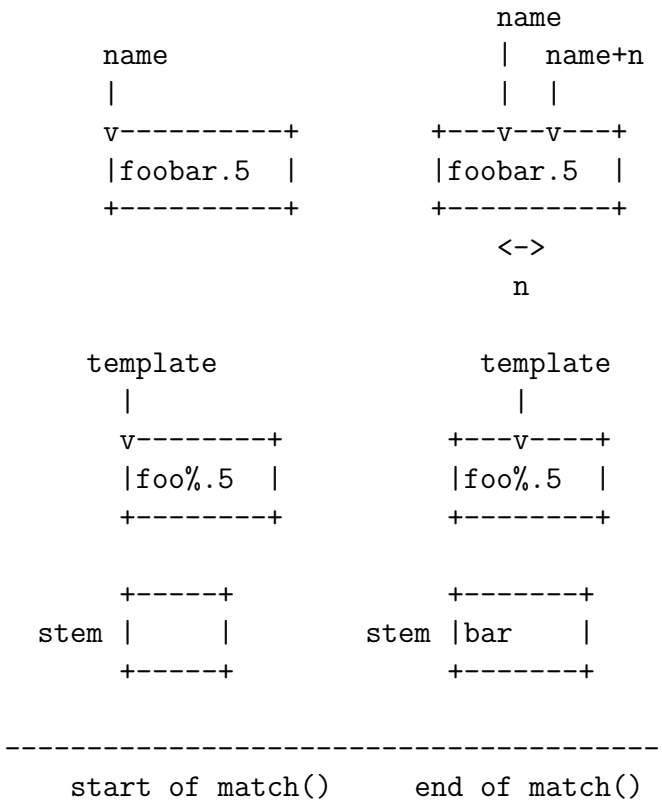


Figure 6.1: `match("foobar.5", "foo\%.5", ...)`.

6.3.2 Substituting the stem: `subst()`

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

⟨*function* `subst` 86⟩≡ (186a)

```
void
subst(char *stem, char *template,   char *dest, int dlen)
{
    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` 37g.

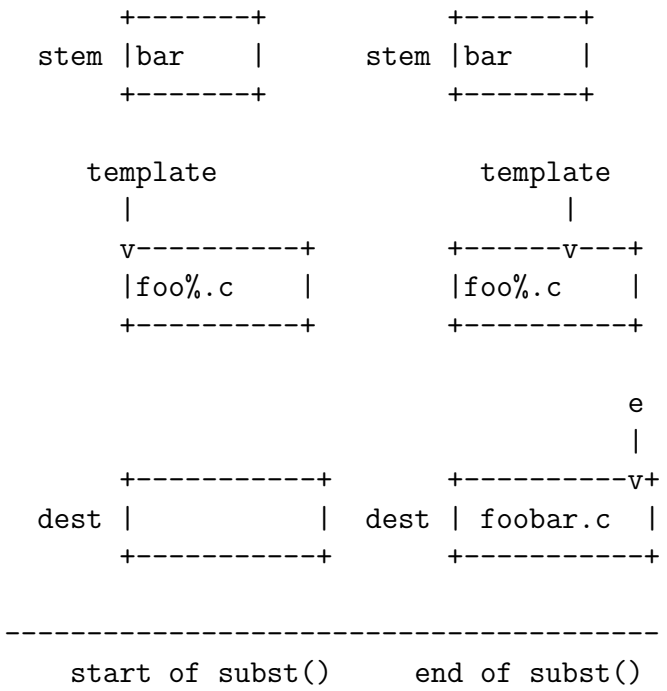


Figure 6.2: `subst("bar", "foo\%.c", ...)`.

6.4 Node cache

The node cache ensures that each file has exactly one `Node`[42a] in the graph. Without it, a header file included by ten source files would produce ten separate nodes, and `mk` would not realize they represent the same file—leading to redundant recipe executions and incorrect modification-time tracking. The cache also turns the graph from a tree into a proper DAG: when `applyrules()`[81] encounters a target it has already visited, it returns the existing node instead of recursing.

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

⟨Sxxx *cases* 87a⟩+≡ (31a) ◁79a 130b▷
```
S_NODE,  /* target name -> node */
```

Each time `newnode()`[42b] 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* 87b⟩≡ (42b)
```
symlook(name, S_NODE, (void *)node);
```
Uses S_NODE 87a and symlook() 32a.

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* 87c⟩≡ (81)
```
sym = symlook(target, S_NODE, nil);
if(sym)
    return sym->u.ptr;
```
Uses S_NODE 87a and symlook() 32a.

Creating a single node per file is important not only to avoid unnecessary calls to `applyrules()`. Indeed, once `mk` finished executing a recipe, `mk` can 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()` called from `newnode()`[42b] 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` 88a⟩≡ (188b)
```
ulong
timeof(char *name, bool force)
{
    ⟨timeof() locals 160f⟩

    ⟨timeof() if name archive member 154f⟩
    if(force)
        return mkmtime(name, true);
    // else
    ⟨timeof() if not force, use time cache 161a⟩
}
```

I will explain in Section 11.9.3 the `force` argument.

⟨*function* `mkmtime` 88b⟩≡ (198)
```
ulong
mkmtime(char *name, bool force)
{
    Dir *d;
    ulong t;
    ⟨mkmtime locals 161e⟩

    ⟨mkmtime() bulk dir optimisation 161f⟩
    d = dirstat(name);
    ⟨mkmtime() check if inexistent file 88c⟩
    t = d->mtime;
    free(d);

    return t;
}
```

Note that if the file does not exist, `mkmtime()`[192c] and `timeof()`[88a] return `0`:

⟨`mkmtime()` *check if inexistent file* 88c⟩≡ (88b)
```
if(d == nil)
    return 0;
```

6.6 Checking the graph and the rules

Once `applyrules()`[81] returned the graph of dependencies, `graph()`[80] can check for special properties of the graph that are signs of mistakes in the `mkfile`:

⟨`graph()` *checking the graph* 88d⟩≡ (80)
```
cyclechk(root);

⟨graph() before ambiguous() 95e⟩
ambiguous(root);
```
Uses `ambiguous()` 92c and `cyclechk()` 89c.

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

6.6.1 Cycle detection

Cycle detection here uses a classic DFS trick: mark on entry, unmark on exit. The set flag while a node is on the recursion stack distinguishes a true back edge (cycle) from a cross edge (harmless DAG sharing). If `mk` used a permanent "visited" flag instead, it would falsely report a cycle the second time it walked into a shared node like `common.h` from a different parent. The downside is that the cost is proportional to the number of paths through the graph, not the number of nodes, which would be a problem for very large graphs but is fine for the `mkfiles` `mk` is built to handle.

GNU Make's behavior here is different and arguably worse: it detects cycles dynamically while building, prints "Circular X ← Y dependency dropped," then keeps going as if the back edge did not exist—which can leave the user with a build that succeeds but produces incorrect output. `mk` takes the strict path: a cycle is always an error, and the static graph means `mk` catches it before any recipe runs.

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* 89a⟩≡
```
foo: foo.5 bar.5
    5l -o foo foo.5 bar.5
%.5: %.c
    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* 89b⟩+≡ (42d) ◁42e 95b▷
```
CYCLE      = 0x0002,
```

This flag is stored in a unique bit in the `Node.flags`[42c] field; it will not conflict with the other node flags that I introduced in Section 3.4.1 (e.g., `NOTMADE`[42e] is `0x0020`).

The function below assumes it is called with the `CYCLE` flag unset for every nodes, which is true because `newnode()`[42b] set `Node.flags` to `0`.

⟨*function* `cyclechk` 89c⟩≡ (190a)
```
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);
```

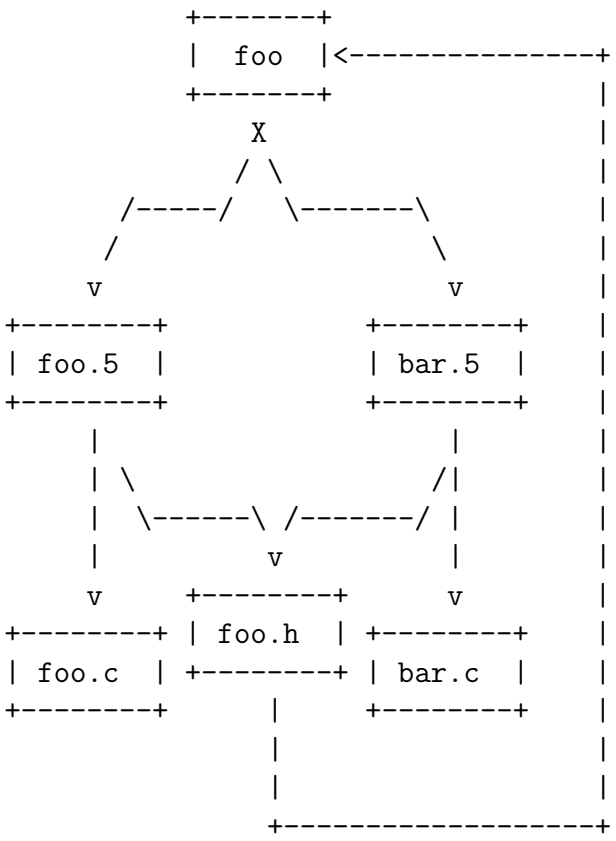


Figure 6.3: A cycle in a graph of dependencies.

```
    n->flags &= ~CYCLE;
}
```

Uses CYCLE 89b and cyclechk() 89c.

Note that it is important for `cyclechk()`[89c] 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 infinite-rule problem cannot exist for simple rules because they refer to concrete filenames and the cycle check catches those. It is unique to meta rules, where applying the same rule with a different stem yields a fresh target name (`foo` → `foo.5` → `foo.5.5` → ...) that the cycle check would never see as a repeat. The fix is to count rule applications along the current DFS branch and cap each rule at `nreps` (default 1). The crucial detail is the `cnt[r->rule]++` before recursing and `cnt[r->rule]--` after: the cap is per-branch, not global, so a meta rule like `%.5: %.c` can still fire many times in independent branches of the graph.

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* 90⟩≡

```
all: foo bar

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

When building the graph for `foo`, `applyrules()`[81] 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* 91a⟩+≡ (36) ◁41d 135c▷
```
int   rule;  /* rule number */
```

⟨*global* `nrules` 91b⟩≡ (187d)
```
static int nrules = 0;
```
Uses `nrules-25` 91b.

⟨`addrule()` *set more fields* 91c⟩+≡ (38b) ◁54b 148k▷
```
r->rule = nrules++;
```
Uses `nrules-25` 91b.

Then, before calling `applyrules()`, `graph()`[80] initializes a map that will count how many times `applyrules()` used a rule:

⟨`graph()` *other locals* 91d⟩≡ (80)
```
// map<ruleid, int>
char *cnt;
```

⟨`graph()` *set cnt for infinite rule detection* 91e⟩≡ (80)
```
cnt = rulecnt();
```
Uses `rulecnt()` 91f.

⟨*function* `rulecnt` 91f⟩≡ (187d)
```
char*
rulecnt(void)
{
    char *s;

    s = Malloc(nrules);
    memset(s, 0, nrules);
    return s;
}
```
Uses `Malloc()` 174a and `nrules-25` 91b.

⟨`graph()` *free cnt* 91g⟩≡ (80)
```
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` 91h⟩≡ (190a)
```
int nreps = 1;
```
Uses `nreps` 91h.

⟨`applyrules()` *infinite rule detection part1* 91i⟩≡ (83e 82b)
```
if(cnt[r->rule] >= nreps)
    continue;

cnt[r->rule]++;
```
Uses `nreps` 91h.

[1] This is one of the motivations for the special pattern character `'&'`.

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* 92a⟩≡ (83e 82b)
```
cnt[r->rule]--;
```

6.6.3 Ambiguous rules detection

A node is ambiguous when it has two arcs with different recipes: `mk` cannot decide which one to run. The detection uses pointer comparison (`master_rule->recipe != a->r->recipe`) rather than `strcmp`, which means two rules that happen to have the same recipe text are still considered ambiguous unless they were literally parsed from the same rule. This is intentional: the pointer identity comes from how `mk` stores recipes. An important refinement: when a simple rule and a meta rule both match a target, the simple rule wins (it is more specific). This lets users write generic pattern rules while overriding them for particular files without triggering an ambiguity error.

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* 92b⟩≡
```
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` 92c⟩≡ (190a)
```
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 94b⟩
            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 94e⟩
}
```

Uses `ambiguous()` 92c, `empty_recipe` 83b, and `trace()` 93a.

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()`[92c] calls `trace()` below for each ambiguous arc:

⟨*function* `trace` 93a⟩≡ (190a)

```
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 93b⟩
        else
            a = nil;
    }
    fprint(STDERR, "\n");
}
```

⟨`trace()` *possibly continue if prereq is also a target* 93b⟩≡ (93a)

```
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* 94a⟩≡
```
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* 94b⟩≡ (92c)
```
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` 38a.

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* 94c⟩+≡ (43b) ◁43d 136c▷
```
short remove;
```

⟨`newarc()` *set other fields* 94d⟩≡ (43e) 136d▷
```
a->remove = false;
```

Adjusting the graph, `togo()`

`ambiguous()`[92c], before returning, removes all the arcs marked as to-be-removed on the current node:

⟨`ambiguous()` *get rid of all skipped arcs* 94e⟩≡ (92c)
```
togo(n);
```
Uses `togo()` 94f.

⟨*function* `togo` 94f⟩≡ (190a)
```
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* 95a⟩≡
```
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* 95b⟩+≡ (42d) ◁89b 96b▷
```
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()`[42b] (remember that `timeof()`[88a] returns `0` for inexistent files):

⟨`newnode()` *adjust flags of node* 95c⟩≡ (42b)
```
if(node->time)
    node->flags = PROBABLE;
```
Uses `PROBABLE` 95b.

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* 95d⟩≡ (82b)
```
node->flags |= PROBABLE;
```
Uses `PROBABLE` 95b.

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()` 95e⟩≡ (88d) 96a▷
```
root->flags |= PROBABLE; /* make sure it doesn't get deleted */
```
Uses `PROBABLE` 95b.

Once `mk` initialized the nodes in the graph of dependencies, `graph()`[80] calls `vacuous()`[96d] to explore the graph to detect and remove vacuous nodes.

⟨`graph()` *before* `ambiguous()` 96a⟩+≡ (88d) ◁95e
```
vacuous(root);
```
Uses `vacuous()` 96d.

Note that `graph()` must call `vacuous()` before `ambiguous()`[92c], to remove the vacuous nodes and arcs, otherwise `ambiguous()` would report ambiguities.

`vacuous()` relies on two extra node flags to operate:

⟨`Node_flag` *cases* 96b⟩+≡ (42d) ◁95b 96c▷
```
VACUOUS    = 0x0200,
```

⟨`Node_flag` *cases* 96c⟩+≡ (42d) ◁96b 146c▷
```
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` 96d⟩≡ (190a)
```
static bool
vacuous(Node *node)
{
    Arc *a;
    bool vac = !(node->flags&PROBABLE);
    ⟨vacuous() other locals 96e⟩

    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 97⟩

    togo(node);
    if(vac) {
        node->flags |= VACUOUS;
    }
    return vac;
}
```
Uses `META` 38a, `PROBABLE` 95b, `READY` 96c, `VACUOUS` 96b, `togo()` 94f, and `vacuous()` 96d.

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* 96e⟩≡ (96d)
```
Arc *a2;
```

⟨vacuous *possibly undelete some arcs* 97⟩≡ (96d)

```
/* 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()`[99], takes a target as a parameter, builds the graph of dependencies for this target (with `graph()`[80]), explores the graph to find outdated files, with a function called `work()`[101c], and schedules recipes to be executed to update those outdated files, with a function called `dorecipe()`[104e]. The following sections will explain the code of `mk()`, `work()`, and `dorecipe()`.

7.1 Mtime vs content-hash change detection

The core question this chapter answers is "has this file changed since the last build?". Every build system has to answer it, and there are only two real strategies. The timestamp strategy compares the file's last-modified time against the mtime of its dependencies: if any dependency is newer than the target, the target is out of date. The content-hash strategy compares a cryptographic hash of the file's contents against a recorded hash from the last build: if the hash changed, the file changed. The choice shapes almost everything else a build system does.

Timestamps are simple, universal, and free. Every UNIX filesystem records `mtime` to one-second resolution, so the check is a single `stat()` call per file and a few integer comparisons. `mk` uses this strategy, as do `make`, `redo`, `tup`, `ninja` (with a bit of caching on top), and every build tool written before 2000. The weakness is that timestamps are an approximate proxy for content: clock skew across machines breaks them, NFS atomicity is subtle, restoring files from a backup resets their mtime and triggers spurious rebuilds, a touched-but- unchanged file forces a rebuild you did not need, and an edited-and-saved-back-to-the-same-bytes file does too. The failure modes are small in absolute terms and tolerable in a local development loop.

Content hashing sidesteps all of that. `Bazel` (Google, 2015, descended from internal Blaze), `Buck` (Facebook), `Nix` (Eelco Dolstra's 2004 thesis work, now a package manager), `Shake` (Neil Mitchell's Haskell library), and `Pants` all hash file contents to decide what is out of date. Hashing is more expensive—every input needs to be read in full every build—but it enables things mtime-based systems cannot reach: hermetic remote build caches that can serve any user who happens to need the same inputs, reproducible builds that detect accidental input changes, and distributed execution that runs compile jobs on a pool of machines. Hashing is what makes Google's build farm and Nix's binary cache possible at all; an mtime-based system cannot ever safely share cached outputs between users or machines because mtimes are meaningless across hosts.

`mk` is firmly on the mtime side. The algorithm in `work()`[101c] walks the dependency graph in depth-first order, compares `time(target) < max(time(deps))` using the `Time` field on each `Node`, and marks the target `NOTMADE`[42e] or `MADE`[42e] accordingly. There is no hash, no cache file, no stored "what did this look like last build" state at all—the filesystem's mtime is the only persistence. The simplicity is the point: the whole `mk` binary is a few thousand lines, most of the state fits in one process's memory, and the build loop is something you can trace by hand. The sharing and reproducibility benefits of hash-based systems are real, but they come with thousands of lines of cache management, remote-execution protocols, and sandboxing—a different kind of tool

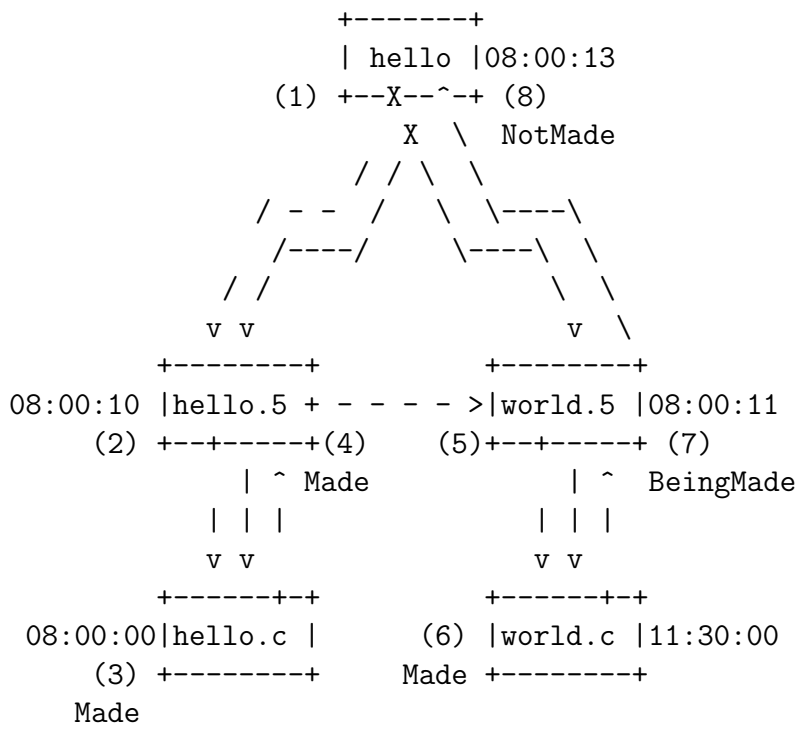


Figure 7.1: Graph of dependencies with building-status labels.

for a different kind of build.

7.2 mk()

`mk()`[99] 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`[42e], `BEINGMADE`[42e], and `MADE`[42e]) 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()` is the heart of the build system. It operates in waves: each call to `work()`[101c] 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()`[115c] 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` 99⟩≡ (190b)

```
void
mk(char *target)
{
    Node *root;
    bool everdid = false;
    bool did = false;
    // enum<WaitupResult>
    int res;
```

```
    ⟨mk() initializations 109b⟩

    root = graph(target);
    ⟨mk() if DEBUG(D_GRAPH) 170a⟩
    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 165d⟩
        }
    }
    if(root->flags&BEINGMADE)
        waitup(EMPTY_CHILDREN_IS_ERROR, (int *)nil);

    ⟨mk() before returning, more waitup() if there was an error 165f⟩
    if(!everdid)
        Bprint(&bout, "mk: '%s' is up to date\n", root->name);
    return;
}
```

Uses BEINGMADE 42e, EMPTY_CHILDREN_IS_ERROR 117b, EMPTY_CHILDREN_IS_OK 117b, NOTMADE 42e, bout 48a, clrmade() 101a, graph() 80, waitup() 115c, and work() 101c.

`mk()` operates in six steps:

1. `mk()` builds the graph of dependencies with `graph()`[80] (see Chapter 6).

2. `mk()` sets the building status of all nodes in the graph to `NOTMADE` with `clrmade()`[101a] (see Section 7.3).

3. `mk()` explores the graph of dependencies repeatedly with `work()` in successive waves to find outdated files (see Section 7.4). 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()`[104e], 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 `EMPTY_CHILDREN_I` flag 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`[117b] 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 11.9.1 why `work()` needs such information for advanced features of `mk`.

7.3 Initializing nodes: `clrmade()`

`clrmade()` below simply marks all nodes as `NOTMADE`[42e]:

⟨*function* `clrmade` 101a⟩≡ (190b)
```
void
clrmade(Node *n)
{
    Arc *a;

    ⟨clrmade() n->flags pretend adjustments 158d⟩
    MADESET(n, NOTMADE);
    for(a = n->arcs; a; a = a->next)
        if(a->n)
            // recurse
            clrmade(a->n);
}
```
Uses `MADESET` 101b, `NOTMADE` 42e, and `clrmade()` 101a.

Note that `Node.flags`[42c] 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` 101b⟩≡ (181d)
```
#define MADESET(n,m) n->flags = (n->flags&~(NOTMADE|BEINGMADE|MADE))|(m)
```

7.4 Exploring the graph: `work()`

`work()`[101c] is the DFS traversal that decides what to build. For each node, it recurses into children first, then examines two booleans: `weoutofdate` (is any prerequisite newer than this node?) and `ready` (are all prerequisites `MADE`[42e]?). A node can be outdated but not ready—for example, if one prerequisite is still `BEINGMADE`[42e] while another has already changed. In that case `work()` returns without scheduling a recipe; the next wave will re-visit the node after the pending prerequisite finishes.

`work()` below performs a DFS on the graph while looking for outdated `NOTMADE`[42e] nodes:

⟨*function* `work` 101c⟩≡ (190b)
```
void
work(Node *node, bool *did,   Node *parent_node, Arc *parent_arc)
{
    ⟨work() locals 102b⟩

    ⟨work() debug 172a⟩
    if(node->flags&BEINGMADE)
        return;
    ⟨work() possibly unpretending node 159a⟩
```

```
    if(node->arcs == nil){
        ⟨work() no arcs, a leaf 102a⟩
    } else {
        ⟨work() some arcs, a node 103a⟩
    }
}
```

Uses BEINGMADE 42e.

The following two sections explain the code of `work()` when the node is a leaf and when it has children.

7.4.1 The leaf case

The leaf case is easy. If a node has no prerequisites and contains an existing file, for instance, `hello.c` in Figure 7.1, then this file is already MADE[42e].

⟨work() *no arcs, a leaf* 102a⟩≡ (101c)
```
/* consider no prerequisite case */
if(node->time == 0){
    ⟨work() print error when inexistent file without prerequisites 102c⟩
} else
    MADESET(node, MADE);
```

Uses MADE 42e and MADESET 101b.

Otherwise, `mk` should report an error:

⟨work() *locals* 102b⟩≡ (101c) 102d▷
```
char cwd[256];
```

⟨work() *print error when inexistent file without prerequisites* 102c⟩≡ (102a)
```
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 165a⟩
else
    Exit();
```

7.4.2 The parent case

The parent case is more complex. For the parent case, `work()`[101c] relies on the two important booleans below:

⟨work() *locals* 102d⟩+≡ (101c) ◁102b 102e▷
```
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[42e].

Here is finally the code of `work()` that handles nodes with children (and using the locals above):

⟨work() *locals* 102e⟩+≡ (101c) ◁102d 159c▷
```
Arc *a;
```

⟨work() *some arcs, a node* 103a⟩≡ (101c)
```
⟨work() adjust weoutofdate if aflag 163c⟩
/*
 *   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 159d⟩
        }
    } else {
        if(node->time == 0){
            weoutofdate = true;
            ⟨work() update ra when no dest in arc and no src 159e⟩
        }
    }
}

if(!ready) /* can't do anything now */
    return;
if(!weoutofdate){
    MADESET(node, MADE);
    return;
}
⟨work() possibly pretending node 158e⟩
// else, out of date

dorecipe(node, did);
return;
```
Uses BEINGMADE 42e, MADE 42e, MADESET 101b, NOTMADE 42e, dorecipe() 104e, outofdate() 103b, and work() 101c.

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 103b⟩≡ (190b)
```
bool
outofdate(Node *node, Arc *arc, bool eval)
{
    ⟨outofdate() locals 149d⟩

    ⟨outofdate() if arc-¿prog 149e⟩
    else
     ⟨outofdate() if arc node is an archive member 157b⟩
     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.

7.5 Scheduling recipes: `dorecipe()`

The job of `dorecipe()`[104e], called from `work()`[101c], is to construct a `Job`[44a] to run a recipe that will update a target `Node`[42a]. 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* 104a⟩≡ (104e) 104b▷
```
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* 104b⟩+≡ (104e) ◁104a 104c▷
```
// list<string> (last = last_alltargets)
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* 104c⟩+≡ (104e) ◁104b 104d▷
```
// 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* 104d⟩+≡ (104e) ◁104c 105a▷
```
// list<string> (last = last_allprereqs)
Word allprereqs;
```

This set will be used when exporting the special variable `$prereqs` in `buildenv()`[125b].

Here is finally the code of `dorecipe()` leveraging the locals above:

⟨*function* `dorecipe` 104e⟩≡ (191a)
```
void
dorecipe(Node *node, bool *did)
{
    // iterators
    Arc *a;
    Node *n;
    Word *w;
    Symtab *s;
    ⟨dorecipe() other locals 104a⟩

    /*
     *   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 105b⟩
    // else

    /*
     *   build the node list
     */
    ⟨dorecipe() build lists of targets and node list 106b⟩

    /*
     *   gather the params for the job
     */
    allprereqs.next = newprereqs.next = nil;
    for(n = nlist; n; n = n->next){
        ⟨dorecipe() build lists of prerequisites 107⟩
        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 42e, MADESET 101b, empty_recipe 83b, newjob() 44c, and run() 108a.

I described newjob()[44c] called above before. I will explain run()[108a], 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 105a⟩+≡ (104e) ◁104d 106a▷

```
char cwd[256];
```

⟨dorecipe() if no recipe found 105b⟩≡ (104e)

```
/*
 *   no recipe? go to buggery!
 */
if(master_rule == nil){
    ⟨dorecipe() when no recipe found, if virtual or norecipe node 146f⟩
    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();
    }
}
```

7.5.1 Building the list of target nodes

To build the list of all targets, `dorecipe()`[104e] can rely on the `Rule.alltargets`[41d] field setup in `addrule()`[38b] (see Section 3.3.4):

⟨`dorecipe()` *other locals* 106a⟩+≡ (104e) ◁105a 152d▷
```
Word *last_alltargets = &alltargets;
char buf[BIGBLOCK];
```
Uses `BIGBLOCK` 181a.

⟨`dorecipe()` *build lists of targets and node list* 106b⟩≡ (104e)
```
nlist->next = nil;
alltargets.next = oldtargets.next = nil;
⟨dorecipe() if regexp rule 135f⟩
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 106c⟩
        n = s->u.ptr;

        ⟨dorecipe() update list of outdated targets 152e⟩

        // add_set(n, nlist)
        if(n == node)
            continue;
        n->next = nlist->next;
        nlist->next = n;
    }
}
```
Uses `META` 38a, `S_NODE` 87a, `newword()` 34d, `subst()` 86, and `symlook()` 32a.

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`[87a] namespace (see Section 6.4) to access the node of a target, and then chains together all the nodes with the `Node.next`[44b] 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()`[115c] to update the modification time and building status of those nodes in the graph.

⟨`dorecipe()` *sanity check s* 106c⟩≡ (106b)
```
if(s == nil)
    continue; /* not a node we are interested in */
```

7.5.2 Building the list of prerequisites

To build the list of prerequisites, `dorecipe()`[104e] 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()`[35b], which treats the list of words as a set and avoids adding duplicates:

⟨`dorecipe()` *build lists of prerequisites* 107⟩≡ (104e)

```
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 152k⟩
            ⟨dorecipe() explain when found arc a making target n out of date 131d⟩
        }
    } else {
        ⟨dorecipe() explain when found target n with no prerequisite 131e⟩
    }
}
```

Uses `addw()` 35b and `outofdate()` 103b.

Chapter 8

Scheduling Jobs

The scheduler has three responsibilities split across three functions, and the split is worth understanding before diving into the code. `run()`[108a] is the producer side: it appends a new job to `jobs`[45a] and, if a processor slot is free, kicks `sched()`[111c] to start it immediately. `sched()` is the launcher: it pops a job, hands it to `execsh()`[192c] to fork a shell, and records the resulting process id in the `events`[110a] slot table. `waitup()`[115c] is the reaper: it blocks on `wait()`, looks the returned pid up in `events` to find which job finished, calls `update()`[116b] to mark the targets `MADE`[42e], and then calls `sched()` again to launch the next queued job. The slot table is the glue: it is the only mapping from kernel-level pids back to graph-level nodes.

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()`[104e] before), which enqueues a job, and `waitup()` (called from `mk()`[99]), 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`[45a] I presented before and runs the scheduler:

⟨*function* `run` 108a⟩≡ (188c)
```
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` 45a, `nproclimit-18` 109a, `nrunning-17` 108b, and `sched()` 111c.

`run()`[108a] relies on the two globals below:

⟨*global* `nrunning` 108b⟩≡ (188c)
```
static int nrunning;
```

⟨*global* `nproclimit` 109a⟩≡ (188c)
```
static int nproclimit;
```

`nrunning`[108b] counts the number of processes currently running a recipe, while `nproclimit`[109a] 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()`[115c], which will wait for some process to finish, and which will call `sched()`[111c] 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()`[192c]) and looks for this variable to setup `nproclimit` with the code below:

⟨`mk()` *initializations* 109b⟩≡ (99) 153e▷
```
nproc(); /* it can be updated dynamically */
```
Uses `nproc()` 109c.

⟨*function* `nproc` 109c⟩≡ (188c)
```
void
nproc(void)
{
    Symtab *sym;
    Word *w;

    if(sym = symlook("NPROC", S_VAR, nil)) {
        w = sym->u.ptr;
        if (!empty_words(w))
            nproclimit = atoi(w->s);
    }
    if(nproclimit < 1)
        nproclimit = 1;
    ⟨nproc() if DEBUG(D_EXEC) 171f⟩

    ⟨nproc() grow nevents if necessary 110c⟩
}
```
Uses `S_VAR` 31a, `empty_words` 62a, `nproclimit-18` 109a, and `symlook()` 32a.

8.2 Scheduling jobs

Before showing the code of `sched()`[111c], I need to present an important data structure used by `sched()`.

8.2.1 `RunEvent` and `events`

`sched()`[111c] 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()`[115c]. 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` 109d⟩≡ (188c)
```
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` 110a⟩≡ (188c)
```
// growing_array<Runevent> (size = nevents (== nproclimit))
static RunEvent *events;
```

⟨*global* `nevents` 110b⟩≡ (188c)
```
static int nevents;
```

`events`[110a] is a growing array. The index of a mapping in this array is called a *slot*. The growing array is initialized in `nproc()`[109c]:

It is easier to see what a slot is by drawing `events` at a single point during a parallel build. Suppose `mk -P3` is building the tiny project from Section 6.1 with `nproclimit=3`. At the moment shown, the two `cc -c` jobs for `hello.o` and `world.o` are already running, the `ar` job for `libutil.a` is waiting on its `.o` prereqs (which are not in this project so it is ready too), and the final link for `hello` is still in the `jobs`[45a] queue because its prereq `hello.o` is `BEINGMADE`:

```
              events[] (size = nproclimit = 3)

      slot  pid    job                 state
      ----  -----  ------------------  ------------------
       0    4217   cc -c hello.c       running  (SIGCHLD
                                                 will land
                                                 in pidslot)
       1    4219   cc -c world.c       running
       2     0     -                   free  (nextslot()
                                               returns 2)

              jobs (linked list, waiting)

      head -> [ link hello   (waits on hello.o BEINGMADE) ]
           -> [ link world   (waits on world.o BEINGMADE) ]
           -> nil
```

Three details hide in this picture. `nextslot()`[111a] uses `pid <= 0` as the "free" predicate, so a slot is reclaimed simply by zeroing its `pid` in `waitup()`—no list splicing. `pidslot()`[111b] is the inverse: given the pid reported by `await()`, it scans `events` linearly until it finds the match, which is fine because `nproclimit` is tiny (the number of CPU cores, not the number of jobs). And the jobs whose prereqs are still `BEINGMADE` never reach this table at all—they sit in `jobs` until `waitup()` calls `update()`[116b] on a finished node, which then re-evaluates whether any waiting job can be enqueued into `events` by a follow-up `sched()` call.

⟨`nproc()` *grow nevents if necessary* 110c⟩≡ (109c)
```
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()` 174a, `Realloc()` 174b, `events-15` 110a, `nevents-16` 110b, and `nproclimit-18` 109a.

`sched()` relies on a few helper functions to operate on `events`. `nextslot()` below finds an available slot in `events`:

⟨*function* `nextslot` 111a⟩≡ (188c)
```
int
nextslot(void)
{
    int i;

    for(i = 0; i < nproclimit; i++)
        if(events[i].pid <= 0)
            return i;
    assert(/*out of slots!!*/ false);
    return 0; /* cyntax */
}
```
Uses `events-15` 110a and `nproclimit-18` 109a.

`pidslot()` below finds the slot of the mapping for a certain process id:

⟨*function* `pidslot` 111b⟩≡ (188c)
```
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) 171e⟩
    return -1;
}
```
Uses `events-15` 110a and `nevents-16` 110b.

8.2.2 `sched()`

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

⟨*function* `sched` 111c⟩≡ (188c)
```
static void
sched(void)
{
    Job *j;
    int slot;
    char *flags;
    ShellEnvVar *e;
    ⟨sched() other locals 128a⟩

    ⟨sched() return if no jobs 112⟩

    // j = pop(jobs)
    j = jobs;
    jobs = j->next;
    ⟨sched() if DEBUG(D_EXEC) 170e⟩

    slot = nextslot();
    events[slot].job = j;

    e = buildenv(j, slot);
    ⟨sched() print recipe command on stdout 128b⟩
```

```
    ⟨sched() if dry mode or touch mode, alternate to execsh 132c⟩
    else {
        ⟨sched() if DEBUG(D_EXEC) print recipe 171a⟩
         flags = "-e";
        ⟨sched() reset flags if NOMINUSE rule 148a⟩

         // launching the job!
         events[slot].pid = execsh(flags, j->r->recipe, nil, e);

         usage();
         nrunning++;
        ⟨sched() if DEBUG(D_EXEC) print pid 171b⟩
    }
}
```

Uses buildenv() 125b, events-15 110a, jobs 45a, nextslot() 111a, nrunning-17 108b, and usage() 134a.

The most important call in the code above is the call to `execsh()`[192c]. `execsh()` 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()`[111c] above increments `nrunning`[108b] 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`[110a].

`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]).

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()`[125b] 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()`[115c], in which case the job queue may be empty. In that case, `sched()` simply returns.

⟨sched() return if no jobs 112⟩≡ (111c)

```
if(jobs == nil){
    usage();
    return;
}
```

Uses jobs 45a and usage() 134a.

8.3 Executing Jobs: `execsh()`

`execsh()`[192c] is the bridge between `mk` and the shell. The two-fork dance below is not gratuitous: there are two reasonable ways to feed a recipe to a shell, and `mk` picks the harder one for good reasons. The simple way is `sh -c "recipe text"`, which would mean stuffing the recipe through the `argv` of `execl()`; the problem is that the recipe may contain quotes, backslashes, and other characters that `mk` would have to re-escape to survive the shell's command-line parsing. The `mk` way is to feed the recipe through standard input: the shell sees raw bytes, no re-escaping is needed, and the recipe can be arbitrarily long. The price is that something must produce those bytes on the writing end of a pipe while the shell consumes them, hence the second fork:

```
mk parent
    |
    | rfork(RFPROC|RFFDG|RFENVG)
    v
```

```
child (will exec the shell)
in[0] = read end, in[1] = write end of pipe
    |
    | fork()
    v
grandchild
(writes recipe bytes into in[1], then exits)

child after grandchild forks:
    dup(in[0], STDIN); close(in[0]); close(in[1]);
    execl(shell, ...)
```

The reason the child (not the grandchild) execs the shell is that the `mk` parent needs to know the shell's pid for `waitup()`[115c], and only the direct child's pid is reported back through `rfork()`. The grandchild that does the writing exits quickly, before the shell finishes; its termination is reaped incidentally by `waitup()` and discarded by the `NOT_A_JOB_PROCESS`[117c] code path.

`execsh()` relies on the global below to know which shell to execute. Under Plan 9, `mk` uses the shell `rc` (see the SHELL book [Pad18]):

⟨*global* `shell` 113a⟩≡ (198)
```
//char  *shell = "/bin/rc";
```

⟨*global* `shellname` 113b⟩≡ (198)
```
//char  *shellname = "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()`[111c]):

⟨*global* `shflags` 113c⟩≡ (185a)
```
char *shflags = "-I"; /* rc flag to force non-interactive mode */
```
Uses `shflags` 113c.

`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` 113d⟩≡ (198)
```
int
execsh(char *shargs, char *shinput, Bufblock *buf, ShellEnvVar *e)
{
    int pid1, pid2;
    fdt in[2]; // pipe descriptors
    int err;
    ⟨execsh() other locals 114a⟩

    ⟨execsh() if buf then create pipe to save output 139b⟩

    pid1 = rfork(RFPROC|RFFDG|RFENVG);
    ⟨execsh() sanity check pid rfork 114c⟩
    // child
    if(pid1 == 0){
        ⟨execsh() in child, if buf, close one side of pipe 139c⟩
        err = pipe(in);
        ⟨execsh() sanity check err pipe 115b⟩
        pid2 = fork();
        ⟨execsh() sanity check pid fork 115a⟩
        // parent of grandchild, the shell interpreter
        if(pid2 != 0){
            // input must come from the pipe
```

```
            dup(in[0], STDIN);
            ⟨execsh() in child, if buf, dup and close 139d⟩
            close(in[0]);
            close(in[1]);
            ⟨execsh() in child, export environment before exec 125c⟩
            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 139e⟩
        close(in[0]);
        // feed the shell
        ⟨execsh() in grandchild, write cmd in pipe 114b⟩
        close(in[1]); // will flush
        _exits(nil);
    }
    // parent
    ⟨execsh() in parent, if buf, close other side of pipe and read output 139g⟩
    return pid1;
}
```

Uses shflags 113c.

`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`[110a] 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* 114a⟩≡ (113d) 139a▷

```
char *endshinput;
```

⟨execsh() *in grandchild, write cmd in pipe* 114b⟩≡ (113d)

```
endshinput = shinput + strlen(shinput);
while(shinput < endshinput){
    n = write(in[1], shinput, endshinput - shinput);
    if(n < 0)
        break;
    shinput += n;
}
```

⟨execsh() *sanity check pid rfork* 114c⟩≡ (113d)

```
if(pid1 < 0){
    perror("mk rfork");
    Exit();
}
```

⟨execsh() *sanity check pid fork* 115a⟩≡ (113d)
```
if(pid2 < 0){
    perror("mk fork");
    Exit();
}
```

⟨execsh() *sanity check err pipe* 115b⟩≡ (113d)
```
if(err < 0){
    perror("pipe");
    Exit();
}
```

8.4 Waiting for jobs to finish

The complication of `waitup()`[115c] comes from a constraint imposed by the kernel: under Plan 9 (and UNIX), `wait()`/`await()` cannot ask "has process X finished?" It can only ask "has *any* child finished?" This means `mk` cannot isolate job processes from other children it may have forked for unrelated reasons—backquote command substitution (Section 11.2), dynamic `mkfile` inclusion (Section 11.3), and custom dependency comparison programs (Section 11.5.7) all spawn processes that will eventually be reported by `wait()` on a future `waitup()` call. This is why `waitup()` has to look up every returned pid in the `events`[110a] slot table and gracefully handle the case where the pid is not a job—the `NOT_A_JOB_PROCESS`[117c] return value covers exactly that case. On Linux, `waitpid(pid, ...)` would let `mk` target a specific child and avoid this whole class of edge cases, but Plan 9's `await()` does not offer that.

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()`[111c] scheduled a job and modified `events`[110a], `mk()`[99] 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()`[116b] to update the graph of dependencies and `sched()` to schedule more jobs:

⟨*function* waitup 115c⟩≡ (188c)
```
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 118b⟩

    ⟨waitup() if retstatus, check process list 151c⟩
again:  /* rogue processes */

    pid = xwaitfor(buf);
    ⟨waitup() if no more children 118a⟩
```

```
    ⟨waitup() if DEBUG(D_EXEC) print pid 171c⟩
    ⟨waitup() if retstatus, check if matching pid 151d⟩

    slot = pidslot(pid);
    ⟨waitup() if slot not found, not a job pid, update process list 151a⟩

    j = events[slot].job;
    usage();
    nrunning--;
    // free events[slot]
    events[slot].pid = -1;

    ⟨waitup() if error in child process, possibly set fake or exit 118c⟩
    // 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 118d⟩
        update(node, fake);
    }

    if(nrunning < nproclimit)
        sched();
    return JOB_ENDED;
}
```

Uses JOB_ENDED 117c, S_NODE 87a, events-15 110a, nproclimit-18 109a, nrunning-17 108b, pidslot() 111b, sched() 111c, symlook() 32a, update() 116b, and usage() 134a.

`waitup()`[115c] above relies on the helper function `waitfor()` 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* waitfor 116a⟩≡ (198)

```
int
xwaitfor(char *msg)
{
    Waitmsg *w;
    int pid;

    // blocking call, wait for any children
    w = wait();
    // no more children
    if(w == nil)
        return -1;
    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()`[115c], marks the target nodes of a job as `MADE`[42e] and updates the modification time of those nodes.

⟨*function* update 116b⟩≡ (190b)

```
void
update(Node *node, bool fake)
{
    Arc *a;

    ⟨update() if fake 165c⟩
    else
       MADESET(node, MADE);
    ⟨update() debug 172b⟩

    ⟨update() if virtual node or inexistent file 146g⟩
    else {
        node->time = timeof(node->name, true);
        ⟨update() unpretend node 159f⟩
        ⟨update() set outofdate prereqs if arc prog 149c⟩
    }
}
```

Uses MADE 42e, MADESET 101b, and timeof() 88a.

I will present the use of the `fake` parameter later in Section 11.12.2.

8.4.3 `waitup()` edge cases

As I mentioned before, the interface of `waitup()`[115c] is complex:

⟨*signature waitup* 117a⟩≡
```
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 117b⟩≡ (181d)
```
enum WaitupParam {
  EMPTY_CHILDREN_IS_OK = 1,
  EMPTY_CHILDREN_IS_ERROR = -1,
  ⟨WaitupParam other cases 140b⟩
};
```

The return value of `waitup()`, of the type below, describes a few possible scenarios:

⟨*type* WaitupResult 117c⟩≡ (181d)
```
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 11.5.7) 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 7.2, `mk()`[99] 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* 118a⟩≡ (115c)
```
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 117c and EMPTY_CHILDREN_IS_OK 117b.

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* 118b⟩≡ (115c) 128c ▷
```
Bufblock *bp;
```

⟨waitup() *if error in child process, possibly set fake or exit* 118c⟩≡ (115c)
```
if(buf[0]){
    bp = newbuf();
    ⟨waitup() if error in child process, print recipe in bp 128d⟩
    fprint(STDERR, "mk: %s: exit status=%s", bp->start, buf);
    freebuf(bp);
    ⟨waitup() when error in child process, delete if DELETE node 147e⟩
    fprint(STDERR, "\n");

    ⟨waitup() when error in child process, if kflag 165b⟩
    else {
        jobs = nil;
        Exit();
    }
}
```
Uses freebuf() 175e, jobs 45a, and newbuf() 175d.

⟨waitup() *skip if node not found* 118d⟩≡ (115c)
```
if(sym == nil)
    continue; /* not interested in this node */
```

8.5 Process management, 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 118e⟩≡ (198)
```
void
Exit(void)
{
    while(waitpid() >= 0)
        ;
    exits("error");
}
```

8.6 Notes (signals) management

Pressing `^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; `mk` and its forked shells all receive it together. The naive response—let `mk` `exit` right away—is wrong for the same reason killing a build with `Ctrl-C` in Make sometimes leaves half-written object files: a child compiler that gets cut mid-write may leave a corrupt file on disk that looks up-to-date the next time you build, and `mk` (or Make) will skip recompiling it. The fix in `killchildren()` below is to clear the job queue (so no new jobs start), keep `waitup()`[115c] alive long enough for all running children to finish their current operation, and only then `Exit()`[192c]. This is the same reason ordinary error handling routes through `Exit()` instead of `exits()` directly.

⟨`main()` *initializations before building* 119a⟩≡ (47b) 122a ▷
```
catchnotes();
```

⟨*function* `catchnotes` 119b⟩≡ (198)
```
void
catchnotes()
{
    atnotify(notifyf, 1);
}
```

⟨*function* `notifyf` 119c⟩≡ (198)
```
int
notifyf(void *a, char *msg)
{
    ⟨notifyf() sanity check not too many notes 119d⟩
    if(strcmp(msg, "interrupt")!=0 && strcmp(msg, "hangup")!=0)
        return 0;
    killchildren(msg);
    return -1;
}
```
Uses `killchildren()` 119e.

⟨`notifyf()` *sanity check not too many notes* 119d⟩≡ (119c)
```
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` 119e⟩≡ (188c)
```
void
killchildren(char *msg)
{
    ⟨killchildren() locals 151e⟩

    jobs = nil;  /* make sure no more get scheduled */
    kflag = true; /* to make sure waitup doesn't exit */

    ⟨killchildren() expunge not-job processes 151f⟩

    while(waitup(EMPTY_CHILDREN_IS_OK, (int *)nil) == JOB_ENDED)
        ;
    Bprint(&bout, "mk: %s\n", msg);
    Exit();
}
```
Uses `EMPTY_CHILDREN_IS_OK` 117b, `JOB_ENDED` 117c, `bout` 48a, `jobs` 45a, `kflag` 164e, and `waitup()` 115c.

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. As a side benefit, 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 Shellenv and shellenv

The symbol table of `mk` (`hash`[31b]) contains already in the `S_VAR`[31a] 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`[33c] 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` 120a⟩≡ (181d)
```
struct ShellEnvVar
{
    // ref<string>, the key
    char   *name;

    // list<ref_own<string>>, the value
    Word   *values;
};
```

All the variables and their values are stored in the following global:

⟨*global* `shellenv` 120b⟩≡ (186b)
```
// growing_array<ShellEnvVar> (endmarker = (nil,nil), size = envinsert.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` 120c⟩≡ (186b)
```
// idx for next free entry in shellenv array
static int nextv;
```

⟨*function* `envinsert` 121a⟩≡ (186b)
```
static void
envinsert(char *name, Word *value)
{
    ⟨envinsert() locals 121b⟩

    ⟨envinsert() grow array if necessary 121c⟩
    shellenv[nextv].name = name;
    shellenv[nextv].values = value;
    nextv++;
}
```
Uses `nextv-7` 120c and `shellenv` 120b.

⟨`envinsert()` *locals* 121b⟩≡ (121a)
```
static int envsize = 0;
```

⟨`envinsert()` *grow array if necessary* 121c⟩≡ (121a)
```
if (nextv >= envsize) {
    envsize += ENVQUANTA;
    shellenv = (ShellEnvVar *) Realloc((char *) shellenv,
                                       envsize*sizeof(ShellEnvVar));
}
```
Uses `ENVQUANTA-6` 121d, `Realloc()` 174b, `nextv-7` 120c, and `shellenv` 120b.

⟨*constant* `ENVQUANTA` 121d⟩≡ (186b)
```
#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` 121e⟩≡ (186b)
```
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 121f⟩
}
```
Uses `freewords()` 35a and `shellenv` 120b.

⟨`envupd()` *if variable not found* 121f⟩≡ (121e)
```
// else
e->name = name;
e->values = value;
envinsert(nil,nil);
```
Uses `envinsert()` 121a.

9.2 Initializing the shell environment: `initenv()`

To initialize `shellenv`[120b], `main()`[47b] calls `initenv()`[122b] before calling `mk()`[99]:

⟨`main()` *initializations before building* 122a⟩+≡ (47b) ◁119a 132f▷
```
initenv();
```
Uses `initenv()` 122b.

⟨*function* `initenv` 122b⟩≡ (186b)
```
void
initenv(void)
{
    char **p;

    nextv = 0; // reset envy

    // 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` 31a, `ecopy()` 122c, `envinsert()` 121a, `nextv-7` 120c, `specialvars-8` 33d, `stow()` 64b, and `symtraverse()` 33a.

I described `symtraverse()`[33a] before. It allows to iterate over a namespace while applying a function, here `ecopy()`:

⟨*function* `ecopy` 122c⟩≡ (186b)
```
static void
ecopy(Symtab *s)
{
    char **p;

    ⟨ecopy() return and do not copy if S_NOEXPORT symbol 152c⟩
    ⟨ecopy() return and do not copy if conflict with mk internal variable 122d⟩
     // else
     envinsert(s->name, s->u.ptr);
}
```
Uses `envinsert()` 121a.

Note that `initenv()` calls `envinsert()`[121a] 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()`[121e] 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* 122d⟩≡ (122c)
```
for(p = specialvars; *p; p++)
    if(strcmp(*p, s->name) == 0)
        return;
```
Uses `specialvars-8` 33d.

9.3 Importing the environment: `readenv()`

`initenv()`[122b] iterates over the `S_VAR`[31a] namespace with `symtraverse()`[33a] to add in `shellenv`[120b] 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()`[47b] calls `inithash()`[34a] 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` 123a⟩≡ (198)

```
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 124b⟩
    while((n = dirread(envdir, &e)) > 0){
        for(i = 0; i < n; i++){
            len = e[i].length;
            ⟨readenv() skip some names 123b⟩

            snprint(name, sizeof name, "/env/%s", e[i].name);
            envfile = open(name, OREAD);
            ⟨readenv() sanity check envfile 124c⟩
            p = Malloc(len+1);
            len2 = read(envfile, p, len);
            ⟨readenv() sanity check len2 124d⟩
            close(envfile);
            ⟨readenv() add null terminator character at end of p 125a⟩
            w = encodenulls(p, len);
            free(p);
            p = strdup(e[i].name);

            // populating symbol table
            setvar(p, (void *) w);
        }
        free(e);
    }
    close(envdir);
}
```

Uses `Malloc()` 174a and `setvar()` 33b.

`readenv()`[192c] 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* 123b⟩≡ (123a)

```
/* 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` 33c, `shname()` 124a, and `symlook()` 32a.

⟨*function* shname 124a⟩≡ (185c)
```
char *
shname(char *a)
{
    Rune r;
    int n;

    while (*a) {
        n = chartorune(&r, a);
        if (!WORDCHR(r))
            break;
        a += n;
    }
    return a;
}
```
Uses WORDCHR 68e.

⟨readenv() *sanity check envdir* 124b⟩≡ (123a)
```
if(envdir < 0)
    return;
```

⟨readenv() *sanity check envfile* 124c⟩≡ (123a)
```
if(envfile < 0)
    continue;
```

⟨readenv() *sanity check len2* 124d⟩≡ (123a)
```
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 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 124e⟩≡ (198)
```
/* 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 125a⟩≡ (123a)
```
if (p[len-1] == '\0')
    len--;
else
    p[len] = '\0';
```

9.4 Adjusting the shell environment: buildenv()

Once shellenv[120b] has been initialized, sched()[111c] can call buildenv() to adjust the environment with the specific values of mk's special variable for a specific job:

⟨*function* buildenv 125b⟩≡ (186b)
```
ShellEnvVar*
buildenv(Job *j, int slot)
{
    ⟨buildenv() locals 137a⟩

    // main variables
    envupd("target", wdup(j->t));
    ⟨buildenv() if regexp rule 135g⟩
    else
        envupd("stem", newword(j->stem));
    envupd("prereq", wdup(j->p));

    // advanced variables
    ⟨buildenv() envupd some variables 137b⟩

    return shellenv;
}
```

Uses envupd() 121e, newword() 34d, shellenv 120b, and wdup() 35c.

Note that as I mentioned in Section 9.1, buildenv()[125b] 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()[111c] updated shellenv[120b] with buildenv()[125b] and called execsh()[192c] with this environment, execsh() forks a shell interpreter and calls exportenv()[192c] in the child process to modify its own environment:

⟨execsh() *in child, export environment before exec* 125c⟩≡ (113d)
```
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` 126a⟩≡ (198)
```
/* 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 126c⟩

    for(;e->name; e++){
        ⟨exportenv() skip entry if not a user variable and no value 126d⟩
        // else
        snprint(name, sizeof name, "/env/%s", e->name);
        ⟨exportenv() if existing symbol but no value, remove from env 127a⟩
        // else
        f = create(name, OWRITE, 0666L);
        ⟨exportenv() sanity check f 127b⟩
        first = true;
        for (w = e->values; w; w = w->next) {
            n = strlen(w->s);
            if (n) {
                ⟨exportenv() write null separator 126b⟩
                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* 126b⟩≡ (126a)
```
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* 126c⟩≡ (126a)
```
Symtab *sym;
bool hasvalue;
```

⟨`exportenv()` *skip entry if not a user variable and no value* 126d⟩≡ (126a)
```
hasvalue = !empty_words(e->values);
sym = symlook(e->name, S_VAR, nil);
if(sym == nil && !hasvalue) /* non-existant null symbol */
    continue;
```
Uses `S_VAR` 31a, `empty_words` 62a, and `symlook()` 32a.

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* 127a⟩≡ (126a)

```
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()` 35a.

⟨`exportenv()` *sanity check f* 127b⟩≡ (126a)

```
if(f < 0) {
    fprint(STDERR, "can't create %s, f=%d\n", name, f);
    perror(name);
    continue;
}
```

Chapter 10

Debugging and Profiling Support TODO

When a recipe fails, `mk` must tell the user what went wrong. The complication is that recipes contain variable references (e.g., `$CFLAGS`) that were expanded before the shell saw them. The `shprint()`[129a] function re-expands a recipe with the current environment so that the error message shows the actual command that was executed, not the template with unexpanded variables.

10.1 Printing jobs: `shprint()`

⟨`sched()` *other locals* 128a⟩≡ (111c) 132b▷
```
Bufblock *buf;
```

⟨`sched()` *print recipe command on stdout* 128b⟩≡ (111c)
```
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` 147g, `bout` 48a, `freebuf()` 175e, `newbuf()` 175d, `nflag` 131f, `shprint()` 129a, and `tflag` 159g.

⟨`waitup()` *other locals* 128c⟩+≡ (115c) ◁118b 147d▷
```
ShellEnvVar *e;
```

⟨`waitup()` *if error in child process, print recipe in* `bp` 128d⟩≡ (118c)
```
e = buildenv(j, slot);
shprint(j->r->recipe, e, bp);
front(bp->start);
```
Uses `buildenv()` 125b, `front()` 128e, and `shprint()` 129a.

⟨*function* `front` 128e⟩≡ (187a)
```
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);
}
```

⟨*function* shprint 129a⟩≡ (187a)

```
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() 130d, insert() 176a, rinsert() 176b, and vexpand() 129b.

10.1.1 Expanding and printing variables

⟨*function* vexpand 129b⟩≡ (187a)

```
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() 176c, mygetenv() 130a, and shname() 124a.

⟨*function* mygetenv 130a⟩≡ (187a)

```
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 33c, S_WESET 130b, symlook() 32a, and wtos() 35d.

⟨Sxxx *cases* 130b⟩+≡ (31a) ◁87a 149b▷

```
S_WESET, /* variable; we set in the mkfile */
```

⟨parse() *when parsing variable definitions, extra setting* 130c⟩≡ (78b)

```
symlook(head->s, S_WESET, (void *)"");
```

Uses S_WESET 130b and symlook() 32a.

10.1.2 Printing quoted strings

⟨*function* copyq 130d⟩≡ (185a)

```
/*
 * 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() 131a and rinsert() 176b.

⟨*function* copysingle 131a⟩≡ (185a)

```
/*
 * 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() 176b.

10.2 Explain mode: mk -e

⟨*global* explain 131b⟩≡ (183)

```
bool explain = false;
```

Uses explain 131b.

⟨main() *-xxx switch cases* 131c⟩+≡ (48d) ◁52d 132a▷

```
case 'e':
    explain = true;
    break;
```

Uses explain 131b.

⟨dorecipe() *explain when found arc* a *making target* n *out of date* 131d⟩≡ (107)

```
if(explain)
    fprint(STDOUT, "%s(%ld) < %s(%ld)\n",
        n->name, n->time, a->n->name, a->n->time);
```

Uses explain 131b.

⟨dorecipe() *explain when found target* n *with no prerequisite* 131e⟩≡ (107)

```
if(explain)
    fprint(STDOUT, "%s has no prerequisites\n", n->name);
```

Uses explain 131b.

10.3 Dry mode: mk -n

⟨*global* nflag 131f⟩≡ (183)

```
bool nflag = false;
```

Uses nflag 131f.

⟨main() *-xxx switch cases* 132a⟩+≡ (48d) ◁131c 132e▷
```
case 'n':
    nflag = true;
    break;
```
Uses nflag 131f.

⟨sched() *other locals* 132b⟩+≡ (111c) ◁128a
```
Node *n;
```

⟨sched() *if dry mode or touch mode, alternate to execsh* 132c⟩≡ (111c)
```
if(nflag || tflag){
    for(n = j->n; n; n = n->next){
        ⟨sched() if touch mode 160b⟩
        n->time = time((long *)nil);
        MADESET(n, MADE);
    }
}
```
Uses MADE 42e, MADESET 101b, nflag 131f, and tflag 159g.

10.4 What-if mode: mk -w*file*

⟨main() *locals* 132d⟩+≡ (47b) ◁52c 163d▷
```
Bufblock *whatif = nil;
```

⟨main() *-xxx switch cases* 132e⟩+≡ (48d) ◁132a 133b▷
```
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() 49a, bufcpy() 176c, insert() 176a, and newbuf() 175d.

⟨main() *initializations before building* 132f⟩+≡ (47b) ◁122a
```
if(whatif){
    insert(whatif, '\0');
    timeinit(whatif->start);
    freebuf(whatif);
}
```
Uses freebuf() 175e, insert() 176a, and timeinit() 132g.

⟨*function* timeinit 132g⟩≡ (188b)
```
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 160e and symlook() 32a.

10.5 Processor utilization: mk -u

⟨*global* uflag 133a⟩≡ (192b)

```
bool uflag = false;
```

Uses uflag 133a.

⟨main() *-xxx switch cases* 133b⟩+≡ (48d) ◁132e 158b▷

```
case 'u':
    uflag = true;
    break;
```

Uses uflag 133a.

⟨*global* tslot 133c⟩≡ (188c)

```
// map<nrunning, int>
static ulong tslot[1000];
```

⟨*global* tick 133d⟩≡ (188c)

```
static ulong tick;
```

⟨main() *setup profiling* 133e⟩≡ (48c) 173b▷

```
usage();
```

Uses usage() 134a.

⟨main() *profile initializations* 133f⟩≡ (48c)

```
usage();
```

Uses usage() 134a.

⟨*function* `usage` 134a⟩≡ (188c)

```
void
usage(void)
{
    ulong t;

    t = time(nil);
    if(tick)
        tslot[nrunning] += t - tick;
    tick = t;
}
```

Uses `nrunning-17` 108b, `tick-22` 133d, and `tslot-21` 133c.

⟨`main()` *print profiling stats if uflag* 134b⟩≡ (47b)

```
if(uflag)
    prusage();
```

Uses `prusage()` 134c and `uflag` 133a.

⟨*function* `prusage` 134c⟩≡ (188c)

```
void
prusage(void)
{
    int i;

    usage();
    for(i = 0; i <= nevents; i++)
        fprint(STDOUT, "%d: %lud\n", i, tslot[i]);
}
```

Uses `nevents-16` 110b, `tslot-21` 133c, and `usage()` 134a.

Chapter 11

Advanced Features TODO

The earlier chapters covered the core pipeline of `mk`: parsing rules, building the dependency graph, finding outdated targets, and scheduling jobs. This chapter covers features that extend that core: regular-expression meta-rules (a more powerful alternative to `%`-patterns), aggregated targets, virtual targets, parallel execution, and the `mkfile` inclusion mechanism.

11.1 Regular-expression rules: `:R:`

⟨`Rule_attr` *cases* 135a⟩≡ (38a) 146a▷
```
REGEXP = 0x0020,
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 135b⟩≡ (75d) 146b▷
```
case 'R':
    *attr |= REGEXP;
    break;
```

⟨`Rule` *other fields* 135c⟩+≡ (36) ◁91a 148i▷
```
Reprog  *pat;  /* reg exp goo */
```

⟨*global* `patrule` 135d⟩≡ (183)
```
Rule *patrule;
```

⟨`addrule()` *if REGEXP attribute* 135e⟩≡ (39a)
```
if(attr&REGEXP){
    patrule = r;
    r->pat = regcomp(target);
}
```
Uses `REGEXP` 135a and `patrule` 135d.

⟨`dorecipe()` *if regexp rule* 135f⟩≡ (106b)
```
if(master_rule->attr&REGEXP){
    last_oldtargets->next = newword(node->name);
    last_alltargets->next = newword(node->name);
}
```
Uses `REGEXP` 135a and `newword()` 34d.

⟨`buildenv()` *if regexp rule* 135g⟩≡ (125b)
```
if(j->r->attr&REGEXP)
    envupd("stem", newword(""));
```
Uses `REGEXP` 135a, `envupd()` 121e, and `newword()` 34d.

⟨*function* `regerror` 136a⟩≡ (187d)
```
//@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` 54a, `mkinline` 54c, and `patrule` 135d.

⟨*constant* `NREGEXP` 136b⟩≡ (181d)
```
#define  NREGEXP  10
```

⟨`Arc` *other fields* 136c⟩+≡ (43b) ◁94c 148l▷
```
char  *match[NREGEXP];
```
Uses `NREGEXP` 136b.

⟨`newarc()` *set other fields* 136d⟩+≡ (43e) ◁94d 148m▷
```
rcopy(a->match, match, NREGEXP);
```
Uses `NREGEXP` 136b.

⟨*function* `rcopy` 136e⟩≡ (198)
```
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* 136f⟩≡ (44a) 152g▷
```
char  **match;
```

⟨`specialvars` *other array elements* 136g⟩≡ (33d) 152f▷
```
"stem0",  /* must be in order from here */
"stem1",
"stem2",
"stem3",
"stem4",
"stem5",
"stem6",
"stem7",
"stem8",
"stem9",
```

⟨buildenv() *locals* 137a⟩≡ (125b) 153h▷
```
char **p;
int i;
```

⟨buildenv() *envupd some variables* 137b⟩≡ (125b) 152i▷
```
/* 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 135a, envupd() 121e, newword() 34d, and specialvars-8 33d.

⟨applyrules *other locals* 137c⟩+≡ (81) ◁84c
```
Resub rmatch[NREGEXP];
```
Uses NREGEXP 136b.

⟨applyrules *other initializations* 137d⟩≡ (81)
```
memset((char*)rmatch, 0, sizeof(rmatch));
```

⟨applyrules() *if regexp rule then continue if some conditions* 137e⟩≡ (83e)
```
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 136b, REGEXP 135a, and patrule 135d.

⟨applyrules() *if regexp rule, adjust buf and rmatch* 137f⟩≡ (83e)
```
if(r->attr&REGEXP)
    regsub(pre->s, buf, sizeof(buf), rmatch, NREGEXP);
```
Uses NREGEXP 136b and REGEXP 135a.

11.2 Shell-command expansion: ‘*cmd*‘

Backquote expansion is the mkfile equivalent of the shell's $(ls *.c): 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. The implementation in bquote()[138a] is subtle because the same Bufblock[174c] 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()[192c] and rewinds buf->current to that point so the output overwrites the command in place. The grandchild-pipe trick from Section 11.2 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* 137g⟩+≡ (57b) ◁60a
```
case '‘':
    if (bquote(bp, buf) == ERROR_0)
        Exit();
    break;
```
Uses bquote() 138a.

⟨*function* `bquote` 138a⟩≡ (188a)
```
/*
 * 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 138c⟩
    if(c == '{'){
        term = '}';  /* rc style */
        ⟨bquote() skip spaces 138c⟩
    } 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 138d⟩
            return OK_1;
        }
        if(c == '\n')
            break; // go to error
        ⟨bquote() handle quotes and escape characters 138b⟩
        rinsert(buf, c);
    }
    SYNERR(line);
    fprint(STDERR, "missing closing %c after `\n", term);
    return ERROR_0;
}
```
Uses `SYNERR` 54d, `insert()` 176a, `mkinline` 54c, `nextrune()` 58a, and `rinsert()` 176b.

⟨`bquote()` *handle quotes and escape characters* 138b⟩≡ (138a)
```
if(c == '\'' || c == '"' || c == '\\'){
    insert(buf, c);
    if(!escapetoken(bp, buf, true, c))
        return ERROR_0;
    continue;
}
```
Uses `escapetoken()` 60b and `insert()` 176a.

⟨`bquote()` *skip spaces* 138c⟩≡ (138)
```
while((c = Bgetrune(bp)) == ' ' || c == '\t')
        ;
```

11.2.2 Adjusting `execsh()`

⟨`bquote()` *execute shell command in* `buf` 138d⟩≡ (138a)
```
initenv();
```

```
  // running the command, passing a buf argument
  execsh(nil, buf->current, buf, shellenv);
```
Uses initenv() 122b and shellenv 120b.

⟨execsh() *other locals* 139a⟩+≡ (113d) ◁114a 139f▷
```
  fdt out[2];
```

⟨execsh() *if buf then create pipe to save output* 139b⟩≡ (113d)
```
  if(buf && pipe(out) < 0){
      perror("pipe");
      Exit();
  }
```

⟨execsh() *in child, if buf, close one side of pipe* 139c⟩≡ (113d)
```
  if(buf)
      close(out[0]);
```

⟨execsh() *in child, if buf, dup and close* 139d⟩≡ (113d)
```
  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* 139e⟩≡ (113d)
```
  if(buf)
      close(out[1]);
```

⟨execsh() *other locals* 139f⟩+≡ (113d) ◁139a
```
  int tot, n;
```

⟨execsh() *in parent, if buf, close other side of pipe and read output* 139g⟩≡ (113d)
```
  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() 176d.

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()`[55], 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. 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* `<|` 140a⟩≡ (61b)
```
if(sep == '<' && *p == '|'){
    sep = '|';
    p++;
}
```

⟨`WaitupParam` *other cases* 140b⟩≡ (117b) 165e▷
```
EMPTY_CHILDREN_IS_ERROR3 = -3,
```

⟨`parse()` *other locals* 140c⟩+≡ (55) ◁78a 148j▷
```
int pid;
```

⟨`parse()` *switch rhead cases* 140d⟩+≡ (55) ◁78b
```
case '|':
    p = wtos(tail, ' ');
    ⟨parse() sanity check p for include program name 140e⟩

    initenv();
    pid = pipecmd(p, shellenv, &newfd);
    ⟨parse() sanity check newfd 140f⟩
    else
        // recursive call
        parse(p, newfd, 0);

    while(waitup(EMPTY_CHILDREN_IS_ERROR3, &pid) >= 0)
        ;
    ⟨parse() sanity check pid after waitup 140g⟩
    break;
```
Uses `EMPTY_CHILDREN_IS_ERROR3` 140b, `initenv()` 122b, `parse()` 55, `shellenv` 120b, `waitup()` 115c, and `wtos()` 35d.

⟨`parse()` *sanity check* `p` *for include program name* 140e⟩≡ (140d)
```
if(*p == '\0'){
    SYNERR(-1);
    fprint(STDERR, "missing include program name\n");
    Exit();
}
```
Uses `SYNERR` 54d.

⟨`parse()` *sanity check* `newfd` 140f⟩≡ (140d)
```
if(newfd < 0){
    fprint(STDERR, "warning: skipping missing program file: ");
    perror(p);
}
```

⟨`parse()` *sanity check* `pid` *after waitup* 140g⟩≡ (140d)
```
if(pid != 0){
    fprint(STDERR, "bad include program status\n");
    Exit();
}
```

⟨*function* pipecmd 141a⟩≡ (198)

```
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 168c, D_EXEC 168a, and shflags 113c.

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()[61b] is hunting for, hence the vargen flag in charin()[62b] that suppresses separator detection between $ and .

11.4.1 Parsing adjustments

⟨charin() *switch rune cases* 141b⟩+≡ (62b) ◁63a

```
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* 142a⟩≡ (62b)

```
    if(vargen){
        SYNERR(-1);
        fprint(STDERR, "missing closing } in pattern generator\n");
    }
```

Uses SYNERR 54d.

⟨varsub() *if variable starts with open brace* 142b⟩≡ (67d)

```
    if(**s == '{')  /* either ${name} or ${name: A%B==C%D}*/
        return expandvar(s);
```

Uses expandvar() 142c.

⟨*function* expandvar 142c⟩≡ (191b)

```
    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 54d, S_VAR 31a, charin() 62b, freebuf() 175e, newword() 34d, symlook() 32a, and varname() 68d.

11.4.2 Substitutions: subsub()

⟨*function* subsub 143⟩≡ (191b)

```
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 144⟩≡ (191b)

```
    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() 62b and stow() 64b.

⟨*function* submatch 145a⟩≡ (191b)
```
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. 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), and `R` (the target is a regular expression). The implementation pattern is the same for all of them: `rhead()`[61b] sets a bit in `Rule.attr` when it sees the letter, `attribute()`[145c] propagates the bit from rules onto the nodes that use them (so `work()`[101c], `dorecipe()`[104e], and `waitup()`[115c] can consult the node directly without re-walking arcs), and the relevant code path checks the flag.

⟨`graph()` *propagate attributes* 145b⟩≡ (80)
```
// propagate attributes in rules to their node
attribute(root);
```
Uses attribute() 145c.

⟨*function* attribute 145c⟩≡ (190a)
```
static void
attribute(Node *n)
{
    Arc *a;

    for(a = n->arcs; a; a = a->next){
```

```
        ⟨attribute() propagate rule attribute to node cases 146d⟩
        // recurse
        if(a->n)
            attribute(a->n);
    }
    ⟨attribute() if virtual node 146e⟩
}
```

Uses attribute() 145c.

11.5.1 Virtual target: :V:

⟨Rule_attr cases 146a⟩+≡ (38a) ◁135a 146h▷

```
VIR    = 0x0010,
```

⟨rhead() when parsing rule attributes, switch rune cases 146b⟩+≡ (75d) ◁135b 147a▷

```
case 'V':
    *attr |= VIR;
    break;
```

⟨Node_flag cases 146c⟩+≡ (42d) ◁96c 147b▷

```
VIRTUAL    = 0x0001,
```

⟨attribute() propagate rule attribute to node cases 146d⟩≡ (145c) 147c▷

```
if(a->r->attr&VIR)
    n->flags |= VIRTUAL;
```

Uses VIR 146a and VIRTUAL 146c.

⟨attribute() if virtual node 146e⟩≡ (145c)

```
if(n->flags&VIRTUAL)
    n->time = 0;
```

Uses VIRTUAL 146c.

⟨dorecipe() when no recipe found, if virtual or norecipe node 146f⟩≡ (105b)

```
if((node->flags&VIRTUAL) || (node->flags&NORECIPE)){
    ⟨dorecipe() when no recipe found, if archive name 154e⟩
    else
        update(node, false);
    ⟨dorecipe() when no recipe found, if tflag 160a⟩
    //bugfix:
    return;
}
```

Uses NORECIPE 148d, VIRTUAL 146c, and update() 116b.

⟨update() if virtual node or inexistent file 146g⟩≡ (116b)

```
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 146c and outofdate() 103b.

11.5.2 Deleting a target when the recipe returns an error: :D:

⟨Rule_attr cases 146h⟩+≡ (38a) ◁146a 147g▷

```
DEL    = 0x0080,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 147a⟩+≡ (75d) ◁146b 147h▷
```
case 'D':
    *attr |= DEL;
    break;
```

⟨Node_flag *cases* 147b⟩+≡ (42d) ◁146c 147i▷
```
DELETE     = 0x0800,
```

⟨attribute() *propagate rule attribute to node cases* 147c⟩+≡ (145c) ◁146d 148e▷
```
if(a->r->attr&DEL)
    n->flags |= DELETE;
```
Uses DEL 146h and DELETE 147b.

⟨waitup() *other locals* 147d⟩+≡ (115c) ◁128c 151b▷
```
Node *n;
bool done;
```

⟨waitup() *when error in child process, delete if DELETE node* 147e⟩≡ (118c)
```
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 147b and delete() 147f.

⟨*function* delete 147f⟩≡ (188b)
```
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 Not printing the recipe (quiet mode): :Q:

⟨Rule_attr *cases* 147g⟩+≡ (38a) ◁146h 148b▷
```
QUIET  = 0x0008,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 147h⟩+≡ (75d) ◁147a 147j▷
```
case 'Q':
    *attr |= QUIET;
    break;
```

11.5.4 Running a shell script without -e: :E:

⟨Node_flag *cases* 147i⟩+≡ (42d) ◁147b 148d▷
```
NOMINUSE   = 0x1000,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 147j⟩+≡ (75d) ◁147h 148c▷
```
case 'E':
    *attr |= NOMINUSE;
    break;
```

⟨sched() *reset flags if NOMINUSE rule* 148a⟩≡ (111c)
```
if (j->r->attr&NOMINUSE)
    flags = nil;
```
Uses NOMINUSE 147i.

11.5.5 Disabling the no-recipe warning, :N:

⟨Rule_attr *cases* 148b⟩+≡ (38a) ◁147g 148f▷
```
NOREC  = 0x0040,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 148c⟩+≡ (75d) ◁147j 148g▷
```
case 'N':
    *attr |= NOREC;
    break;
```

⟨Node_flag *cases* 148d⟩+≡ (42d) ◁147i 158c▷
```
NORECIPE   = 0x0400,
```

⟨attribute() *propagate rule attribute to node cases* 148e⟩+≡ (145c) ◁147c
```
if(a->r->attr&NOREC)
    n->flags |= NORECIPE;
```
Uses NOREC 148b and NORECIPE 148d.

11.5.6 Forbidding metarules to match virtual targets: :n:

⟨Rule_attr *cases* 148f⟩+≡ (38a) ◁148b
```
NOVIRT = 0x0100,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 148g⟩+≡ (75d) ◁148c 149a▷
```
case 'n':
    *attr |= NOVIRT;
    break;
```

⟨applyrules() *skip this meta rule and continue if some conditions* 148h⟩+≡ (83e) ◁84d
```
if ((r->attr&NOVIRT) && lasta != &head && (lasta->r->attr&VIR))
    continue;
```
Uses NOVIRT 148f and VIR 146a.

11.5.7 Custom-dependency comparison program: :P:

⟨Rule *other fields* 148i⟩+≡ (36) ◁135c
```
char  *prog;  /* to use in out of date */
```

⟨parse() *other locals* 148j⟩+≡ (55) ◁140c
```
char *prog;
```

⟨addrule() *set more fields* 148k⟩+≡ (38b) ◁91c
```
r->prog = prog;
```

⟨Arc *other fields* 148l⟩+≡ (43b) ◁136c
```
char  *prog;
```

⟨newarc() *set other fields* 148m⟩+≡ (43e) ◁136d
```
a->prog = r->prog;
```

⟨rhead() *other locals* 148n⟩+≡ (61b) ◁75c
```
char *pp;
```

⟨rhead() *when parsing rule attributes, switch rune cases* 149a⟩+≡ (75d) ◁148g
```
case 'P':
    pp = utfrune(p, ':');
    if (pp == nil || *pp == 0)
        goto eos;
    *pp = 0;
    *prog = strdup(p);
    *pp = ':';
    p = pp;
    break;
```

⟨Sxxx *cases* 149b⟩+≡ (31a) ◁130b 152b▷
```
S_OUTOFDATE, /* n1\377n2 -> 2(outofdate) or 1(not outofdate) */
```

⟨update() *set outofdate prereqs if arc prog* 149c⟩≡ (116b)
```
for(a = node->arcs; a; a = a->next)
    if(a->prog)
        outofdate(node, a, true);
```
Uses outofdate() 103b.

⟨outofdate() *locals* 149d⟩≡ (103b)
```
char buf[3*NAMEBLOCK];
char *str = nil;
Symtab *sym;
int ret;
```
Uses NAMEBLOCK 84b.

⟨outofdate() *if arc-¿prog* 149e⟩≡ (103b)
```
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 149b, pcmp() 149f, and symlook() 32a.

⟨*function* pcmp 149f⟩≡ (190b)
```
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 140b, NAMEBLOCK 84b, bout 48a, and waitup() 115c.

Process

⟨*struct* Process 150a⟩≡ (188c)
```
struct Process {
    int pid;
    int status;

    // Extra
    // double_list<ref_own<Process> backward, forward
    Process *b, *f;
};
```

⟨*global* phead 150b⟩≡ (188c)
```
// double_list<ref_own<Process> (next = Process.f)
static Process *phead;
```

⟨*global* pfree 150c⟩≡ (188c)
```
// double_list<ref_own<Process> (next = Process.f)
static Process *pfree;
```

⟨*function* pnew 150d⟩≡ (188c)
```
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() 174a, pfree-20 150c, and phead-19 150b.

⟨*function* pdelete 150e⟩≡ (188c)
```
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-20 150c and phead-19 150b.

`waitup()` adjustments

⟨`waitup()` *if slot not found, not a job pid, update process list* 151a⟩≡ (115c)
```
if(slot < 0){
    ⟨waitup() if DEBUG(D_EXEC) and slot ¡ 0 171d⟩
     pnew(pid, buf[0]? 1:0);
     goto again;
}
```
Uses `pnew()` 150d.

⟨`waitup()` *other locals* 151b⟩+≡ (115c) ◁147d
```
Process *p;
```

⟨`waitup()` *if retstatus, check process list* 151c⟩≡ (115c)
```
/* 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()` 150e and `phead-19` 150b.

⟨`waitup()` *if retstatus, check if matching pid* 151d⟩≡ (115c)
```
if(retstatus && pid == *retstatus){
    *retstatus = buf[0]? 1:0;
    return -1;
}
```

`killchildren()` adjustments

⟨`killchildren()` *locals* 151e⟩≡ (119e)
```
Process *p;
```

⟨`killchildren()` *expunge not-job processes* 151f⟩≡ (119e)
```
for(p = phead; p; p = p->f)
    expunge(p->pid, msg);
```
Uses `phead-19` 150b.

⟨*function* `expunge` 151g⟩≡ (198)
```
void
expunge(int pid, char *msg)
{
    postnote(PNPROC, pid, msg);
}
```

11.6 Variable attributes

11.6.1 Private variables: `=U=`

⟨`rhead()` *when parsing variable attributes, switch rune cases* 151h⟩≡ (79c)
```
case 'U':
    *attr = 1;
    break;
```

⟨`parse()` *when parsing variable definitions, if variable with attr* 152a⟩≡ (78b)
```
if(attr)
    symlook(head->s, S_NOEXPORT, (void *)"");
```
Uses `S_NOEXPORT` 152b and `symlook()` 32a.

⟨`Sxxx` *cases* 152b⟩+≡ (31a) ◁149b 154d▷
```
S_NOEXPORT, /* var -> noexport */ // set of noexport variables
```

⟨`ecopy()` *return and do not copy if* `S_NOEXPORT` *symbol* 152c⟩≡ (122c)
```
if(symlook(s->name, S_NOEXPORT, nil))
    return;
```
Uses `S_NOEXPORT` 152b and `symlook()` 32a.

11.7 Advanced `mk` variables

11.7.1 `$target` versus `$alltargets`

⟨`dorecipe()` *other locals* 152d⟩+≡ (104e) ◁106a 152j▷
```
Word oldtargets;
Word *last_oldtargets = &oldtargets;
```

⟨`dorecipe()` *update list of outdated targets* 152e⟩≡ (106b)
```
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` 163a, `newword()` 34d, and `outofdate()` 103b.

⟨`specialvars` *other array elements* 152f⟩+≡ (33d) ◁136g 153a▷
```
"alltarget",
```

⟨`Job` *other fields* 152g⟩+≡ (44a) ◁136f 153b▷
```
Word  *at; /* all targets */
```

⟨`newjob()` *setting other fields* 152h⟩≡ (44c) 153c▷
```
j->at = alltargets;
```

⟨`buildenv()` *envupd some variables* 152i⟩+≡ (125b) ◁137b 153d▷
```
envupd("alltarget", wdup(j->at));
```
Uses `envupd()` 121e and `wdup()` 35c.

11.7.2 `$prereq` versus `$newprereq`

⟨`dorecipe()` *other locals* 152j⟩+≡ (104e) ◁152d
```
Word newprereqs;
```

⟨`dorecipe()` *when outofdate node, update list of newprereqs* 152k⟩≡ (107)
```
addw(&newprereqs, a->n->name);
```
Uses `addw()` 35b.

⟨specialvars *other array elements* 153a⟩+≡ (33d) ◁152f 153g▷
```
"newprereq",
```

⟨Job *other fields* 153b⟩+≡ (44a) ◁152g
```
Word  *np; /* new prerequisites */
```

⟨newjob() *setting other fields* 153c⟩+≡ (44c) ◁152h
```
j->np = newprereqs;
```

⟨buildenv() *envupd some variables* 153d⟩+≡ (125b) ◁152i 153i▷
```
envupd("newprereq", wdup(j->np));
```
Uses envupd() 121e and wdup() 35c.

11.7.3 $NREP

⟨mk() *initializations* 153e⟩+≡ (99) ◁109b 164h▷
```
nrep();  /* it can be updated dynamically */
```

⟨*function* nrep 153f⟩≡ (190a)
```
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) 170d⟩
}
```

11.7.4 $pid

⟨specialvars *other array elements* 153g⟩+≡ (33d) ◁153a 153j▷
```
"pid",
```

⟨buildenv() *locals* 153h⟩+≡ (125b) ◁137a 154b▷
```
char buf[256];
```

⟨buildenv() *envupd some variables* 153i⟩+≡ (125b) ◁153d 153k▷
```
snprint(buf, sizeof buf, "%d", getpid());
envupd("pid", newword(buf));
```
Uses envupd() 121e and newword() 34d.

11.7.5 $nproc

⟨specialvars *other array elements* 153j⟩+≡ (33d) ◁153g 154a▷
```
"nproc",
```

⟨buildenv() *envupd some variables* 153k⟩+≡ (125b) ◁153i 154c▷
```
snprint(buf, sizeof buf, "%d", slot);
envupd("nproc", newword(buf));
```
Uses envupd() 121e and newword() 34d.

11.8 Dealing with archives (libraries)

⟨specialvars *other array elements* 154a⟩+≡ (33d) ◁153j
```
"newmember",
```

⟨buildenv() *locals* 154b⟩+≡ (125b) ◁153h
```
Word *w, *v, **l;
char *cp, *qp;
```

⟨buildenv() *envupd some variables* 154c⟩+≡ (125b) ◁153k
```
// 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() 121e and wdup() 35c.

⟨Sxxx *cases* 154d⟩+≡ (31a) ◁152b 156b▷
```
S_AGG,  /* aggregate -> time */
```

⟨dorecipe() *when no recipe found, if archive name* 154e⟩≡ (146f)
```
if(strchr(node->name, '(') && node->time == 0)
    MADESET(node, MADE);
```
Uses MADE 42e and MADESET 101b.

⟨timeof() *if name archive member* 154f⟩≡ (88a)
```
if(utfrune(name, '('))
    return atimeof(force, name);  /* archive */
```
Uses atimeof() 154g.

⟨*function* atimeof 154g⟩≡ (185d)
```
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 154d, S_TIME 160e, atimes() 156a, split() 157a, and symlook() 32a.

⟨*function* atouch 155⟩≡ (185d)

```
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 160e, split() 157a, and symlook() 32a.

⟨*function* atimes 156a⟩≡ (185d)

```
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 181a, S_TIME 160e, and symlook() 32a.

⟨Sxxx *cases* 156b⟩+≡ (31a) ◁154d 160e▷

```
S_BITCH, /* bitched about aggregate not there */
```

⟨*function* type 156c⟩≡ (185d)

```
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 156b, bout 48a, and symlook() 32a.

⟨*function* split 157a⟩≡ (185d)

```
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() 156c.

⟨outofdate() *if arc node is an archive member* 157b⟩≡ (103b)

```
if(strchr(arc->n->name, '(') && arc->n->time == 0)
   /* missing archive member */
   return true;
```

11.9 Optimizations

11.9.1 Missing-intermediates optimization: `mk -I`

The motivation for "pretending" is historical. 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 to pretend a missing intermediate file is up-to-date, provided the next level up in the graph is also up-to-date. 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()`[101c] 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` 158a⟩≡ (183)
```
bool iflag = false;
```
Uses `iflag` 158a.

⟨`main()` *-xxx switch cases* 158b⟩+≡ (48d) ◁133b 159h▷
```
case 'i':
    iflag = true;
    break;
```
Uses `iflag` 158a.

⟨`Node_flag` *cases* 158c⟩+≡ (42d) ◁148d
```
CANPRETEND = 0x0008,
PRETENDING = 0x0010,
```

⟨`clrmade() n->flags` *pretend adjustments* 158d⟩≡ (101a)
```
n->flags &= ~(CANPRETEND|PRETENDING);
if(strchr(n->name, '(') == nil || n->time)
    n->flags |= CANPRETEND;
```
Uses `CANPRETEND` 158c and `PRETENDING` 158c.

⟨`work()` *possibly pretending node* 158e⟩≡ (103a)
```
/*
 *   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` 158c, `MADE` 42e, `MADESET` 101b, `PRETENDING` 158c, `bout` 48a, `explain` 131b, `iflag` 158a, `outofdate()` 103b, `unpretend()` 159b, and `work()` 101c.

⟨work() *possibly unpretending node* 159a⟩≡ (101c)
```
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 42e, PRETENDING 158c, explain 131b, outofdate() 103b, and unpretend() 159b.

⟨*function* unpretend 159b⟩≡ (190b)
```
static void
unpretend(Node *n)
{
    MADESET(n, NOTMADE);
    n->flags &= ~(CANPRETEND|PRETENDING);
    n->time = 0;
}
```
Uses CANPRETEND 158c, MADESET 101b, NOTMADE 42e, and PRETENDING 158c.

⟨work() *locals* 159c⟩+≡ (101c) ◁102e
```
Arc *ra = nil;
```

⟨work() *update* ra *when outofdate* node *with arc* a 159d⟩≡ (103a)
```
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* 159e⟩≡ (103a)
```
if(ra == nil)
    ra = a;
```

⟨update() *unpretend node* 159f⟩≡ (116b)
```
node->flags &= ~(CANPRETEND|PRETENDING);
```
Uses CANPRETEND 158c and PRETENDING 158c.

11.9.2 Touching-mode optimization: mk -t

⟨*global* tflag 159g⟩≡ (183)
```
bool tflag = false;
```
Uses tflag 159g.

⟨main() *-xxx switch cases* 159h⟩+≡ (48d) ◁158b 163b▷
```
case 't':
    tflag = true;
    break;
```
Uses tflag 159g.

⟨dorecipe() *when no recipe found, if tflag* 160a⟩≡ (146f)
```
if(tflag){
    if(!(node->flags&VIRTUAL))
        touch(node->name);
    else if(explain)
        Bprint(&bout, "no touch of virtual '%s'\n", node->name);
}
```
Uses VIRTUAL 146c, bout 48a, explain 131b, tflag 159g, and touch() 160c.

⟨sched() *if touch mode* 160b⟩≡ (132c)
```
if(tflag){
    if(!(n->flags&VIRTUAL))
        touch(n->name);
    else if(explain)
        Bprint(&bout, "no touch of virtual '%s'\n", n->name);
}
```
Uses VIRTUAL 146c, bout 48a, explain 131b, tflag 159g, and touch() 160c.

⟨*function* touch 160c⟩≡ (188b)
```
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() 155, bout 48a, and nflag 131f.

⟨*function* chgtime 160d⟩≡ (198)
```
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));
}
```

11.9.3 Time cache

⟨Sxxx *cases* 160e⟩+≡ (31a) ◁156b 162c▷
```
S_TIME,  /* file -> time */
```

⟨timeof() *locals* 160f⟩≡ (88a) 161b▷
```
ulong t;
```

⟨`timeof()` *if not force, use time cache* 161a⟩≡ (88a)
```
⟨timeof() check time cache 161c⟩
t = mkmtime(name, false);
⟨timeof() update time cache 161d⟩
return t;
```

⟨`timeof()` *locals* 161b⟩+≡ (88a) ◁160f
```
Symtab *sym;
```

⟨`timeof()` *check time cache* 161c⟩≡ (161a)
```
sym = symlook(name, S_TIME, nil);
if (sym)
    return sym->u.value;  /* uggh */
```
Uses `S_TIME` 160e and `symlook()` 32a.

⟨`timeof()` *update time cache* 161d⟩≡ (161a)
```
if(t == 0)
    return 0;
symlook(name, S_TIME, (void*)t);  /* install time in cache */
```
Uses `S_TIME` 160e and `symlook()` 32a.

11.9.4 Bulk time optimisation

There is a layered cache here. The first layer (Section 11.9.3) is per-file: the first call to `timeof()`[88a] for a given path remembers the result. The second layer is per-directory: rather than `stat`-ing each file individually, `mk` reads the entire directory once and prepopulates `S_TIME`[160e] for every entry. On projects with many files in one directory this turns N system calls into one. The trade-off is the first-touch cost—if `mk` only ever needs the time of one file in a 10 000-file directory, the bulk read is wasted—but in practice `mk` visits most files in a directory anyway during graph construction.

The cautionary tale in the `%subtle:` note above is real and worth understanding. On a case-insensitive filesystem like VFAT, `dirstat("helloc.c")` succeeds even when the on-disk name is `HELLOC.C`. But the bulk read populates `S_TIME` under the on-disk name `HELLOC.C`, so a later cache lookup for `helloc.c` misses, returns 0 ("does not exist"), and `mk` concludes the source is missing. The lesson: any time you cache results keyed by a string, you must canonicalize the key the same way on insert and on lookup, or the cache will silently shadow real data.

⟨`mkmtime` *locals* 161e⟩≡ (88b) 162a▷
```
//char *s, *ss;
//char carry;
//Symtab *sym;
```

⟨`mkmtime()` *bulk dir optimisation* 161f⟩≡ (88b)
```
⟨mkmtime() cleanup name 162b⟩
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* 162a⟩+≡ (88b) ◁161e

```
char buf[4096];
```

⟨mkmtime() *cleanup name* 162b⟩≡ (161f)

```
strecpy(buf, buf + sizeof buf - 1, name);
cleanname(buf);
name = buf;
```

⟨Sxxx *cases* 162c⟩+≡ (31a) ◁160e

```
S_BULKED, /* we have bulked this dir */
```

⟨*function* bulkmtime 162d⟩≡ (198)

```
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 162c, dirtime() 162e, and symlook() 32a.

⟨*function* dirtime 162e⟩≡ (198)

```
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 160e and symlook() 32a.

11.10 Recompiling everything: mk -a

⟨*global* aflag 163a⟩≡ (183)

```
bool aflag = false;
```

Uses aflag 163a.

⟨main() *-xxx switch cases* 163b⟩+≡ (48d) ◁159h 164f▷

```
case 'a':
    aflag = true;
    iflag = true;
    break;
```

Uses aflag 163a and iflag 158a.

⟨work() *adjust weoutofdate if aflag* 163c⟩≡ (103a)

```
if(aflag)
    weoutofdate = true;
```

Uses aflag 163a.

11.11 Recursive mk

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). $MKFLAGS and $MKARGS solve this: main()[47b] 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 -k, -n, or CFLAGS=-O2 settings the user originally typed.

⟨main() *locals* 163d⟩+≡ (47b) ◁132d 168d▷

```
Bufblock *buf = newbuf();
```

Uses newbuf() 175d.

⟨main() *add* argv[0] *in* buf 163e⟩≡ (51b 48d)

```
bufcpy(buf, argv[0], strlen(argv[0]));
insert(buf, ' ');
```

Uses bufcpy() 176c and insert() 176a.

⟨main() *add* argv[i] *in* buf 164a⟩≡ (49c)
```
bufcpy(buf, argv[i], strlen(argv[i]));
insert(buf, ' ');
```
Uses bufcpy() 176c and insert() 176a.

⟨main() *set variables for recursive mk* 164b⟩≡ (48c)
```
⟨main() set MKFLAGS variable 164c⟩
⟨main() set MKARGS variable 164d⟩
```

⟨main() *set MKFLAGS variable* 164c⟩≡ (164b)
```
if (buf->current != buf->start) {
    buf->current--;
    insert(buf, '\0');
}
symlook("MKFLAGS", S_VAR, (void*) stow(buf->start));
```
Uses S_VAR 31a, insert() 176a, stow() 64b, and symlook() 32a.

⟨main() *set MKARGS variable* 164d⟩≡ (164b)
```
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 31a, bufcpy() 176c, freebuf() 175e, insert() 176a, stow() 64b, and symlook() 32a.

11.12 mk -k

11.12.1 kflag and runerrs

⟨*global* kflag 164e⟩≡ (183)
```
bool kflag = false;
```
Uses kflag 164e.

⟨main() *-xxx switch cases* 164f⟩+≡ (48d) ◁163b 168e▷
```
case 'k':
    kflag = true;
    break;
```
Uses kflag 164e.

⟨*global* runerrs 164g⟩≡ (190b)
```
int runerrs;
```

⟨mk() *initializations* 164h⟩+≡ (99) ◁153e
```
runerrs = 0;
```
Uses runerrs 164g.

11.12.2 Adjusting `mk()`, `work()`, and `waitup()`

⟨`work()` *when inexistent target without prerequisites, if kflag* 165a⟩≡ (102c)
```
if(kflag){
    node->flags |= BEINGMADE;
    runerrs++;
}
```
Uses `BEINGMADE` 42e, `kflag` 164e, and `runerrs` 164g.

⟨`waitup()` *when error in child process, if kflag* 165b⟩≡ (118c)
```
if(kflag){
    runerrs++;
    fake = true;
}
```
Uses `kflag` 164e and `runerrs` 164g.

⟨`update()` *if fake* 165c⟩≡ (116b)
```
if(fake)
    MADESET(node, BEINGMADE);
```
Uses `BEINGMADE` 42e and `MADESET` 101b.

⟨`mk()` *if no child to waitup and root not MADE, possibly break* 165d⟩≡ (99)
```
if(res > 0){
    if(root->flags&(NOTMADE|BEINGMADE)){
        assert(/*must be run errors*/ runerrs);
        break; /* nothing more waiting */
    }
}
```
Uses `BEINGMADE` 42e, `NOTMADE` 42e, and `runerrs` 164g.

⟨`WaitupParam` *other cases* 165e⟩+≡ (117b) ◁140b
```
EMPTY_CHILDREN_IS_ERROR2 = -2,
```

⟨`mk()` *before returning, more* `waitup()` *if there was an error* 165f⟩≡ (99)
```
while(jobs)
    waitup(EMPTY_CHILDREN_IS_ERROR2, (int *)nil);
assert(/*target didnt get done*/ runerrs || (root->flags&MADE));
```
Uses `EMPTY_CHILDREN_IS_ERROR2` 165e, `MADE` 42e, `jobs` 45a, `runerrs` 164g, and `waitup()` 115c.

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.

At its core, `mk` solves a simple problem: given a graph of dependencies and a set of recipes, figure out what is out of date and rebuild it in the right order. Yet this simple idea touches many interesting topics: parsing a small declarative language, building and traversing a dependency graph, detecting cycles, comparing file modification times, running recipes in parallel while respecting dependencies, and pattern-matching with `%` rules to avoid repetition.

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.
- *Stem-based pattern matching*: the `%` metarule extracts a stem and substitutes it into prerequisite names. This is a simple form of unification—the same idea behind URL routing in web frameworks (`/users/:id/posts`) and generic rewrite rules in any template system.
- *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`, the Shell book [Pad18] is the next logical step after this book. Many features of `mk` are inspired by features from the shell: the ability to call programs easily, to use pipes and redirections, to loop over files, and to use variables are all inherited from `rc`. In fact, `mk` invokes `rc -e` to execute each recipe, so understanding how `rc` processes `-e` and `-c` flags is directly relevant.

- The Shell book [Pad18] explains the internals of `rc`, the shell that `mk` uses to execute recipes. Understanding how `rc` handles `fork()`, `exec()`, pipes, and redirections will clarify what happens when `mk` runs a recipe.
- 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 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 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. Hashes also make it possible to detect when a command's flags change, something timestamp-based systems like `mk` cannot track.
- *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.
- *Incremental testing:* Some build systems (e.g., Bazel, sbt) can determine which tests are affected by a code change and re-run only those, rather than the entire test suite.

Despite these additions, the fundamental algorithm remains the same: build a dependency graph, find what is out of date, and rebuild in topological order. `mk` implements this core cleanly in about 4000 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 -d`) 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` 168a⟩≡ (181d)
```
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` 168b⟩≡ (183)
```
// bitset<enum<dxxx>>
int debug;
```

⟨*function* `DEBUG` 168c⟩≡ (181d)
```
#define DEBUG(x) (debug&(x))
```

⟨`main()` *locals* 168d⟩+≡ (47b) ◁163d
```
char *s;
```

⟨`main()` *-xxx switch cases* 168e⟩+≡ (48d) ◁164f
```
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` 168a, `D_GRAPH` 168a, `D_PARSE` 168a, and `debug` 168b.

A.1 Dumping the rules: `mk -dp`

⟨`main()` *if* `DEBUG(D_PARSE)` 169a⟩≡ (51c)
```
if(DEBUG(D_PARSE)){
    dumpw("default targets", target1);
    dumpr("rules", rules);
    dumpr("metarules", metarules);
    dumpv("variables");
}
```
Uses `DEBUG` 168c, `D_PARSE` 168a, `dumpr()` 169c, `dumpv()` 169d, `dumpw()` 169b, `metarules` 37e, `rules` 37b, and `target1` 51f.

⟨*dumper dumpw* 169b⟩≡ (192a)
```
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` 48a.

⟨*dumper dumpr* 169c⟩≡ (192a)
```
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` 48a and `wtos()` 35d.

⟨*dumper dumpv* 169d⟩≡ (192a)
```
void
dumpv(char *s)
{
    Bprint(&bout, "%s:\n", s);
    symtraverse(S_VAR, &print1);
}
```
Uses `S_VAR` 31a, `bout` 48a, `print1()` 169e, and `symtraverse()` 33a.

⟨*function* `print1` 169e⟩≡ (192a)
```
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` 48a.

A.2 Dumping the graph: mk -dg

⟨mk() *if* DEBUG(D_GRAPH) 170a⟩≡ (99)
```
if(DEBUG(D_GRAPH)){
    dumpn("new target\n", root);
    Bflush(&bout);
}
```
Uses DEBUG 168c, D_GRAPH 168a, bout 48a, and dumpn() 170b.

⟨*dumper dumpn* 170b⟩≡ (192a)
```
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 48a and dumpa() 170c.

⟨*dumper dumpa* 170c⟩≡ (192a)
```
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 48a and dumpn() 170b.

⟨nrep() *if* DEBUG(D_GRAPH) 170d⟩≡ (153f)
```
if(DEBUG(D_GRAPH))
    Bprint(&bout, "nreps = %d\n", nreps);
```

A.3 Tracing jobs: mk -de

⟨sched() *if* DEBUG(D_EXEC) 170e⟩≡ (111c)
```
if(DEBUG(D_EXEC))
    fprint(STDOUT, "firing up job for target %s\n", wtos(j->t, ' '));
```
Uses DEBUG 168c, D_EXEC 168a, and wtos() 35d.

⟨sched() *if* DEBUG(D_EXEC) *print recipe* 171a⟩≡ (111c)
```
if(DEBUG(D_EXEC))
    fprint(STDOUT, "recipe='%s'\n", j->r->recipe);
Bflush(&bout);
```
Uses DEBUG 168c, D_EXEC 168a, and bout 48a.

⟨sched() *if* DEBUG(D_EXEC) *print pid* 171b⟩≡ (111c)
```
if(DEBUG(D_EXEC))
    fprint(STDOUT, "pid for target %s = %d\n", wtos(j->t, ' '), events[slot].pid);
```
Uses DEBUG 168c, D_EXEC 168a, events-15 110a, and wtos() 35d.

⟨waitup() *if* DEBUG(D_EXEC) *print pid* 171c⟩≡ (115c)
```
if(DEBUG(D_EXEC))
    fprint(STDOUT, "waitup got pid=%d, status='%s'\n", pid, buf);
```
Uses DEBUG 168c and D_EXEC 168a.

⟨waitup() *if* DEBUG(D_EXEC) *and slot ¡ 0* 171d⟩≡ (151a)
```
  if(DEBUG(D_EXEC))
      fprint(STDERR, "mk: wait returned unexpected process %d\n", pid);
```
Uses DEBUG 168c and D_EXEC 168a.

⟨pidslot() *if* DEBUG(D_EXEC) 171e⟩≡ (111b)
```
if(DEBUG(D_EXEC))
    fprint(STDERR, "mk: wait returned unexpected process %d\n", pid);
```
Uses DEBUG 168c and D_EXEC 168a.

⟨nproc() *if* DEBUG(D_EXEC) 171f⟩≡ (109c)
```
if(DEBUG(D_EXEC))
    fprint(STDERR, "nprocs = %d\n", nproclimit);
```
Uses DEBUG 168c, D_EXEC 168a, and nproclimit-18 109a.

⟨*dumper dumpj* 171g⟩≡ (192a)
```
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 48a and wtos() 35d.

A.4 Tracing function calls: mk -dt

⟨applyrules *debug* 171h⟩≡ (81)
```
if(DEBUG(D_TRACE))
    print("applyrules(%lux='%s')\n", target, target);
```
Uses DEBUG 168c and D_TRACE 168a.

⟨newnode() *debug* 171i⟩≡ (42b)
```
if(DEBUG(D_TRACE))
    print("newnode(%s), time = %d\n", name, node->time);
```
Uses DEBUG 168c and D_TRACE 168a.

⟨`work()` *debug* 172a⟩≡ (101c)

```
if(DEBUG(D_TRACE))
    print("work(%s) flags=0x%x time=%lud\n", node->name, node->flags, node->time);
```

Uses `DEBUG` 168c and `D_TRACE` 168a.

⟨`update()` *debug* 172b⟩≡ (116b)

```
if(DEBUG(D_TRACE))
    print("update(): node %s time=%lud flags=0x%x\n", node->name, node->time, node->flags);
```

Uses `DEBUG` 168c and `D_TRACE` 168a.

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, which is useful for profiling the build system rather than the programs being built.

⟨*global* `buf` 173a⟩≡ (192b)
```
short buf[10000];
```

⟨`main()` *setup profiling* 173b⟩+≡ (48c) ◁133e
```
#ifdef PROF
    {
        extern int etext();
        monitor(main, etext, buf, sizeof buf, 300);
    }
#endif
```

⟨*function* `symstat` 173c⟩≡ (184c)
```
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` 31c, `bout` 48a, and `hash-3` 31b.

Appendix C

Utilities

This appendix collects the utility code used throughout `mk`: memory allocation wrappers, string buffers (`Bufblock`), word-list manipulation, and the `mk -f` flag handling.

C.1 Memory management

⟨*function* `Malloc` 174a⟩≡ (184a)

```
void*
Malloc(int n)
{
    void *s;

    s = malloc(n);
    if(!s) {
        fprint(STDERR, "mk: cannot alloc %d bytes\n", n);
        Exit();
    }
    return s;
}
```

⟨*function* `Realloc` 174b⟩≡ (184a)

```
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` 174c⟩≡ (181d)

```
struct Bufblock
{
    char   *start;
    char   *end;
```

```
    // between start and end
    char   *current;

    // Extra
    ⟨Bufblock extra fields 175a⟩
};
```

⟨Bufblock *extra fields* 175a⟩≡ (174c)
```
struct Bufblock *next;
```
Uses Bufblock 174c.

⟨*global* freelist 175b⟩≡ (184b)
```
static Bufblock *freelist;
```

⟨*constant* QUANTA 175c⟩≡ (184b)
```
#define QUANTA 4096
```

⟨*constructor* newbuf 175d⟩≡ (184b)
```
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() 174a, QUANTA-13 175c, and freelist-12 175b.

⟨*destructor* freebuf 175e⟩≡ (184b)
```
void
freebuf(Bufblock *p)
{
    p->next = freelist;
    freelist = p;
}
```
Uses freelist-12 175b.

⟨*macro* isempty 175f⟩≡ (179)
```
#define isempty(buf) (buf->current == buf->start)
```

⟨*macro* resetbuf 175g⟩≡ (179)
```
#define resetbuf(buf) do { buf->current = buf->start; } while(0)
```

⟨*macro* bufcontent 175h⟩≡ (179)
```
#define bufcontent(buf) buf->start
```

⟨*function* `insert` 176a⟩≡ (184b)

```
void
insert(Bufblock *buf, int c)
{

    if (buf->current >= buf->end)
        growbuf(buf);
    *buf->current++ = c;
}
```

Uses `growbuf()` 176d.

⟨*function* `rinsert` 176b⟩≡ (184b)

```
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()` 176d.

⟨*function* `bufcpy` 176c⟩≡ (184b)

```
void
bufcpy(Bufblock *buf, char *cp, int n)
{

    while (n--)
        insert(buf, *cp++);
}
```

Uses `insert()` 176a.

⟨*function* `growbuf` 176d⟩≡ (184b)

```
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-13 175c, Realloc() 174b, and freelist-12 175b.

C.3 File management

⟨*function* maketmp 177⟩≡ (198)

```
char*
maketmp(void)
{
    static char temp[] = "/tmp/mkargXXXXXX";

    mktemp(temp);
    return temp;
}
```

Appendix D

Examples of `mkfiles` TODO

D.1 The `mkfile` of `mk`

D.2 The `mkfiles` of Plan 9

D.2.1 `/$objtype/mkfile` for the ARM

D.2.2 `/sys/src/mkfile.proto`

D.2.3 `/sys/src/cmd/mkone`

D.2.4 `/sys/src/cmd/mklib`

Appendix E

Extra Code

E.1 mk/

E.1.1 mk/fns.h

⟨mk/fns.h 179⟩≡

```
// 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 175f⟩
⟨macro resetbuf 175g⟩
⟨macro bufcontent 175h⟩

// 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 181a⟩≡ (181d)
```
#define BIGBLOCK 20000
```

⟨*function* RERR 181b⟩≡ (181d)
```
//#define RERR(r)  (fprint(STDERR, "mk: %s:%d: rule error; ", (r)->file, (r)->line))
```

⟨*function* SEP 181c⟩≡ (181d)
```
//#define SEP(c)  (((c)==' ')||((c)=='\t')||((c)=='\n'))
```

⟨mk/mk.h 181d⟩≡
```
#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 174c⟩

⟨struct Word 34b⟩
```

```
// used by main and parse.c
extern Word *target1;
```

⟨*struct* Envy 120a⟩

```
extern ShellEnvVar *shellenv;
```

⟨*struct* Rule 36⟩

```
extern Rule *rules, *metarules, *patrule;
```

⟨*enum* Rule_attr 38a⟩

⟨*macro* empty_recipe 83b⟩
⟨*macro* empty_prereqs 83c⟩
⟨*macro* empty_words 62a⟩

⟨*constant* NREGEXP 136b⟩

⟨*struct* Arc 43b⟩

⟨*struct* Node 42a⟩

⟨*enum* Node_flag 42d⟩
⟨*function* MADESET 101b⟩

⟨*struct* Job 44a⟩

```
extern Job *jobs;
```

⟨*struct* Symtab 30⟩

⟨*enum* Namespace 31a⟩

⟨*type* WaitupParam 117b⟩
⟨*type* WaitupResult 117c⟩

```
extern int debug;
extern bool nflag, tflag, iflag, kflag, aflag;
extern int mkinline;
extern char *infile;
extern bool explain;
extern int runerrs;
```

⟨*function* SYNERR 54d⟩
⟨*function* RERR 181b⟩
⟨*constant* NAMEBLOCK 84b⟩
⟨*constant* BIGBLOCK 181a⟩

⟨*function* SEP 181c⟩
⟨*function* WORDCHR 68e⟩

⟨*enum* Dxxx 168a⟩
⟨*function* DEBUG 168c⟩

⟨*function* PERCENT 37g⟩

```
//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())
typedef struct Shell {
    char* shell;
    char* shellname;
//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);
} Shell;
//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 43b, Bufblock 174c, Job 44a, Node 42a, Rule 36, Shell 181d, ShellEnvVar 120a, Symtab 30, and Word 34b.

E.1.3 mk/globals.c

⟨mk/globals.c 183⟩≡

```
#include "mk.h"

// used by DEBUG which is used by many files
⟨global debug 168b⟩

⟨global infile 54a⟩
⟨global mkinline 54c⟩

⟨global rules 37b⟩
⟨global metarules 37e⟩
⟨global patrule 135d⟩

// was in main.c, but used also by parse.c
⟨global target1 51f⟩

⟨global nflag 131f⟩
⟨global tflag 159g⟩
```

⟨*global* `iflag` 158a⟩
⟨*global* `kflag` 164e⟩
⟨*global* `aflag` 163a⟩

⟨*global* `explain` 131b⟩

⟨*global* `jobs` 45a⟩

⟨*global* `bout` 48a⟩

E.1.4 mk/utils.c

⟨`mk/utils.c` 184a⟩≡
```
#include "mk.h"
```

⟨*function* `Malloc` 174a⟩

⟨*function* `Realloc` 174b⟩

```
// maketmp() is back in Plan9.c
```

E.1.5 mk/bufblock.c

⟨`mk/bufblock.c` 184b⟩≡
```
#include "mk.h"
```

⟨*global* `freelist` 175b⟩
⟨*constant* `QUANTA` 175c⟩

⟨*constructor* `newbuf` 175d⟩

⟨*destructor* `freebuf` 175e⟩

⟨*function* `growbuf` 176d⟩

⟨*function* `bufcpy` 176c⟩

⟨*function* `insert` 176a⟩

⟨*function* `rinsert` 176b⟩

E.1.6 mk/symtab.c

⟨`mk/symtab.c` 184c⟩≡
```
#include "mk.h"
```

⟨*constant* `NHASH` 31c⟩
⟨*constant* `HASHMUL` 32d⟩
⟨*global* `hash` 31b⟩

⟨*function* `symlook` 32a⟩

⟨*function* `symtraverse` 33a⟩

⟨*function* `symstat` 173c⟩

E.1.7 mk/rc.c

⟨mk/rc.c 185a⟩≡
```
#include "mk.h"

⟨global termchars 79d⟩
⟨global shflags 113c⟩

/*
 * This file contains functions that depend on rc's syntax.  Most
 * of the routines extract strings observing rc's escape conventions
 */

⟨function squote 63b⟩

⟨function charin 62b⟩

⟨function expandquote 66c⟩

⟨function escapetoken 60b⟩

⟨function copysingle 131a⟩

⟨function copyq 130d⟩
```

E.1.8 mk/word.c

⟨mk/word.c 185b⟩≡
```
#include "mk.h"

⟨constructor newword 34d⟩


⟨function wtos 35d⟩

⟨function wdup 35c⟩

⟨destructor freewords 35a⟩

// was in recipe.c before
⟨function addw 35b⟩
```

E.1.9 mk/var.c

⟨mk/var.c 185c⟩≡
```
#include "mk.h"

⟨function setvar 33b⟩

⟨function shname 124a⟩
```

E.1.10 mk/archive.c

⟨mk/archive.c 185d⟩≡
```
#include "mk.h"
#include <ar.h>
```

```
static void atimes(char *);
static char *split(char*, char**);
```

⟨*function* `atimeof` 154g⟩

⟨*function* `atouch` 155⟩

⟨*function* `atimes` 156a⟩

⟨*function* `type` 156c⟩

⟨*function* `split` 157a⟩

E.1.11 mk/match.c

⟨`mk/match.c` 186a⟩≡
```
#include "mk.h"
```

⟨*function* `match` 85a⟩

⟨*function* `subst` 86⟩

E.1.12 mk/env.c

⟨`mk/env.c` 186b⟩≡
```
#include "mk.h"
```

⟨*constant* `ENVQUANTA` 121d⟩

⟨*global* `shellenv` 120b⟩
⟨*global* `nextv` 120c⟩

⟨*global* `specialvars` 33d⟩

```
// encodenulls() is back in Plan9.c
// readenv() is back in Plan9.c
extern void readenv(void);
// exportenv() is back in plan9.c
```

⟨*function* `inithash` 34a⟩

⟨*function* `envinsert` 121a⟩

⟨*function* `envupd` 121e⟩

⟨*function* `ecopy` 122c⟩

⟨*function* `initenv` 122b⟩

⟨*function* `buildenv` 125b⟩

E.1.13 mk/parse.c

⟨`mk/parse.c` 186c⟩≡
```
#include        "mk.h"
```

```
void    ipop(void);
void    ipush(void);
static int   rhead(char *, Word **, Word **, int *, char **);
static char* rbody(Biobuf*);
```

⟨*function* `parse` 55⟩

⟨*function* `addrules` 41c⟩

⟨*function* `rhead` 61b⟩

⟨*function* `rbody` 73c⟩

⟨*struct* `input` 76g⟩
⟨*global* `inputs` 77a⟩

⟨*function* `ipush` 77c⟩

⟨*function* `ipop` 77d⟩

E.1.14 mk/shprint.c

⟨`mk/shprint.c` 187a⟩≡
```
#include "mk.h"

static char *vexpand(char*, ShellEnvVar*, Bufblock*);
static char *shquote(char*, Rune, Bufblock*);
static char *shbquote(char*, Bufblock*);
```

⟨*function* `shprint` 129a⟩

⟨*function* `mygetenv` 130a⟩

⟨*function* `vexpand` 129b⟩

⟨*function* `front` 128e⟩

E.1.15 mk/job.c

⟨`mk/job.c` 187b⟩≡
```
#include "mk.h"
```

E.1.16 mk/arc.c

⟨`mk/arc.c` 187c⟩≡
```
#include "mk.h"
```

E.1.17 mk/rule.c

⟨`mk/rule.c` 187d⟩≡
```
#include "mk.h"
```

⟨*global* `lr` 37d⟩

⟨*global* `lmr` 37f⟩

⟨*global* `nrules` 91b⟩

```
static int rcmp(Rule *r, char *target, Word *tail);
```

⟨*function* `addrule` 38b⟩

⟨*function* `rcmp` 40d⟩

⟨*function* `rulecnt` 91f⟩

⟨*function* `regerror` 136a⟩

E.1.18 mk/lex.c

⟨`mk/lex.c` 188a⟩≡

```
#include "mk.h"

static int bquote(Biobuf*, Bufblock*);
```

⟨*function* `assline` 57b⟩

⟨*function* `bquote` 138a⟩

⟨*function* `nextrune` 58a⟩

E.1.19 mk/file.c

⟨`mk/file.c` 188b⟩≡

```
#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` 88a⟩

⟨*function* `touch` 160c⟩

⟨*function* `delete` 147f⟩

⟨*function* `timeinit` 132g⟩

E.1.20 mk/run.c

⟨`mk/run.c` 188c⟩≡

```
#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` 109d⟩

⟨*global* `events` 110a⟩
⟨*global* `nevents` 110b⟩
⟨*global* `nrunning` 108b⟩
⟨*global* `nproclimit` 109a⟩

⟨*struct* `Process` 150a⟩
⟨*global* `phead` 150b⟩
⟨*global* `pfree` 150c⟩

⟨*function* `run` 108a⟩

```
// shell is back in Plan9.c
// shellname is back in Plan9.c
```

⟨*function* `sched` 111c⟩

```
// execsh() is back in Plan9.c
// xwaitfor() is back in Plan9.c
extern int xwaitfor(char *msg);
```

⟨*function* `waitup` 115c⟩

⟨*function* `nproc` 109c⟩

⟨*function* `nextslot` 111a⟩

⟨*function* `pidslot` 111b⟩

⟨*function* `pnew` 150d⟩

⟨*function* `pdelete` 150e⟩

```
// 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` 119e⟩

⟨*global* `tslot` 133c⟩
⟨*global* `tick` 133d⟩

⟨*function* `usage` 134a⟩

⟨*function* `prusage` 134c⟩

```
// pipecmd() is back in Plan9.c
```
Uses Process 150a and RunEvent 109d.

E.1.21 mk/graph.c

⟨mk/graph.c 190a⟩≡
```
#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 91h⟩

⟨*function* graph 80⟩

⟨*function* applyrules 81⟩

⟨*function* nrep 153f⟩

⟨*function* togo 94f⟩

⟨*function* vacuous 96d⟩

⟨*constructor* newnode 42b⟩

```
// rcopy() is back in Plan9.c
extern void rcopy(char **to, Resub *match, int n);
```

⟨*constructor* newarc 43e⟩

⟨*function* trace 93a⟩

⟨*function* cyclechk 89c⟩

⟨*function* ambiguous 92c⟩

⟨*function* attribute 145c⟩

E.1.22 mk/mk.c

⟨mk/mk.c 190b⟩≡
```
#include "mk.h"

void clrmade(Node*);
void work(Node*, bool*, Node*, Arc*);
```

⟨*global* runerrs 164g⟩

⟨*function* mk 99⟩

⟨*function* `clrmade` 101a⟩

⟨*function* `unpretend` 159b⟩

⟨*function* `work` 101c⟩

⟨*function* `update` 116b⟩

⟨*function* `pcmp` 149f⟩

⟨*function* `outofdate` 103b⟩

E.1.23 mk/recipe.c

⟨`mk/recipe.c` 191a⟩≡

```
#include "mk.h"
```

⟨*constructor* `newjob` 44c⟩

⟨*function* `dorecipe` 104e⟩

E.1.24 mk/varsub.c

⟨`mk/varsub.c` 191b⟩≡

```
#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` 67d⟩

⟨*function* `varname` 68d⟩

⟨*function* `varmatch` 69b⟩

⟨*function* `expandvar` 142c⟩

⟨*function* `extractpat` 144⟩

⟨*function* `subsub` 143⟩

⟨*function* `submatch` 145a⟩

⟨*function* `nextword` 65⟩

⟨*function* `stow` 64b⟩

E.1.25 mk/dumpers.c

⟨mk/dumpers.c 192a⟩≡

```
#include "mk.h"

void dumpa(char*, Arc*);

⟨dumper dumpn 170b⟩

⟨dumper dumpa 170c⟩

⟨dumper dumpj 171g⟩

⟨function print1 169e⟩

⟨dumper dumpv 169d⟩

⟨dumper dumpr 169c⟩

⟨dumper dumpw 169b⟩
```

E.1.26 mk/main.c

⟨mk/main.c 192b⟩≡

```
#include "mk.h"

⟨constant MKFILE 51d⟩

⟨global version 47a⟩

// see also globals.c

⟨global uflag 133a⟩

void badusage(void);

#ifdef PROF
⟨global buf 173a⟩
#endif

⟨function main 47b⟩

⟨function badusage 49a⟩
```

E.1.27 mk/Posix.c

⟨mk/Posix.c 192c⟩≡

```
// 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 168c, D_EXEC 168a, IWS 192c, Realloc() 174b, S_INTERNAL 33c, ShellEnvVar 120a, __anon_struct_1 192c, growbuf() 176d, killchildren() 119e, newword() 34d, setvar() 33b, shflags 113c, shname() 124a, sigmsgs-5 192c, stab 192c, symlook() 32a, and wtos() 35d.

E.1.28 mk/Plan9.c

⟨mk/Plan9.c 198⟩≡

```
#include "mk.h"

// could be in utils.c
⟨function maketmp 177⟩

// could be in env.c
⟨function encodenulls 124e⟩
```

⟨*function* `readenv` 123a⟩
⟨*function* `exportenv` 126a⟩

```
// could be in file.c
```
⟨*function* `chgtime` 160d⟩
⟨*function* `dirtime` 162e⟩
⟨*function* `bulkmtime` 162d⟩
⟨*function* `mkmtime` 88b⟩

```
// could be in run.c
Shell rc = {
  .shellname = "rc",
  .shell = "/bin/rc",
};

Shell* shell = &rc;
```

⟨*global* `shell` 113a⟩
⟨*global* `shellname` 113b⟩

```
// could be in run.c
```
⟨*function* `execsh` 113d⟩
⟨*function* `waitfor` 116a⟩

```
// could be in run.c
```
⟨*function* `Exit` 118e⟩

```
// back in run.c
extern void killchildren(char *msg);
```

⟨*function* `notifyf` 119c⟩
⟨*function* `catchnotes` 119b⟩
⟨*function* `expunge` 151g⟩

```
// could be in run.c
```
⟨*function* `pipecmd` 141a⟩

```
// could be in graph.c
```
⟨*function* `rcopy` 136e⟩

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()`: 38b, 41c
`addrules()`: 41c, 52f, 73a
`addw()`: 35b, 107, 152k
`aflag`: 152e, 163a, 163a, 163b, 163c
`ambiguous()`: 88d, 92c, 92c
`applyrules()`: 80, 81, 82b, 83e
`Arc`: 43b, 43c, 181d
`Arc.match`: 136c
`Arc.n`: 43b
`Arc.next`: 43c
`Arc.prog`: 148l
`Arc.r`: 43b
`Arc.remove`: 94c
`Arc.stem`: 43d
`Arc` (typedef): 181d
`assline()`: 55, 57b
`atimeof()`: 154f, 154g
`atimes()`: 154g, 156a
`atouch()`: 155, 160c
`attribute()`: 145b, 145c, 145c
`badusage()`: 48d, 49a, 51b, 132e
`BEINGMADE`: 42e, 99, 101c, 103a, 104e, 165a, 165c, 165d
`BIGBLOCK`: 106a, 156a, 181a
`bout`: 48a, 48b, 99, 119e, 128b, 149f, 156c, 158e, 160a, 160b, 160c, 169b, 169c, 169d, 169e, 170a, 170b, 170c, 171a, 171g, 173c
`bquote()`: 137g, 138a
`Bufblock`: 174c, 175a, 181d
`Bufblock.current`: 174c
`Bufblock.end`: 174c
`Bufblock.next`: 175a
`Bufblock.start`: 174c
`Bufblock` (typedef): 181d
`bufcontent`: 57b, 175h
`bufcpy()`: 69c, 69d, 71a, 129b, 132e, 163e, 164a, 164d, 176c
`buildenv()`: 111c, 125b, 128d
`bulkmtime()`: 162d
`CANPRETEND`: 158c, 158d, 158e, 159b, 159f
`catchnotes()`: 192c

charin(): 39a, 62b, 73b, 142c, 144
chgtime(): 192c
clrmade(): 99, 101a, 101a
copyq(): 129a, 130d
copysingle(): 130d, 131a
CYCLE: 89b, 89c
cyclechk(): 88d, 89c, 89c
DEBUG: 141a, 168c, 169a, 170a, 170e, 171a, 171b, 171c, 171d, 171e, 171f, 171h, 171i, 172a, 172b, 192c
debug: 168b, 168e
DEL: 146h, 147c
DELETE: 147b, 147c, 147e
delete(): 147e, 147f
dirtime(): 162d, 162e
dorecipe(): 103a, 104e
dumpa(): 170b, 170c
dumpj(): 171g
dumpn(): 170a, 170b, 170c
dumpr(): 169a, 169c
dumpv(): 169a, 169d
dumpw(): 169a, 169b
Dxxx: 168a
D_EXEC: 141a, 168a, 168e, 170e, 171a, 171b, 171c, 171d, 171e, 171f, 192c
D_GRAPH: 168a, 168e, 170a
D_PARSE: 168a, 168e, 169a
D_TRACE: 168a, 171h, 171i, 172a, 172b
ecopy(): 122b, 122c
EMPTY_CHILDREN_IS_ERROR2: 165e, 165f
EMPTY_CHILDREN_IS_ERROR3: 140b, 140d, 149f
EMPTY_CHILDREN_IS_ERROR: 99, 117b
EMPTY_CHILDREN_IS_OK: 99, 117b, 118a, 119e
EMPTY_CHILDREN: 117c, 118a
empty_prereqs: 83a, 83c, 83d, 84d, 84e
empty_recipe: 83a, 83b, 84d, 92c, 104e
empty_words: 62a, 109c, 126d
envinsert(): 121a, 121f, 122b, 122c
ENVQUANTA-6: 121c, 121d
envupd(): 121e, 125b, 135g, 137b, 152i, 153d, 153i, 153k, 154c
Envy (typedef): 192c
escapetoken(): 60a, 60b, 138b
events-15: 110a, 110c, 111a, 111b, 111c, 115c, 171b
execsh(): 192c
Exit(): 192c
expandquote(): 66b, 66c
expandvar(): 142b, 142c
explain: 131b, 131b, 131c, 131d, 131e, 158e, 159a, 160a, 160b
exportenv(): 192c
expunge(): 192c
extractpat(): 144
freebuf(): 35d, 55, 65, 67d, 69a, 73c, 118c, 128b, 132f, 142c, 164d, 175e

freelist-12: 175b, 175d, 175e, 176d
freewords(): 35a, 121e, 127a
front(): 128d, 128e
graph(): 80, 99
growbuf(): 139g, 176a, 176b, 176d, 192c
hash-3: 31b, 32a, 32e, 33a, 173c
HASHMUL-2: 32c, 32d
iflag: 158a, 158a, 158b, 158e, 163b
infile: 54a, 54b, 55, 77c, 77d, 136a
initenv(): 122a, 122b, 138d, 140d
inithash(): 34a, 48c
Input: 76g, 77a, 77b, 77c, 77d
Input.file: 76g
Input.line: 76g
Input.next: 77b
inputs-14: 77a, 77a, 77c, 77d
insert(): 35d, 57b, 59c, 65, 68d, 69c, 69d, 71a, 73c, 129a, 132e, 132f, 138a, 138b, 163e, 164a, 164c, 164d, 176a, 176c
ipop(): 76f, 77d
ipush(): 76e, 77c
isempty: 57b, 59c, 65, 69a, 69c, 175f
IWS: 192c, 192c
Job: 44a, 45b, 181d
Job.at: 152g
Job.match: 136f
Job.n: 44a
Job.next: 45b
Job.np: 153b
Job.p: 44a
Job.r: 44a
Job.stem: 44a
Job.t: 44a
Job (typedef): 181d
jobs: 45a, 108a, 111c, 112, 118c, 119e, 165f
JOB_ENDED: 115c, 117c, 119e
kflag: 119e, 164e, 164e, 164f, 165a, 165b
killchildren(): 119c, 119e, 192c
lmr-24: 37f, 39a
lr-23: 37d, 38c
MADE: 42e, 102a, 103a, 116b, 132c, 154e, 158e, 159a, 165f
MADESET: 101a, 101b, 102a, 103a, 104e, 116b, 132c, 154e, 158e, 159b, 165c
main-11(): 47b
maketmp(): 192c
Malloc(): 32e, 34d, 38b, 42b, 43e, 44c, 77c, 91f, 110c, 123a, 150d, 174a, 175d
membername(): 192c
META: 38a, 39a, 94b, 96d, 106b
metarules: 37e, 39a, 83e, 169a
mk(): 52b, 52e, 52f, 99
MKFILE-9: 51c, 51d

mkinline: 52f, 54c, 55, 58b, 58c, 59b, 60b, 74a, 77c, 77d, 136a, 138a
mkmtime(): 192c
mygetenv(): 129b, 130a
NAMEBLOCK: 84a, 84b, 84c, 149d, 149f
Namespace: 31a
nevents-16: 110b, 110c, 111b, 134c
newarc(): 43e, 82b, 83d, 83e, 84e
newbuf(): 35d, 55, 65, 68d, 73c, 118c, 128b, 132e, 163d, 175d
newjob(): 44c, 104e
newnode(): 42b, 81
newword(): 34d, 35b, 35c, 52f, 64c, 65, 106b, 125b, 135f, 135g, 137b, 142c, 152e, 153i, 153k, 192c
nextrune(): 57b, 58a, 60b, 138a
nextslot(): 111a, 111c
nextv-7: 120c, 121a, 121c, 122b
nextword(): 64b, 65
nflag: 128b, 131f, 131f, 132a, 132c, 160c
NHASH-1: 31b, 31c, 32c, 33a, 173c
Node: 42a, 43b, 44b, 181d
Node.arcs: 43a
Node.flags: 42c
Node.name: 42a
Node.next: 44b
Node.time: 42a
Node (typedef): 181d
Node_flag: 42d
NOMINUSE: 147i, 148a
NOPLAN9DEFINES-4: 192c
NOREC: 148b, 148e
NORECIPE: 146f, 148d, 148e
notifyf(): 192c
NOTMADE: 42e, 99, 101a, 103a, 159b, 165d
NOT_A_JOB_PROCESS: 117c
NOVIRT: 148f, 148h
nproc(): 109b, 109c
nproclimit-18: 108a, 109a, 109c, 110c, 111a, 115c, 171f
NREGEXP: 136b, 136c, 136d, 137c, 137e, 137f
nreps: 91h, 91h, 91i
nrules-25: 91b, 91b, 91c, 91f
nrunning-17: 108a, 108b, 111c, 115c, 134a
outofdate(): 103a, 103b, 107, 146g, 149c, 152e, 158e, 159a
parse(): 49c, 51c, 55, 76a, 140d
patrule: 135d, 135e, 136a, 137e
pcmp(): 149e, 149f
pdelete(): 150e, 151c
PERCENT: 37g, 86
pfree-20: 150c, 150d, 150e
phead-19: 150b, 150d, 150e, 151c, 151f
pidslot(): 111b, 115c
pipecmd(): 192c

pnew(): 150d, 151a
PRETENDING: 158c, 158d, 158e, 159a, 159b, 159f
print1(): 169d, 169e
PROBABLE: 95b, 95c, 95d, 95e, 96d
Process: 150a, 188c
Process.b: 150a
Process.f: 150a
Process.pid: 150a
Process.status: 150a
Process (typedef): 188c
prusage(): 134b, 134c
QUANTA-13: 175c, 175d, 176d
QUIET: 128b, 147g
rbody(): 73a, 73c
rc: 192c
rcopy(): 192c
readenv(): 192c
READY: 96c, 96d
Realloc(): 110c, 121c, 174b, 176d, 192c
regerror(): 136a
REGEXP: 39a, 73b, 135a, 135e, 135f, 135g, 137b, 137e, 137f
resetbuf: 57b, 69c, 69d, 175g
rinsert(): 57b, 60a, 60b, 65, 66c, 68d, 73c, 74b, 129a, 130d, 131a, 138a, 176b
Rule: 36, 37c, 39c, 181d
Rule.alltargets: 41d
Rule.attr: 37h
Rule.chain: 39c
Rule.file: 37a
Rule.line: 37a
Rule.next: 37c
Rule.pat: 135c
Rule.prereqs: 36
Rule.prog: 148i
Rule.recipe: 36
Rule.rule: 91a
Rule.target: 36
Rule (typedef): 181d
rulecnt(): 91e, 91f
rules: 37b, 38c, 169a
Rule_attr: 38a
run(): 104e, 108a
runerrs: 164g, 164h, 165a, 165b, 165d, 165f
RunEvent: 109d, 188c
RunEvent.job: 109d
RunEvent.pid: 109d
RunEvent (typedef): 188c
sched(): 108a, 111c, 115c
setvar(): 33b, 78b, 123a, 192c
Shell: 181d, 181d

shell: 192c
Shell.shell: 181d
Shell.shellname: 181d
Shell (typedef): 181d
shellenv: 120b, 121a, 121c, 121e, 125b, 138d, 140d
ShellEnvVar: 120a, 181d, 192c
ShellEnvVar.name: 120a
ShellEnvVar.values: 120a
ShellEnvVar (typedef): 181d
shflags: 113c, 113c, 113d, 141a, 192c
shname(): 123b, 124a, 129b, 192c
shprint(): 128b, 128d, 129a
sigmsgs-5: 192c, 192c
specialvars-8: 33d, 34a, 122b, 122d, 137b
split(): 154g, 155, 157a
squote(): 63a, 63b
stab: 192c, 192c
stow(): 64b, 122b, 144, 164c, 164d
submatch(): 145a
subst(): 83e, 86, 106b
symlook(): 32a, 33b, 34a, 40a, 40c, 47b, 78e, 82b, 87b, 87c, 106b, 109c, 115c, 123b, 126d, 130a, 130c, 132g, 142c, 149e, 152a, 152c, 154g, 155, 156a, 156c, 161c, 161d, 162d, 162e, 164c, 164d, 192c
symstat(): 173c
Symtab: 30, 31d, 181d
Symtab.name: 30
Symtab.next: 31d
Symtab.space: 30
Symtab.u: 30
Symtab (typedef): 181d
symtraverse(): 33a, 122b, 169d
SYNERR: 54d, 57a, 60b, 63b, 69a, 76b, 78c, 138a, 140e, 142a, 142c
S_AGG: 154d, 154g
S_BITCH: 156b, 156c
S_BULKED: 162c, 162d
S_INTERNAL: 33c, 34a, 123b, 130a, 192c
S_NODE: 87a, 87b, 87c, 106b, 115c
S_NOEXPORT: 152a, 152b, 152c
S_OUTOFDATE: 149b, 149e
S_OVERRIDE: 78e, 79a
S_TARGET: 39b, 40a, 40c, 82b
S_TIME: 132g, 154g, 155, 156a, 160e, 161c, 161d, 162e
S_VAR: 31a, 33b, 47b, 109c, 122b, 126d, 142c, 164c, 164d, 169d
S_WESET: 130a, 130b, 130c
target1: 51f, 52b, 73b, 169a
termchars: 79d, 79d
tflag: 128b, 132c, 159g, 159g, 159h, 160a, 160b
tick-22: 133d, 134a
timeinit(): 132f, 132g
timeof(): 42b, 88a, 116b

togo(): 94e, 94f, 96d
touch(): 160a, 160b, 160c
trace(): 92c, 93a
tslot-21: 133c, 134a, 134c
type(): 156c, 157a
uflag: 133a, 133a, 133b, 134b
unpretend(): 158e, 159a, 159b
update(): 115c, 116b, 146f
usage(): 111c, 112, 115c, 133e, 133f, 134a, 134c
VACUOUS: 96b, 96d
vacuous(): 96a, 96d, 96d
varname(): 67d, 68d, 142c
varsub(): 67c, 67d
version-10: 47a, 47a
vexpand(): 129a, 129b
VIR: 52f, 146a, 146d, 148h
VIRTUAL: 146c, 146d, 146e, 146f, 146g, 160a, 160b
waitup(): 99, 115c, 119e, 140d, 149f, 165f
WaitupParam: 117b
WaitupResult: 117c
wdup(): 35c, 73b, 125b, 152i, 153d, 154c
Word: 34b, 34c, 181d
Word.next: 34c
Word.s: 34b
Word (typedef): 181d
WORDCHR: 68d, 68e, 124a
work(): 99, 101c, 103a, 158e
wtos(): 35d, 76a, 130a, 140d, 169c, 170e, 171b, 171g, 192c
xwaitfor(): 192c
__anon_struct_1.msg: 192c
__anon_struct_1.sig: 192c
__anon_struct_1: 192c, 192c
__anon_struct_2.ptr: 30
__anon_struct_2.value: 30
__anon_struct_2: 30, 30

Bibliography

[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) 11

[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, 11

[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, 23

[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) 11

[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, 11

[OL96] Andy Oram and Mike Loukides. *Programming with GNU Software*. O'Reilly, 1996. cited page(s) 11

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

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 114, 116, 123, 167

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 167

[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, 55, 58, 167

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 33, 48, 50, 55, 63, 114, 116

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 13, 15, 16, 60, 112, 113, 166, 167

[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) 11