cook a file construction tool

Peter Miller

ABSTRACT

This document describes **cook**, a maintenance tool designed to construct files. **Cook** may be used to maintain consistency between executable files and the associated source files that are used to generate them. The consistency is designated by the relative last-modified times of files and is thus automatically adjusted each time a file is edited, compiled or otherwise modified. **Cook** validates the consistency of a system of files and executes all commands necessary to maintain that consistency.

Page 2

1. Introduction

Cook is a tool for constructing files. It is given a set of files to create, and instructions detailing how to construct them. In any non-trivial program there will be prerequisites to performing the actions necessary to creating any file, such as extraction from a source-control system. **Cook** provides a mechanism to define these.

When a program is being developed or maintained, the programmer will typically change one file of several which comprise the program. **Cook** examines the last-modified times of the files to see when the prerequisites of a file have changed, implying that the file needs to be recreated as it is logically out of date.

Cook also provides a facility for implicit recipes, allowing users to specify how to form a file with a given suffix from a file with a different suffix. For example, to create *filename***.o** from *filename***.c**

1.1 How to Use this Manual

This manual is divided into two parts.

The first part is tutorial introduction to **cook**. This part runs from chapter 4 to chapter 5.

The second part is for reference and details precisely how **cook** works. This part runs from chapter 6 to chapter 14.

Users familiar with other programs similar to **cook** are advised to skim the tutorial part before diving into the reference part.

2. *Ancient History*

Cook was originally developed because I was marooned on an operating system without anything even vaguely resembling make. This was in 1988. Since I had to write my own, I added a few improvements. When I finally escaped back to UNIX, it took only two days to port **cook** to SystemV. I have since deleted all code for that original operating system, although clues to its identity are still present.

After I had **cook** up on UNIX, the progress the world had made caught up with me. It was gratifying that many of the features other make-oid authors had thought necessary were either already present, or easily and seamlessly added.

Cook was written with portability in mind. This does not means it is entirely portable, unfortunately. **Cook** has been tested on SystemV R2 and SystemV R3 and SunOS R4. It should also be portable to others.

If you have any trouble getting **cook** working, please e-mail me so I know where I went wrong.

3. License

4. Cook from the Outside

This chapter is part of the tutorial on how to use the **cook** program. It focuses on how to use **cook**, without needing to know how **cook** works internally.

4.1 What can cook do for me?

By far the most common use of cook, by experts and beginners alike, is to issue the command

```
cook
```

and cook will consult its cookbook to see what needs to be done.

In general, **cook** is used to take a set of files and chew on them in some way to produce another set of files; such as the source files for a program, and how to turn them into the executable program file. In order for **cook** to do anything useful, it nees to know what to do. "What to do" is contained in a file called *Howto.cook* in the same directory as the files it is going to work on. You need to execute the `cook` command in the same directory as all of the files.

4.2 What is cook doing?

The *Howto.cook* file was written by the same person who wrote the source files. It contains a set of recipes; each of which, among other things, contain commands for how to manipulate the files. The **cook** program echos each of the commands it is about to execute, so that you can watch what it is doing as it goes.

If the *Howto.cook* file contained only commands, you would be better off using a shell script. In addition to the commands is information telling **cook** which files need to be constructed before other files can be, and from this information **cook** determines the order in which to execute the commands. Also, **cook** examines other information to determine which commands it need not do, because the associated files are already up-to-date.

4.3 What can cook always do?

If you are in a directory with a *Howto.cook* file, you can expect a few common requests to work

cook clobber	This command can be expected to remove any files from the directory which **cook** is able to reconstruct.
cook all	This is the default action, and so can be obtained by a simple `cook` request. It causes **cook** to construct some specific file or set of files.
cook clean	This is similar to "cook clobber" above, but it only removes intermediate files, and not not the final file or files which "cook all" constructs.

In addition to the above, many *Howto.cook* files will also define

cook install	If a program or library or document is constructed in the directory, the this command will install it into the correct place in the system.
cook uninstall	The reverse of the above, it removes something from the system.

4.4 If something goes wrong

Most errors while **cook** is constructing file are caused by errors in the source files, and not the *Howto.cook* file. In general, you can fix the problems in the source files, and execute the **cook** command again, and **cook** will resume from the command which incurred the error.

To help you while editing the files with the errors, **cook** keeps a listing file of all the commands it executed, and any output of those commands, in a file called *Howto.list* in the current directory.

You may want **cook** to find all the errors it can before you do any editing, do do this, use the **-Continue** option (it may be abbreviated to **-c** for convenience).

5. *Cook from a Cookbook*

This chapter describes the contents and meaning of a cookbook, a file which contains information **cook** needs to do its job. It focuses on what a cookbook looks like, and touches on a few areas of how **cook** works does its job.

5.1 What does Cook do?

The basic building block for **cook** is the concept of a *recipe*. A recipe has three parts:

1. one or more files which the recipe constructs, known as the *targets* of the recipe
2. zero or more files which are used by the recipe to construct the target, known as the *ingredients* of the recipe
3. one or more commands to execute which construct the targets from the ingredients, known as the *body* of the recipe.

When a number of recipes are given, some recipes may describe how to cook the ingredients of other recipes. When **cook** is asked to construct a particular target it automatically determines the correct order to perform the recipe bodies to cook the requested target.

Cook would not be especially useful if you had to give explicit recipes for how to cook every little thing. As a result, **cook** has the concept of an *implicit* recipe. An implicit recipe is very similar to an explicit recipe, except that the targets and ingredients of the recipe are *patterns* to be matched to file names, rather than explicit file names. This means it is possible to write a recipe, for example which constructs a files with a name ending in '**.o**' from a file of the same name, but ending in '**.c**' rather than '**.o**'.

In addition to recipes, **cook** needs to know *when* to construct targets from ingredients. **Cook** has been designed to cook as little as possible. "As little as possible" is determined by examining when each file was last modified, and only constructing targets when that are out of date with the ingredients.

5.1.1 When is Cook useful?

From the above description, **cook** may be described as a tool for maintaining consistency of sets of files.

5.1.2 When is Cook not useful?

Cook is not useful for maintaining consistency of sets of things which are *within* files and thus **cook** is unable to determine when they were modified. For example, **cook** is not useful for maintaining consistency of sets of records within a database.

5.2 How do I tell Cook what to do?

Sets of recipes are gathered together into cookbooks. When **cook** is executed it looks for a cookbook of the name *Howto.cook* in the current directory. If you did not name a file to be constructed on the command line, the first target in the cookbook will be constructed.

The best way to understand how to write recipes is an example. In this example, a program, *prog*, is composed of three files: *foo.c*, *bar.c* and *baz.c*. To inform **cook** of this, the cookbook

```
#include "c"

prog: foo.o bar.o baz.o
     {
        cc -o prog foo.o bar.o baz.o;
     }
```

is sufficient for *prog* to be constructed.

This cookbook has two parts. The line

```
#include "c"
```

tells **cook** to refer to a system cookbook which tells it, among other things, how to construct a *something***.o** file from a *something*.**c** file.

The second part is a recipe. The first line of this recipe

```
prog: foo.o bar.o baz.o
        ...
```

names the target, *prog*, and the ingredients, *foo.o*, *bar.o* and *baz.o*.

The next three lines

```
...
        {
           cc -o prog foo.o bar.o baz.o;
        }
```

are the recipe body, which consists of a single *cc*(1) command to be executed. Recipe bodies are always within **{** curly braces **}**, and commands always end with a semicolon (**;**).

Thus, to update *prog* after any of the source files have been edited, it is only necessary to issue the command

```
cook prog
```

This could be simplified further, because **cook** will cook the targets of the first recipe by default; in this case, *prog*.

The power of cook becomes more apparent when include files are considered. If the files *foo.c* and *baz.c* include the file *defs.h*, this would automatically be detected by **cook**. If *defs.h* were to be edited, and **cook** re-executed, this would cause **cook** to recompile both *foo.c* and *baz.c*, and relink *prog.*

5.2.1 The common program case

The above example may be simplified even further. If the four files *foo.c*, *bar.c*, *baz.c* and *defs.h* all resided in a directory with a path of */some/where/prog*, the the *Howto.cook* file in that directory need only contain

```
#include "c"
#include "program"
```

for *prog* to be cooked. This is because the "`program`" cookbook looks for all of the *something*.**c** files in the current directory, compiles them all, and links them into a program named after the current directory.

The default target in the "`program`" cookbook is called *all*. The ingredient of *all* is the program named after the current directory. Two other targets are supplied by this cookbook:

clean removes all of the *something*.**o** files from the current directory.

clobber removes the program named after the current directory, and also removes all of the *something*.**o** files from the current directory.

5.3 Creating a Cookbook

To use **cook** you will usually need to define a cookbook, by creating a file, usually called *Howto.cook* in the current directory, with your favorite text editor.

This file has a specific format. The format has been designed to be easy to learn, even for the casual user. Much of the power of **cook** is contained in how it works, without complicating the format of the cookbook.

Example of what a cookbook looks like are scattered throughout this document. The following example is the entire cookbook for **cook** itself.

```
#include "c"
#include "yacc"
#include "usr.local"
#include "program"
```

As you can see, even for a complex program like **cook** the cookbook is remarkably simple.

6. The Command Line

Cook may be invoked from the command line by a command of the form

> **cook** [*option*...] [*filename*...]

Options and filenames may be arbitrarily mixed on the command line; no processing is done until all options and filenames on the command line have been scanned.

Cook will attempt to create the named files from the recipes given to it. The recipes are contained in a file called *Howto.cook* in the current directory. This file may, in turn, include other files containing additional recipes.

If no *filename*s are given on the command line the targets of the first recipe defined are cooked.

6.1 Options

The valid options for **cook** are listed below. Any other options (words on the command line beginning with '**-**') will cause a diagnostic message to be issued.

-Action Execute the commands given in the recipes. This is the default.

-NoAction Do not execute the commands given in the recipes.

-Continue If cooking a target should fail, continue with other recipes for which the failed target is not an ingredient, directly or indirectly.

-NoContinue If cooking a target should fail, **cook** will exit. This is the default.

-Errok When a command is executed, the exit code will be ignored.

-NoErrok When a command is executed, if the exit code is positive it will be deemed to fail, and thus the recipe containing it to have failed. This is the default.

-Force Always perform the actions of recipes, irrespective of the last-modified times of any of the ingredients.

This option is useful if something beyond the scope of the cookbook has been modified; for example, a bug fix in a compiler.

-NoForce Perform the actions of the recipes if any of the ingredients are logically out of date. This is the default.

-Help Provide information about how to execute **cook** on *stdout*, and perform no other function.

-Include *filename*
Search the named directory before the standard places for included cookbooks. Each directory so named will be scanned in the order given. The standard places are *$HOME/.cook* then */usr/local/lib/cook*.

-List Causes **cook** to automatically redirect the *stdout* and *stderr* of the session. Output will continue to come to the terminal, unless **cook** is executing in the background. The name of the file will be the name of the cookbook with any suffix removed and "`.list`" appended; this will usually be *Howto.list*. This is the default.

-List *filename* Causes **cook** to automatically redirect the *stdout* and *stderr* of the session into the named file. Output will continue to come to the terminal, unless **cook** is executing in the background.

-NoList No automatic redirection of the session of the session will be made.

-NoList *filename*
No automatic redirection of the session of the session will be made, however subsequent **-List** options will default to listing to the named file.

-Meter After each command is executed, print a summary of the command's CPU usage.

-NoMeter Do not print a CPU usage summary after each command. This is the default.

-Precious When commands in the body of a recipe fail, no not delete the targets of the recipe.

-NoPrecious When commands in the body of a recipe fail, delete the targets of the recipe. This is the default.

-Silent Do not echo commands before they are executed.

-NoSilent Echo commands before they are executed. This is the default.

-Touch Update the last-modified times of the target files, rather than execute the actions bound to recipes.

This can be useful if you have made a modification to a file that you know will make a system of files logically out of date, but has no significance; for example, adding a comment to a widely used include file.

-NoTouch Execute the actions bound to recipes, rather than update the last-modified times of the target files. This is the default.

-TTy When listing, also send the output stream to the terminal. This is the default.

-NoTTy When listing, do not send the output to the terminal.

name=*value* Assign the *value* to the named variable. The value may contain spaces if you can convince the shell to pass them through.

In the above descriptions, a shorter form is indicated by the uppercase letters; for example, the **-NoTouch** option may be abbreviated to **-nt**. **Cook** is case insensitive to the options, so you may arbitrarily mix cases within the options.

Two options are provided for tracing the inferences **cook** makes when attempting to cook a target.

-TRace **Cook** will emit copious amounts of information about the inferences it is making when cooking targets. This option may be used when you think **cook** is acting strangely, or are just curious.

-NoTRace **Cook** will not emit information about the inferences it is making when cooking targets. This is the default.

7. Cookbook Language Definition

This chapter defines that language which cookbooks are written in. While some of its properties are similar to C, do not be misled.

A number of sections appear within this chapter.

1. The *Lexical Analysis* section describes what the words of the cookbook language look like.
2. The *Preprocessor* section describes the include mechanism and the conditional compilation mechanism.
3. The *Syntax Descriptions* section describes how to read the syntax definitions in the following section.
4. The *Syntax and Semantics* section describes how words in the cookbook may be combined to form valid constructs (the *syntax*), and what these constructs mean (the *semantics*).

The sections are laid out in the recommended reading order.

7.1 Lexical Analysis

The cookbook is made of a number of recipes, which are in turn made of words. This section describes what constitutes a word, and what does not.

7.1.1 Words and Keywords
Words are made of sequences of almost any character, and are separated by white space (including end-of-line) or the special symbols. **Cook** is always case sensitive when reading cookbooks.

The characters **:;={}[]** are the special symbols, and are words in themselves, needing no delimiting.

In addition to the special symbols, some words, known as *keywords*, have special meaning to **cook**. The keywords are:

if	then	else	set
loop	loopstop	fail	unsetenv

You will meet the keywords in later sections.

7.1.2 Escape Sequences
The character **** is the *escape* character. If a character is preceded by a **** any specialness, if it had any, will be removed. If it had no specialness it may have some added.

This means that, if you want to use **if** as a word, rather than a keyword, at least one of its characters needs to be escaped, for example **\if**.

The escape sequences which are special are as follows.

\b	The backspace character
\f	The form feed character
\n	The newline or linefeed character
\r	The carriage return character
\t	The horizontal tab character
*****nnn*	A character with a value of *nnn*, where *nnn* is an octal number of at most 3 digits.

An escaped end-of-line is totally ignored. It should be noted that a cookbook may not have any non-printing ascii characters in it other than space, tab and end-of-line.

7.1.3 Quoting
Words, and sections of words, may be quoted. If any part of a word is quoted it cannot be a keyword.

This means that, if you want to use **if** as a word, rather than a keyword, at least one of its characters needs to be quoted, for example **'if'**.

Both single (**'**) and double (quotes are understood by **cook**, and one may enclose the other. If a quote is escaped it does not open or close a quote as it usually would.

Cook does not like newlines within quotes. This is a generally good heuristic for catching unbalanced quotes.

7.1.4 Comments
Comments are delimited on the left by **/***, and on the right by ***/**, If the **/** character has been escaped or quoted, it doesn't introduce a comment. Comments may be nested. Comments may span multiple lines. Comments are replaced by one logical space.

7.2 *Preprocessor*

The preprocessor may be thought of as doing a little work before the *Syntax and Semantics* section has its turn.

The preprocessor is driven by *preprocessor directives*. A preprocessor directive is a line which starts with a hash (**#**) character. Each of the preprocessor directives is described below.

7.2.1 *include*

The most common preprocessor directive is

```
#include "filename"
```

This preprocessor directive is processed as if the contents of the named file had appeared in the cookbook, rather than the preprocessor include directive.

The most common use of the #include directive is to include system cookbooks.

The standard places to search are first any path specified with the **-Include** command line option, and then *$HOME/.cook* and then */usr/local/lib/cook* in that order.

7.2.2 *if*

The #if directive may be used to conditionally pass tokens to the syntax and semantics processing. Directives take the form

```
#if expression1
something1
#elif expression2
something2
#else
something3
#endif
```

There may be any number of `elif` clauses, and the `else` clause is optional. Only one of the *somethings* will be passed through.

7.2.3 *ifdef*

This directive takes a similar form to the `if` directive, but with a different first line:

```
#ifdef variable
```

This is syntactic sugar for

```
#if [defined symbol]
```

This is of most use in bracketing `#include` directives.

7.2.4 *ifndef*

This directive takes a similar form to the `if` directive, but with a different first line:

```
#ifndef variable
```

This is syntactic sugar for

```
#if [not [defined symbol]]
```

This is of most use in bracketing `#include` directives.

7.2.5 *pragma*

This is for the addition of extensions.

7.2.5.1 *once*

This directive is to ensure that include files in which it appears are included exactly once.

This directive has the form

```
#pragma once
```

7.2.5.2 *unknown extensions*

Any extensions not recognised will be ignored.

7.3 Syntax Descriptions

In the syntax descriptions which follow there are several meta symbols, defined as follows.

=	A definition symbol.
|	The alternative symbol.
.	The end-of-production symbol.
/**text****/**	This is a comment.
'*text***'**	The *text* is a literal terminal symbol of the grammar being defined.

Like all real languages, the format is free form, no columns or end-of-line are significant.

As an example, the syntax description format is described using itself.

```
syntax
    = /* empty */
    | syntax production
    .
```

A syntax definition may consist of zero or more productions.

```
production
    = name '=' term '.'
    .
```

A production names a non-terminal symbol of the grammar being defined on the left, and its expansion on the right.

```
term
    = factor
    | term '|' term
    .
```

A term may consist of a factor; or a term may consist of zero or more of the above terms separated by *alternative* symbols.

```
factor
    = /* empty */
    | factor name
    | factor literal
    .
```

A factor consists of zero or more symbol names (terminal or non-terminal) or literals in a row.

A symbol is terminal if it is not named on the left-hand-side of a production, or it is a literal.

7.4 Syntax and Semantics

7.4.1 Overall Structure

The general form of the cookbook is defined as

```
cookbook
    = /* empty */
    | cookbook statement
    .
```

A cookbook is defined as a sequence of statements. Each statement statement is executed. For a definition of what it means when a statement is executed, see the individual statement definitions.

The nonterminal symbol *statement* will be defined in the sections below.

Please note that a statement is not always evaluated when is is read, but at specific, well defined times.

7.4.2 The Compound Statement

A nonterminal symbol which will be referred to below is the *compound_statement* symbol, defined as follows:

```
compound_statement
    = '{' statements '}'
    .
statements
    = /* empty */
    | statements statement
    .
```

The compound statement may be used anywhere a statement may be, and in particular

```
statement
    = compound_statement
    .
```

7.4.3 Variables and Expressions

Cook provides variables to the user to simplify things.

7.4.3.1 The Assignment Statement

It is possible to assign to variables with the following statement.

```
statement
    = expr '=' exprs ';'
    .
```

When this statement is executed, the variable whose name the left hand expression evaluates to will be assigned the value that the right hand expression list evaluates to.

7.4.3.2 Expressions

Many definitions make reference to the *expr, elist* and *exprs* nonterminal symbols. These are defined as follows.

The *elist* is a list of at least one expression, whereas the *exprs* is a list of zero or more expressions.

```
elist
    = expr
    | elist expr
    .
exprs
    = /* empty */
    | exprs expr
    .
```

An expression is composed of variable references, function invocations, words, or concatenation of expressions. The concatenation is implied by abutting the two parts of the expression together, e.g.: "`[fred]>thing`" is an indirection on *fred* concatenated with the literal word "`>thing`".

```
expr
```

```
        = WORD
        | '[' elist ']'
        | expr cat expr
        .
```

When an [*elist*] expression is evaluated, the *elist* is evaluated first. If the result is a single word, then a variable of that name is searched for. If found the value of an expression of this form is the value of the variable.

If there is no variable of the given name, or the *elist* evaluated to more than one word, the first word is taken to be a built-in function name. If there is no function of this name it is an error.

The *cat* operator works as one would expect, joining the last word of the left expression and the first word of the right expression together, and otherwise leaving the order of the expressions alone. One usually uses the trivial case of single word expressions.

7.4.4 Recipes

A number of forms of *statement* are concerned with telling **cook** how to cook things. There are three forms, the *explicit* recipe, the *implicit* recipe, and the *ingredients* recipe.

7.4.4.1 The Explicit Recipe Statement

The explicit recipe has the form

```
statement
    = elist ':' exprs flags compound_statement
    .
```

The target(s) of the recipe are to the left of the colon, and the ingredients, if any, are to the right. The statements, usually commands, to perform to cook the target(s) are contained in the compound statement. The expressions are only evaluated into words when the recipe is instanciated.

The *flags* are defined as follows.

```
flags
    = /* empty */
    | 'set' words
    .
```

A number of flags may be used

errok
: If the errok flag is specified, the commands within the actions bound to the recipe will always be successful.

forced
: If the forced flag is specified, the actions bound to the recipe will always be evaluated.

precious
: If the precious flag is specified, if the actions bound to the recipe fail, the targets of the recipe will not be deleted, as is the default.

silent
: If the silent flag is specified, the command within the actions bound to the recipe will not be echoed.

wildpath
: If the wildpath flag is specified in an implicit recipe, leading path on pattern matches will be prepended to reconstructions.

meter
: If the meter flag is specified, a summary of the CPU usage by the commands within this recipe will be printed after each command. The silent options override this option.

clearstat
: The date-time-modified of the files named in executed commands will be removed from the time cache. This is essential for commands such as *rm*(1) and *mv*(1).

7.4.4.2 The Implicit Recipe Statement

Implicit recipes are distinguished from explicit recipes in that and implicit recipe has a target with a '**%**' character in it.

7.4.4.2.1 Simple Form

In general the user will rarely need to use the implicit recipe form, as there are a huge range of implicit recipes already defined in the system default recipes.

An example of this recipe form is

```
%: %.Z
     {
        uncompress %;
     }
```

This recipe tells **cook** how to use the *uncompress*(1) program.

7.4.4.2.2 Complex Form

The implicit recipe recipe has a second form

```
statement
     = elist ':' exprs1 ':' exprs2 flags compound_statement
     .
```

In this form, the ingredients specified in *exprs1* are used to determine the applicability of the recipe; if these are all constructable then the recipe will be applied, if any are not constructable then the recipe will not be applied. If the recipe is applied, the ingredients specified in *exprs2* are required to be constructable. The *exprs2* section is known as the *forced ingredients* section.

An example of this is the C recipe

```
%.o: %.c: [collect c_incl %.c]
     {
        cc -c %.c;
     }
```

This recipe is applied if the *%.c* file can be constructed, and is not applied if it cannot be constructed. The include dependencies are only expressed if the recipe is going to be applied; but if they are expressed, they *must* be constructable. This means that absent include files generate an error.

The naive form of this recipe

```
%.o: %.c [collect c_incl %.c]
     {
        cc -c %.c;
     }
```

will attempt to apply the *c_incl* command before the *%.c* file is guaranteed to exist. This is because the *exprs2* is performed after the *exprs1* all exist (because they are constructable, they have been constructed). In this naive form, absent include files result in the recipe not being applied.

7.4.4.3 The Ingredients Recipe Statement

The ingredients recipe has the form

```
statement
     = elist ':' elist ';'
     .
```

The target(s) of the recipe are to the left of the colon, and the prerequisites are to the right. There are no statements to perform to cook the targets of this recipe, it is simply supplementary to any other recipe, usually an implicit recipe.

The expressions are only evaluated into words when the recipe is instanciated.

7.4.5 Commands

Commands may take several forms in **cook**. They all have one thing in common; they execute a command.

```
statement
     = command
     .
```

7.4.5.1 The Simple Command Statement
The simplest command form is

```
command
    = simple_command
    .
simple_command
    = elist flags ';'
    .
```

When executed, the *elist* is evaluated into a word list and used as a command to be passed to the operating system. While this is usually sufficient, some programs, notably *newgrp*(1), require slightly more.

7.4.5.2 The Data Command Statement
For programs which require *stdin* to be supplied by **cook** to perform their functions, the data command statement has been provided.

```
command
    = simple_command
      'data'
          expr
      'dataend'
    .
```

In this form, the *expr* is evaluated and used as input to the command. Between the **data** and **dataend** keywords the definition of the special symbols and whitespace change. There are only two special symbols, **[** and **]**, to allow functions and variable references to appear in the expression. In addition, whitespace ceases to have its usual specialness; it is handed to the command, instead.

The **data** keyword must be the last on a line, whitespace after the **data** keyword up to and including end-of-line, will *not* be given to the command.

The **dataend** keyword must appear alone on a line, optionally surrounded by whitespace; it is is not alone, it is not a **dataend** keyword and will not terminate the expression.

An example of this may be useful.

```
/usr/fred/%: %
    {
        newgrp fred;
            data
                cp % /usr/fred/%
            dataend
    }
```

If the directory */usr/fred* has read-only permissions for others, and group write permissions, and belonged to group *fred*, and you were a member of group *fred*, the above implicit recipe could be used to copy the file.

7.4.5.3 The Fail Statement
Cook can be forced to think that a recipe has failed by the uses of the **fail** statement.

```
statement
    = 'fail' ';'
    .
```

This is hugely useful when programs do not return a useful exit status, but *do* fail to produce the goods.

Another variation of this statement is

```
statement
    = 'fail' 'backtrack' ';'
    .
```

This enables you to write a recipe which will succeed if all of the ingredients are up-to-date, but cause **cook** to backtrack if any of the ingredients are out-of-date. It is useful with some software configuration management systems.

7.4.6 Flow Control
This section details statement forms which the casual user will never need.

7.4.6.1 The If Statement
The if statement has one of two forms.

```
statement
    = 'if' expr 'then' statement
    | 'if' expr 'then' statement 'else' statement
    .
```

In nested if statements, the **else** will bind to the closest **else**-less **if**.

An expression is false if and only if all of its words are null or it has no words.

7.4.6.2 The Loop and Loopend Statements
Looping is provided for in **cook** by the generic infinite loop construct defined below. A facility is provided to break out of a loop at any point.

```
statement
    = 'loop' statement
    | 'loopstop' ';'
    .
```

The statement following the **loop** directive is executed repeatedly forever. The **loopstop** statement is only semantically valid within the scope of a **loop** statement.

8. *Built-In Functions*

This chapter defines each of the builtin functions of *cook*.

A builtin function is invoked by using an expression of the form

[*func-name arg arg ...***]**

in most places where a literal word is valid.

8.1 *and*

This function requires at least two arguments, upon which it forms a logical conjunction. The value returned is "1" (true) if none of the arguments are "" (false), otherwise "" (false) is returned.

8.2 *catenate*

This function requires zero or more arguments. If no arguments are supplied, the result is an empty word list. If one or more arguments are supplied, the result is a word list of one word being the catenation of all of the arguments.

8.3 *collect*

The arguments are interpreted as a command to be passed to the operating system. The result is one word for each white-space separated word of the output of the command.

8.4 *collect_lines*

The arguments are interpreted as a command to be passed to the operating system. The result one word for each line of the output of the command.

8.5 *count*

This function requires zero or more arguments. The result is a word list of one word containing the (decimal) length of the argument word list.

8.6 *defined*

This function requires a single argument, the name of a variable to be tested for existence. It returns "1" (true) if the named variable is defined and "" (false) if it is not.

8.7 *dir*

This function requires one or more arguments, the names of files which will have their directory parts extracted. This function may also be referred to as *dirname*.

8.8 *downcase*

This function requires one or more arguments, words to be forced into lower case.

8.9 *entryname*

This function requires one or more arguments, the names of files which will have their entry name parts extracted.

8.10 *execute*

This function requires at least one argument, and executes the command given by the arguments. If the executed command returns an error code the resulting value is "" (false), otherwise it is "1" (true).

8.11 *exists*

This function requires one argument, being the name of a file to test for existence. The resulting wordlist is "" (false) if the file does not exist, and "1" (true) if the file does exist.

8.12 *find_command*

This function requires at least one argument, being the names of commands to search for in $PATH. The resulting word list contains either "" (false) or a fully qualified path name for each command given.

8.13 fromto

This function requires at least two arguments. Fromto gives the user access to the wildcard transformations available to **cook**. The first argument is the "from" form, the second argument is the "to" form. All other arguments are mapped from one to the other.

8.14 getenv

Each argument is treated as the name of an environment variable. The result is the value of each argument variable, or "" if it does not exist.

8.15 glob

Each argument is treated as a *sh*(1) file name pattern, and expanded accordingly.

8.16 head

This function requires zero or more arguments. The wordlist returned is empty if there were no arguments, or the first argument if there were arguments.

8.17 if

This function requires one or more arguments, the arguments before the "then" word are used as a condition. If the condition is true the words between the "then" word and the "else" word are the result, otherwise the words after the "else" word are the value. The "else" clause is optional. There is no way to escape the "then" and "else" words.

8.18 in

This function requires one or more arguments. The wordlist returned is a single word: "1" (true) if the first argument is equal to any of the later ones; "" (false) if not.

8.19 matches

This function requires one or more arguments, the first is a pattern, subsequent are strings to match against this pattern. The resulting wordlist contains "" (false) if did not match and "1" (true) if it did.

8.20 match_mask

This function requires one or more arguments, the first is a pattern, subsequent are strings to match against this pattern. The resulting wordlist contains those arguments which mtached the pattern given as the first argument.

8.21 mtime

This function requires one argument, the name of a file to fetch the last-modified time of. The resulting wordlist is "" (false) is the file does not exist, or a string containing a (comparable) representation of the date and time the files was last modified.

8.22 not

This function requires zero or more arguments, the value to be logically negated. It returns "1" (true) if all of the arguments are "" (false), or there are no arguments; and returns "" (false) otherwise. This is symmetric with the definition of true and false for **if**.

8.23 operating_system

This function requires zero or more arguments. The resulting wordlist contains the values of various attributes of the operating system, as named in the arguments. If no attributes are named "name" is assumed. Below is a list of attributes:

name	The generic name of the operating system **cook** presently being run under. For example: if you were running on a Berkeley 4.2BSD Unix, this would return "`unix`".

version — The specific version of operating system, within name, **cook** is presently being run under. For example: if you were running on a Berkeley 4.2BSD Unix, this would return `"4.2BSD"`.

This function may also be referred to as "os".

8.24 or

This function requires at least two arguments, upon which it forms a logical disjunction. The value returned is "1" (true) if any one of the arguments is not "" (false), otherwise "" (false) is returned.

8.25 pathname

The function requires one or more arguments, being files names to be replaced with their full path names.

8.26 prepost

This function must have at least two arguments. The first argument is a prefix and the second argument is a suffix. The resulting word list is the third and later arguments each given the prefix and suffix as defined by the first and second arguments.

8.27 quote

Each argument is quoted by single quotes, with special characters escaped as necessary.

8.28 resolve

This builtin function is used to resolve file names when using the *search_list* variable to locate files. This builtin function produces resolved file names as output. This is useful when taking partial copies of a source to perform controlled updates. The targets of recipes are always cooked into the current directory.

8.29 sort

The arguments are sorted lexicographically.

8.30 stringset

Logical operations are performed on sets of strings. These include conjunction (**|**), disjunction (**&**) and difference (**-**).

8.31 tail

This function requires zero or more arguments. The word list returned will be empty if there is less than two arguments, otherwise it will consist of the second and later arguments.

8.32 upcase

This function requires one or more arguments, words to be forced into upper case.

8.33 uptodate

This function requires one or more arguments, filenames to be brought up-to-date. The result are true ("1") if no error occurred, or false ("") if some error occurred.

9. Predefined Variables

A number of variables are defined by **cook** at run-time.

9.1 need

The ingredients of the recipe currently being cooked.

9.2 search_list

This variable may be set to a list of directories to be searched for targets and ingredients. This list is initially the current directory (.) and will always have the current directory prepended if it is not present. This is useful when taking partial copies of a source to perform controlled updates. Use the *resolve* builtin function to determine what file name cook actually found. The targets of recipes are always cooked into the current directory.

9.3 self

The name **cook** was invoked as, usually "cook". Be careful what you call cook, because anything with the string "cook" in it will be changed, including (but not limited to) file suffixes and environment variable names.

9.4 target

The target of the recipe currently being cooked. Specifically, the target which caused the recipe to be invoked.

9.5 targets

The targets of the recipe currently being cooked. This includes all targets of the recipe, should there be more than one.

9.6 younger

The subset of the ingredients of the recipe currently being cooked which are younger than the target.

9.7 version

The version of **cook** currently executing.

10. Actions when Cooking

This section describes what **cook** does when you ask it to cook something.

Cook performs the following actions in the order stated.

10.1 Scan the COOK Environment Variable

The **COOK** environment variable is looked for. If it is found, it is treated as if it consisted of **cook** command line arguments. Only the **-Help** option is illegal. This could result is very strange behaviour if used incorrectly.

This feature is supplied to override **cook**'s default with your own preferences.

10.2 Scan the Command Line

The command line is scanned as defined in chapter 3.

10.3 Locate the Cookbook

The current directory is scanned for the cookbook. Names which a cookbook may have include

howto.cook	Howto.cook	.howto.cook
how.to.cook	How.to.cook	.how.to.cook
cookfile	Cookfile	.cookrc
cook.file	Cook.file	.cook.rc

The first so named file found in the current directory will be used. The order of search is not defined. You are strongly advised to have just *one* of these name forms in any directory. The name *Howto.cook* is the preferred form.

10.4 Form the Listing Filename

The listing file, if not explicitly named in the environment variable or on the command line, will be the name of the cookbook, with any suffix removed and '`.list`' appended.

10.5 Create the Listing file

The listing file is created. If **cook** is executing in the background, or the **-NoTTy** option has been specified, *stdout* and *stderr* will be redirected into the listing file. If **cook** is executing in the foreground, and the **-NoTTy** option has not been specified, *stdout* and *stderr* will be redirected into a pipe to a *tee*(1) command; which will, in turn, copy the output into the named file.

A heading line with the name of the file and the date, is generated.

10.6 Scan the Cookbook

When **cook** reads the cookbook it evaluates all of the statements it finds in it. Usually these statements instanciate recipes, although other things are possible.

Recipes contain statements that are not evaluated immediately, but which are remembered for later execution when cooking a target. The meaning of a cookbook is defined in chapter X.

10.7 Determine targets to cook

If no target files are named on the command line, the targets of the first defined explicit or ingredients recipe. It is an error if this is none.

10.8 Cooking a Target

Each of the targets, in the order given, are cooked.

To cook a target each the following steps is performed in the order given:

1. **Cook** scans through the instanciated ingredients recipes in the order they were defined. All ingredients recipes with the target in their target list are used.

   If a recipe is used, then any ingredients are recursively cooked. If any of the ingredients are younger than the target, all other explicit or implicit recipes with the same target will be deemed to be out of date.[1]

2. **Cook** then scans through the instanciated explicit recipes in the order they were defined. All explicit recipes with the target in their target list are used.

   If a recipe is a used, the ingredients are recursively cooked. If any ingredients are out of date the recipe body will be performed.

3. If the target was not in the target list of any explicit recipe, **cook** then scans the instanciated implicit recipes in the order they were defined.

   Implicit recipe targets and ingredients may contain a wildcard character (**%**), which is why they are implicit. When expressions are evaluated into word lists in an implicit recipe, any word containing the wildcard character (**%**) will be expanded out by the current wildcard expansion.

   If the target matches a pattern in the targets of an implicit recipe, it is a candidate. Each ingredient of a candidate recipe is recursively cooked. If any ingredient cannot be cooked, then the implicit recipe is not used. If all ingredients can be cooked, then the implicit recipe is used.

   If an implicit recipe is a used, the forced ingredients are recursively cooked. It is an error if a forced ingredient cannot be constructed. After the forced ingredients are constructed, the recipe body is performed.

   Only the first implicit recipe to get to this point is used. The scan stops at this point.

4. If the target is not the subject of any ingredients or explicit recipe, and no implicit recipes can be applied, then two things can happen.

   - If the file exists, then it is up to date, or
   - If the file does not exist then **cook** doesn't know how.

If a command in the body of any recipe fail, **cook** will not that body any further, and will not perform the body of any recipe for which the target of the failed actions was an ingredient, directly or indirectly.

Cook will trap recursive looping of targets.

- If the file exists, the it is up to date, or
- If the file does not exist then **cook** doesn't know how.

1. A target which does not exist yet is considered to be infinitely ancient, and thus everything is younger than it.

11. Option Precedence

At various points in the description there are a number of flags and options with the same, or similar, names. These are in fact different levels of the same option.

The different levels, from highest precedence to lowest, are as follows.

Error
: This level is used to disable undesirable side effects when an error occurs.

Command Line
: Options specified on the command line override almost everything. There are some isolated cases where there is no equivalent command line option. They are in scope for the entire **cook** session.

Execute
: When a command attached to a recipe is executed, the flags in the '**set**' clause are given this precedence. They are in scope for the duration of the execution of the command they are bound to.

Recipe
: When a recipe is considered for use, the flags in the '**set**' clause are given the precedence. They are in scope for the evaluation of the ingredients names and the execution of the recipe body; they are not in scope while cooking the ingredients.

Cookbook
: When a '**set**' statement is encountered in the cookbook, the option are given this priority. They are in scope until the end of the **cook** session.

Environment Variable
: When the options in the **COOK** environment variable are set, they are given this precedence. They are in scope for the entire **cook** session.

Default
: All options have a default setting. The defaults noted in chapter 3 are given this precedence. They are in scope for the entire **cook** session.

12. File name patterns

The tough part about designing a pattern matcher for something like cook is that the patterns must be reversible. That is, it must be possible to use the same string both as a pattern to be matched against and as a template for building a string once a pattern has matched. Rather like the difference between the left and right sides of an editor search-and-replace command in an editor using the same description for both the search pattern and the replace template. This is why classic regular expressions have not been used. They tend to be slow to match, too.

This matcher has eleven match "fields", referenced as **%** and **%0** to **%9**. The **%** character can be escaped as **%%**. The **%** and **%1** to **%9** forms match any character except **/**. The **%0** form matches all characters, but must be either empty, or have whole path components, including the trailing **/** on each component.

A few examples will make this clearer:

string	does not match
%.c	snot/fred.c
%1/%2.c	etc/boo/fred.c

string	matches	setting
%.c	fred.c	%="fred"
%1/%2.c	snot/fred.c	%1="snot" %2="fred"
%0%5.c	fred.c	%0="" %5="fred"
%0%6.c	snot/fred.c	%0="snot/" %6="fred"
%0%7.c	etc/boo/fred.c	%0="etc/boo/" %7="fred"
/usr/%1/%1%2/%3.%2%4	/usr/man/man1/fred.1x	%1="man" %2="1" %3="fred" %4="x"

The **%0** behaviour is designed to allow patterns to range over subtrees in a controlled manner. Note that the use of this sort of pattern in a recipe will result in deeper searches than the naive recipe designer would expect.

13. Supplied Cookbooks

A number of cookbooks are supplied with **cook**. To make use of one, a preprocessor directive of the form

```
#include "whichone"
```

must appear the the start of your cookbook.

Cook does not have any "builtin" recipes. All recipes are stored in text files, so they are more easily read, understood, copied, hacked or corrected. The supplied cookbooks live in the */usr/local/lib/cook* directory.

You may supply your own "system" recipes, by placing cookbooks into a directory called *$HOME/.cook* or using the **-Include** command line option, possibly in your *$COOK* environment variable.

13.1 as

This cookbook defines how to use the assembler.

13.1.1 recipes

%.o: %.s	Construct object files from assembler source files.

13.1.2 variables

as	The assembler command. Not altered if already defined.
as_flags	Options to pass the assembler command. Not altered if already defined. The default is empty.
as_src	Assembler source files in the current directory.
dot_src	Source files constructable in the current directory (unioned with existing setting, if necessary).
dot_obj	Object files constructable in the current directory (unioned with existing setting, if necessary).
dot_clean	Files which may be removed from the current directory in a clean target.

13.2 c

This cookbook describes how to work with C files. Include file dependencies are automatically determined.

13.2.1 recipes

%.o: %.c	Construct object files form C source files, with automatic include file dependency detection.
%.ln: %.c	Construct lint object files from C source files, with automatic include file dependency detection.

13.2.2 variables

c_incl	The C include dependency sniffer command. Not altered if already defined.
cc	The C compiler command. Not altered if already defined.
lint	The lint command. Not altered if already defined.
cc_flags	Options to pass to the C compiler command. Not altered if already defined. The default is "-O".
cc_include_flags	Options passed to the C compiler and c_incl controlling include file searching. Not altered if already defined. The default is empty.
cc_link_flags	Options passed to the C compiler when linking, these are typically library search paths and libraries. Not altered if already defined. The default is empty.

cc_src	C source files in the current directory.
dot_src	Source files constructable in the current directory (unioned with existing setting, if necessary).
dot_obj	Object files constructable in the current directory (unioned with existing setting, if necessary).
dot_clean	Files which may be removed from the current directory in a clean target.
dot_lint_obj	Lint object files constructable in the current directory (unioned with existing setting, if necessary).

13.3 home

This cookbook defined where certain directories are, and some common uses of those directories, relative to $HOME.

13.3.1 variables

home	The current users' home directory.
bin	The directory to place program binaries into.
include	The directory to place include files into.
lib	The directory to place libraries into.
cc_include_flags	The [include] directory is appended to the search options.
cc_link_flags	The [lib] directory is appended to the search options.

13.4 lex

This cookbook describes how to work with lex files.

13.4.1 recipes

%.l: %.c	Construct C source files from lex source files.

13.4.2 variables

lex	The lex command. Not altered if already defined.
lex_flags	Options to pass to the lex command. Not altered if already defined. The default is empty.
lex_src	Lex source files in the current directory.
dot_src	Source files constructable in the current directory (unioned with existing setting, if necessary).
dot_obj	Object files constructable in the current directory (unioned with existing setting, if necessary).
dot_clean	Files which may be removed from the current directory in a clean target.
dot_lint_obj	Lint object files constructable in the current directory (unioned with existing setting, if necessary).

13.5 library

This cookbook defines how to construct a library.

If an include file (or files) are defined for this library, you will have to append them to [install] in your *Howto.cook* file.

13.5.1 variables

all — targets of the all recipe

install — targets of the install recipe

me — The name of the library to be constructed. Defaults to the last component of the pathname of the current directory.

ar — The archive command.

install — targets of the install command. Only defined if the [lib] variable is defined.

13.5.2 recipes

all — construct the targets defined in [all].

clean — remove the files named in [dot_clean].

clobber — remove the files name in [dot_clean] and [all].

install — Construct the files named in [install]. Only defined if the [lib] variable is defined.

uninstall — Remove the files named in [install]. Only defined if the [lib] variable is defined.

13.6 print

This cookbook is used to print files. It will almost certainly need to be changed for every site.

13.6.1 recipes

%.lw: %.ps — Print a PostScript file.

%.lp: % — Print a text file.

13.6.2 variables

lp — The print command. Not altered if already defined.

lp_flags — Options passed to the print command. Not altered if already defined. Defaults to empty.

13.7 program

This cookbook defines how to construct a program.

If you program uses any libraries, you will have to append them to [cc_link_flags] in your *Howto.cook* file.

13.7.1 variables

all — Targets of the all recipe.

install — targets of the install recipe

me — The name of the program to be constructed. Defaults to the last component of the pathname of the current directory.

install — targets of the install command.

13.7.2 recipes

all — Construct the targets named in [all].

clean — Remove the files named in [dot_clean].

clobber — Remove the files named in [dot_clean] and [all].

install — Construct the files named in [install]. Only defined if the [lib] variable is defined.

uninstall — Remove the files named in [install]. Only defined if the [lib] variable is defined.

13.8 sccs

This cookbook is used to extract files from SCCS.

13.8.1 recipes

%: SCCS/s.%	Extract files from SCCS.

13.8.2 variables

get	The sccs get command.

13.9 text

This cookbook is used to process text documents.

Include file dependencies are automatically detected. The requirements for various preprocessors are automatically detected (eg eqn, tbl, pic, graf).

13.9.1 recipes

%.ps: %.t	PostScript for generic *roff source.
%: %.t	Straight text from *roff source.

13.9.2 variables

text_incl	The text_incl command (finds include dependencies). Not altered if already set.
text_roff	The text_roff command (finds preprocessor requirements). Not altered if already set.
roff_flags	Arguments passed to text_roff, and indirectly to the *roff program. Not altered if already set. Defaults to empty.

13.10 usr

This cookbook defined where certain directories are, relative to /usr.

13.10.1 variables

bin	The directory to place program binaries into.
include	The directory to place include files into.
lib	The directory to place libraries into.

13.11 usr.local

This cookbook defined where certain directories are, and some common uses of those directories, elative to /usr/local.
.H 3 "variables"

bin	The directory to place program binaries into.
include	The directory to place include files into.
lib	The directory to place libraries into.
cc_include_flags	The [include] directory is added to the search options.
cc_link_flags	The [lib] directory is added to the search options.

13.12 yacc

This cookbook describes how to use yacc.

You will have to add "-d" to the [yacc_flags] variable if you want %.h files generated.

If a *y.output* file is constructed, it will be moved to *%.list*.

13.12.1 recipes

%.c %.h: %.y — Construct C source and header files from yacc source files. Applied if -d in [yacc_flags].

%.c: %.y — Construct C source files from yacc source files. Applied if -d not in [yacc_flags].

13.12.2 variables

yacc_src — Yacc source files in the current directory.

dot_src — Source files constructable in the current directory (unioned with existing setting, if necessary).

dot_obj — Object files constructable in the current directory (unioned with existing setting, if necessary).

dot_clean — Files which may be removed from the current directory in a clean target.

dot_lint_obj — Lint object files constructable in the current directory (unioned with existing setting, if necessary).

13.13 yacc_many

This cookbook describes how to use yacc. The difference with the "yacc" cookbook is that this cookbook allows you to have more that one yacc generated parser in the same program, by using the classic sed(1) hack of the output.

14. Glossary

This document employs a number of terms specific to **cook**.

body — A set of statements, usually commands, to be performed to *cook* the *target*s of a *recipe* after the *ingredient*s exist.

command — A command is a list of words to be passed to the *operating system* to be executed.

cook — When used as a verb, refers to the actions **cook** would perform to create a *target*, according to some *recipe*.

cookbook — A file containing input for **cook**, usually *recipe*s.

explicit recipe — An explicit recipe is one where the *target*s contain no patterns. That is, there are no percent ('**%**') characters in any of the *target*s.

flag — A flag modifies the behaviour of a cook session, *recipe* or command.

forced ingredient — A files which must exist before a *target* file of an *implicit recipe* may be cooked. The inability to construct a forced ingredient is an error.

function — A function is an action applied to a word list.

gate — A gate is a condition which allows the conditional application of a *recipe*.

implicit recipe — An implicit recipe is a recipe with patterns in the *target*s. That is, there is a percent ('**%**') character in at least one of the *target*s.

ingredient — A files which must exist before a *target* file may be cooked. In an *implicit recipe* the inability to construct of an ingredient means that the *recipe* will not be applied. In an explicit recipe the inability to construct an ingredient is an error.

last-modified time — UNIX imbues files with several attributes. One of these is a time-stamp of when the file was last modified. Usually this is when the file was last written to.

recipe — A *recipe* consists of several parts.

1. A set of *target*s to be cooked,
2. A set of ingredients of those *target*s, and
3. An optional set of forced ingredients.
4. An optional set of flags.
5. An optional gate.
6. An optional body .

target — The object of a *recipe*, a thing which is cooked.

touch — UNIX imbues files with several attributes. One of these is a time-stamp of when the file was last modified. Usually this is when the file was last written to, however it is possible to simply adjust this attribute, rather than actually writing to the file; this is colloquially known as *touch*ing a file.

variable — A variable is a named place holder for a value. The value may be changed.

CONTENTS