Principia Softwarica: The Build System `mk`

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Andrew Hume

March 24, 2026

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

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

Contents

1 Introduction 7
- 1.1 Motivations 7
- 1.2 The Plan 9 build system: `mk` 8
- 1.3 Other build systems 8
- 1.4 Getting started 10
- 1.5 Requirements 11
- 1.6 About this document 11
- 1.7 Copyright 11
- 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` 36
  - 3.3.3 `metarules` 37
  - 3.3.4 Adding rules 37
- 3.4 Graph 41
  - 3.4.1 `Node` 41
  - 3.4.2 `Arc` 42
- 3.5 `Jobs` 43

4 main() **46**
- 4.1 main() skeleton . . . 46
- 4.2 mk -*flag* arguments processing . . . 47
- 4.3 mk *var*=*values* arguments processing . . . 48
- 4.4 mk remaining arguments processing . . . 49
- 4.5 Using the mkfile or mk -f*file* . . . 49
- 4.6 Building the target(s) . . . 50
  - 4.6.1 Default target: target1 . . . 50
  - 4.6.2 Building Sequentiallly . . . 51
  - 4.6.3 Building in Parallel . . . 51

5 Parsing the mkfile **53**
- 5.1 parse() . . . 53
  - 5.1.1 Assembling a line: assline() . . . 55
  - 5.1.2 Parsing the head of a line: rhead() . . . 59
  - 5.1.3 Splitting a string in words: stow() . . . 62
- 5.2 Parsing Rules: *target*:*prereqs* . . . 69
  - 5.2.1 Parsing the recipe: rbody() . . . 71
  - 5.2.2 Parsing rule attributes . . . 72
- 5.3 Parsing Included files: <*file* . . . 73
- 5.4 Parsing Variable definitions: *var*=*values* . . . 75
  - 5.4.1 Overriding variable definitions . . . 76
  - 5.4.2 Parsing variable attributes . . . 76

6 Building the Graph of Dependencies **78**
- 6.1 graph() and applyrules() . . . 78
- 6.2 Finding the simple rule(s) for a target . . . 79
- 6.3 Finding matching metarules . . . 80
  - 6.3.1 Matching a pattern: match() . . . 82
  - 6.3.2 Substituting the stem: subst() . . . 82
- 6.4 Node cache . . . 83
- 6.5 timeof() . . . 84
- 6.6 Checking the graph and the rules . . . 85
  - 6.6.1 Cycle detection . . . 85
  - 6.6.2 Infinite rule detection . . . 87
  - 6.6.3 Ambiguous rules detection . . . 88

7 Finding Outdated Files **94**
- 7.1 mk() . . . 94
- 7.2 Initializing nodes: clrmade() . . . 96
- 7.3 Exploring the graph: work() . . . 97
  - 7.3.1 The leaf case . . . 97
  - 7.3.2 The parent case . . . 98
- 7.4 Scheduling recipes: dorecipe() . . . 99
  - 7.4.1 Building the list of target nodes . . . 101
  - 7.4.2 Building the list of prerequisites . . . 102

8 Scheduling Jobs **103**
8.1 Enqueuing jobs: `run()` 103
8.2 Scheduling jobs 104
8.2.1 `RunEvent` and `events` 104
8.2.2 `sched()` 105
8.3 Executing Jobs: `execsh()` 107
8.4 Waiting for jobs to finish 108
8.4.1 `waitup()` 109
8.4.2 `update()` 110
8.4.3 `waitup()` edge cases 110
8.5 Process management, `Exit()` 112
8.6 Notes (signals) management 112

9 The Shell Environment **114**
9.1 `Shellenv` and `shellenv` 114
9.2 Initializing the shell environment: `initenv()` 115
9.3 Importing the environment: `readenv()` 116
9.4 Adjusting the shell environment: `buildenv()` 119
9.5 Exporting the shell environment: `exportenv()` 119

10 Debugging and Profiling Support TODO **122**
10.1 Printing jobs: `shprint()` 122
10.1.1 Expanding and printing variables 123
10.1.2 Printing quoted strings 124
10.2 Explain mode: `mk -e` 125
10.3 Dry mode: `mk -n` 125
10.4 What-if mode: `mk -w`*file* 126
10.5 Processor utilization: `mk -u` 127

11 Advanced Features TODO **129**
11.1 Regular-expression rules: `:R:` 129
11.2 Shell-command expansion: ‘*cmd*‘ 131
11.2.1 Parsing backquotes: `bquote()` 131
11.2.2 Adjusting `execsh()` 132
11.3 Dynamic `mkfile`: `<|`*prog* 133
11.4 Substitution variables: `$`*`name`*`:`*`pattern`*`=`*`subst`* 135
11.4.1 Parsing adjustments 135
11.4.2 Substitutions: `subsub()` 136
11.5 Rule attributes 139
11.5.1 Virtual target: `:V:` 139
11.5.2 Deleting a target when the recipe returns an error: `:D:` 140
11.5.3 Not printing the recipe (quiet mode): `:Q:` 141
11.5.4 Running a shell script without `-e`: `:E:` 141
11.5.5 Disabling the no-recipe warning, `:N:` 141
11.5.6 Forbidding metarules to match virtual targets: `:n:` 142
11.5.7 Custom-dependency comparison program: `:P:` 142
11.6 Variable attributes 145
11.6.1 Private variables: `=U=` 145
11.7 Advanced `mk` variables 145
11.7.1 `$target` versus `$alltargets` 145

11.7.2 $prereq versus $newprereq 146
11.7.3 $NREP 146
11.7.4 $pid 147
11.7.5 $nproc 147
11.8 Dealing with archives (libraries) 147
11.9 Optimizations 151
11.9.1 Missing-intermediates optimization: mk -I 151
11.9.2 Touching-mode optimization: mk -t 153
11.9.3 Time cache 154
11.9.4 Bulk time optimisation 155
11.10 Recompiling everything: mk -a 156
11.11 Recursive mk 157
11.12 mk -k 157
11.12.1 kflag and runerrs 157
11.12.2 Adjusting mk(), work(), and waitup() 158

12 Conclusion **159**
12.1 Patterns and techniques 159
12.2 Connections to other books 159
12.3 Beyond mk 160

A Debugging **161**
A.1 Dumping the rules: mk -dp 162
A.2 Dumping the graph: mk -dg 163
A.3 Tracing jobs: mk -de 163
A.4 Tracing function calls: mk -dt 164

B Profiling **166**

C Utilities **167**
C.1 Memory management 167
C.2 Buffer management 167
C.3 File management 170

D Examples of mkfiles TODO **171**
D.1 The mkfile of mk 171
D.2 The mkfiles of Plan 9 171
D.2.1 /$objtype/mkfile for the ARM 171
D.2.2 /sys/src/mkfile.proto 171
D.2.3 /sys/src/cmd/mkone 171
D.2.4 /sys/src/cmd/mklib 171

E Extra Code **172**
E.1 mk/ 172
E.1.1 mk/fns.h 172
E.1.2 mk/mk.h 174
E.1.3 mk/globals.c 176
E.1.4 mk/utils.c 177
E.1.5 mk/bufblock.c 177
E.1.6 mk/symtab.c 177
E.1.7 mk/rc.c 178

E.1.8 mk/word.c 178
E.1.9 mk/var.c 178
E.1.10 mk/archive.c 178
E.1.11 mk/match.c 179
E.1.12 mk/env.c 179
E.1.13 mk/parse.c 179
E.1.14 mk/shprint.c 180
E.1.15 mk/job.c 180
E.1.16 mk/arc.c 180
E.1.17 mk/rule.c 180
E.1.18 mk/lex.c 181
E.1.19 mk/file.c 181
E.1.20 mk/run.c 181
E.1.21 mk/graph.c 183
E.1.22 mk/mk.c 183
E.1.23 mk/recipe.c 184
E.1.24 mk/varsub.c 184
E.1.25 mk/dumpers.c 185
E.1.26 mk/main.c 185
E.1.27 mk/Posix.c 185
E.1.28 mk/Plan9.c 191

Glossary **193**

Index **194**

References **200**

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.

`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 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

[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/`.
[7] `https://www.gnu.org/software/make/`

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.

- 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 `Makefiles`. Unlike Ant, CMake acts as a frontend to Make (and other build systems). Indeed, CMake is a *meta build system*. Instead of writing `Makefiles`, the user of CMake creates `CMakeLists.txt` files containing builtin (portable) commands to compile source code. CMake then generates from those configuration files regular `Makefiles` that can be processed by Make. CMake can also generate files for IDEs such as Apple's Xcode or Microsoft's Visual Studio.

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

[8] http://ant.apache.org/
[9] https://cmake.org/

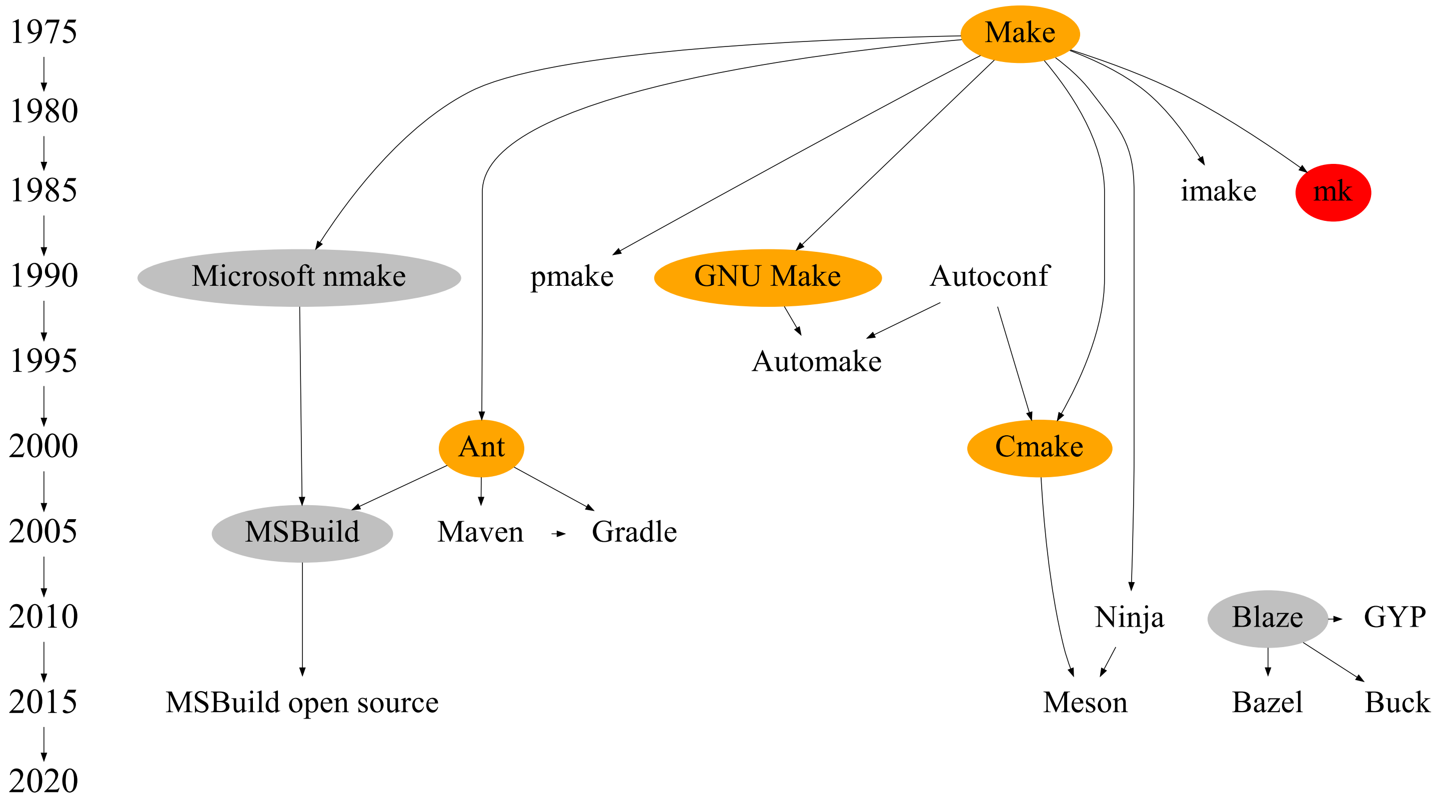


Figure 1.1: Build systems timeline

Note that the lack of portability of `Makefiles` 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), `Makefiles` are now more portable because the same command-line programs are available on more platforms.

Figure 1.1 presents a timeline of major build systems. I think `mk` represents the best compromise for this book: it implements the essential features of a build system while still having a small and understandable codebase (5500 LOC).

1.4 Getting started

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

[10]https://www.cygwin.com/
[11]https://www.macports.org/
[12]http://brew.sh/
[13]https://9fans.github.io/plan9port/
[14]https://github.com/aryx/goken9cc

```
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.

1.6 About this document

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

1.7 Copyright

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

1.8 Acknowledgments

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

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.

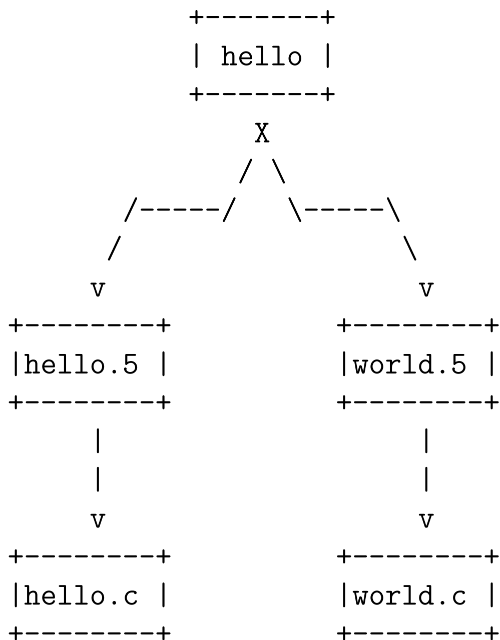


Figure 2.1: Graph of dependencies for `hello`.

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.

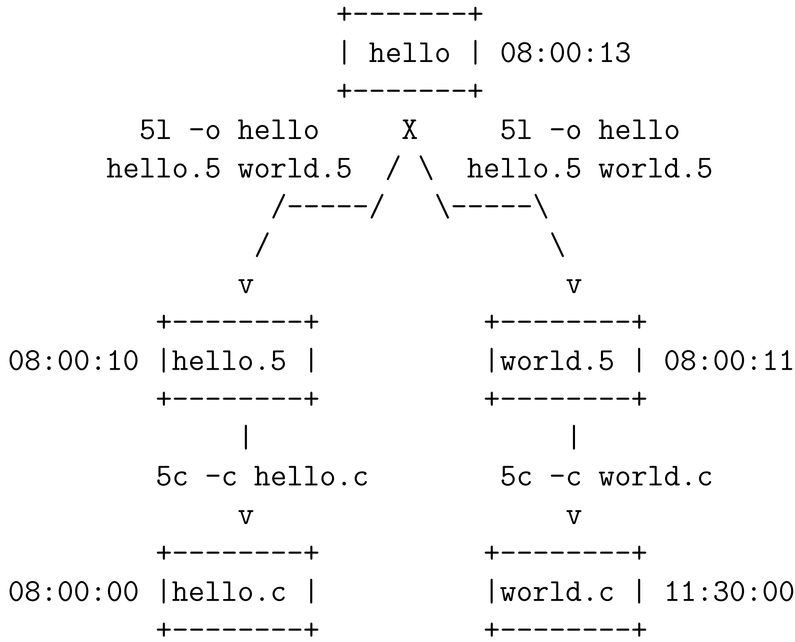


Figure 2.2: Graph of dependencies for `hello`, with labels.

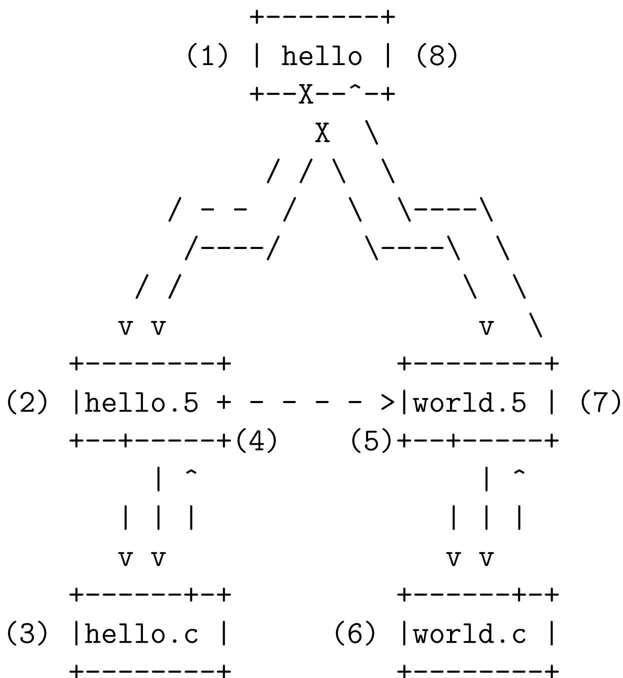


Figure 2.3: Depth-first search traversal of the graph dependencies for `hello`.

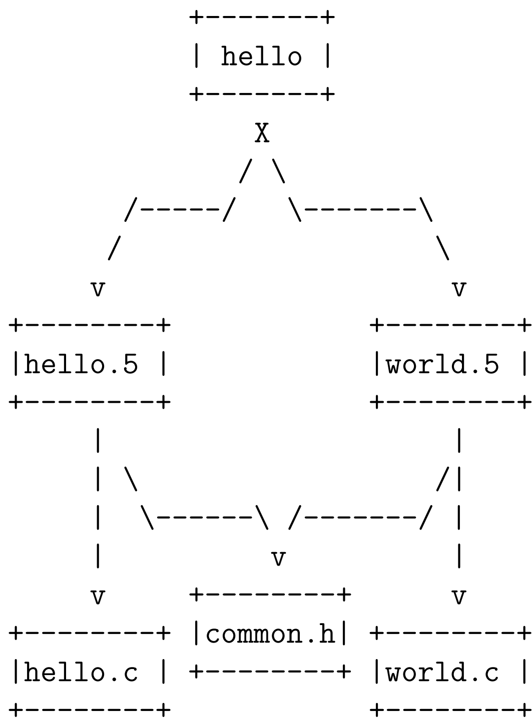


Figure 2.4: Graph of dependencies for `hello` with `common.h`.

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`.

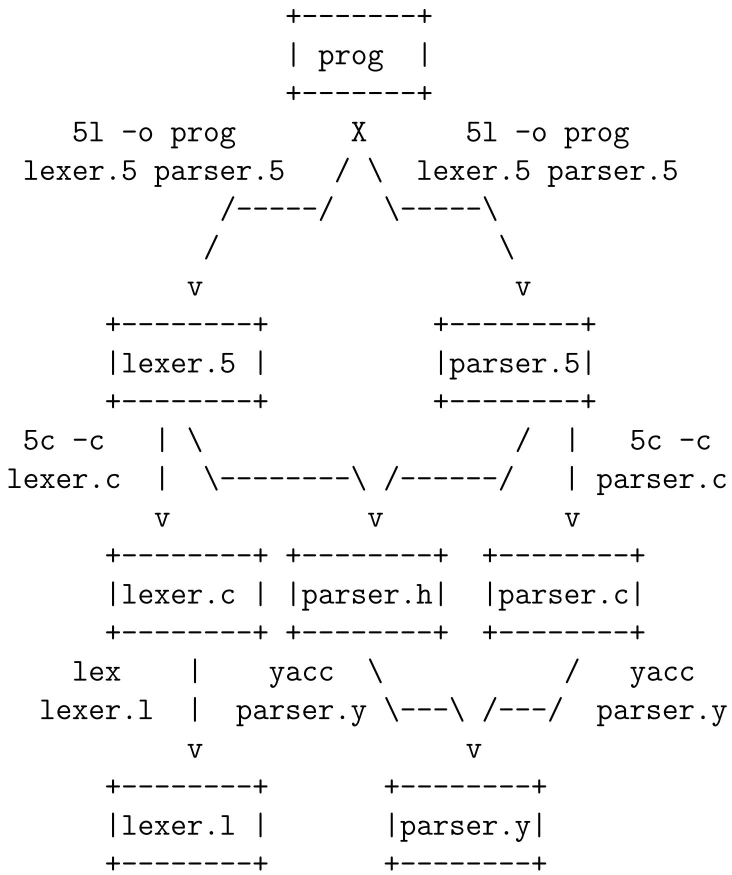


Figure 2.5: Graph with many-to-one dependencies.

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.

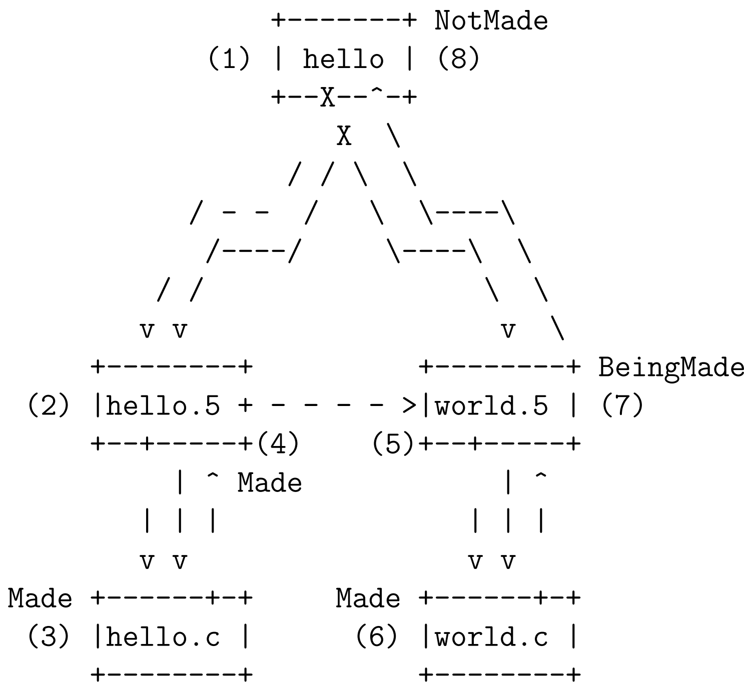


Figure 2.6: Graph of dependencies with building-status labels.

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`.

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

2.4 Code organization

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

2.5 Software architecture

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

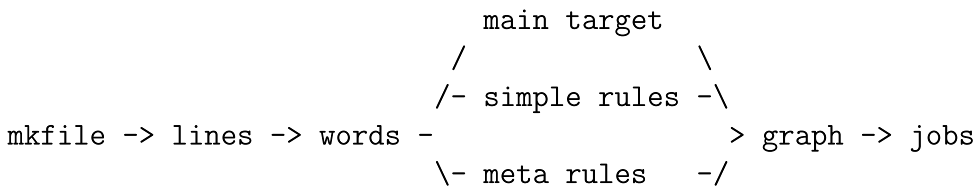
1. *Parse* the `mkfile` (via `parse()` [53e]) to extract the rules and meta rules in the file
2. *Build* the graph of dependencies (via `graph()` [78]) for a specific target given the rules extracted previously
3. *Find* outdated files in the graph (via `work()` [97a])
4. *Schedule* jobs (via `sched()` [105e]) that will run the shell recipes (via `execsh()` [185c]) to update the outdated files

Starting from the top of Figure 2.7, the function `main()` [46b], 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()` [55b]), which is then split in words, which are then analyzed to populate important globals such as `rules` [36c] and `metarules` [37b], which contain the lists of extracted rules. `parse()` also sets the global `target1` [50e] 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()` [94] (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.

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

Figure 2.7: Control flow diagram of `mk`.



(use global symbol table)

Figure 2.8: Data flow diagram of `mk`.

Function	Ch.	File	Entities	LOC
data structures and constants	3	`mk.h`	`Symtab`[30] `Word`[34a] `Rule`[36a] `Node`[41d] `Arc`[42f] `Job`[44a]	375
symbol table and cache	3	`symtab.c`	`hash`[31b] `symlook()`[31e] `symtraverse()`[32e]	89
variables	3	`var.c`	`setvar()`[33a]	28
list of strings (words)	3	`word.c`	`newword()`[34c] `wtos()`[35b]	90
globals	3	`globals.c`	`rules`[36c] `metarules`[37b] `jobs`[45a]	63
adding rules	3	`rule.c`	`addrule()`[38a]	163
function prototypes	3	`fns.h`		140
entry point	4	`main.c`	`main()`[46b]	331
lexer	5	`lex.c`	`assline()`[55b] `nextrune()`[56a]	160
parser	5	`parse.c`	`parse()`[53e] `rhead()`[59b] `rbody()`[71a]	418
escaping methods for `rc`	5	`rc.c`	`escapetoken()`[58b] `charin()`[60c] `squote()`[61b]	211
parsing and expanding variables	5	`varsub.c`	`stow()`[62b] `varsub()`[65d] `varname()`[66d]	439
building and checking the graph	6	`graph.c`	`graph()`[78] `applyrules()`[79a] `cyclechk()`[86b]	455
pattern matching and substituting	6	`match.c`	`match()`[82a] `subst()`[82c]	71
file and time management	6	`file.c`	`timeof()`[85a]	104
finding outdated files in the graph	7	`mk.c`	`mk()`[94] `work()`[97a] `outofdate()`[99a] `update()`[110a]	343
constructing a job	7	`recipe.c`	`dorecipe()`[100a]	197
scheduling jobs	8	`run.c`	`run()`[103a] `RunEvent`[104c] `sched()`[105e] `waitup()`[109a]	455
shell environment	9	`env.c`	`buildenv()`[119b] `shellenv`[114b] `envinsert()`[114d]	218
printing shell commands	10	`shprint.c`	`shprint()`[123a]	109
handling archives (libraries)	11	`archive.c`	`atimeof()`[148e]	178
dumpers	A	`dumpers.c`	`dumpv()`[162d] `dumpr()`[162c] `dumpn()`[163b]	105
memory management	C	`utils.c`	`Malloc()`[167a] `Realloc()`[167b]	34
string buffer management	C	`bufblock.c`	`newbuf()`[168d] `insert()`[169a]	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()`[79a] 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()`[100a] 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()`[103a] (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`[42d]). 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()`[109a] (at the right in Figure 2.7) to wait for those jobs to finish. Once a job is finished, `waitup()` calls `update()`[110a] 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()`[46b] 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`[174d] 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⟩≡ (174d)
```
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⟩≡ (174d)
```
enum Namespace {
    S_VAR, /* variable -> value */ // value is a list of words
    ⟨Sxxx cases 33b⟩
};
```

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⟩≡ (177c)
```
// hash<(string * enum<Namespace>), 'a> (next = Symtab.next in bucket)
static Symtab *hash[NHASH];
```
Uses NHASH-1 31c.

⟨*constant* `NHASH` 31c⟩≡ (177c)
```
#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).

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` 31e⟩≡ (177c)
```
Symtab*
symlook(char *sym, int space, void *install)
{
    Symtab *s;
    long h;
    ⟨symlook() other locals 32a⟩

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

    // 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 32d⟩
}
```
Uses hash-3 31b.

⟨symlook() *other locals* 32a⟩≡ (31e)
```
char *p;
```

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

⟨*constant* HASHMUL 32c⟩≡ (177c)
```
#define HASHMUL 79L /* this is a good value */
```

If `symlook()`[31e] does not find the key and the install parameter is not `nil`, it creates a new symbol:

⟨symlook() *if symbol not found* 32d⟩≡ (31e)
```
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() 167a and hash-3 31b.

`Malloc()`[167a], 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 32e⟩≡ (177c)
```
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` 33a⟩≡ (178c)
```
void
setvar(char *name, void *value)
{
    symlook(name, S_VAR, value)->u.ptr = value;
}
```
Uses `S_VAR` 31a and `symlook()` 31e.

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

Thus, to get the value of the special variable `stem`, call the `symlook()`[31e] function with the pair `("stem", S_INTERNAL)`.

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

⟨*global* `specialvars` 33c⟩≡ (179b)
```
static char *specialvars[] =
{
    "target",
    "prereq",
    "stem",

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

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

`specialvars`[33c] is used to initialize entries in the symbol table in `inithash()` (called from `main()`[46b]):

⟨*function* `inithash` 33d⟩≡ (179b)
```
void
inithash(void)
{
    char **p;

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

    readenv();    /* o.s. dependent */
}
```
Uses `S_INTERNAL` 33b, `specialvars-8` 33c, and `symlook()` 31e.

`readenv()`[185c] 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` 34a⟩≡ (174d)
```
struct Word
{
    // ref_own<string>
    char   *s;

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

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

`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` 34c⟩≡ (178b)
```
Word*
newword(char *s)
{
    Word *w;

    w = (Word *)Malloc(sizeof(Word));
    w->s = strdup(s);
    w->next = nil;
    return w;
}
```
Uses `Malloc()` 167a.

`freewords()` frees a list of words:

⟨*destructor* `freewords` 34d⟩≡ (178b)
```
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` 34e⟩≡ (178b)
```
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()` 34c.

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

⟨*function* `wdup` 35a⟩≡ (178b)

```
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()` 34c.

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

⟨*function* `wtos` 35b⟩≡ (178b)

```
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()` 168e, `insert()` 169a, and `newbuf()` 168d.

`wtos()`[35b] relies on the `Bufblock`[167c] 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`[177b], 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 36a⟩≡ (174d)
```
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 37e⟩
    ⟨Rule debug fields 36b⟩

    // Extra
    ⟨Rule extra fields 36d⟩
};
```

Rule.prereqs contains a list of words, hence the use of a pointer to a Word[34a]. 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* 36b⟩≡ (36a)
```
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 36c⟩≡ (176)
```
// 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()[38a] function).

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

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

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

⟨*global* `lr` 37a⟩≡ (180d)
```
// 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` 37b⟩≡ (176)
```
// 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` 37c⟩≡ (180d)
```
// 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` 37d⟩≡ (174d)
```
#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`[36c], a meta rule contains also the `META` *rule attribute* in `Rule.attr`:

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

⟨*enum* `Rule_attr` 37f⟩≡ (174d)
```
enum Rule_attr {
    META    = 0x0001,
    ⟨Rule_attr cases 129a⟩
};
```

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` 38a⟩≡ (180d)
```
void
addrule(char *target, Word *prereqs, char *recipe,
        Word *alltargets, int attr, int hline, char *prog)
{
    Rule *r = nil;
    ⟨addrule() other locals 39c⟩

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

    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 40d⟩

    ⟨addrule() indexing r by target in S_TARGET 39d⟩

    ⟨addrule() if meta rule 38c⟩
    else {
        ⟨addrule() if simple rule 38b⟩
    }
}
```
Uses `Malloc()` 167a.

If the rule is a simple rule, `mk` populates `rules`[36c]:

⟨`addrule()` *if simple rule* 38b⟩≡ (38a)
```
⟨addrule() return if reuse, to not add the rule in a list 40e⟩
// else

// add_list(r, rules, lr)
if(rules == nil)
    rules = lr = r;
else {
    lr->next = r;
    lr = r;
}
```
Uses `lr-23` 37a and `rules` 36c.

If the rule is a meta rule, `mk` populates `metarules`[37b]. `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* 38c⟩≡ (38a)
```
if(charin(target, "%&") || (attr&REGEXP)){
    r->attr |= META;
    ⟨addrule() return if reuse, to not add the rule in a list 40e⟩
    // else
    ⟨addrule() if REGEXP attribute 129e⟩

    // add_list(r, metarules, lmr)
```

```
    if(metarules == nil)
        metarules = lmr = r;
    else {
        lmr->next = r;
        lmr = r;
    }
}
```

Uses META 37f, REGEXP 129a, charin() 60c, lmr-24 37c, and metarules 37b.

I will describe the function charin()[60c] and the rule attribute REGEXP[129a] 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()[79a] 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* 39a⟩+≡ (31a) ◁33b 76d▷

```
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[36a]:

⟨Rule *extra fields* 39b⟩+≡ (36a) ◁36d

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

Uses Rule 36a.

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

⟨addrule() *other locals* 39c⟩≡ (38a) 40a▷

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

⟨addrule() *indexing* r *by target in* S_TARGET 39d⟩≡ (38a)

```
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 39a and symlook() 31e.

Remember that the last parameter of `symlook()`[31e], 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* 40a⟩+≡ (38a) ◁39c

```
bool reuse;
```

⟨`addrule()` *find if rule already exists, set* `reuse`*, update* `r` 40b⟩≡ (38a)

```
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` 39a and `symlook()` 31e.

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

⟨*function* `rcmp` 40c⟩≡ (180d)

```
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()`[38a] overwrites (reuses) a previous rule, the `Rule.next`[36d] field of this rule should not be modified. Otherwise, `Rule.next` should be set to `nil`:

⟨`addrule()` *set more fields* 40d⟩≡ (38a) 41c▷

```
if(!reuse){
    r->next = nil;
}
```

⟨`addrule()` *return if reuse, to not add the rule in a list* 40e⟩≡ (38)

```
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` 41a⟩≡ (179c)
```
void
addrules(Word *targets, Word *prereqs, char *recipe,
         int attr, int hline, char *prog)
{
    Word *w;

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

    ⟨addrules() set target1 69d⟩
    for(w = targets; w; w = w->next)
        addrule(w->s, prereqs, recipe, targets, attr, hline, prog);
}
```
Uses `addrule()` 38a.

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

⟨`Rule` *other fields* 41b⟩+≡ (36a) ◁37e 87b▷
```
// ref<list<ref_own<string>>
Word   *alltargets; /* all the targets */
```

⟨`addrule()` *set more fields* 41c⟩+≡ (38a) ◁40d 53b▷
```
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` 41d⟩≡ (174d)
```
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 42e⟩
    ⟨Node other fields 42b⟩

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

The function below constructs a new node:

⟨*constructor* `newnode` 42a⟩≡ (183a)
```
static Node*
newnode(char *name)
{
    Node *node;

    node = (Node *)Malloc(sizeof(Node));
    ⟨newnode() update node cache 84b⟩

    node->name = name;
    // call to timeof()!
    node->time = timeof(name, false);
    node->flags = 0;
    ⟨newnode() adjust flags of node 92b⟩

    node->arcs = nil;
    node->next = nil;
    ⟨newnode() debug 164i⟩
    return node;
}
```
Uses `Malloc()` 167a and `timeof()` 85a.

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

⟨`Node` *other fields* 42b⟩≡ (41d)
```
// bitset<enum<Node_flag>>
ushort  flags;
```

⟨*enum* `Node_flag` 42c⟩≡ (174d)
```
enum Node_flag {
    ⟨Node_flag cases 42d⟩
};
```

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* 42d⟩≡ (42c) 86a ▷
```
NOTMADE    = 0x0020,
BEINGMADE  = 0x0040,
MADE       = 0x0080,
```

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

3.4.2 Arc

A `Node`[41d] contains also a set of arcs where each arc contains a pointer to another node (a prerequisite):

⟨`Node` *arcs field* 42e⟩≡ (41d)
```
// list<ref_own<Arc>> (next = Arc.next)
Arc  *arcs;
```

⟨*struct* `Arc` 42f⟩≡ (174d)
```
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 43b⟩
```

```
    //Extra
    ⟨Arc extra fields 43a⟩
};
```
Uses `Node` 41d.

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* 43a⟩≡ (42f)
```
  // list<ref_own<arc> (head = Node.arcs)
  struct Arc *next;
```
Uses `Arc` 42f.

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* 43b⟩≡ (42f) 91a ▷
```
  // 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` 43c⟩≡ (183a)
```
  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 91b⟩
      return a;
  }
```
Uses `Malloc()` 167a.

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`[42d] 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⟩≡ (174d)
```
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 130f⟩

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

The list of target nodes of a job are chained together through an extra field in `Node`[41d]:

⟨`Node` *extra fields* 44b⟩≡ (41d)
```
// list<ref<Node>> (head = Job.n)
struct Node *next;  /* list for a rule */
```
Uses `Node` 41d.

The function below constructs a new job:

⟨*constructor* `newjob` 44c⟩≡ (184a)
```
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 146d⟩

    j->next = nil;
    return j;
}
```
Uses `Malloc()` 167a.

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

⟨*global* `jobs` 45a⟩≡ (176)

```
// 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.

Chapter 4

main()

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

4.1 main() skeleton

⟨*global* version 46a⟩≡ (185b)
```
static char *version = "@(#)mk general release 4 (plan 9)";
```
Uses version-10 46a.

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

⟨*function* main 46b⟩≡ (185b)
```
void
main(int argc, char **argv)
{
    ⟨main() locals 48b⟩
    Symtab* sym;

    // Initializing

    ⟨main() initializations 47b⟩

    // Parsing the mkfile

    ⟨main() parsing mkfile, call parse() 50b⟩

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

    ⟨main() initializations before building 112c⟩

    //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() 50d⟩

    // Reporting (optional)
```

```
    ⟨main() print profiling stats if uflag 128b⟩

    // Exiting

    exits(nil);
}
```
Uses `S_VAR` 31a and `symlook()` 31e.

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` 47a⟩≡ (176)
```
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* 47b⟩≡ (46b) 47c ▷
```
Binit(&bout, STDOUT, OWRITE);
```
Uses `bout` 47a.

`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* 47c⟩+≡ (46b) ◁ 47b
```
⟨main() argv processing part 1, -xxx 47d⟩
⟨main() setup profiling 127e⟩
inithash();
⟨main() argv processing part 2, xxx=yyy 48c⟩
⟨main() set variables for recursive mk 157d⟩
⟨main() argv processing part 3, skip xxx=yyy 49d⟩
⟨main() profile initializations 127f⟩
```
Uses `inithash()` 33d.

`inithash()`[33d] 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* 47d⟩≡ (47c)
```
USED(argc);
for(argv++; *argv && (**argv == '-'); argv++)
{
    ⟨main() add argv[0] in buf 157b⟩

    switch(argv[0][1]) {
    ⟨main() -xxx switch cases 50a⟩
    default:
        badusage();
    }
}
```
Uses `badusage()` 48a.

⟨*function* badusage 48a⟩≡ (185b)
```
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()`[53e] 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* 48b⟩≡ (46b) 49f▷
```
char *temp = nil;
fdt tfd = -1;
Biobuf tb;
int i;
```

⟨`main()` *argv processing part 2, xxx=yyy* 48c⟩≡ (47c)
```
for(i = 0; argv[i]; i++)
  if(utfrune(argv[i], '=')){
    ⟨main() add argv[i] in buf 157c⟩

    ⟨main() create temporary file if not exist yet and set tb 49a⟩
    Bprint(&tb, "%s\n", argv[i]);
    ⟨main() mark argv[i] for skipping 49e⟩
  }

if(tfd >= 0){
    Bflush(&tb);
    seek(tfd, 0L, SEEK__START);
    parse("<command line args>", tfd, true);
    remove(temp);
}
```
Uses `parse()` 53e.

`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

[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.

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()`[46b] passes `true` to `parse()` in the code above.

⟨`main()` *create temporary file if not exist yet and set* `tb` 49a⟩≡ (48c)
```
if(tfd < 0){
    temp = maketmp();
    ⟨main() when creating temporary file, sanity check temp 49b⟩
    tfd = create(temp, ORDWR, 0600);
    ⟨main() when creating temporary file, sanity check tfd 49c⟩
    Binit(&tb, tfd, OWRITE);
}
```

The code above omits the error management code shown below:

⟨`main()` *when creating temporary file, sanity check temp* 49b⟩≡ (49a)
```
if(temp == nil) {
    perror("temp file");
    Exit();
}
```

⟨`main()` *when creating temporary file, sanity check tfd* 49c⟩≡ (49a)
```
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* 49d⟩≡ (47c)
```
/* 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* 49e⟩≡ (48c)
```
/*
 *   assignment args become null strings
 */
*argv[i] = '\0';
```

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

4.5 Using the `mkfile` or `mk -f`*file*

I described before in Section 1.4 the use of `-f` to change the default file used by `mk`:

⟨`main()` *locals* 49f⟩+≡ (46b) ◁48b 50f▷
```
char *f = nil;
```

⟨main() *-xxx switch cases* 50a⟩≡ (47d) 51c ▷
```
case 'f':
    if(*++argv == nil)
        badusage();
    f = *argv;
    ⟨main() add argv[0] in buf 157b⟩
    break;
```
Uses badusage() 48a.

The parse()[53e] function, called below, will process the mkfile (or another file if -f was used) and modify rules[36c], metarules[37b], 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() 50b⟩≡ (46b)
```
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) 162a⟩
```
Uses MKFILE-9 50c and parse() 53e.

⟨*constant* MKFILE 50c⟩≡ (185b)
```
#define MKFILE  "mkfile"
```

4.6 Building the target(s)

Once parse()[53e] processed the mkfile and modified some globals, mk is ready to build a target by calling mk[94]. 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() 50d⟩≡ (46b)
```
if(*argv == nil){
    ⟨main() when no target arguments 51a⟩
} else {
    ⟨main() if sequential mode and target arguments given 51d⟩
    else {
       ⟨main() parallel mode and target arguments given 51e⟩
    }
}
```

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 50e⟩≡ (176)
```
Word *target1;
```

Section 5.2 contains the code in parse()[53e] 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()[94]:

⟨main() *locals* 50f⟩+≡ (46b) ◁ 49f 51b ▷
```
Word *w;
```

⟨main() *when no target arguments* 51a⟩≡ (50d)

```
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() 94 and target1 50e.

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* 51b⟩+≡ (46b) ◁50f 126d▷

```
bool sflag = false;
```

⟨main() *-xxx switch cases* 51c⟩+≡ (47d) ◁50a 125c▷

```
case 's':
    sflag = true;
    break;
```

⟨main() *if sequential mode and target arguments given* 51d⟩≡ (50d)

```
if(sflag){
    for(; *argv; argv++)
        if(**argv)
            mk(*argv);
}
```

Uses mk() 94.

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

⟨main() *parallel mode and target arguments given* 51e⟩≡ (50d)

```
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` 139c, `addrules()` 41a, `mk()` 94, `mkinline` 53c, and `newword()` 34c.

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

The `VIR`[139c] 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()`[46b], I can explain the different components in the building pipeline, starting in this chapter with the parsing functions.

I mentioned before `parse()`[53e], 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`[36c], `metarules`[37b], 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.

5.1 `parse()`

Before showing the code of `parse()`[53e], 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` 53a⟩≡ (176)
```
char *infile;
```

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

⟨`addrule()` *set more fields* 53b⟩+≡ (38a) ◁41c 87d▷
```
r->file = infile;
```
Uses `infile` 53a.

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

⟨*global* `mkinline` 53c⟩≡ (176)
```
int mkinline;
```

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

⟨*function* `SYNERR` 53d⟩≡ (174d)
```
#define SYNERR(l) (fprint(STDERR, "mk: %s:%d: syntax error; ", \
                            infile, ((l)>=0)? (l) : mkinline))
```

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

⟨*function* `parse` 53e⟩≡ (179c)
```
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 69b⟩

    ⟨parse() sanity check fd 54⟩
    ⟨parse() start, push 74c⟩

    // 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 55a⟩
        }
    }
    close(fd);
    freebuf(buf);
    ⟨parse() end, pop 74d⟩
}
```

Uses `assline()` 55b, `freebuf()` 168e, `infile` 53a, `mkinline` 53c, and `newbuf()` 168d.

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()`[55b]). 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()`[59b] 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`[36c] and `metarules`[37b], or the symbol table. I will describe in the next sections the cases of the `switch` above.

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

```
⟨parse() sanity check fd 54⟩≡                                              (53e)
  if(fd < 0){
```

```
        perror(f);
        Exit();
    }
```

⟨parse() *switch rhead cases* 55a⟩≡ (53e) 69c ▷

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

Uses SYNERR 53d.

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 55b⟩≡ (181a)

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

        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 57b⟩
            default:
                rinsert(buf, c);
                break;
            }
        }
    eof:
        insert(buf, '\0');
        return *(bufcontent(buf)) != '\0';
    }
```

Uses bufcontent 168h, insert() 169a, isempty 168f, nextrune() 56a, resetbuf 168g, and rinsert() 169b.

`insert()`[169a], `rinsert()`[169b] (for "Rune insert"), `resetbuf()`[168g], `bufcontent()`[168h], and `isempty()`[168f], called above, are all functions (or macros) operating on a string buffer and are described in Appendix C.2.

[1] This function has a confusing name.

The code of `assline()`[55b] 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()`[53e]. 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.

Escaped newline, `nextrune()`

`assline()`[55b] 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` 56a⟩≡ (181a)

```
/*
 * 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 56c⟩
        ⟨nextrune() handle mkinline 56b⟩
        return c;
    }
}
```

⟨`nextrune()` *handle mkinline* 56b⟩≡ (56a)

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

Uses `mkinline` 53c.

`nextrune()`[56a] 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* 56c⟩≡ (56a)

```
if (c == '\\') {
    if (Bgetrune(bp) == '\n') {
        // an escaped newline!
        mkinline++;
        if (elide)
            continue;
        // else
        return ' ';
    }
    // else, it was just \
    Bungetrune(bp);
}
```

Uses `mkinline` 53c.

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()`[55b] recognizes and skips comments:

⟨`assline()` *other locals* 57a⟩≡ (55b)
```
int prevc;
```

⟨`assline()` *switch character cases* 57b⟩≡ (55b) 58a ▷
```
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 57e⟩
   ⟨assline() when processing comments, if not only comment on the line 57c⟩
    // else, skip lines with only a comment
    break;
```
Uses `mkinline` 53c.

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* 57c⟩≡ (57b)
```
if (!isempty(buf)) {
     insert(buf, '\0');
     return true;
 }
```
Uses `insert()` 169a and `isempty` 168f.

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

⟨*example of escaped newline* 57d⟩≡
```
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* 57e⟩≡ (57b)
```
if (prevc == '\\')
    break;  /* propagate escaped newlines??*/
```

Quoted characters

Assembling a line sounds like an easy ask, but as you have just seen `assline()`[55b] is not trivial: it must handle escaped newlines (through `nextrune()`[56a]), 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* 58a⟩+≡ (55b) ◁57b 131g▷

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

Uses `escapetoken()` 58b and `rinsert()` 169b.

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` 58b⟩≡ (178a)

```
/*
 * 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 59a⟩
        }
        // else
        rinsert(buf, c);
    }
    // must have reached EOF
    SYNERR(line);
```

```
        fprint(STDERR, "missing closing %c\n", esc);
        return ERROR_0;
    }
```

Uses SYNERR 53d, mkinline 53c, nextrune() 56a, and rinsert() 169b.

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* 59a⟩≡ (58b)

```
    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()`[53e] 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` 59b⟩≡ (179c)

```
    static int
    rhead(char *line, Word **h, Word **t,    int *attr, char **prog)
    {
        char *p;
        int sep; // one of : = <
        ⟨rhead() other locals 72e⟩

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

        sep = *p;
        *p++ = '\0';
        ⟨rhead() adjust sep if dynamic mkfile <| 133h⟩
        ⟨rhead() adjust attr and prog 72d⟩

        // potentially expand variables in head
        *h = stow(line);
        ⟨rhead() sanity check h 60a⟩
        // potentially expand variables in tail
        *t = stow(p);

        return sep;
    }
```

`rhead()`[59b] relies on the function `charin()`[60c] 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

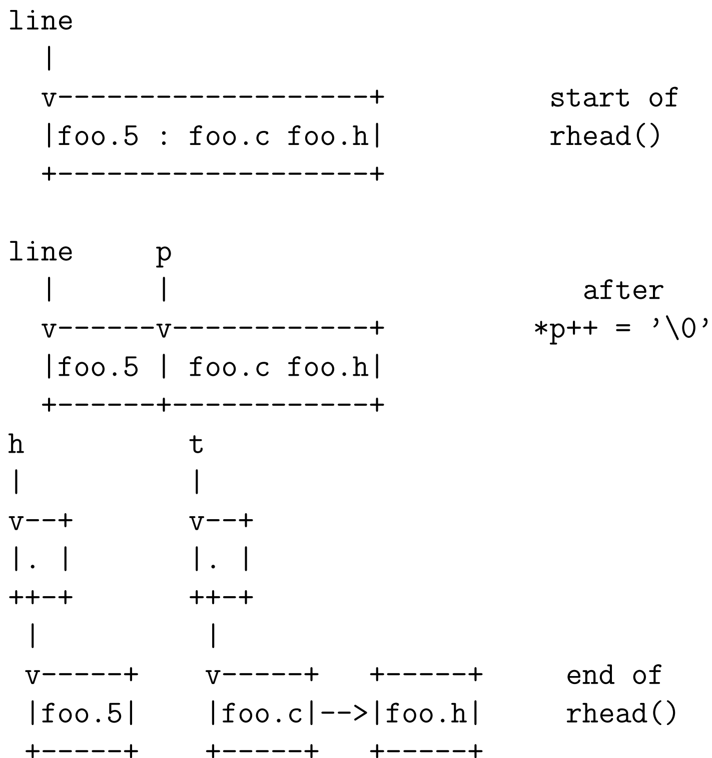


Figure 5.1: Evolution of local variables in `rhead()`.

processed by `stow()`[62b] (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* 60a⟩≡ (59b)

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

⟨*macro* `empty_words` 60b⟩≡ (174d)

```
#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()`[38a] called `charin()` to check whether the target contains a special pattern character. `rhead()`[59b] 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` 60c⟩≡ (178a)

```
/*
 * 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 61a⟩
        default:
            if(utfrune(pat, r) && !vargen)
                return cp;
            break;
        }
        cp += n;
    }
    ⟨charin() sanity check vargen 135b⟩
    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()`[56a] 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()`[55b] needs special code to handle quoted strings (because they may contain a `'#'` that needs to be treated differently), `charin()`[60c] needs also special code to handle quoted strings, because they may contain one of the special characters `rhead()`[59b] is looking for (`':'`, `'='`, or `'<'`).

⟨charin() *switch rune cases* 61a⟩≡ (60c) 135a ▷

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

Uses `squote()` 61b.

⟨*function* squote 61b⟩≡ (178a)

```
/*
 * 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 62a⟩
```

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

Uses SYNERR 53d.

⟨squote() *return, unless double quote* 62a⟩≡ (61b)

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

5.1.3 Splitting a string in words: stow()

stow()[62b] and nextword()[63b] 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()[59b] found the position of the special character in the line assembled by assline()[55b], 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 62b⟩≡ (184b)

```
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 63a⟩
    return head;
}
```

Uses nextword() 63b.

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* 63a⟩≡ (62b)
```
  if (!head)
      head = newword("");
```
Uses `newword()` 34c.

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

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` 63b⟩≡ (184b)
```
  /*
   * 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 65b⟩

      buf = newbuf();

  restart:
      head = lastw = nil;
      empty = true;
      ⟨nextword() skipping leading white space 64a⟩

      while(*cp){
          cp += chartorune(&r, cp);
          switch(r)
          {
          case ' ':
          case '\t':
          case '\n':
              goto out;
          ⟨nextword() switch rune cases 64b⟩
          default:
              empty = false;
              rinsert(buf, r);
              break;
          }
```

```
    }
out:
    *s = cp;
    if(!isempty(buf)){
        ⟨nextword() when buffer not empty, if there was already an head 69a⟩
         else {
            insert(buf, '\0');
            head = newword(buf->start);
        }
    }
    freebuf(buf);
    return head;
}
```

Uses freebuf() 168e, insert() 169a, isempty 168f, newbuf() 168d, newword() 34c, and rinsert() 169b.

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()`[63b].

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

⟨`nextword()` *skipping leading white space* 64a⟩≡ (63b)

```
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()`[63b] must handle quoted strings. As opposed to `assline()`[55b], which inputs a quoted string, or `charin()`[60c], which skips over a quoted string, `nextword()` expands a quoted string and stores the expansion in the buffer `b`. Indeed, `stow()`[62b] 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* 64b⟩≡ (63b) 65c ▷

```
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() 64c.

⟨*function* `expandquote` 64c⟩≡ (178a)

```
/*
 * 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 65a⟩
        }
        rinsert(buf, r);
    }
    return nil;
}
```

Uses rinsert() 169b.

⟨expandquote() *return, unless double quote* 65a⟩≡ (64c)

```
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()`[63b], 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* 65b⟩≡ (63b)

```
// list<ref_own<Word>
Word *w;
```

This is why `nextword()` returns a list of words, and why `stow()`[62b] 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()`[65d] 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* 65c⟩+≡ (63b) ◁64b

```
case '$':
    w = varsub(&cp);
    ⟨nextword() when in variable case, if w is nil 66b⟩
    empty = false;

    ⟨nextword() when in variable case, if non-space chars before var 67d⟩
    ⟨nextword() when in variable case, if head is not empty 67e⟩
    else
        head = lastw = w;

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

Uses varsub() 65d.

`varsub()` in turn relies on `varname()`[66d] to extract the name of the variable from the string pointed by `s` (`varname()` also increments `s` by side effect), and `varmatch()`[67c] to grab the value of the variable from the symbol table:

⟨*function* varsub 65d⟩≡ (184b)

```
Word*
varsub(char **s)
{
    Bufblock *buf;
```

```
    Word *w;

    ⟨varsub() if variable starts with open brace 135c⟩
    // else

    buf = varname(s);
    ⟨varsub() sanity check buf 66a⟩
    w = varmatch(buf->start);

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

Uses freebuf() 168e and varname() 66d.

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* 66a⟩≡ (65d)

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

⟨nextword() *when in variable case, if w is nil* 66b⟩≡ (65c)

```
if(w == nil){
   ⟨nextword() when in variable case, if w is nil and no char before 66c⟩
    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* 66c⟩≡ (66b)

```
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 66d⟩≡ (184b)

```
/*
 * 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 67b⟩
    *s = cp;
    insert(buf, '\0');
```

```
    return buf;
}
```

Uses `WORDCHR` 67a, `insert()` 169a, `newbuf()` 168d, and `rinsert()` 169b.

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

⟨*function* `WORDCHR` 67a⟩≡ (174d)

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

⟨`varname()` *sanity check buf* 67b⟩≡ (66d)

```
if (isempty(buf)){
    SYNERR(-1);
    fprint(STDERR, "missing variable name <%s>\n", *s);
    freebuf(buf);
    return nil;
}
```

Uses `SYNERR` 53d, `freebuf()` 168e, and `isempty` 168f.

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

⟨*function* `varmatch` 67c⟩≡ (184b)

```
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* 67d⟩≡ (65c)

```
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()` 169c, `insert()` 169a, `isempty` 168f, and `resetbuf` 168g.

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* 67e⟩≡ (65c)

```
if(head){
    // merge the last and first words
    bufcpy(buf, lastw->s, strlen(lastw->s));
```

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



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

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

                (before)          (after)

                   adjusting the first
                          word
```

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

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

Uses `bufcpy()` 169c, `insert()` 169a, and `resetbuf` 168g.

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* 69a⟩≡ (63b)

```
    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()` 169c and `insert()` 169a.

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()`[53e] to process the rules, definitions, and inclusions. Those actions are based on the information returned by `rhead()`[59b]: 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* 69b⟩≡ (53e) 72c ▷

```
    char *body;
```

⟨`parse()` *switch rhead cases* 69c⟩+≡ (53e) ◁ 55a 73c ▷

```
    case ':':
        body = rbody(&in);
        addrules(head, tail, body,   attr, hline, prog);
        break;
```

Uses `addrules()` 41a and `rbody()` 71a.

The action above relies on `rbody()`[71a] to read the body of a rule, that is its recipe. I will explain `rbody()` in the next section.

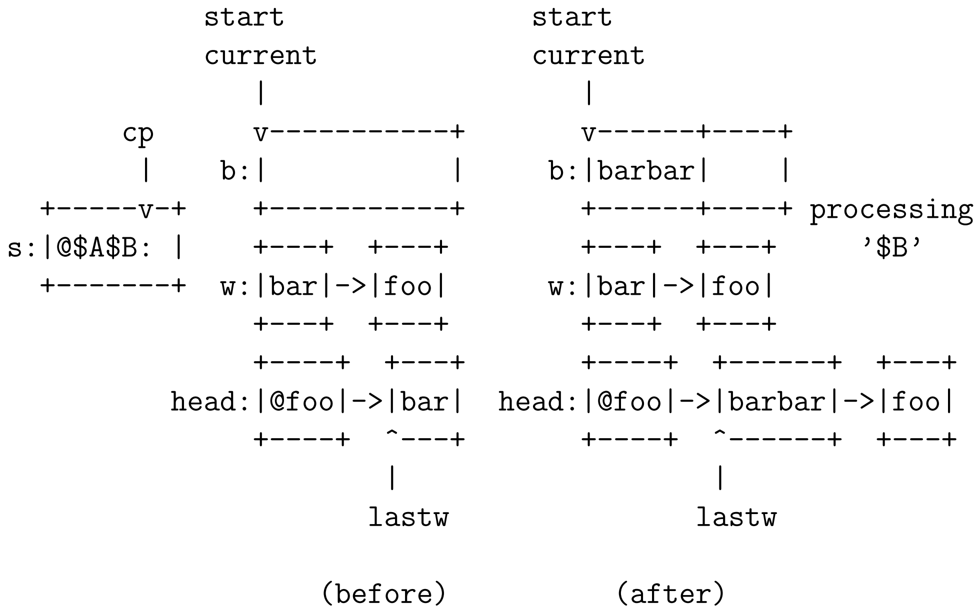
I have described `addrules()`[41a] called above in Section 3.3.4. `addrules()` can handle rules with multiple targets by calling `addrule()`[38a] for each target.

I mentioned also in Section 4.6.1 that `mk`, during parsing, sets the global `target1`[50e] to contain the first target found in the mkfile. Here is finally the code that sets `target1`:

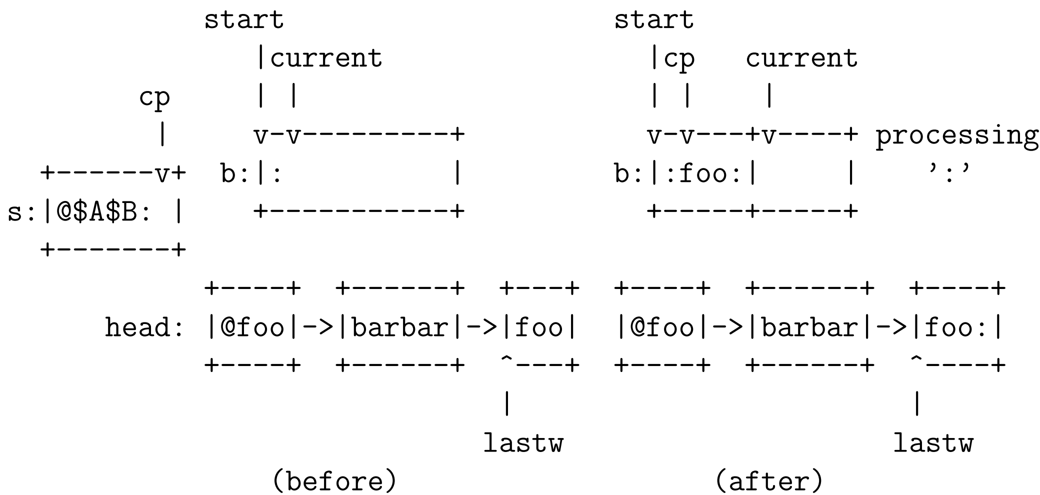
⟨`addrules()` *set* `target1` 69d⟩≡ (41a)

```
    /* 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, "%&"))
```

where A=foo bar
B=bar foo



merging the last and
first word



adjusting the last
word

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

```
            break;
        if(w == nil) // head does not contain any pattern
            target1 = wdup(targets);
    }
```

Uses REGEXP 129a, charin() 60c, target1 50e, and wdup() 35a.

Note the use of charin()[60c] 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[129a] used above, see Section 11.1.

5.2.1 Parsing the recipe: rbody()

rbody(), like assline()[55b], 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 71a⟩≡ (179c)

```
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 72a⟩
            else
              if (r != ' ' && r != '\t') {
                Bungetrune(in);
                break;
            }
        } else
            // not in first column
            rinsert(buf, r);

        lastr = r;
        ⟨rbody() handle mkinline 71b⟩
    }

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

    return p;
}
```

Uses freebuf() 168e, insert() 169a, newbuf() 168d, and rinsert() 169b.

⟨rbody() *handle mkinline* 71b⟩≡ (71a)

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

Uses `mkinline` 53c.

If `rbody()`[71a] 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* 72a⟩≡ (71a)
```
  if (r == '#')
      rinsert(buf, r); // the shell recognize comments too
```
Uses `rinsert()` 169b.

If a rule has no recipe, `rbody()` returns the empty string to its caller `parse()`[53e]. 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* 72b⟩≡
```
  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`[37e], but before they are stored in a local variable in `parse()`[53e]:

⟨`parse()` *other locals* 72c⟩+≡ (53e) ◁69b 73b▷
```
  // bitset<Rule_attr>
  int attr;
```

This variable is passed by address to `rhead()`[59b] (see the call to `rhead()` in `parse()`). `rhead()` then initializes this variable and adjusts it depending on the separator:

⟨`rhead()` *adjust* `attr` *and* `prog` 72d⟩≡ (59b)
```
  *attr = 0; // Nothing
  *prog = nil;
  // variable attributes
  ⟨rhead() if sep is = 77a⟩
  // rule attributes
  ⟨rhead() if sep is : 73a⟩
```

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

⟨`rhead()` *other locals* 72e⟩≡ (59b) 142i▷
```
  Rune r;
  int n;
```

⟨rhead() *if sep is* : 73a⟩≡ (72d)

```
  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 129b⟩
          //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()`[59b] is `'<'`.

A file inclusion in an `mkfile` will result in the opening of a new file, hence the following additional variables in `parse()`[53e]:

⟨parse() *other locals* 73b⟩+≡ (53e) ◁72c 75b▷

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

Here is the code using `newfd`:

⟨parse() *switch rhead cases* 73c⟩+≡ (53e) ◁69c 75c▷

```
  case '<':
      p = wtos(tail, ' ');
      ⟨parse() when parsing included file, sanity check p 73d⟩
      newfd = open(p, OREAD);
      ⟨parse() when parsing included file, sanity check newfd 74a⟩
      else
          // recurse
          parse(p, newfd, false);
      break;
```

Uses `parse()` 53e and `wtos()` 35b.

⟨parse() *when parsing included file, sanity check p* 73d⟩≡ (73c)

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

Uses `SYNERR` 53d.

⟨parse() *when parsing included file, sanity check newfd* 74a⟩≡ (73c)
```
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()[62b], which performs quote and variable expansions. Thus, you can also use variables in the filename to include, as in

⟨*tests/mkfile/mkincludearc* 74b⟩≡
```
</$objtype/mkfile
```

I described wtos()[35b] 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[53a] and mkinline[53c] must be saved before the call and restored after, hence the following calls in parse():

⟨parse() *start, push* 74c⟩≡ (53e)
```
ipush();
```
Uses ipush() 74h.

⟨parse() *end, pop* 74d⟩≡ (53e)
```
ipop();
```
Uses ipop() 75a.

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

⟨*struct* input 74e⟩≡ (179c)
```
struct Input
{
    char *file;
    int line;

    // Extra
    ⟨Input extra fields 74g⟩
};
```

⟨*global* inputs 74f⟩≡ (179c)
```
// list<ref_own<Input>> (next = Input.next)
static struct Input *inputs = nil;
```
Uses Input 74e and inputs-14 74f.

⟨Input *extra fields* 74g⟩≡ (74e)
```
// list<ref_own<Input>> (head = inputs)
struct Input *next;
```
Uses Input 74e.

⟨*function* ipush 74h⟩≡ (179c)
```
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 74e, Malloc() 167a, infile 53a, inputs-14 74f, and mkinline 53c.

⟨*function* ipop 75a⟩≡ (179c)

```
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 74e, infile 53a, inputs-14 74f, and mkinline 53c.

5.4 Parsing Variable definitions: *var=values*

The final action of parse()[53e] manages variable definitions, when the special character returned by rhead()[59b] is '='.

⟨parse() *other locals* 75b⟩+≡ (53e) ◁73b 134b▷

```
bool set = true;
```

⟨parse() *switch rhead cases* 75c⟩+≡ (53e) ◁73c 134c▷

```
case '=':
    ⟨parse() when parsing variable definitions, sanity check head 76a⟩
    ⟨parse() when parsing variable definitions, override handling 76c⟩
    if(set){
        setvar(head->s, (void *) tail);
        ⟨parse() when parsing variable definitions, extra setting 124c⟩
    }
    ⟨parse() when parsing variable definitions, if variable with attr 145f⟩
    break;
```

Uses setvar() 33a.

The code above relies mainly on the function `setvar()`[33a] 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* 76a⟩≡ (75c)
```
if(head->next){
    SYNERR(-1);
    fprint(STDERR, "multiple vars on left side of assignment\n");
    Exit();
}
```
Uses `SYNERR` 53d.

Note also that both `head` and `tail` used above are the results of calls to `stow()`[62b] 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* 76b⟩≡
```
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()`[53e] 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* 76c⟩≡ (75c)
```
if(symlook(head->s, S_OVERRIDE, nil)){
    set = varoverride;
} else {
    set = true;
    if(varoverride)
        symlook(head->s, S_OVERRIDE, (void *)"");
}
```
Uses `S_OVERRIDE` 76d and `symlook()` 31e.

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* 76d⟩+≡ (31a) ◁39a 84a▷
```
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* 76e⟩≡
```
MYVAR=U= foo.5 bar.5 # a private variable
```

Here is the code in rhead()[59b] to extract variable attributes:

⟨rhead() *if sep is* = 77a⟩≡ (72d)
```
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 145e⟩
            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 77b⟩≡ (178a)
```
char *termchars = "=' \t"; /*used in parse.c to isolate assignment attribute*/
```
Uses termchars 77b.

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* 77c⟩≡
```
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()`[78], 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`[36c] and `metarules`[37b] populated by `parse()`[53e] 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()`[79a], which is the function containing the logic to build the graph of dependencies.

⟨*function* `graph` 78⟩≡ (183a)
```
Node*
graph(char *target)
{
    Node *root;
    ⟨graph() other locals 87e⟩

    ⟨graph() set cnt for infinite rule detection 88a⟩
    root = applyrules(target, cnt);
    ⟨graph() free cnt 88c⟩

    ⟨graph() checking the graph 85d⟩
    ⟨graph() propagate attributes 139a⟩

    return root;
}
```
Uses `applyrules()` 79a.

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()`[42a]), 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` 79a⟩≡ (183a)
```
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 79b⟩

    ⟨applyrules debug 164h⟩
    ⟨applyrules check node cache if target is already there 84c⟩
    // else

    target = strdup(target);
    node = newnode(target); // calls timeof() internally
    head.next = nil;
    ⟨applyrules other initializations 131d⟩

    // apply regular rules with target as a head (modifies lasta)
    ⟨applyrules() apply regular rules 79c⟩

    // apply meta rules (modifies lasta)
    ⟨applyrules() apply meta rules 80e⟩

    node->arcs = head.next;

    return node;
}
```
Uses `newnode()` 42a.

The code above relies on a local variable `head` containing an `Arc`[42f] 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()`[38a] with the list of rules, or in Section 4.6.3 with a list of words).

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

6.2 Finding the simple rule(s) for a target

The first step to find the dependencies of a node is to look for all the rules mentioning the node as a target. Fortunately, `applyrules()`[79a] can rely on the symbol table and the `S_TARGET`[39a] namespace to quickly access all the rules mentioning a specific target (as explained in Section 3.3.4):

⟨`applyrules` *other locals* 79b⟩≡ (79a) 81a ▷
```
Symtab *sym;
Rule *r;
Word *pre;
Arc *arc;
```

⟨`applyrules()` *apply regular rules* 79c⟩≡ (79a)
```
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 80a⟩
```

```
    ⟨applyrules() infinite rule detection part1 88e⟩
    ⟨applyrules() when found a regular rule for target node, set flags 92c⟩

    ⟨applyrules() if no prerequisites in rule r 80d⟩
    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 88f⟩
}
```

Uses S_TARGET 39a, applyrules() 79a, newarc() 43c, and symlook() 31e.

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* 80a⟩≡ (79c)

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

Uses empty_prereqs 80c and empty_recipe 80b.

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

⟨*macro* empty_recipe 80b⟩≡ (174d)

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

⟨*macro* empty_prereqs 80c⟩≡ (174d)

```
#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* 80d⟩≡ (79c)

```
// 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 80c and newarc() 43c.

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()`[79a] needs to go through all the meta rules stored in the global `metarules`[37b]:

⟨applyrules() *apply meta rules* 80e⟩≡ (79a)

```
for(r = metarules; r; r = r->next){
    ⟨applyrules() skip this meta rule and continue if some conditions 81d⟩
    ⟨applyrules() if regexp rule then continue if some conditions 131e⟩
    else {
```

```
            if(match(node->name, r->target, stem)) {
                ⟨applyrules() infinite rule detection part1 88e⟩

                ⟨applyrules() if no prerequisites in meta rule r 81e⟩
                else
                    for(pre = r->prereqs; pre; pre = pre->next) {
                        ⟨applyrules() if regexp rule, adjust buf and rmatch 131f⟩
                        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 88f⟩
            }
        }
    }
```

Uses `applyrules()` 79a, `metarules` 37b, `newarc()` 43c, and `subst()` 82c.

The code above relies on the functions `match()`[82a] and `subst()`[82c], 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* 81a⟩+≡ (79a) ◁79b 81c▷

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

Uses `NAMEBLOCK` 81b.

⟨*constant* `NAMEBLOCK` 81b⟩≡ (174d)

```
  #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* 81c⟩+≡ (79a) ◁81a 131c▷

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

Uses `NAMEBLOCK` 81b.

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

⟨`applyrules()` *skip this meta rule and continue if some conditions* 81d⟩≡ (80e) 142c▷

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

Uses `empty_prereqs` 80c and `empty_recipe` 80b.

Some meta rules can also contain virtual targets:

⟨`applyrules()` *if no prerequisites in meta rule r* 81e⟩≡ (80e)

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

Uses `empty_prereqs` 80c and `newarc()` 43c.

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

6.3.1 Matching a pattern: `match()`

`match()` below iterates over two string parameters (`name` and `template`) at the same time by incrementing both pointers until it finds the special pattern character ('%' or '&') in the `template` parameter. Once it found the pattern character, it computes the length of the string matched by the pattern character and it makes sure the rest of the template also matches the end of the name. Finally it modifies the `stem` buffer passed from `applyrules()`[79a]. Figure 6.1 illustrates how `match()` works on a simple example.

⟨*function* `match` 82a⟩≡ (179a)
```
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 82b⟩

    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* 82b⟩≡ (82a)
```
if(*template == '&')
    return !charin(stem, "./");
```

6.3.2 Substituting the stem: `subst()`

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

⟨*function* `subst` 82c⟩≡ (179a)
```
void
subst(char *stem, char *template,   char *dest, int dlen)
{
```

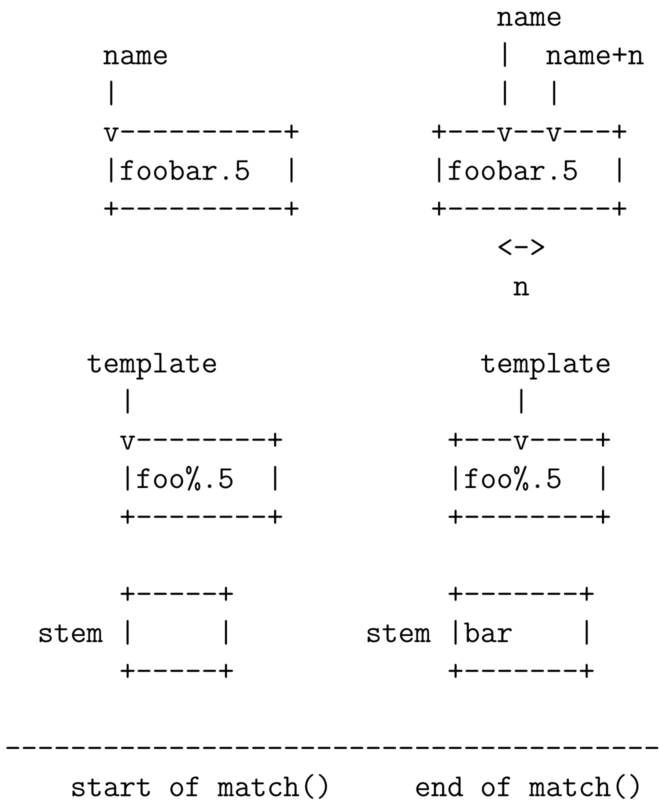


Figure 6.1: `match("foobar.5", "foo\%.5", ...)`.

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

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

Uses `PERCENT` 37d.

6.4 Node cache

The node cache ensures that each file has exactly one `Node`[41d] 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

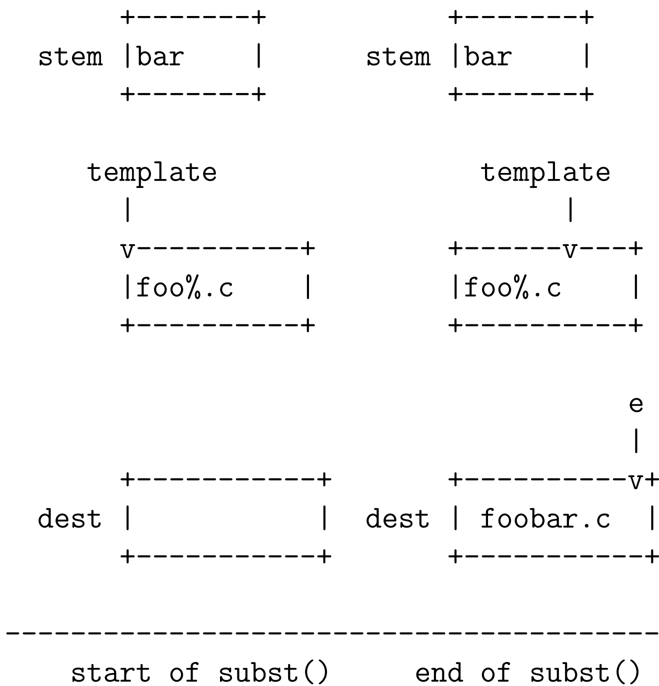


Figure 6.2: `subst("bar", "foo\%.c", ...)`.

to redundant recipe executions and incorrect modification-time tracking. The cache also turns the graph from a tree into a proper DAG: when `applyrules()`[79a] 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* 84a⟩+≡ (31a) ◁76d 124b▷
```
S_NODE,  /* target name -> node */
```

Each time `newnode()`[42a] 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* 84b⟩≡ (42a)
```
symlook(name, S_NODE, (void *)node);
```
Uses `S_NODE` 84a and `symlook()` 31e.

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* 84c⟩≡ (79a)
```
sym = symlook(target, S_NODE, nil);
if(sym)
    return sym->u.ptr;
```
Uses `S_NODE` 84a and `symlook()` 31e.

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()`[42a] 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` 85a⟩≡ (181b)
```
ulong
timeof(char *name, bool force)
{
    ⟨timeof() locals 154d⟩

    ⟨timeof() if name archive member 148d⟩
    if(force)
        return mkmtime(name, true);
    // else
    ⟨timeof() if not force, use time cache 154e⟩
}
```

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

⟨*function* `mkmtime` 85b⟩≡ (191)
```
ulong
mkmtime(char *name, bool force)
{
    Dir *d;
    ulong t;
    ⟨mkmtime locals 155a⟩

    ⟨mkmtime() bulk dir optimisation 155b⟩
    d = dirstat(name);
    ⟨mkmtime() check if inexistent file 85c⟩
    t = d->mtime;
    free(d);

    return t;
}
```

Note that if the file does not exist, `mkmtime()`[185c] and `timeof()`[85a] return `0`:

⟨`mkmtime()` *check if inexistent file* 85c⟩≡ (85b)
```
if(d == nil)
    return 0;
```

6.6 Checking the graph and the rules

Once `applyrules()`[79a] returned the graph of dependencies, `graph()`[78] can check for special properties of the graph that are signs of mistakes in the `mkfile`:

⟨`graph()` *checking the graph* 85d⟩≡ (78)
```
cyclechk(root);

⟨graph() before ambiguous() 92d⟩
ambiguous(root);
```
Uses `ambiguous()` 89 and `cyclechk()` 86b.

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

6.6.1 Cycle detection

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

⟨*tests/mk/mkfile-with-cycle* 85e⟩≡

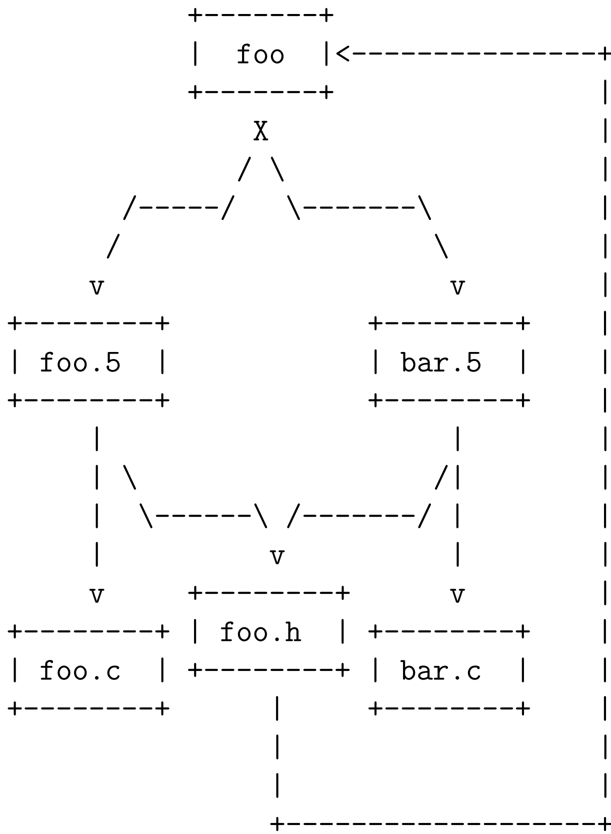


Figure 6.3: A cycle in a graph of dependencies.

```
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* 86a⟩+≡ (42c) ◁42d 92a▷
```
  CYCLE      = 0x0002,
```

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

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

⟨*function* `cyclechk` 86b⟩≡ (183a)

```
static void
cyclechk(Node *n)
{
    Arc *a;

    if((n->flags&CYCLE)){
        fprint(STDERR, "mk: cycle in graph detected at target %s\n", n->name);
        Exit();
    }
    n->flags |= CYCLE;
    for(a = n->arcs; a; a = a->next)
        if(a->n)
            cyclechk(a->n);
    n->flags &= ~CYCLE;
}
```
Uses `CYCLE` 86a and `cyclechk()` 86b.

Note that it is important for `cyclechk()`[86b] to remove the `CYCLE` flag at the very end before returning. Indeed, the graph of dependencies can be a DAG, in which case the same node may be referenced from multiple parents (e.g., `foo.h` in Figure 6.3). Without the last instruction in `cyclechk()`, `mk` would report another (this time wrong) cycle when visiting `foo.h` for the second time from `bar.5` instead of `foo.5`.

6.6.2 Infinite rule detection

The second mistake `mk` can detect is the presence of meta rules that `mk` could instantiate infinitely. Here is an example of an `mkfile` illustrating the issue:

⟨*tests/mk/mkfile-infinite* 87a⟩≡
```
all: foo bar

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

When building the graph for `foo`, `applyrules()`[79a] 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* 87b⟩+≡ (36a) ◁41b 129c▷
```
int   rule;  /* rule number */
```

⟨*global* `nrules` 87c⟩≡ (180d)
```
static int nrules = 0;
```
Uses `nrules-25` 87c.

⟨`addrule()` *set more fields* 87d⟩+≡ (38a) ◁53b 142f▷
```
r->rule = nrules++;
```
Uses `nrules-25` 87c.

Then, before calling `applyrules()`, `graph()`[78] initializes a map that will count how many times `applyrules()` used a rule:

⟨`graph()` *other locals* 87e⟩≡ (78)
```
// map<ruleid, int>
char *cnt;
```

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

⟨graph() *set cnt for infinite rule detection* 88a⟩≡ (78)
```
cnt = rulecnt();
```
Uses rulecnt() 88b.

⟨*function* rulecnt 88b⟩≡ (180d)
```
char*
rulecnt(void)
{
    char *s;

    s = Malloc(nrules);
    memset(s, 0, nrules);
    return s;
}
```
Uses Malloc() 167a and nrules-25 87c.

⟨graph() *free cnt* 88c⟩≡ (78)
```
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 88d⟩≡ (183a)
```
int nreps = 1;
```
Uses nreps 88d.

⟨applyrules() *infinite rule detection part1* 88e⟩≡ (80e 79c)
```
if(cnt[r->rule] >= nreps)
    continue;

cnt[r->rule]++;
```
Uses nreps 88d.

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* 88f⟩≡ (80e 79c)
```
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* 88g⟩≡
```
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` 89⟩≡ (183a)

```
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 90d⟩
            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 91c⟩
}
```

Uses `ambiguous()` 89, `empty_recipe` 80b, and `trace()` 90a.

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

⟨*function* `trace` 90a⟩≡ (183a)
```
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 90b⟩
        else
            a = nil;
    }
    fprint(STDERR, "\n");
}
```

⟨`trace()` *possibly continue if prereq is also a target* 90b⟩≡ (90a)
```
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* 90c⟩≡
```
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* 90d⟩≡ (89)
```
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` 37f.

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* 91a⟩+≡ (42f) ◁43b 130c▷

```
short remove;
```

⟨`newarc()` *set other fields* 91b⟩≡ (43c) 130d▷

```
a->remove = false;
```

Adjusting the graph, `togo()`

`ambiguous()`[89], before returning, removes all the arcs marked as to-be-removed on the current node:

⟨`ambiguous()` *get rid of all skipped arcs* 91c⟩≡ (89)

```
togo(n);
```

Uses `togo()` 91d.

⟨*function* `togo` 91d⟩≡ (183a)

```
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* 91e⟩≡

```
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* 92a⟩+≡ (42c) ◁86a 92f▷
```
  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()`[42a] (remember that `timeof()`[85a] returns `0` for inexistent files):

⟨`newnode()` *adjust flags of node* 92b⟩≡ (42a)
```
  if(node->time)
      node->flags = PROBABLE;
```
Uses `PROBABLE` 92a.

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* 92c⟩≡ (79c)
```
  node->flags |= PROBABLE;
```
Uses `PROBABLE` 92a.

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()` 92d⟩≡ (85d) 92e▷
```
  root->flags |= PROBABLE; /* make sure it doesn't get deleted */
```
Uses `PROBABLE` 92a.

Once `mk` initialized the nodes in the graph of dependencies, `graph()`[78] calls `vacuous()`[92h] to explore the graph to detect and remove vacuous nodes.

⟨`graph()` *before* `ambiguous()` 92e⟩+≡ (85d) ◁92d
```
  vacuous(root);
```
Uses `vacuous()` 92h.

Note that `graph()` must call `vacuous()` before `ambiguous()`[89], to remove the vacuous nodes and arcs, otherwise `ambiguous()` would report ambiguities.

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

⟨`Node_flag` *cases* 92f⟩+≡ (42c) ◁92a 92g▷
```
  VACUOUS     = 0x0200,
```

⟨`Node_flag` *cases* 92g⟩+≡ (42c) ◁92f 139e▷
```
  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` 92h⟩≡ (183a)
```
  static bool
  vacuous(Node *node)
  {
      Arc *a;
      bool vac = !(node->flags&PROBABLE);
      ⟨vacuous() other locals 93a⟩
```

```
    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 93b⟩

    togo(node);
    if(vac) {
        node->flags |= VACUOUS;
    }
    return vac;
}
```

Uses `META` 37f, `PROBABLE` 92a, `READY` 92g, `VACUOUS` 92f, `togo()` 91d, and `vacuous()` 92h.

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* 93a⟩≡ (92h)

```
Arc *a2;
```

⟨`vacuous` *possibly undelete some arcs* 93b⟩≡ (92h)

```
/* 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()`[94], takes a target as a parameter, builds the graph of dependencies for this target (with `graph()`[78]), explores the graph to find outdated files, with a function called `work()`[97a], and schedules recipes to be executed to update those outdated files, with a function called `dorecipe()`[100a]. The following sections will explain the code of `mk()`, `work()`, and `dorecipe()`.

7.1 mk()

`mk()`[94] 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`[42d], `BEINGMADE`[42d], and `MADE`[42d]) 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()`[97a] 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()`[109a] 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 94⟩≡ (183b)

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

    ⟨mk() initializations 104a⟩

    root = graph(target);
    ⟨mk() if DEBUG(D_GRAPH) 163a⟩
```

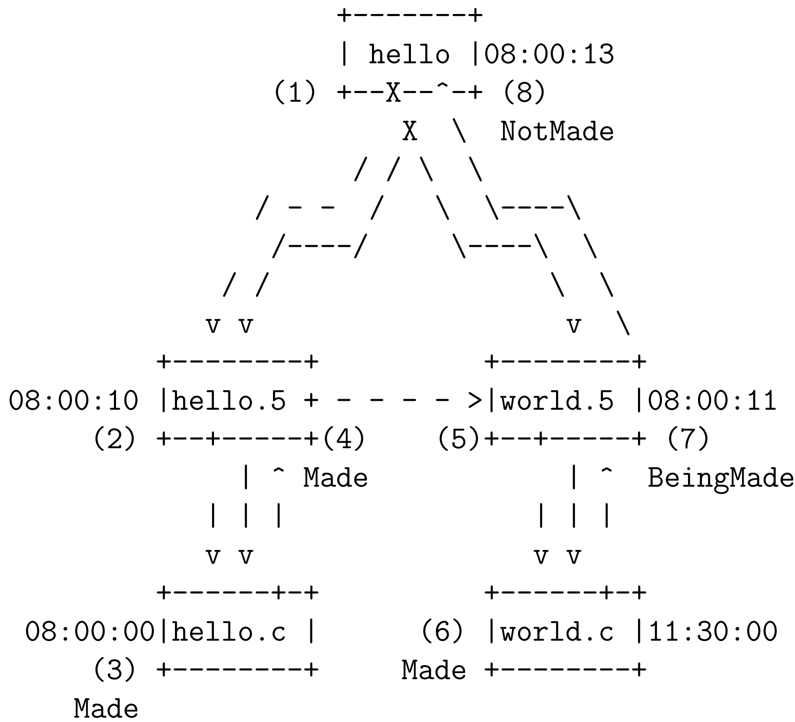


Figure 7.1: Graph of dependencies with building-status labels.

```
    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 158f⟩
        }
    }
    if(root->flags&BEINGMADE)
        waitup(EMPTY_CHILDREN_IS_ERROR, (int *)nil);

    ⟨mk() before returning, more waitup() if there was an error 158h⟩
    if(!everdid)
        Bprint(&bout, "mk: '%s' is up to date\n", root->name);
    return;
}
```

Uses `BEINGMADE` 42d, `EMPTY_CHILDREN_IS_ERROR` 111a, `EMPTY_CHILDREN_IS_OK` 111a, `NOTMADE` 42d, `bout` 47a, `clrmade()` 96a, `graph()` 78, `waitup()` 109a, and `work()` 97a.

`mk()` operates in six steps:

1. `mk()` builds the graph of dependencies with `graph()`[78] (see Chapter 6).
2. `mk()` sets the building status of all nodes in the graph to `NOTMADE` with `clrmade()`[96a] (see Section 7.2).
3. `mk()` explores the graph of dependencies repeatedly with `work()` in successive waves to find outdated files (see Section 7.3). When `work()` finds up-to-date files, it marks them as `MADE`, (e.g., `hello.c` in Figure 7.1). When it finds outdated files, `work()` schedules a recipe to be executed with `dorecipe()`[100a], 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[111a] 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.

7.2 Initializing nodes: clrmade()

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

⟨*function* clrmade 96a⟩≡ (183b)
```
void
clrmade(Node *n)
{
    Arc *a;

    ⟨clrmade() n->flags pretend adjustments 152a⟩
    MADESET(n, NOTMADE);
    for(a = n->arcs; a; a = a->next)
        if(a->n)
            // recurse
            clrmade(a->n);
}
```
Uses MADESET 96b, NOTMADE 42d, and clrmade() 96a.

Note that Node.flags[42b] 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 96b⟩≡ (174d)
```
#define MADESET(n,m) n->flags = (n->flags&~(NOTMADE|BEINGMADE|MADE))|(m)
```

7.3 Exploring the graph: `work()`

`work()`[97a] 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`[42d]?). A node can be outdated but not ready—for example, if one prerequisite is still `BEINGMADE`[42d] 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`[42d] nodes:

⟨*function* `work` 97a⟩≡ (183b)
```
void
work(Node *node, bool *did,   Node *parent_node, Arc *parent_arc)
{
    ⟨work() locals 97c⟩

    ⟨work() debug 165a⟩
    if(node->flags&BEINGMADE)
        return;
    ⟨work() possibly unpretending node 152c⟩

    if(node->arcs == nil){
        ⟨work() no arcs, a leaf 97b⟩
    } else {
        ⟨work() some arcs, a node 98c⟩
    }
}
```
Uses `BEINGMADE` 42d.

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

7.3.1 The leaf case

The leaf case is easy. 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`[42d].

⟨`work()` *no arcs, a leaf* 97b⟩≡ (97a)
```
/* consider no prerequisite case */
if(node->time == 0){
    ⟨work() print error when inexistent file without prerequisites 97d⟩
} else
    MADESET(node, MADE);
```
Uses `MADE` 42d and `MADESET` 96b.

Otherwise, `mk` should report an error:

⟨`work()` *locals* 97c⟩≡ (97a) 98a ▷
```
char cwd[256];
```

⟨`work()` *print error when inexistent file without prerequisites* 97d⟩≡ (97b)
```
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 158c⟩
else
    Exit();
```

7.3.2 The parent case

The parent case is more complex. For the parent case, `work()`[97a] relies on the two important booleans below:

⟨`work()` *locals* 98a⟩+≡ (97a) ◁97c 98b▷

```
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`[42d].

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

⟨`work()` *locals* 98b⟩+≡ (97a) ◁98a 153b▷

```
Arc *a;
```

⟨`work()` *some arcs, a node* 98c⟩≡ (97a)

```
⟨work() adjust weoutofdate if aflag 156d⟩
/*
 *   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 153c⟩
        }
    } else {
        if(node->time == 0){
            weoutofdate = true;
            ⟨work() update ra when no dest in arc and no src 153d⟩
        }
    }
}

if(!ready) /* can't do anything now */
    return;
if(!weoutofdate){
    MADESET(node, MADE);
    return;
}
⟨work() possibly pretending node 152b⟩
// else, out of date

dorecipe(node, did);
return;
```

Uses `BEINGMADE` 42d, `MADE` 42d, `MADESET` 96b, `NOTMADE` 42d, `dorecipe()` 100a, `outofdate()` 99a, and `work()` 97a.

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` 99a⟩≡ (183b)
```
bool
outofdate(Node *node, Arc *arc, bool eval)
{
    ⟨outofdate() locals 142m⟩

    ⟨outofdate() if arc-¿prog 143a⟩
    else
     ⟨outofdate() if arc node is an archive member 151b⟩
     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.

7.4 Scheduling recipes: `dorecipe()`

The job of `dorecipe()`[100a], called from `work()`[97a], is to construct a `Job`[44a] to run a recipe that will update a target `Node`[41d]. 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* 99b⟩≡ (100a) 99c ▷
```
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* 99c⟩+≡ (100a) ◁ 99b 99d ▷
```
// 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* 99d⟩+≡ (100a) ◁ 99c 99e ▷
```
// 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* 99e⟩+≡ (100a) ◁ 99d 100b ▷
```
// list<string> (last = last_allprereqs)
Word allprereqs;
```

This set will be used when exporting the special variable `$prereqs` in `buildenv()`[119b].

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

⟨*function* `dorecipe` 100a⟩≡ (184a)
```
void
dorecipe(Node *node, bool *did)
{
    // iterators
    Arc *a;
    Node *n;
    Word *w;
    Symtab *s;
    ⟨dorecipe() other locals 99b⟩

    /*
     *   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 100c⟩
    // else

    /*
     *   build the node list
     */
    ⟨dorecipe() build lists of targets and node list 101b⟩

    /*
     *   gather the params for the job
     */
    allprereqs.next = newprereqs.next = nil;
    for(n = nlist; n; n = n->next){
        ⟨dorecipe() build lists of prerequisites 102b⟩
        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` 42d, `MADESET` 96b, `empty_recipe` 80b, `newjob()` 44c, and `run()` 103a.

I described `newjob()`[44c] called above before. I will explain `run()`[103a], 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* 100b⟩+≡ (100a) ◁99e 101a▷
```
char cwd[256];
```

⟨`dorecipe()` *if no recipe found* 100c⟩≡ (100a)
```
/*
```

```
 *   no recipe? go to buggery!
 */
if(master_rule == nil){
    ⟨dorecipe() when no recipe found, if virtual or norecipe node 140a⟩
    else {
        if(getwd(cwd, sizeof cwd))
            fprint(STDERR, "mk: no recipe to make '%s' in directory %s\n",
                   node->name, cwd);
        else
            fprint(STDERR, "mk: no recipe to make '%s'\n", node->name);
        Exit();
    }
}
```

7.4.1 Building the list of target nodes

To build the list of all targets, `dorecipe()`[100a] can rely on the `Rule.alltargets`[41b] field setup in `addrule()`[38a] (see Section 3.3.4):

⟨dorecipe() *other locals* 101a⟩+≡ (100a) ◁100b 145i▷

```
Word *last_alltargets = &alltargets;
char buf[BIGBLOCK];
```

Uses BIGBLOCK 174a.

⟨dorecipe() *build lists of targets and node list* 101b⟩≡ (100a)

```
nlist->next = nil;
alltargets.next = oldtargets.next = nil;
⟨dorecipe() if regexp rule 129f⟩
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 102a⟩
        n = s->u.ptr;

        ⟨dorecipe() update list of outdated targets 146a⟩

        // add_set(n, nlist)
        if(n == node)
            continue;
        n->next = nlist->next;
        nlist->next = n;
    }
}
```

Uses META 37f, S_NODE 84a, newword() 34c, subst() 82c, and symlook() 31e.

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`[84a] 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()`[109a] to update the modification time and building status of those nodes in the graph.

⟨`dorecipe()` *sanity check s* 102a⟩≡ (101b)

```
if(s == nil)
     continue; /* not a node we are interested in */
```

7.4.2 Building the list of prerequisites

To build the list of prerequisites, `dorecipe()`[100a] 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()`[34e], which treats the list of words as a set and avoids adding duplicates:

⟨`dorecipe()` *build lists of prerequisites* 102b⟩≡ (100a)

```
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 146g⟩
               ⟨dorecipe() explain when found arc a making target n out of date 125d⟩
          }
     } else {
          ⟨dorecipe() explain when found target n with no prerequisite 125e⟩
     }
}
```

Uses `addw()` 34e and `outofdate()` 99a.

Chapter 8

Scheduling Jobs

In this chapter, you will see the remaining important functions of `mk` that I introduced in Figure 2.7. I mentioned `run()`[103a] (called from `dorecipe()`[100a] before), which enqueues a job, and `waitup()`[109a] (called from `mk()`[94]), which waits for a job to finish. Both functions eventually call `sched()`[105e], which takes a job from the job queue and schedules it for execution by a shell with `execsh()`[185c]. 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` 103a⟩≡ (181c)

```
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` 103c, `nrunning-17` 103b, and `sched()` 105e.

`run()`[103a] relies on the two globals below:

⟨*global* `nrunning` 103b⟩≡ (181c)

```
static int nrunning;
```

⟨*global* `nproclimit` 103c⟩≡ (181c)

```
static int nproclimit;
```

`nrunning`[103b] counts the number of processes currently running a recipe, while `nproclimit`[103c] 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()[109a], which will wait for some process to finish, and which will call sched()[105e] 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()[185c]) and looks for this variable to setup nproclimit with the code below:

⟨mk() *initializations* 104a⟩≡ (94) 146l ▷
```
nproc(); /* it can be updated dynamically */
```
Uses nproc() 104b.

⟨*function* nproc 104b⟩≡ (181c)
```
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) 164f⟩

    ⟨nproc() grow nevents if necessary 105b⟩
}
```
Uses S_VAR 31a, empty_words 60b, nproclimit-18 103c, and symlook() 31e.

8.2 Scheduling jobs

Before showing the code of sched()[105e], I need to present an important data structure used by sched().

8.2.1 RunEvent and events

sched()[105e] 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()[109a]. 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 104c⟩≡ (181c)
```
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 104d⟩≡ (181c)
```
// growing_array<Runevent> (size = nevents (== nproclimit))
static RunEvent *events;
```

⟨*global* nevents 105a⟩≡ (181c)
```
static int nevents;
```

events[104d] is a growing array. The index of a mapping in this array is called a *slot*. The growing array is initialized in nproc()[104b]:

⟨nproc() *grow nevents if necessary* 105b⟩≡ (104b)
```
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() 167a, Realloc() 167b, events-15 104d, nevents-16 105a, and nproclimit-18 103c.

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

⟨*function* nextslot 105c⟩≡ (181c)
```
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 104d and nproclimit-18 103c.

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

⟨*function* pidslot 105d⟩≡ (181c)
```
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) 164e⟩
    return -1;
}
```
Uses events-15 104d and nevents-16 105a.

8.2.2 sched()

Here is finally the code of sched():

⟨*function* sched 105e⟩≡ (181c)
```
static void
sched(void)
{
```

```
    Job *j;
    int slot;
    char *flags;
    ShellEnvVar *e;
    ⟨sched() other locals 122a⟩

    ⟨sched() return if no jobs 106⟩

    // j = pop(jobs)
    j = jobs;
    jobs = j->next;
    ⟨sched() if DEBUG(D_EXEC) 163e⟩

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

    e = buildenv(j, slot);
    ⟨sched() print recipe command on stdout 122b⟩

    ⟨sched() if dry mode or touch mode, alternate to execsh 126c⟩
    else {
       ⟨sched() if DEBUG(D_EXEC) print recipe 164a⟩
        flags = "-e";
       ⟨sched() reset flags if NOMINUSE rule 141f⟩

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

        usage();
        nrunning++;
       ⟨sched() if DEBUG(D_EXEC) print pid 164b⟩
    }
}
```

Uses `buildenv()` 119b, `events-15` 104d, `jobs` 45a, `nextslot()` 105c, `nrunning-17` 103b, and `usage()` 128a.

The most important call in the code above is the call to `execsh()`[185c]. `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()`[105e] above increments `nrunning`[103b] 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`[104d].

`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()`[119b] 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()`[109a], in which case the job queue may be empty. In that case, `sched()` simply returns.

⟨`sched()` *return if no jobs* 106⟩≡ (105e)

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

Uses `jobs` 45a and `usage()` 128a.

8.3 Executing Jobs: `execsh()`

`execsh()`[185c] 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` 107a⟩≡ (191)
```
//char  *shell = "/bin/rc";
```

⟨*global* `shellname` 107b⟩≡ (191)
```
//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()`[105e]):

⟨*global* `shflags` 107c⟩≡ (178a)
```
char *shflags = "-I"; /* rc flag to force non-interactive mode */
```
Uses `shflags` 107c.

`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` 107d⟩≡ (191)
```
int
execsh(char *shargs, char *shinput, Bufblock *buf, ShellEnvVar *e)
{
    int pid1, pid2;
    fdt in[2]; // pipe descriptors
    int err;
    ⟨execsh() other locals 108a⟩

    ⟨execsh() if buf then create pipe to save output 133b⟩

    pid1 = rfork(RFPROC|RFFDG|RFENVG);
    ⟨execsh() sanity check pid rfork 108c⟩
    // child
    if(pid1 == 0){
        ⟨execsh() in child, if buf, close one side of pipe 133c⟩
        err = pipe(in);
        ⟨execsh() sanity check err pipe 108e⟩
        pid2 = fork();
        ⟨execsh() sanity check pid fork 108d⟩
        // 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 133d⟩
            close(in[0]);
            close(in[1]);
            ⟨execsh() in child, export environment before exec 119c⟩
            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 133e⟩
        close(in[0]);
        // feed the shell
```

```
        ⟨execsh() in grandchild, write cmd in pipe 108b⟩
        close(in[1]); // will flush
        _exits(nil);
    }
    // parent
    ⟨execsh() in parent, if buf, close other side of pipe and read output 133g⟩
    return pid1;
}
```

Uses shflags 107c.

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[104d] and looked for by waitup()[109a]. 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 108a⟩≡                                    (107d) 133a▷
  char *endshinput;

⟨execsh() in grandchild, write cmd in pipe 108b⟩≡                      (107d)
  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 108c⟩≡                                (107d)
  if(pid1 < 0){
      perror("mk rfork");
      Exit();
  }

⟨execsh() sanity check pid fork 108d⟩≡                                 (107d)
  if(pid2 < 0){
      perror("mk fork");
      Exit();
  }

⟨execsh() sanity check err pipe 108e⟩≡                                 (107d)
  if(err < 0){
      perror("pipe");
      Exit();
  }
```

8.4 Waiting for jobs to finish

The interface of waitup() is complex. I will focus first on the easy case where I do not care about the arguments passed to waitup(), and describe later the use of those arguments and the different edge cases of waitup().

8.4.1 waitup()

Once sched()[105e] scheduled a job and modified events[104d], mk()[94] 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()[110a] to update the graph of dependencies and sched() to schedule more jobs:

⟨*function* waitup 109a⟩≡ (181c)

```
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 111d⟩

    ⟨waitup() if retstatus, check process list 144e⟩
again:  /* rogue processes */

    pid = xwaitfor(buf);
    ⟨waitup() if no more children 111c⟩
    ⟨waitup() if DEBUG(D_EXEC) print pid 164c⟩
    ⟨waitup() if retstatus, check if matching pid 145a⟩

    slot = pidslot(pid);
    ⟨waitup() if slot not found, not a job pid, update process list 144c⟩

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

    ⟨waitup() if error in child process, possibly set fake or exit 111e⟩
    // 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 112a⟩
        update(node, fake);
    }

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

Uses JOB_ENDED 111b, S_NODE 84a, events-15 104d, nproclimit-18 103c, nrunning-17 103b, pidslot() 105d, sched() 105e, symlook() 31e, update() 110a, and usage() 128a.

waitup()[109a] 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 109b⟩≡ (191)
```
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()`[109a], marks the target nodes of a job as `MADE`[42d] and updates the modification time of those nodes.

⟨*function* update 110a⟩≡ (183b)
```
void
update(Node *node, bool fake)
{
    Arc *a;

    ⟨update() if fake 158e⟩
    else
       MADESET(node, MADE);
    ⟨update() debug 165b⟩

    ⟨update() if virtual node or inexistent file 140b⟩
    else {
        node->time = timeof(node->name, true);
        ⟨update() unpretend node 153e⟩
        ⟨update() set outofdate prereqs if arc prog 142l⟩
    }
}
```
Uses `MADE` 42d, `MADESET` 96b, and `timeof()` 85a.

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()`[109a] is complex:

⟨*signature waitup* 110b⟩≡
```
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` 111a⟩≡ (174d)
```
enum WaitupParam {
  EMPTY_CHILDREN_IS_OK = 1,
  EMPTY_CHILDREN_IS_ERROR = -1,
  ⟨WaitupParam other cases 134a⟩
};
```

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

⟨*type* `WaitupResult` 111b⟩≡ (174d)
```
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 7.1, `mk()`[94] 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* 111c⟩≡ (109a)
```
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` 111b and `EMPTY_CHILDREN_IS_OK` 111a.

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* 111d⟩≡ (109a) 122c ▷
```
Bufblock *bp;
```

⟨`waitup()` *if error in child process, possibly set fake or exit* 111e⟩≡ (109a)
```
if(buf[0]){
    bp = newbuf();
    ⟨waitup() if error in child process, print recipe in bp 122d⟩
    fprint(STDERR, "mk: %s: exit status=%s", bp->start, buf);
    freebuf(bp);
    ⟨waitup() when error in child process, delete if DELETE node 140h⟩
    fprint(STDERR, "\n");

    ⟨waitup() when error in child process, if kflag 158d⟩
    else {
```

```
        jobs = nil;
        Exit();
    }
}
```

Uses freebuf() 168e, jobs 45a, and newbuf() 168d.

⟨waitup() *skip if node not found* 112a⟩≡ (109a)

```
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 112b⟩≡ (191)

```
void
Exit(void)
{
    while(waitpid() >= 0)
        ;
    exits("error");
}
```

8.6 Notes (signals) management

⟨main() *initializations before building* 112c⟩≡ (46b) 115f▷

```
catchnotes();
```

⟨*function* catchnotes 112d⟩≡ (191)

```
void
catchnotes()
{
    atnotify(notifyf, 1);
}
```

⟨*function* notifyf 112e⟩≡ (191)

```
int
notifyf(void *a, char *msg)
{
    ⟨notifyf() sanity check not too many notes 113a⟩
    if(strcmp(msg, "interrupt")!=0 && strcmp(msg, "hangup")!=0)
        return 0;
    killchildren(msg);
    return -1;
}
```

Uses killchildren() 113b.

⟨`notifyf()` *sanity check not too many notes* 113a⟩≡ (112e)

```
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` 113b⟩≡ (181c)

```
void
killchildren(char *msg)
{
    ⟨killchildren() locals 145b⟩

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

    ⟨killchildren() expunge not-job processes 145c⟩

    while(waitup(EMPTY_CHILDREN_IS_OK, (int *)nil) == JOB_ENDED)
        ;
    Bprint(&bout, "mk: %s\n", msg);
    Exit();
}
```

Uses `EMPTY_CHILDREN_IS_OK` 111a, `JOB_ENDED` 111b, `bout` 47a, `jobs` 45a, `kflag` 157g, and `waitup()` 109a.

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.

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`[33b] 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` 114a⟩≡ (174d)

```
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` 114b⟩≡ (179b)

```
// 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` 114c⟩≡ (179b)

```
// idx for next free entry in shellenv array
static int nextv;
```

⟨*function* `envinsert` 114d⟩≡ (179b)

```
static void
envinsert(char *name, Word *value)
{
    ⟨envinsert() locals 115a⟩
```

```
    ⟨envinsert() grow array if necessary 115b⟩
    shellenv[nextv].name = name;
    shellenv[nextv].values = value;
    nextv++;
}
```

Uses nextv-7 114c and shellenv 114b.

⟨envinsert() *locals* 115a⟩≡ (114d)

```
static int envsize = 0;
```

⟨envinsert() *grow array if necessary* 115b⟩≡ (114d)

```
if (nextv >= envsize) {
    envsize += ENVQUANTA;
    shellenv = (ShellEnvVar *) Realloc((char *) shellenv,
                                      envsize*sizeof(ShellEnvVar));
}
```

Uses ENVQUANTA-6 115c, Realloc() 167b, nextv-7 114c, and shellenv 114b.

⟨*constant* ENVQUANTA 115c⟩≡ (179b)

```
#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 115d⟩≡ (179b)

```
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 115e⟩
}
```

Uses freewords() 34d and shellenv 114b.

⟨envupd() *if variable not found* 115e⟩≡ (115d)

```
// else
e->name = name;
e->values = value;
envinsert(nil,nil);
```

Uses envinsert() 114d.

9.2 Initializing the shell environment: `initenv()`

To initialize `shellenv`[114b], `main()`[46b] calls `initenv()`[116a] before calling `mk()`[94]:

⟨main() *initializations before building* 115f⟩+≡ (46b) ◁112c 126f▷

```
initenv();
```

Uses initenv() 116a.

⟨*function* `initenv` 116a⟩≡ (179b)

```
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()` 116b, `envinsert()` 114d, `nextv-7` 114c, `specialvars-8` 33c, `stow()` 62b, and `symtraverse()` 32e.

I described `symtraverse()`[32e] before. It allows to iterate over a namespace while applying a function, here `ecopy()`:

⟨*function* `ecopy` 116b⟩≡ (179b)

```
static void
ecopy(Symtab *s)
{
    char **p;

    ⟨ecopy() return and do not copy if S_NOEXPORT symbol 145h⟩
    ⟨ecopy() return and do not copy if conflict with mk internal variable 116c⟩
     // else
     envinsert(s->name, s->u.ptr);
}
```

Uses `envinsert()` 114d.

Note that `initenv()` calls `envinsert()`[114d] 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()`[115d] 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* 116c⟩≡ (116b)

```
for(p = specialvars; *p; p++)
    if(strcmp(*p, s->name) == 0)
        return;
```

Uses `specialvars-8` 33c.

9.3 Importing the environment: `readenv()`

`initenv()`[116a] iterates over the `S_VAR`[31a] namespace with `symtraverse()`[32e] to add in `shellenv`[114b] the user variables, for instance, a variable `$CFLAGS` defined in the `mkfile`. In fact, the `S_VAR` namespace contains also the variables from the environment of `mk` itself. Indeed `main()`[46b] calls `inithash()`[33d] to initialize the symbol table, and `inithash()` calls `readenv()` below to populate the symbol table with variables from the environment (e.g., `$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` 117a⟩≡ (191)
```
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 118a⟩
    while((n = dirread(envdir, &e)) > 0){
        for(i = 0; i < n; i++){
            len = e[i].length;
            ⟨readenv() skip some names 117b⟩

            snprint(name, sizeof name, "/env/%s", e[i].name);
            envfile = open(name, OREAD);
            ⟨readenv() sanity check envfile 118b⟩
            p = Malloc(len+1);
            len2 = read(envfile, p, len);
            ⟨readenv() sanity check len2 118c⟩
            close(envfile);
            ⟨readenv() add null terminator character at end of p 119a⟩
            w = encodenulls(p, len);
            free(p);
            p = strdup(e[i].name);

            // populating symbol table
            setvar(p, (void *) w);
        }
        free(e);
    }
    close(envdir);
}
```
Uses `Malloc()` 167a and `setvar()` 33a.

`readenv()`[185c] 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* 117b⟩≡ (117a)
```
/* 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` 33b, `shname()` 117c, and `symlook()` 31e.

⟨*function* `shname` 117c⟩≡ (178c)
```
char *
```

```
shname(char *a)
{
    Rune r;
    int n;

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

Uses WORDCHR 67a.

⟨readenv() *sanity check envdir* 118a⟩≡ (117a)

```
if(envdir < 0)
    return;
```

⟨readenv() *sanity check envfile* 118b⟩≡ (117a)

```
if(envfile < 0)
    continue;
```

⟨readenv() *sanity check len2* 118c⟩≡ (117a)

```
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 118d⟩≡ (191)

```
/* 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 119a⟩≡ (117a)

```
if (p[len-1] == '\0')
    len--;
else
    p[len] = '\0';
```

9.4 Adjusting the shell environment: buildenv()

Once shellenv[114b] has been initialized, sched()[105e] can call buildenv() to adjust the environment with the specific values of mk's special variable for a specific job:

⟨*function* buildenv 119b⟩≡ (179b)

```
ShellEnvVar*
buildenv(Job *j, int slot)
{
    ⟨buildenv() locals 131a⟩

    // main variables
    envupd("target", wdup(j->t));
    ⟨buildenv() if regexp rule 129g⟩
    else
        envupd("stem", newword(j->stem));
    envupd("prereq", wdup(j->p));

    // advanced variables
    ⟨buildenv() envupd some variables 131b⟩

    return shellenv;
}
```

Uses envupd() 115d, newword() 34c, shellenv 114b, and wdup() 35a.

Note that as I mentioned in Section 9.1, buildenv()[119b] 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()[105e] updated shellenv[114b] with buildenv()[119b] and called execsh()[185c] with this environment, execsh() forks a shell interpreter and calls exportenv()[185c] in the child process to modify its own environment:

⟨execsh() *in child, export environment before exec* 119c⟩≡ (107d)

```
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` 120a⟩≡ (191)
```
/* 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 120c⟩

    for(;e->name; e++){
        ⟨exportenv() skip entry if not a user variable and no value 120d⟩
        // else
        snprint(name, sizeof name, "/env/%s", e->name);
        ⟨exportenv() if existing symbol but no value, remove from env 121a⟩
        // else
        f = create(name, OWRITE, 0666L);
        ⟨exportenv() sanity check f 121b⟩
        first = true;
        for (w = e->values; w; w = w->next) {
            n = strlen(w->s);
            if (n) {
                ⟨exportenv() write null separator 120b⟩
                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* 120b⟩≡ (120a)
```
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* 120c⟩≡ (120a)
```
Symtab *sym;
bool hasvalue;
```

⟨`exportenv()` *skip entry if not a user variable and no value* 120d⟩≡ (120a)
```
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` 60b, and `symlook()` 31e.

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* 121a⟩≡ (120a)

```
  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()` 34d.

⟨`exportenv()` *sanity check f* 121b⟩≡ (120a)

```
  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()`[123a] 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* 122a⟩≡ (105e) 126b▷
```
  Bufblock *buf;
```

⟨`sched()` *print recipe command on stdout* 122b⟩≡ (105e)
```
  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` 141b, `bout` 47a, `freebuf()` 168e, `newbuf()` 168d, `nflag` 125f, `shprint()` 123a, and `tflag` 153f.

⟨`waitup()` *other locals* 122c⟩+≡ (109a) ◁111d 140g▷
```
  ShellEnvVar *e;
```

⟨`waitup()` *if error in child process, print recipe in* `bp` 122d⟩≡ (111e)
```
  e = buildenv(j, slot);
  shprint(j->r->recipe, e, bp);
  front(bp->start);
```
Uses `buildenv()` 119b, `front()` 122e, and `shprint()` 123a.

⟨*function* `front` 122e⟩≡ (180a)
```
  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 123a⟩≡ (180a)

```
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() 124d, insert() 169a, rinsert() 169b, and vexpand() 123b.

10.1.1 Expanding and printing variables

⟨*function* vexpand 123b⟩≡ (180a)

```
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() 169c, mygetenv() 124a, and shname() 117c.

⟨*function* mygetenv 124a⟩≡ (180a)
```
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 33b, S_WESET 124b, symlook() 31e, and wtos() 35b.

⟨Sxxx *cases* 124b⟩+≡ (31a) ◁84a 142k▷
```
S_WESET, /* variable; we set in the mkfile */
```

⟨parse() *when parsing variable definitions, extra setting* 124c⟩≡ (75c)
```
symlook(head->s, S_WESET, (void *)"");
```
Uses S_WESET 124b and symlook() 31e.

10.1.2 Printing quoted strings

⟨*function* copyq 124d⟩≡ (178a)
```
/*
 * 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()` 125a and `rinsert()` 169b.

⟨*function* `copysingle` 125a⟩≡ (178a)

```
/*
 * 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()` 169b.

10.2 Explain mode: `mk -e`

⟨*global* `explain` 125b⟩≡ (176)

```
bool explain = false;
```

Uses `explain` 125b.

⟨`main()` *-xxx switch cases* 125c⟩+≡ (47d) ◁51c 126a▷

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

Uses `explain` 125b.

⟨`dorecipe()` *explain when found arc* `a` *making target* `n` *out of date* 125d⟩≡ (102b)

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

Uses `explain` 125b.

⟨`dorecipe()` *explain when found target* `n` *with no prerequisite* 125e⟩≡ (102b)

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

Uses `explain` 125b.

10.3 Dry mode: `mk -n`

⟨*global* `nflag` 125f⟩≡ (176)

```
bool nflag = false;
```

Uses `nflag` 125f.

⟨main() *-xxx switch cases* 126a⟩+≡ (47d) ◁125c 126e▷
```
case 'n':
    nflag = true;
    break;
```
Uses `nflag` 125f.

⟨sched() *other locals* 126b⟩+≡ (105e) ◁122a
```
Node *n;
```

⟨sched() *if dry mode or touch mode, alternate to execsh* 126c⟩≡ (105e)
```
if(nflag || tflag){
    for(n = j->n; n; n = n->next){
        ⟨sched() if touch mode 153i⟩
        n->time = time((long *)nil);
        MADESET(n, MADE);
    }
}
```
Uses `MADE` 42d, `MADESET` 96b, `nflag` 125f, and `tflag` 153f.

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

⟨main() *locals* 126d⟩+≡ (46b) ◁51b 157a▷
```
Bufblock *whatif = nil;
```

⟨main() *-xxx switch cases* 126e⟩+≡ (47d) ◁126a 127b▷
```
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()` 48a, `bufcpy()` 169c, `insert()` 169a, and `newbuf()` 168d.

⟨main() *initializations before building* 126f⟩+≡ (46b) ◁115f
```
if(whatif){
    insert(whatif, '\0');
    timeinit(whatif->start);
    freebuf(whatif);
}
```
Uses `freebuf()` 168e, `insert()` 169a, and `timeinit()` 126g.

⟨*function* timeinit 126g⟩≡ (181b)
```
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` 154c and `symlook()` 31e.

10.5 Processor utilization: `mk -u`

⟨*global* `uflag` 127a⟩≡ (185b)

```
  bool uflag = false;
```

Uses `uflag` 127a.

⟨`main()` *-xxx switch cases* 127b⟩+≡ (47d) ◁126e 151d▷

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

Uses `uflag` 127a.

⟨*global* `tslot` 127c⟩≡ (181c)

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

⟨*global* `tick` 127d⟩≡ (181c)

```
  static ulong tick;
```

⟨`main()` *setup profiling* 127e⟩≡ (47c) 166b▷

```
  usage();
```

Uses `usage()` 128a.

⟨`main()` *profile initializations* 127f⟩≡ (47c)

```
  usage();
```

Uses `usage()` 128a.

⟨*function* usage 128a⟩≡ (181c)
```
void
usage(void)
{
    ulong t;

    t = time(nil);
    if(tick)
        tslot[nrunning] += t - tick;
    tick = t;
}
```
Uses nrunning-17 103b, tick-22 127d, and tslot-21 127c.

⟨main() *print profiling stats if uflag* 128b⟩≡ (46b)
```
if(uflag)
    prusage();
```
Uses prusage() 128c and uflag 127a.

⟨*function* prusage 128c⟩≡ (181c)
```
void
prusage(void)
{
    int i;

    usage();
    for(i = 0; i <= nevents; i++)
        fprint(STDOUT, "%d: %lud\n", i, tslot[i]);
}
```
Uses nevents-16 105a, tslot-21 127c, and usage() 128a.

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* 129a⟩≡ (37f) 139c ▷

```
REGEXP = 0x0020,
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 129b⟩≡ (73a) 139d ▷

```
case 'R':
    *attr |= REGEXP;
    break;
```

⟨`Rule` *other fields* 129c⟩+≡ (36a) ◁ 87b 142d ▷

```
Reprog  *pat;  /* reg exp goo */
```

⟨*global* `patrule` 129d⟩≡ (176)

```
Rule *patrule;
```

⟨`addrule()` *if REGEXP attribute* 129e⟩≡ (38c)

```
if(attr&REGEXP){
    patrule = r;
    r->pat = regcomp(target);
}
```

Uses `REGEXP` 129a and `patrule` 129d.

⟨`dorecipe()` *if regexp rule* 129f⟩≡ (101b)

```
if(master_rule->attr&REGEXP){
    last_oldtargets->next = newword(node->name);
    last_alltargets->next = newword(node->name);
}
```

Uses `REGEXP` 129a and `newword()` 34c.

⟨`buildenv()` *if regexp rule* 129g⟩≡ (119b)

```
if(j->r->attr&REGEXP)
    envupd("stem", newword(""));
```

Uses `REGEXP` 129a, `envupd()` 115d, and `newword()` 34c.

⟨*function* regerror 130a⟩≡ (180d)
```
//@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 53a, mkinline 53c, and patrule 129d.

⟨*constant* NREGEXP 130b⟩≡ (174d)
```
#define  NREGEXP  10
```

⟨Arc *other fields* 130c⟩+≡ (42f) ◁91a 142g▷
```
char  *match[NREGEXP];
```
Uses NREGEXP 130b.

⟨newarc() *set other fields* 130d⟩+≡ (43c) ◁91b 142h▷
```
rcopy(a->match, match, NREGEXP);
```
Uses NREGEXP 130b.

⟨*function* rcopy 130e⟩≡ (191)
```
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* 130f⟩≡ (44a) 146c▷
```
char  **match;
```

⟨specialvars *other array elements* 130g⟩≡ (33c) 146b▷
```
"stem0",  /* must be in order from here */
"stem1",
"stem2",
"stem3",
"stem4",
"stem5",
"stem6",
"stem7",
"stem8",
"stem9",
```

⟨buildenv() *locals* 131a⟩≡ (119b) 147c ▷
```
char **p;
int i;
```

⟨buildenv() *envupd some variables* 131b⟩≡ (119b) 146e ▷
```
/* 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 129a, envupd() 115d, newword() 34c, and specialvars-8 33c.

⟨applyrules *other locals* 131c⟩+≡ (79a) ◁ 81c
```
Resub rmatch[NREGEXP];
```
Uses NREGEXP 130b.

⟨applyrules *other initializations* 131d⟩≡ (79a)
```
memset((char*)rmatch, 0, sizeof(rmatch));
```

⟨applyrules() *if regexp rule then continue if some conditions* 131e⟩≡ (80e)
```
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 130b, REGEXP 129a, and patrule 129d.

⟨applyrules() *if regexp rule, adjust buf and rmatch* 131f⟩≡ (80e)
```
if(r->attr&REGEXP)
    regsub(pre->s, buf, sizeof(buf), rmatch, NREGEXP);
```
Uses NREGEXP 130b and REGEXP 129a.

11.2 Shell-command expansion: ‘*cmd*‘

11.2.1 Parsing backquotes: bquote()

⟨assline() *switch character cases* 131g⟩+≡ (55b) ◁ 58a
```
case '`':
    if (bquote(bp, buf) == ERROR_0)
        Exit();
    break;
```
Uses bquote() 132a.

⟨*function* bquote 132a⟩≡ (181a)

```
/*
 * 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 132c⟩
    if(c == '{'){
        term = '}';  /* rc style */
        ⟨bquote() skip spaces 132c⟩
    } 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 132d⟩
            return OK_1;
        }
        if(c == '\n')
            break; // go to error
        ⟨bquote() handle quotes and escape characters 132b⟩
        rinsert(buf, c);
    }
    SYNERR(line);
    fprint(STDERR, "missing closing %c after `\n", term);
    return ERROR_0;
}
```

Uses SYNERR 53d, insert() 169a, mkinline 53c, nextrune() 56a, and rinsert() 169b.

⟨bquote() *handle quotes and escape characters* 132b⟩≡ (132a)

```
if(c == '\'' || c == '"' || c == '\\'){
    insert(buf, c);
    if(!escapetoken(bp, buf, true, c))
        return ERROR_0;
    continue;
}
```

Uses escapetoken() 58b and insert() 169a.

⟨bquote() *skip spaces* 132c⟩≡ (132)

```
while((c = Bgetrune(bp)) == ' ' || c == '\t')
        ;
```

11.2.2 Adjusting execsh()

⟨bquote() *execute shell command in* buf 132d⟩≡ (132a)

```
initenv();
```

```
  // running the command, passing a buf argument
  execsh(nil, buf->current, buf, shellenv);
```
Uses `initenv()` 116a and `shellenv` 114b.

⟨`execsh()` *other locals* 133a⟩+≡ (107d) ◁108a 133f▷
```
  fdt out[2];
```

⟨`execsh()` *if buf then create pipe to save output* 133b⟩≡ (107d)
```
  if(buf && pipe(out) < 0){
      perror("pipe");
      Exit();
  }
```

⟨`execsh()` *in child, if buf, close one side of pipe* 133c⟩≡ (107d)
```
  if(buf)
      close(out[0]);
```

⟨`execsh()` *in child, if buf, dup and close* 133d⟩≡ (107d)
```
  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* 133e⟩≡ (107d)
```
  if(buf)
      close(out[1]);
```

⟨`execsh()` *other locals* 133f⟩+≡ (107d) ◁133a
```
  int tot, n;
```

⟨`execsh()` *in parent, if buf, close other side of pipe and read output* 133g⟩≡ (107d)
```
  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()` 169d.

11.3 Dynamic `mkfile`: `<|`*prog*

⟨`rhead()` *adjust sep if dynamic mkfile* `<|` 133h⟩≡ (59b)
```
  if(sep == '<' && *p == '|'){
      sep = '|';
      p++;
  }
```

⟨WaitupParam *other cases* 134a⟩≡ (111a) 158g ▷

```
  EMPTY_CHILDREN_IS_ERROR3 = -3,
```

⟨parse() *other locals* 134b⟩+≡ (53e) ◁ 75b 142e ▷

```
  int pid;
```

⟨parse() *switch rhead cases* 134c⟩+≡ (53e) ◁ 75c

```
  case '|':
      p = wtos(tail, ' ');
      ⟨parse() sanity check p for include program name 134d⟩

      initenv();
      pid = pipecmd(p, shellenv, &newfd);
      ⟨parse() sanity check newfd 134e⟩
      else
          // recursive call
          parse(p, newfd, 0);

      while(waitup(EMPTY_CHILDREN_IS_ERROR3, &pid) >= 0)
          ;
      ⟨parse() sanity check pid after waitup 134f⟩
      break;
```

Uses EMPTY_CHILDREN_IS_ERROR3 134a, initenv() 116a, parse() 53e, shellenv 114b, waitup() 109a, and wtos() 35b.

⟨parse() *sanity check* p *for include program name* 134d⟩≡ (134c)

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

Uses SYNERR 53d.

⟨parse() *sanity check* newfd 134e⟩≡ (134c)

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

⟨parse() *sanity check* pid *after waitup* 134f⟩≡ (134c)

```
  if(pid != 0){
      fprint(STDERR, "bad include program status\n");
      Exit();
  }
```

⟨*function* pipecmd 134g⟩≡ (191)

```
  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` 161c, `D_EXEC` 161a, and `shflags` 107c.

11.4 Substitution variables: $*name*:*pattern*=*subst*

11.4.1 Parsing adjustments

⟨`charin()` *switch rune cases* 135a⟩+≡ (60c) ◁61a

```
    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* 135b⟩≡ (60c)

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

Uses `SYNERR` 53d.

⟨`varsub()` *if variable starts with open brace* 135c⟩≡ (65d)

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

Uses `expandvar()` 136a.

⟨*function* expandvar 136a⟩≡ (184b)
```
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 53d, S_VAR 31a, charin() 60c, freebuf() 168e, newword() 34c, symlook() 31e, and varname() 66d.

11.4.2 Substitutions: subsub()

⟨*function* subsub 136b⟩≡ (184b)
```
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 138a⟩≡ (184b)

```
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()` 60c and `stow()` 62b.

⟨*function* submatch 138b⟩≡ (184b)

```
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

⟨graph() *propagate attributes* 139a⟩≡ (78)
```
// propagate attributes in rules to their node
attribute(root);
```
Uses attribute() 139b.

⟨*function* attribute 139b⟩≡ (183a)
```
static void
attribute(Node *n)
{
    Arc *a;

    for(a = n->arcs; a; a = a->next){
        ⟨attribute() propagate rule attribute to node cases 139f⟩
        // recurse
        if(a->n)
            attribute(a->n);
    }
    ⟨attribute() if virtual node 139g⟩
}
```
Uses attribute() 139b.

11.5.1 Virtual target: :V:

⟨Rule_attr *cases* 139c⟩+≡ (37f) ◁129a 140c▷
```
VIR    = 0x0010,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 139d⟩+≡ (73a) ◁129b 140d▷
```
case 'V':
    *attr |= VIR;
    break;
```

⟨Node_flag *cases* 139e⟩+≡ (42c) ◁92g 140e▷
```
VIRTUAL    = 0x0001,
```

⟨attribute() *propagate rule attribute to node cases* 139f⟩≡ (139b) 140f▷
```
if(a->r->attr&VIR)
    n->flags |= VIRTUAL;
```
Uses VIR 139c and VIRTUAL 139e.

⟨attribute() *if virtual node* 139g⟩≡ (139b)
```
if(n->flags&VIRTUAL)
    n->time = 0;
```
Uses VIRTUAL 139e.

⟨dorecipe() *when no recipe found, if virtual or norecipe node* 140a⟩≡ (100c)
```
  if((node->flags&VIRTUAL) || (node->flags&NORECIPE)){
      ⟨dorecipe() when no recipe found, if archive name 148c⟩
      else
          update(node, false);
      ⟨dorecipe() when no recipe found, if tflag 153h⟩
      //bugfix:
      return;
  }
```
Uses NORECIPE 141i, VIRTUAL 139e, and update() 110a.

⟨update() *if virtual node or inexistent file* 140b⟩≡ (110a)
```
  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 139e and outofdate() 99a.

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

⟨Rule_attr *cases* 140c⟩+≡ (37f) ◁139c 141b▷
```
  DEL    = 0x0080,
```

⟨rhead() *when parsing rule attributes, switch rune cases* 140d⟩+≡ (73a) ◁139d 141c▷
```
  case 'D':
      *attr |= DEL;
      break;
```

⟨Node_flag *cases* 140e⟩+≡ (42c) ◁139e 141d▷
```
  DELETE     = 0x0800,
```

⟨attribute() *propagate rule attribute to node cases* 140f⟩+≡ (139b) ◁139f 141j▷
```
  if(a->r->attr&DEL)
      n->flags |= DELETE;
```
Uses DEL 140c and DELETE 140e.

⟨waitup() *other locals* 140g⟩+≡ (109a) ◁122c 144d▷
```
  Node *n;
  bool done;
```

⟨waitup() *when error in child process, delete if DELETE node* 140h⟩≡ (111e)
```
  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 140e and delete() 141a.

⟨*function* delete 141a⟩≡ (181b)
```
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* 141b⟩+≡ (37f) ◁140c 141g▷
```
QUIET  = 0x0008,
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 141c⟩+≡ (73a) ◁140d 141e▷
```
case 'Q':
    *attr |= QUIET;
    break;
```

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

⟨`Node_flag` *cases* 141d⟩+≡ (42c) ◁140e 141i▷
```
NOMINUSE   = 0x1000,
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 141e⟩+≡ (73a) ◁141c 141h▷
```
case 'E':
    *attr |= NOMINUSE;
    break;
```

⟨`sched()` *reset flags if NOMINUSE rule* 141f⟩≡ (105e)
```
if (j->r->attr&NOMINUSE)
    flags = nil;
```
Uses `NOMINUSE` 141d.

11.5.5 Disabling the no-recipe warning, `:N:`

⟨`Rule_attr` *cases* 141g⟩+≡ (37f) ◁141b 142a▷
```
NOREC  = 0x0040,
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 141h⟩+≡ (73a) ◁141e 142b▷
```
case 'N':
    *attr |= NOREC;
    break;
```

⟨`Node_flag` *cases* 141i⟩+≡ (42c) ◁141d 151e▷
```
NORECIPE   = 0x0400,
```

⟨`attribute()` *propagate rule attribute to node cases* 141j⟩+≡ (139b) ◁140f
```
if(a->r->attr&NOREC)
    n->flags |= NORECIPE;
```
Uses `NOREC` 141g and `NORECIPE` 141i.

11.5.6 Forbidding metarules to match virtual targets: `:n:`

⟨`Rule_attr` *cases* 142a⟩+≡ (37f) ◁141g
```
  NOVIRT = 0x0100,
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 142b⟩+≡ (73a) ◁141h 142j▷
```
  case 'n':
      *attr |= NOVIRT;
      break;
```

⟨`applyrules()` *skip this meta rule and continue if some conditions* 142c⟩+≡ (80e) ◁81d
```
  if ((r->attr&NOVIRT) && lasta != &head && (lasta->r->attr&VIR))
      continue;
```
Uses `NOVIRT` 142a and `VIR` 139c.

11.5.7 Custom-dependency comparison program: `:P:`

⟨`Rule` *other fields* 142d⟩+≡ (36a) ◁129c
```
  char  *prog;  /* to use in out of date */
```

⟨`parse()` *other locals* 142e⟩+≡ (53e) ◁134b
```
  char *prog;
```

⟨`addrule()` *set more fields* 142f⟩+≡ (38a) ◁87d
```
  r->prog = prog;
```

⟨`Arc` *other fields* 142g⟩+≡ (42f) ◁130c
```
  char  *prog;
```

⟨`newarc()` *set other fields* 142h⟩+≡ (43c) ◁130d
```
  a->prog = r->prog;
```

⟨`rhead()` *other locals* 142i⟩+≡ (59b) ◁72e
```
  char *pp;
```

⟨`rhead()` *when parsing rule attributes, switch rune cases* 142j⟩+≡ (73a) ◁142b
```
  case 'P':
      pp = utfrune(p, ':');
      if (pp == nil || *pp == 0)
          goto eos;
      *pp = 0;
      *prog = strdup(p);
      *pp = ':';
      p = pp;
      break;
```

⟨`Sxxx` *cases* 142k⟩+≡ (31a) ◁124b 145g▷
```
  S_OUTOFDATE, /* n1\377n2 -> 2(outofdate) or 1(not outofdate) */
```

⟨`update()` *set outofdate prereqs if arc prog* 142l⟩≡ (110a)
```
  for(a = node->arcs; a; a = a->next)
      if(a->prog)
          outofdate(node, a, true);
```
Uses `outofdate()` 99a.

⟨`outofdate()` *locals* 142m⟩≡ (99a)
```
  char buf[3*NAMEBLOCK];
  char *str = nil;
  Symtab *sym;
  int ret;
```
Uses `NAMEBLOCK` 81b.

⟨outofdate() *if arc-¿prog* 143a⟩≡ (99a)

```
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 142k, pcmp() 143b, and symlook() 31e.

⟨*function* pcmp 143b⟩≡ (183b)

```
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 134a, NAMEBLOCK 81b, bout 47a, and waitup() 109a.

Process

⟨*struct* Process 143c⟩≡ (181c)

```
struct Process {
    int pid;
    int status;

    // Extra
    // double_list<ref_own<Process> backward, forward
    Process *b, *f;
};
```

⟨*global* phead 143d⟩≡ (181c)

```
// double_list<ref_own<Process> (next = Process.f)
static Process *phead;
```

⟨*global* pfree 143e⟩≡ (181c)

```
// double_list<ref_own<Process> (next = Process.f)
static Process *pfree;
```

⟨*function* `pnew` 144a⟩≡ (181c)

```
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()` 167a, `pfree-20` 143e, and `phead-19` 143d.

⟨*function* `pdelete` 144b⟩≡ (181c)

```
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` 143e and `phead-19` 143d.

`waitup()` adjustments

⟨`waitup()` *if slot not found, not a job pid, update process list* 144c⟩≡ (109a)

```
if(slot < 0){
   ⟨waitup() if DEBUG(D_EXEC) and slot ¡ 0 164d⟩
    pnew(pid, buf[0]? 1:0);
    goto again;
}
```

Uses `pnew()` 144a.

⟨`waitup()` *other locals* 144d⟩+≡ (109a) ◁140g

```
Process *p;
```

⟨`waitup()` *if retstatus, check process list* 144e⟩≡ (109a)

```
/* 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()` 144b and `phead-19` 143d.

⟨`waitup()` *if retstatus, check if matching pid* 145a⟩≡ (109a)

```
if(retstatus && pid == *retstatus){
    *retstatus = buf[0]? 1:0;
    return -1;
}
```

`killchildren()` adjustments

⟨`killchildren()` *locals* 145b⟩≡ (113b)

```
Process *p;
```

⟨`killchildren()` *expunge not-job processes* 145c⟩≡ (113b)

```
for(p = phead; p; p = p->f)
    expunge(p->pid, msg);
```

Uses `phead-19` 143d.

⟨*function* `expunge` 145d⟩≡ (191)

```
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* 145e⟩≡ (77a)

```
case 'U':
    *attr = 1;
    break;
```

⟨`parse()` *when parsing variable definitions, if variable with attr* 145f⟩≡ (75c)

```
if(attr)
    symlook(head->s, S_NOEXPORT, (void *)"");
```

Uses `S_NOEXPORT` 145g and `symlook()` 31e.

⟨`Sxxx` *cases* 145g⟩+≡ (31a) ◁142k 148b▷

```
S_NOEXPORT, /* var -> noexport */ // set of noexport variables
```

⟨`ecopy()` *return and do not copy if* `S_NOEXPORT` *symbol* 145h⟩≡ (116b)

```
if(symlook(s->name, S_NOEXPORT, nil))
    return;
```

Uses `S_NOEXPORT` 145g and `symlook()` 31e.

11.7 Advanced `mk` variables

11.7.1 `$target` versus `$alltargets`

⟨`dorecipe()` *other locals* 145i⟩+≡ (100a) ◁101a 146f▷

```
Word oldtargets;
Word *last_oldtargets = &oldtargets;
```

⟨dorecipe() *update list of outdated targets* 146a⟩≡ (101b)

```
  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 156b, newword() 34c, and outofdate() 99a.

⟨specialvars *other array elements* 146b⟩+≡ (33c) ◁130g 146h▷

```
  "alltarget",
```

⟨Job *other fields* 146c⟩+≡ (44a) ◁130f 146i▷

```
  Word  *at; /* all targets */
```

⟨newjob() *setting other fields* 146d⟩≡ (44c) 146j▷

```
  j->at = alltargets;
```

⟨buildenv() *envupd some variables* 146e⟩+≡ (119b) ◁131b 146k▷

```
  envupd("alltarget", wdup(j->at));
```

Uses envupd() 115d and wdup() 35a.

11.7.2 $prereq versus $newprereq

⟨dorecipe() *other locals* 146f⟩+≡ (100a) ◁145i

```
  Word newprereqs;
```

⟨dorecipe() *when outofdate node, update list of newprereqs* 146g⟩≡ (102b)

```
  addw(&newprereqs, a->n->name);
```

Uses addw() 34e.

⟨specialvars *other array elements* 146h⟩+≡ (33c) ◁146b 147b▷

```
  "newprereq",
```

⟨Job *other fields* 146i⟩+≡ (44a) ◁146c

```
  Word  *np; /* new prerequisites */
```

⟨newjob() *setting other fields* 146j⟩+≡ (44c) ◁146d

```
  j->np = newprereqs;
```

⟨buildenv() *envupd some variables* 146k⟩+≡ (119b) ◁146e 147d▷

```
  envupd("newprereq", wdup(j->np));
```

Uses envupd() 115d and wdup() 35a.

11.7.3 $NREP

⟨mk() *initializations* 146l⟩+≡ (94) ◁104a 158b▷

```
  nrep();  /* it can be updated dynamically */
```

⟨*function* nrep 147a⟩≡ (183a)
```
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) 163d⟩
}
```

11.7.4 $pid

⟨specialvars *other array elements* 147b⟩+≡ (33c) ◁146h 147e▷
```
"pid",
```

⟨buildenv() *locals* 147c⟩+≡ (119b) ◁131a 147h▷
```
char buf[256];
```

⟨buildenv() *envupd some variables* 147d⟩+≡ (119b) ◁146k 147f▷
```
snprint(buf, sizeof buf, "%d", getpid());
envupd("pid", newword(buf));
```
Uses envupd() 115d and newword() 34c.

11.7.5 $nproc

⟨specialvars *other array elements* 147e⟩+≡ (33c) ◁147b 147g▷
```
"nproc",
```

⟨buildenv() *envupd some variables* 147f⟩+≡ (119b) ◁147d 148a▷
```
snprint(buf, sizeof buf, "%d", slot);
envupd("nproc", newword(buf));
```
Uses envupd() 115d and newword() 34c.

11.8 Dealing with archives (libraries)

⟨specialvars *other array elements* 147g⟩+≡ (33c) ◁147e
```
"newmember",
```

⟨buildenv() *locals* 147h⟩+≡ (119b) ◁147c
```
Word *w, *v, **l;
char *cp, *qp;
```

⟨buildenv() *envupd some variables* 148a⟩+≡ (119b) ◁147f

```
// 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() 115d and wdup() 35a.

⟨Sxxx *cases* 148b⟩+≡ (31a) ◁145g 150a▷

```
S_AGG,  /* aggregate -> time */
```

⟨dorecipe() *when no recipe found, if archive name* 148c⟩≡ (140a)

```
if(strchr(node->name, '(') && node->time == 0)
    MADESET(node, MADE);
```

Uses MADE 42d and MADESET 96b.

⟨timeof() *if name archive member* 148d⟩≡ (85a)

```
if(utfrune(name, '('))
    return atimeof(force, name);  /* archive */
```

Uses atimeof() 148e.

⟨*function* atimeof 148e⟩≡ (178d)

```
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 148b, S_TIME 154c, atimes() 149b, split() 151a, and symlook() 31e.

⟨*function* atouch 149a⟩≡ (178d)

```
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 154c, split() 151a, and symlook() 31e.

⟨*function* atimes 149b⟩≡ (178d)

```
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 174a, S_TIME 154c, and symlook() 31e.

⟨Sxxx *cases* 150a⟩+≡ (31a) ◁148b 154c▷

```
S_BITCH, /* bitched about aggregate not there */
```

⟨*function* type 150b⟩≡ (178d)

```
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 150a, bout 47a, and symlook() 31e.

⟨*function* split 151a⟩≡ (178d)
```
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() 150b.

⟨outofdate() *if arc node is an archive member* 151b⟩≡ (99a)
```
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

⟨*global* iflag 151c⟩≡ (176)
```
bool iflag = false;
```
Uses iflag 151c.

⟨main() *-xxx switch cases* 151d⟩+≡ (47d) ◁127b 153g▷
```
case 'i':
    iflag = true;
    break;
```
Uses iflag 151c.

⟨Node_flag *cases* 151e⟩+≡ (42c) ◁141i
```
CANPRETEND = 0x0008,
PRETENDING = 0x0010,
```

⟨clrmade() n->flags *pretend adjustments* 152a⟩≡ (96a)

```
n->flags &= ~(CANPRETEND|PRETENDING);
if(strchr(n->name, '(') == nil || n->time)
    n->flags |= CANPRETEND;
```

Uses CANPRETEND 151e and PRETENDING 151e.

⟨work() *possibly pretending node* 152b⟩≡ (98c)

```
/*
 *   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 151e, MADE 42d, MADESET 96b, PRETENDING 151e, bout 47a, explain 125b, iflag 151c, outofdate() 99a, unpretend() 153a, and work() 97a.

⟨work() *possibly unpretending node* 152c⟩≡ (97a)

```
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 42d, PRETENDING 151e, explain 125b, outofdate() 99a, and unpretend() 153a.

⟨*function* unpretend 153a⟩≡ (183b)
```
static void
unpretend(Node *n)
{
    MADESET(n, NOTMADE);
    n->flags &= ~(CANPRETEND|PRETENDING);
    n->time = 0;
}
```
Uses CANPRETEND 151e, MADESET 96b, NOTMADE 42d, and PRETENDING 151e.

⟨work() *locals* 153b⟩+≡ (97a) ◁98b
```
Arc *ra = nil;
```

⟨work() *update* ra *when outofdate* node *with arc* a 153c⟩≡ (98c)
```
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* 153d⟩≡ (98c)
```
if(ra == nil)
    ra = a;
```

⟨update() *unpretend node* 153e⟩≡ (110a)
```
node->flags &= ~(CANPRETEND|PRETENDING);
```
Uses CANPRETEND 151e and PRETENDING 151e.

11.9.2 Touching-mode optimization: mk -t

⟨*global* tflag 153f⟩≡ (176)
```
bool tflag = false;
```
Uses tflag 153f.

⟨main() *-xxx switch cases* 153g⟩+≡ (47d) ◁151d 156c▷
```
case 't':
    tflag = true;
    break;
```
Uses tflag 153f.

⟨dorecipe() *when no recipe found, if tflag* 153h⟩≡ (140a)
```
if(tflag){
    if(!(node->flags&VIRTUAL))
        touch(node->name);
    else if(explain)
        Bprint(&bout, "no touch of virtual '%s'\n", node->name);
}
```
Uses VIRTUAL 139e, bout 47a, explain 125b, tflag 153f, and touch() 154a.

⟨sched() *if touch mode* 153i⟩≡ (126c)
```
if(tflag){
    if(!(n->flags&VIRTUAL))
        touch(n->name);
    else if(explain)
        Bprint(&bout, "no touch of virtual '%s'\n", n->name);
}
```
Uses VIRTUAL 139e, bout 47a, explain 125b, tflag 153f, and touch() 154a.

⟨*function* touch 154a⟩≡ (181b)

```
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() 149a, bout 47a, and nflag 125f.

⟨*function* chgtime 154b⟩≡ (191)

```
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* 154c⟩+≡ (31a) ◁150a 155e▷

```
S_TIME,  /* file -> time */
```

⟨timeof() *locals* 154d⟩≡ (85a) 154f▷

```
ulong t;
```

⟨timeof() *if not force, use time cache* 154e⟩≡ (85a)

```
⟨timeof() check time cache 154g⟩
t = mkmtime(name, false);
⟨timeof() update time cache 154h⟩
return t;
```

⟨timeof() *locals* 154f⟩+≡ (85a) ◁154d

```
Symtab *sym;
```

⟨timeof() *check time cache* 154g⟩≡ (154e)

```
sym = symlook(name, S_TIME, nil);
if (sym)
    return sym->u.value;  /* uggh */
```

Uses S_TIME 154c and symlook() 31e.

⟨timeof() *update time cache* 154h⟩≡ (154e)

```
if(t == 0)
    return 0;
symlook(name, S_TIME, (void*)t);  /* install time in cache */
```

Uses S_TIME 154c and symlook() 31e.

11.9.4 Bulk time optimisation

⟨mkmtime *locals* 155a⟩≡ (85b) 155c ▷
```
//char *s, *ss;
//char carry;
//Symtab *sym;
```

⟨mkmtime() *bulk dir optimisation* 155b⟩≡ (85b)
```
⟨mkmtime() cleanup name 155d⟩
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* 155c⟩+≡ (85b) ◁ 155a
```
char buf[4096];
```

⟨mkmtime() *cleanup name* 155d⟩≡ (155b)
```
strecpy(buf, buf + sizeof buf - 1, name);
cleanname(buf);
name = buf;
```

⟨Sxxx *cases* 155e⟩+≡ (31a) ◁ 154c
```
S_BULKED, /* we have bulked this dir */
```

⟨*function* bulkmtime 155f⟩≡ (191)
```
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 155e, dirtime() 156a, and symlook() 31e.

⟨*function* dirtime 156a⟩≡ (191)

```
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 154c and symlook() 31e.

11.10 Recompiling everything: mk -a

⟨*global* aflag 156b⟩≡ (176)

```
bool aflag = false;
```

Uses aflag 156b.

⟨main() *-xxx switch cases* 156c⟩+≡ (47d) ◁153g 157h▷

```
case 'a':
    aflag = true;
    iflag = true;
    break;
```

Uses aflag 156b and iflag 151c.

⟨work() *adjust weoutofdate if aflag* 156d⟩≡ (98c)

```
if(aflag)
    weoutofdate = true;
```

Uses aflag 156b.

11.11 Recursive mk

⟨main() *locals* 157a⟩+≡ (46b) ◁126d 161d▷
```
Bufblock *buf = newbuf();
```
Uses newbuf() 168d.

⟨main() *add* argv[0] *in* buf 157b⟩≡ (50a 47d)
```
bufcpy(buf, argv[0], strlen(argv[0]));
insert(buf, ' ');
```
Uses bufcpy() 169c and insert() 169a.

⟨main() *add* argv[i] *in* buf 157c⟩≡ (48c)
```
bufcpy(buf, argv[i], strlen(argv[i]));
insert(buf, ' ');
```
Uses bufcpy() 169c and insert() 169a.

⟨main() *set variables for recursive mk* 157d⟩≡ (47c)
```
⟨main() set MKFLAGS variable 157e⟩
⟨main() set MKARGS variable 157f⟩
```

⟨main() *set MKFLAGS variable* 157e⟩≡ (157d)
```
if (buf->current != buf->start) {
    buf->current--;
    insert(buf, '\0');
}
symlook("MKFLAGS", S_VAR, (void*) stow(buf->start));
```
Uses S_VAR 31a, insert() 169a, stow() 62b, and symlook() 31e.

⟨main() *set MKARGS variable* 157f⟩≡ (157d)
```
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() 169c, freebuf() 168e, insert() 169a, stow() 62b, and symlook() 31e.

11.12 mk -k

11.12.1 kflag and runerrs

⟨*global* kflag 157g⟩≡ (176)
```
bool kflag = false;
```
Uses kflag 157g.

⟨main() *-xxx switch cases* 157h⟩+≡ (47d) ◁156c 161e▷
```
case 'k':
    kflag = true;
    break;
```
Uses kflag 157g.

⟨*global* runerrs 158a⟩≡ (183b)

```
int runerrs;
```

⟨mk() *initializations* 158b⟩+≡ (94) ◁146l

```
runerrs = 0;
```

Uses runerrs 158a.

11.12.2 Adjusting mk(), work(), and waitup()

⟨work() *when inexistent target without prerequisites, if kflag* 158c⟩≡ (97d)

```
if(kflag){
    node->flags |= BEINGMADE;
    runerrs++;
}
```

Uses BEINGMADE 42d, kflag 157g, and runerrs 158a.

⟨waitup() *when error in child process, if kflag* 158d⟩≡ (111e)

```
if(kflag){
    runerrs++;
    fake = true;
}
```

Uses kflag 157g and runerrs 158a.

⟨update() *if fake* 158e⟩≡ (110a)

```
if(fake)
    MADESET(node, BEINGMADE);
```

Uses BEINGMADE 42d and MADESET 96b.

⟨mk() *if no child to waitup and root not MADE, possibly break* 158f⟩≡ (94)

```
if(res > 0){
    if(root->flags&(NOTMADE|BEINGMADE)){
        assert(/*must be run errors*/ runerrs);
        break; /* nothing more waiting */
    }
}
```

Uses BEINGMADE 42d, NOTMADE 42d, and runerrs 158a.

⟨WaitupParam *other cases* 158g⟩+≡ (111a) ◁134a

```
EMPTY_CHILDREN_IS_ERROR2 = -2,
```

⟨mk() *before returning, more* waitup() *if there was an error* 158h⟩≡ (94)

```
while(jobs)
    waitup(EMPTY_CHILDREN_IS_ERROR2, (int *)nil);
assert(/*target didnt get done*/ runerrs || (root->flags&MADE));
```

Uses EMPTY_CHILDREN_IS_ERROR2 158g, MADE 42d, jobs 45a, runerrs 158a, and waitup() 109a.

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 161a⟩≡ (174d)
```
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 161b⟩≡ (176)
```
// bitset<enum<dxxx>>
int debug;
```

⟨*function* DEBUG 161c⟩≡ (174d)
```
#define DEBUG(x) (debug&(x))
```

⟨main() *locals* 161d⟩+≡ (46b) ◁157a
```
char *s;
```

⟨main() *-xxx switch cases* 161e⟩+≡ (47d) ◁157h
```
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 161a, D_GRAPH 161a, D_PARSE 161a, and debug 161b.

A.1 Dumping the rules: `mk -dp`

⟨main() *if* DEBUG(D_PARSE) 162a⟩≡ (50b)

```
if(DEBUG(D_PARSE)){
    dumpw("default targets", target1);
    dumpr("rules", rules);
    dumpr("metarules", metarules);
    dumpv("variables");
}
```

Uses DEBUG 161c, D_PARSE 161a, dumpr() 162c, dumpv() 162d, dumpw() 162b, metarules 37b, rules 36c, and target1 50e.

⟨*dumper dumpw* 162b⟩≡ (185a)

```
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 47a.

⟨*dumper dumpr* 162c⟩≡ (185a)

```
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 47a and wtos() 35b.

⟨*dumper dumpv* 162d⟩≡ (185a)

```
void
dumpv(char *s)
{
    Bprint(&bout, "%s:\n", s);
    symtraverse(S_VAR, &print1);
}
```

Uses S_VAR 31a, bout 47a, print1() 162e, and symtraverse() 32e.

⟨*function* print1 162e⟩≡ (185a)

```
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 47a.

A.2 Dumping the graph: mk -dg

⟨mk() *if* DEBUG(D_GRAPH) 163a⟩≡ (94)

```
if(DEBUG(D_GRAPH)){
     dumpn("new target\n", root);
     Bflush(&bout);
}
```

Uses DEBUG 161c, D_GRAPH 161a, bout 47a, and dumpn() 163b.

⟨*dumper dumpn* 163b⟩≡ (185a)

```
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 47a and dumpa() 163c.

⟨*dumper dumpa* 163c⟩≡ (185a)

```
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 47a and dumpn() 163b.

⟨nrep() *if* DEBUG(D_GRAPH) 163d⟩≡ (147a)

```
if(DEBUG(D_GRAPH))
     Bprint(&bout, "nreps = %d\n", nreps);
```

A.3 Tracing jobs: mk -de

⟨sched() *if* DEBUG(D_EXEC) 163e⟩≡ (105e)

```
if(DEBUG(D_EXEC))
     fprint(STDOUT, "firing up job for target %s\n", wtos(j->t, ' '));
```

Uses DEBUG 161c, D_EXEC 161a, and wtos() 35b.

⟨sched() *if* DEBUG(D_EXEC) *print recipe* 164a⟩≡ (105e)

```
  if(DEBUG(D_EXEC))
      fprint(STDOUT, "recipe='%s'\n", j->r->recipe);
  Bflush(&bout);
```

Uses DEBUG 161c, D_EXEC 161a, and bout 47a.

⟨sched() *if* DEBUG(D_EXEC) *print pid* 164b⟩≡ (105e)

```
  if(DEBUG(D_EXEC))
      fprint(STDOUT, "pid for target %s = %d\n", wtos(j->t, ' '), events[slot].pid);
```

Uses DEBUG 161c, D_EXEC 161a, events-15 104d, and wtos() 35b.

⟨waitup() *if* DEBUG(D_EXEC) *print pid* 164c⟩≡ (109a)

```
  if(DEBUG(D_EXEC))
      fprint(STDOUT, "waitup got pid=%d, status='%s'\n", pid, buf);
```

Uses DEBUG 161c and D_EXEC 161a.

⟨waitup() *if* DEBUG(D_EXEC) *and slot ¡ 0* 164d⟩≡ (144c)

```
   if(DEBUG(D_EXEC))
       fprint(STDERR, "mk: wait returned unexpected process %d\n", pid);
```

Uses DEBUG 161c and D_EXEC 161a.

⟨pidslot() *if* DEBUG(D_EXEC) 164e⟩≡ (105d)

```
  if(DEBUG(D_EXEC))
      fprint(STDERR, "mk: wait returned unexpected process %d\n", pid);
```

Uses DEBUG 161c and D_EXEC 161a.

⟨nproc() *if* DEBUG(D_EXEC) 164f⟩≡ (104b)

```
  if(DEBUG(D_EXEC))
      fprint(STDERR, "nprocs = %d\n", nproclimit);
```

Uses DEBUG 161c, D_EXEC 161a, and nproclimit-18 103c.

⟨*dumper dumpj* 164g⟩≡ (185a)

```
  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 47a and wtos() 35b.

A.4 Tracing function calls: mk -dt

⟨applyrules *debug* 164h⟩≡ (79a)

```
  if(DEBUG(D_TRACE))
      print("applyrules(%lux='%s')\n", target, target);
```

Uses DEBUG 161c and D_TRACE 161a.

⟨newnode() *debug* 164i⟩≡ (42a)

```
  if(DEBUG(D_TRACE))
      print("newnode(%s), time = %d\n", name, node->time);
```

Uses DEBUG 161c and D_TRACE 161a.

⟨`work()` *debug* 165a⟩≡ (97a)

```
  if(DEBUG(D_TRACE))
      print("work(%s) flags=0x%x time=%lud\n", node->name, node->flags, node->time);
```

Uses `DEBUG` 161c and `D_TRACE` 161a.

⟨`update()` *debug* 165b⟩≡ (110a)

```
  if(DEBUG(D_TRACE))
      print("update(): node %s time=%lud flags=0x%x\n", node->name, node->time, node->flags);
```

Uses `DEBUG` 161c and `D_TRACE` 161a.

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 166a⟩≡ (185b)

```
short buf[10000];
```

⟨main() *setup profiling* 166b⟩+≡ (47c) ◁127e

```
#ifdef PROF
    {
        extern int etext();
        monitor(main, etext, buf, sizeof buf, 300);
    }
#endif
```

⟨*function* symstat 166c⟩≡ (177c)

```
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 47a, 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` 167a⟩≡ (177a)

```
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` 167b⟩≡ (177a)

```
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` 167c⟩≡ (174d)

```
struct Bufblock
{
    char    *start;
    char    *end;
```

```
    // between start and end
    char   *current;

    // Extra
    ⟨Bufblock extra fields 168a⟩
};
```

⟨Bufblock *extra fields* 168a⟩≡ (167c)

```
struct Bufblock *next;
```

Uses Bufblock 167c.

⟨*global* freelist 168b⟩≡ (177b)

```
static Bufblock *freelist;
```

⟨*constant* QUANTA 168c⟩≡ (177b)

```
#define QUANTA 4096
```

⟨*constructor* newbuf 168d⟩≡ (177b)

```
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() 167a, QUANTA-13 168c, and freelist-12 168b.

⟨*destructor* freebuf 168e⟩≡ (177b)

```
void
freebuf(Bufblock *p)
{
    p->next = freelist;
    freelist = p;
}
```

Uses freelist-12 168b.

⟨*macro* isempty 168f⟩≡ (172)

```
#define isempty(buf) (buf->current == buf->start)
```

⟨*macro* resetbuf 168g⟩≡ (172)

```
#define resetbuf(buf) do { buf->current = buf->start; } while(0)
```

⟨*macro* bufcontent 168h⟩≡ (172)

```
#define bufcontent(buf) buf->start
```

⟨*function* insert 169a⟩≡ (177b)
```
void
insert(Bufblock *buf, int c)
{

    if (buf->current >= buf->end)
        growbuf(buf);
    *buf->current++ = c;
}
```
Uses growbuf() 169d.

⟨*function* rinsert 169b⟩≡ (177b)
```
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() 169d.

⟨*function* bufcpy 169c⟩≡ (177b)
```
void
bufcpy(Bufblock *buf, char *cp, int n)
{

    while (n--)
        insert(buf, *cp++);
}
```
Uses insert() 169a.

⟨*function* growbuf 169d⟩≡ (177b)
```
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` 168c, `Realloc()` 167b, and `freelist-12` 168b.

C.3 File management

⟨*function* `maketmp` 170⟩≡ (191)

```
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 172⟩≡

```
// 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 168f⟩
⟨macro resetbuf 168g⟩
⟨macro bufcontent 168h⟩

// 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 174a⟩≡ (174d)

```
#define BIGBLOCK 20000
```

⟨*function* RERR 174b⟩≡ (174d)

```
//#define RERR(r)  (fprint(STDERR, "mk: %s:%d: rule error; ", (r)->file, (r)->line))
```

⟨*function* SEP 174c⟩≡ (174d)

```
//#define SEP(c)  (((c)==' ')||((c)=='\t')||((c)=='\n'))
```

⟨mk/mk.h 174d⟩≡

```
#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 167c⟩

⟨struct Word 34a⟩
```

```
// used by main and parse.c
extern Word *target1;


⟨struct Envy 114a⟩

extern ShellEnvVar *shellenv;

⟨struct Rule 36a⟩

extern Rule *rules, *metarules, *patrule;

⟨enum Rule_attr 37f⟩

⟨macro empty_recipe 80b⟩
⟨macro empty_prereqs 80c⟩
⟨macro empty_words 60b⟩

⟨constant NREGEXP 130b⟩

⟨struct Arc 42f⟩

⟨struct Node 41d⟩

⟨enum Node_flag 42c⟩
⟨function MADESET 96b⟩

⟨struct Job 44a⟩

extern Job *jobs;

⟨struct Symtab 30⟩

⟨enum Namespace 31a⟩

⟨type WaitupParam 111a⟩
⟨type WaitupResult 111b⟩

extern int debug;
extern bool nflag, tflag, iflag, kflag, aflag;
extern int mkinline;
extern char *infile;
extern bool explain;
extern int runerrs;

⟨function SYNERR 53d⟩
⟨function RERR 174b⟩
⟨constant NAMEBLOCK 81b⟩
⟨constant BIGBLOCK 174a⟩

⟨function SEP 174c⟩
⟨function WORDCHR 67a⟩

⟨enum Dxxx 161a⟩
⟨function DEBUG 161c⟩

⟨function PERCENT 37d⟩

//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` 42f, `Bufblock` 167c, `Job` 44a, `Node` 41d, `Rule` 36a, `Shell` 174d, `ShellEnvVar` 114a, `Symtab` 30, and `Word` 34a.

E.1.3 mk/globals.c

⟨mk/globals.c 176⟩≡
```
#include "mk.h"

// used by DEBUG which is used by many files
⟨global debug 161b⟩

⟨global infile 53a⟩
⟨global mkinline 53c⟩

⟨global rules 36c⟩
⟨global metarules 37b⟩
⟨global patrule 129d⟩

// was in main.c, but used also by parse.c
⟨global target1 50e⟩

⟨global nflag 125f⟩
⟨global tflag 153f⟩
```

```
⟨global iflag 151c⟩
⟨global kflag 157g⟩
⟨global aflag 156b⟩

⟨global explain 125b⟩

⟨global jobs 45a⟩

⟨global bout 47a⟩
```

E.1.4 mk/utils.c

```
⟨mk/utils.c 177a⟩≡
  #include "mk.h"

  ⟨function Malloc 167a⟩

  ⟨function Realloc 167b⟩

  // maketmp() is back in Plan9.c
```

E.1.5 mk/bufblock.c

```
⟨mk/bufblock.c 177b⟩≡
  #include "mk.h"

  ⟨global freelist 168b⟩
  ⟨constant QUANTA 168c⟩

  ⟨constructor newbuf 168d⟩

  ⟨destructor freebuf 168e⟩

  ⟨function growbuf 169d⟩

  ⟨function bufcpy 169c⟩

  ⟨function insert 169a⟩

  ⟨function rinsert 169b⟩
```

E.1.6 mk/symtab.c

```
⟨mk/symtab.c 177c⟩≡
  #include "mk.h"

  ⟨constant NHASH 31c⟩
  ⟨constant HASHMUL 32c⟩
  ⟨global hash 31b⟩

  ⟨function symlook 31e⟩

  ⟨function symtraverse 32e⟩

  ⟨function symstat 166c⟩
```

E.1.7 mk/rc.c

⟨mk/rc.c 178a⟩≡
```
#include "mk.h"

⟨global termchars 77b⟩
⟨global shflags 107c⟩

/*
 * This file contains functions that depend on rc's syntax.  Most
 * of the routines extract strings observing rc's escape conventions
 */

⟨function squote 61b⟩

⟨function charin 60c⟩

⟨function expandquote 64c⟩

⟨function escapetoken 58b⟩

⟨function copysingle 125a⟩

⟨function copyq 124d⟩
```

E.1.8 mk/word.c

⟨mk/word.c 178b⟩≡
```
#include "mk.h"

⟨constructor newword 34c⟩


⟨function wtos 35b⟩

⟨function wdup 35a⟩

⟨destructor freewords 34d⟩

// was in recipe.c before
⟨function addw 34e⟩
```

E.1.9 mk/var.c

⟨mk/var.c 178c⟩≡
```
#include "mk.h"

⟨function setvar 33a⟩

⟨function shname 117c⟩
```

E.1.10 mk/archive.c

⟨mk/archive.c 178d⟩≡
```
#include "mk.h"
#include <ar.h>
```

```
static void atimes(char *);
static char *split(char*, char**);

⟨function atimeof 148e⟩

⟨function atouch 149a⟩

⟨function atimes 149b⟩

⟨function type 150b⟩

⟨function split 151a⟩
```

E.1.11 mk/match.c

```
⟨mk/match.c 179a⟩≡
  #include "mk.h"

  ⟨function match 82a⟩

  ⟨function subst 82c⟩
```

E.1.12 mk/env.c

```
⟨mk/env.c 179b⟩≡
  #include "mk.h"

  ⟨constant ENVQUANTA 115c⟩

  ⟨global shellenv 114b⟩
  ⟨global nextv 114c⟩

  ⟨global specialvars 33c⟩

  // encodenulls() is back in Plan9.c
  // readenv() is back in Plan9.c
  extern void readenv(void);
  // exportenv() is back in plan9.c

  ⟨function inithash 33d⟩

  ⟨function envinsert 114d⟩

  ⟨function envupd 115d⟩

  ⟨function ecopy 116b⟩

  ⟨function initenv 116a⟩

  ⟨function buildenv 119b⟩
```

E.1.13 mk/parse.c

```
⟨mk/parse.c 179c⟩≡
  #include       "mk.h"
```

```
void    ipop(void);
void    ipush(void);
static int   rhead(char *, Word **, Word **, int *, char **);
static char* rbody(Biobuf*);

⟨function parse 53e⟩

⟨function addrules 41a⟩

⟨function rhead 59b⟩

⟨function rbody 71a⟩

⟨struct input 74e⟩
⟨global inputs 74f⟩

⟨function ipush 74h⟩

⟨function ipop 75a⟩
```

E.1.14 mk/shprint.c

⟨mk/shprint.c 180a⟩≡

```
#include "mk.h"

static char *vexpand(char*, ShellEnvVar*, Bufblock*);
static char *shquote(char*, Rune, Bufblock*);
static char *shbquote(char*, Bufblock*);

⟨function shprint 123a⟩

⟨function mygetenv 124a⟩

⟨function vexpand 123b⟩

⟨function front 122e⟩
```

E.1.15 mk/job.c

⟨mk/job.c 180b⟩≡

```
#include "mk.h"
```

E.1.16 mk/arc.c

⟨mk/arc.c 180c⟩≡

```
#include "mk.h"
```

E.1.17 mk/rule.c

⟨mk/rule.c 180d⟩≡

```
#include "mk.h"

⟨global lr 37a⟩
```

```
⟨global lmr 37c⟩

⟨global nrules 87c⟩

static int rcmp(Rule *r, char *target, Word *tail);


⟨function addrule 38a⟩


⟨function rcmp 40c⟩

⟨function rulecnt 88b⟩

⟨function regerror 130a⟩
```

E.1.18 mk/lex.c

⟨mk/lex.c 181a⟩≡

```
#include "mk.h"

static int bquote(Biobuf*, Bufblock*);

⟨function assline 55b⟩

⟨function bquote 132a⟩

⟨function nextrune 56a⟩
```

E.1.19 mk/file.c

⟨mk/file.c 181b⟩≡

```
#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 85a⟩

⟨function touch 154a⟩

⟨function delete 141a⟩

⟨function timeinit 126g⟩
```

E.1.20 mk/run.c

⟨mk/run.c 181c⟩≡

```
#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 104c⟩

⟨global events 104d⟩
⟨global nevents 105a⟩
⟨global nrunning 103b⟩
⟨global nproclimit 103c⟩

⟨struct Process 143c⟩
⟨global phead 143d⟩
⟨global pfree 143e⟩

⟨function run 103a⟩

// shell is back in Plan9.c
// shellname is back in Plan9.c

⟨function sched 105e⟩

// execsh() is back in Plan9.c
// xwaitfor() is back in Plan9.c
extern int xwaitfor(char *msg);

⟨function waitup 109a⟩

⟨function nproc 104b⟩

⟨function nextslot 105c⟩

⟨function pidslot 105d⟩


⟨function pnew 144a⟩

⟨function pdelete 144b⟩

// 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 113b⟩

⟨global tslot 127c⟩
⟨global tick 127d⟩

⟨function usage 128a⟩

⟨function prusage 128c⟩
```

```
  // pipecmd() is back in Plan9.c
```
Uses `Process` 143c and `RunEvent` 104c.

E.1.21 mk/graph.c

⟨mk/graph.c 183a⟩≡
```
  #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 88d⟩

  ⟨function graph 78⟩

  ⟨function applyrules 79a⟩

  ⟨function nrep 147a⟩

  ⟨function togo 91d⟩

  ⟨function vacuous 92h⟩

  ⟨constructor newnode 42a⟩

  // rcopy() is back in Plan9.c
  extern void rcopy(char **to, Resub *match, int n);

  ⟨constructor newarc 43c⟩


  ⟨function trace 90a⟩

  ⟨function cyclechk 86b⟩

  ⟨function ambiguous 89⟩

  ⟨function attribute 139b⟩
```

E.1.22 mk/mk.c

⟨mk/mk.c 183b⟩≡
```
  #include "mk.h"

  void clrmade(Node*);
  void work(Node*, bool*, Node*, Arc*);

  ⟨global runerrs 158a⟩

  ⟨function mk 94⟩
```

⟨*function* clrmade 96a⟩

⟨*function* unpretend 153a⟩

⟨*function* work 97a⟩

⟨*function* update 110a⟩

⟨*function* pcmp 143b⟩

⟨*function* outofdate 99a⟩

E.1.23 mk/recipe.c

⟨mk/recipe.c 184a⟩≡

```
#include "mk.h"
```

⟨*constructor* newjob 44c⟩

⟨*function* dorecipe 100a⟩

E.1.24 mk/varsub.c

⟨mk/varsub.c 184b⟩≡

```
#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 65d⟩

⟨*function* varname 66d⟩

⟨*function* varmatch 67c⟩

⟨*function* expandvar 136a⟩

⟨*function* extractpat 138a⟩

⟨*function* subsub 136b⟩

⟨*function* submatch 138b⟩

⟨*function* nextword 63b⟩

⟨*function* stow 62b⟩

E.1.25 mk/dumpers.c

```
⟨mk/dumpers.c 185a⟩≡
  #include "mk.h"

  void dumpa(char*, Arc*);

  ⟨dumper dumpn 163b⟩

  ⟨dumper dumpa 163c⟩

  ⟨dumper dumpj 164g⟩

  ⟨function print1 162e⟩

  ⟨dumper dumpv 162d⟩

  ⟨dumper dumpr 162c⟩

  ⟨dumper dumpw 162b⟩
```

E.1.26 mk/main.c

```
⟨mk/main.c 185b⟩≡
  #include "mk.h"

  ⟨constant MKFILE 50c⟩

  ⟨global version 46a⟩

  // see also globals.c

  ⟨global uflag 127a⟩

  void badusage(void);

  #ifdef PROF
  ⟨global buf 166a⟩
  #endif

  ⟨function main 46b⟩

  ⟨function badusage 48a⟩
```

E.1.27 mk/Posix.c

```
⟨mk/Posix.c 185c⟩≡

  // 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 161c, D_EXEC 161a, IWS 185c, Realloc() 167b, S_INTERNAL 33b, ShellEnvVar 114a, __anon_struct_1 185c, growbuf() 169d, killchildren() 113b, newword() 34c, setvar() 33a, shflags 107c, shname() 117c, sigmsgs-5 185c, stab 185c, symlook() 31e, and wtos() 35b.

E.1.28 mk/Plan9.c

⟨mk/Plan9.c 191⟩≡

```
  #include "mk.h"

  // could be in utils.c
  ⟨function maketmp 170⟩

  // could be in env.c
  ⟨function encodenulls 118d⟩
```

```
⟨function readenv 117a⟩
⟨function exportenv 120a⟩

// could be in file.c
⟨function chgtime 154b⟩
⟨function dirtime 156a⟩
⟨function bulkmtime 155f⟩
⟨function mkmtime 85b⟩

// could be in run.c
Shell rc = {
  .shellname = "rc",
  .shell = "/bin/rc",
};

Shell* shell = &rc;

⟨global shell 107a⟩
⟨global shellname 107b⟩

// could be in run.c
⟨function execsh 107d⟩
⟨function waitfor 109b⟩

// could be in run.c
⟨function Exit 112b⟩

// back in run.c
extern void killchildren(char *msg);

⟨function notifyf 112e⟩
⟨function catchnotes 112d⟩
⟨function expunge 145d⟩

// could be in run.c
⟨function pipecmd 134g⟩

// could be in graph.c
⟨function rcopy 130e⟩
```

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(): 38a, 41a
addrules(): 41a, 51e, 69c
addw(): 34e, 102b, 146g
aflag: 146a, 156b, 156b, 156c, 156d
ambiguous(): 85d, 89, 89
applyrules(): 78, 79a, 79c, 80e
Arc: 42f, 43a, 174d
Arc.match: 130c
Arc.n: 42f
Arc.next: 43a
Arc.prog: 142g
Arc.r: 42f
Arc.remove: 91a
Arc.stem: 43b
Arc (typedef): 174d
assline(): 53e, 55b
atimeof(): 148d, 148e
atimes(): 148e, 149b
atouch(): 149a, 154a
attribute(): 139a, 139b, 139b
badusage(): 47d, 48a, 50a, 126e
BEINGMADE: 42d, 94, 97a, 98c, 100a, 158c, 158e, 158f
BIGBLOCK: 101a, 149b, 174a
bout: 47a, 47b, 94, 113b, 122b, 143b, 150b, 152b, 153h, 153i, 154a, 162b, 162c, 162d, 162e, 163a, 163b, 163c, 164a, 164g, 166c
bquote(): 131g, 132a
Bufblock: 167c, 168a, 174d
Bufblock.current: 167c
Bufblock.end: 167c
Bufblock.next: 168a
Bufblock.start: 167c
Bufblock (typedef): 174d
bufcontent: 55b, 168h
bufcpy(): 67d, 67e, 69a, 123b, 126e, 157b, 157c, 157f, 169c
buildenv(): 105e, 119b, 122d
bulkmtime(): 155f
CANPRETEND: 151e, 152a, 152b, 153a, 153e
catchnotes(): 185c

charin(): 38c, 60c, 69d, 136a, 138a
chgtime(): 185c
clrmade(): 94, 96a, 96a
copyq(): 123a, 124d
copysingle(): 124d, 125a
CYCLE: 86a, 86b
cyclechk(): 85d, 86b, 86b
DEBUG: 134g, 161c, 162a, 163a, 163e, 164a, 164b, 164c, 164d, 164e, 164f, 164h, 164i, 165a, 165b, 185c
debug: 161b, 161e
DEL: 140c, 140f
DELETE: 140e, 140f, 140h
delete(): 140h, 141a
dirtime(): 155f, 156a
dorecipe(): 98c, 100a
dumpa(): 163b, 163c
dumpj(): 164g
dumpn(): 163a, 163b, 163c
dumpr(): 162a, 162c
dumpv(): 162a, 162d
dumpw(): 162a, 162b
Dxxx: 161a
D_EXEC: 134g, 161a, 161e, 163e, 164a, 164b, 164c, 164d, 164e, 164f, 185c
D_GRAPH: 161a, 161e, 163a
D_PARSE: 161a, 161e, 162a
D_TRACE: 161a, 164h, 164i, 165a, 165b
ecopy(): 116a, 116b
EMPTY_CHILDREN_IS_ERROR2: 158g, 158h
EMPTY_CHILDREN_IS_ERROR3: 134a, 134c, 143b
EMPTY_CHILDREN_IS_ERROR: 94, 111a
EMPTY_CHILDREN_IS_OK: 94, 111a, 111c, 113b
EMPTY_CHILDREN: 111b, 111c
empty_prereqs: 80a, 80c, 80d, 81d, 81e
empty_recipe: 80a, 80b, 81d, 89, 100a
empty_words: 60b, 104b, 120d
envinsert(): 114d, 115e, 116a, 116b
ENVQUANTA-6: 115b, 115c
envupd(): 115d, 119b, 129g, 131b, 146e, 146k, 147d, 147f, 148a
Envy (typedef): 185c
escapetoken(): 58a, 58b, 132b
events-15: 104d, 105b, 105c, 105d, 105e, 109a, 164b
execsh(): 185c
Exit(): 185c
expandquote(): 64b, 64c
expandvar(): 135c, 136a
explain: 125b, 125b, 125c, 125d, 125e, 152b, 152c, 153h, 153i
exportenv(): 185c
expunge(): 185c
extractpat(): 138a
freebuf(): 35b, 53e, 63b, 65d, 67b, 71a, 111e, 122b, 126f, 136a, 157f, 168e

freelist-12: 168b, 168d, 168e, 169d
freewords(): 34d, 115d, 121a
front(): 122d, 122e
graph(): 78, 94
growbuf(): 133g, 169a, 169b, 169d, 185c
hash-3: 31b, 31e, 32d, 32e, 166c
HASHMUL-2: 32b, 32c
iflag: 151c, 151c, 151d, 152b, 156c
infile: 53a, 53b, 53e, 74h, 75a, 130a
initenv(): 115f, 116a, 132d, 134c
inithash(): 33d, 47c
Input: 74e, 74f, 74g, 74h, 75a
Input.file: 74e
Input.line: 74e
Input.next: 74g
inputs-14: 74f, 74f, 74h, 75a
insert(): 35b, 55b, 57c, 63b, 66d, 67d, 67e, 69a, 71a, 123a, 126e, 126f, 132a, 132b, 157b, 157c, 157e, 157f, 169a, 169c
ipop(): 74d, 75a
ipush(): 74c, 74h
isempty: 55b, 57c, 63b, 67b, 67d, 168f
IWS: 185c, 185c
Job: 44a, 45b, 174d
Job.at: 146c
Job.match: 130f
Job.n: 44a
Job.next: 45b
Job.np: 146i
Job.p: 44a
Job.r: 44a
Job.stem: 44a
Job.t: 44a
Job (typedef): 174d
jobs: 45a, 103a, 105e, 106, 111e, 113b, 158h
JOB_ENDED: 109a, 111b, 113b
kflag: 113b, 157g, 157g, 157h, 158c, 158d
killchildren(): 112e, 113b, 185c
lmr-24: 37c, 38c
lr-23: 37a, 38b
MADE: 42d, 97b, 98c, 110a, 126c, 148c, 152b, 152c, 158h
MADESET: 96a, 96b, 97b, 98c, 100a, 110a, 126c, 148c, 152b, 153a, 158e
main-11(): 46b
maketmp(): 185c
Malloc(): 32d, 34c, 38a, 42a, 43c, 44c, 74h, 88b, 105b, 117a, 144a, 167a, 168d
membername(): 185c
META: 37f, 38c, 90d, 92h, 101b
metarules: 37b, 38c, 80e, 162a
mk(): 51a, 51d, 51e, 94
MKFILE-9: 50b, 50c

mkinline: 51e, 53c, 53e, 56b, 56c, 57b, 58b, 71b, 74h, 75a, 130a, 132a
mkmtime(): 185c
mygetenv(): 123b, 124a
NAMEBLOCK: 81a, 81b, 81c, 142m, 143b
Namespace: 31a
nevents-16: 105a, 105b, 105d, 128c
newarc(): 43c, 79c, 80d, 80e, 81e
newbuf(): 35b, 53e, 63b, 66d, 71a, 111e, 122b, 126e, 157a, 168d
newjob(): 44c, 100a
newnode(): 42a, 79a
newword(): 34c, 34e, 35a, 51e, 63a, 63b, 101b, 119b, 129f, 129g, 131b, 136a, 146a, 147d, 147f, 185c
nextrune(): 55b, 56a, 58b, 132a
nextslot(): 105c, 105e
nextv-7: 114c, 114d, 115b, 116a
nextword(): 62b, 63b
nflag: 122b, 125f, 125f, 126a, 126c, 154a
NHASH-1: 31b, 31c, 32b, 32e, 166c
Node: 41d, 42f, 44b, 174d
Node.arcs: 42e
Node.flags: 42b
Node.name: 41d
Node.next: 44b
Node.time: 41d
Node (typedef): 174d
Node_flag: 42c
NOMINUSE: 141d, 141f
NOPLAN9DEFINES-4: 185c
NOREC: 141g, 141j
NORECIPE: 140a, 141i, 141j
notifyf(): 185c
NOTMADE: 42d, 94, 96a, 98c, 153a, 158f
NOT_A_JOB_PROCESS: 111b
NOVIRT: 142a, 142c
nproc(): 104a, 104b
nproclimit-18: 103a, 103c, 104b, 105b, 105c, 109a, 164f
NREGEXP: 130b, 130c, 130d, 131c, 131e, 131f
nreps: 88d, 88d, 88e
nrules-25: 87c, 87c, 87d, 88b
nrunning-17: 103a, 103b, 105e, 109a, 128a
outofdate(): 98c, 99a, 102b, 140b, 142l, 146a, 152b, 152c
parse(): 48c, 50b, 53e, 73c, 134c
patrule: 129d, 129e, 130a, 131e
pcmp(): 143a, 143b
pdelete(): 144b, 144e
PERCENT: 37d, 82c
pfree-20: 143e, 144a, 144b
phead-19: 143d, 144a, 144b, 144e, 145c
pidslot(): 105d, 109a
pipecmd(): 185c

pnew(): 144a, 144c
PRETENDING: 151e, 152a, 152b, 152c, 153a, 153e
print1(): 162d, 162e
PROBABLE: 92a, 92b, 92c, 92d, 92h
Process: 143c, 181c
Process.b: 143c
Process.f: 143c
Process.pid: 143c
Process.status: 143c
Process (typedef): 181c
prusage(): 128b, 128c
QUANTA-13: 168c, 168d, 169d
QUIET: 122b, 141b
rbody(): 69c, 71a
rc: 185c
rcopy(): 185c
readenv(): 185c
READY: 92g, 92h
Realloc(): 105b, 115b, 167b, 169d, 185c
regerror(): 130a
REGEXP: 38c, 69d, 129a, 129e, 129f, 129g, 131b, 131e, 131f
resetbuf: 55b, 67d, 67e, 168g
rinsert(): 55b, 58a, 58b, 63b, 64c, 66d, 71a, 72a, 123a, 124d, 125a, 132a, 169b
Rule: 36a, 36d, 39b, 174d
Rule.alltargets: 41b
Rule.attr: 37e
Rule.chain: 39b
Rule.file: 36b
Rule.line: 36b
Rule.next: 36d
Rule.pat: 129c
Rule.prereqs: 36a
Rule.prog: 142d
Rule.recipe: 36a
Rule.rule: 87b
Rule.target: 36a
Rule (typedef): 174d
rulecnt(): 88a, 88b
rules: 36c, 38b, 162a
Rule_attr: 37f
run(): 100a, 103a
runerrs: 158a, 158b, 158c, 158d, 158f, 158h
RunEvent: 104c, 181c
RunEvent.job: 104c
RunEvent.pid: 104c
RunEvent (typedef): 181c
sched(): 103a, 105e, 109a
setvar(): 33a, 75c, 117a, 185c
Shell: 174d, 174d

shell: 185c
Shell.shell: 174d
Shell.shellname: 174d
Shell (typedef): 174d
shellenv: 114b, 114d, 115b, 115d, 119b, 132d, 134c
ShellEnvVar: 114a, 174d, 185c
ShellEnvVar.name: 114a
ShellEnvVar.values: 114a
ShellEnvVar (typedef): 174d
shflags: 107c, 107c, 107d, 134g, 185c
shname(): 117b, 117c, 123b, 185c
shprint(): 122b, 122d, 123a
sigmsgs-5: 185c, 185c
specialvars-8: 33c, 33d, 116a, 116c, 131b
split(): 148e, 149a, 151a
squote(): 61a, 61b
stab: 185c, 185c
stow(): 62b, 116a, 138a, 157e, 157f
submatch(): 138b
subst(): 80e, 82c, 101b
symlook(): 31e, 33a, 33d, 39d, 40b, 46b, 76c, 79c, 84b, 84c, 101b, 104b, 109a, 117b, 120d, 124a, 124c, 126g, 136a, 143a, 145f, 145h, 148e, 149a, 149b, 150b, 154g, 154h, 155f, 156a, 157e, 157f, 185c
symstat(): 166c
Symtab: 30, 31d, 174d
Symtab.name: 30
Symtab.next: 31d
Symtab.space: 30
Symtab.u: 30
Symtab (typedef): 174d
symtraverse(): 32e, 116a, 162d
SYNERR: 53d, 55a, 58b, 61b, 67b, 73d, 76a, 132a, 134d, 135b, 136a
S_AGG: 148b, 148e
S_BITCH: 150a, 150b
S_BULKED: 155e, 155f
S_INTERNAL: 33b, 33d, 117b, 124a, 185c
S_NODE: 84a, 84b, 84c, 101b, 109a
S_NOEXPORT: 145f, 145g, 145h
S_OUTOFDATE: 142k, 143a
S_OVERRIDE: 76c, 76d
S_TARGET: 39a, 39d, 40b, 79c
S_TIME: 126g, 148e, 149a, 149b, 154c, 154g, 154h, 156a
S_VAR: 31a, 33a, 46b, 104b, 116a, 120d, 136a, 157e, 157f, 162d
S_WESET: 124a, 124b, 124c
target1: 50e, 51a, 69d, 162a
termchars: 77b, 77b
tflag: 122b, 126c, 153f, 153f, 153g, 153h, 153i
tick-22: 127d, 128a
timeinit(): 126f, 126g
timeof(): 42a, 85a, 110a

togo(): 91c, 91d, 92h
touch(): 153h, 153i, 154a
trace(): 89, 90a
tslot-21: 127c, 128a, 128c
type(): 150b, 151a
uflag: 127a, 127a, 127b, 128b
unpretend(): 152b, 152c, 153a
update(): 109a, 110a, 140a
usage(): 105e, 106, 109a, 127e, 127f, 128a, 128c
VACUOUS: 92f, 92h
vacuous(): 92e, 92h, 92h
varname(): 65d, 66d, 136a
varsub(): 65c, 65d
version-10: 46a, 46a
vexpand(): 123a, 123b
VIR: 51e, 139c, 139f, 142c
VIRTUAL: 139e, 139f, 139g, 140a, 140b, 153h, 153i
waitup(): 94, 109a, 113b, 134c, 143b, 158h
WaitupParam: 111a
WaitupResult: 111b
wdup(): 35a, 69d, 119b, 146e, 146k, 148a
Word: 34a, 34b, 174d
Word.next: 34b
Word.s: 34a
Word (typedef): 174d
WORDCHR: 66d, 67a, 117c
work(): 94, 97a, 98c, 152b
wtos(): 35b, 73c, 124a, 134c, 162c, 163e, 164b, 164g, 185c
xwaitfor(): 185c
__anon_struct_1.msg: 185c
__anon_struct_1.sig: 185c
__anon_struct_1: 185c, 185c
__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) 11

[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) 8, 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) 11

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 108, 109, 117, 160

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 160

[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, 54, 56, 160

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 32, 47, 49, 54, 61, 108, 109, 110

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 13, 15, 16, 58, 106, 107, 159, 160

[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