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.

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to see when the prerequisites of a file have changed, implying that the file needs to be recreated as it is logically out of date.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

cook version 1.9

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

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*...]

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.

-No_Action 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.

-No_Continue 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.

-No_Errok 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.

-FingerPrint When *cook* examined a file to determine if it has changed, it uses the last-modified time information available in the file system. There are times when this is altered, but the file contents do not actually change. The fingerprinting facility examines the file when it appears to have changed, and compares the old fingerprint against the present file contents. If the fingerprint did not change, the last-modified time in the file system is ignored. Note that this has implications if you are in the habit of using the *touch*(1) command – *cook* will do nothing until you actually change the file.

-No_FingerPrint
Do not use fingerprints to supplement the last-modified time file information. 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.

-No_Force 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.

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

-No_List *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.

-No_Meter 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.

-No_Precious 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.

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

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

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

-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.

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

-Update This option causes **cook** to check the last-modified time of the targets of recipes, and updates them if necessary, to make sure they are consistent with (younger than) the last-modified times of the ingredients. This results in more system calls, and can slow things down on some systems.

-No_Update Do not update the file last-modified times after performing the body of a recipe. This is the default.

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 **-No_Touch** 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.

-No_TRace **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 include-cooked

This directive looks similar to the one above, but do not be deceived.

```
#include-cooked filename...
```

You may name several filenames on the line, and they may be expressions.

The search path used for these files is the same as that used for other cooked files, see the *search_list* variable and the *resolve* built-in function for more information. The order in which you set the *search_list* and the the *#include-cooked* directives is important. Always set the *search_list* variable first, if you are going to use it.

Files included in this way are checked, after they have been read, to make sure they are up-to-date. If they are not, **cook** brings them up-to-date and then re-reads the cookbook and starts over.

You will only get a warning if the files are not found. Usually, **cook** will either succeed in constructing them, in which case they will be present the second time around, or a fatal error will result from attempting to construct them. Note that it is possible to go into an infinite loop, if the files are constantly out-of-date.

The commonest use of this construct is maintaining include file dependency lists for source files.

```
obj = [fromto %.c %.o [glob *.c]];

%.o: %.c
{
        [cc] [cc_flags] -c %.c;
}

%.d: %.c
{
        c_incl -prefix "'%.o %.d: %.c'" -suffix "';'"
                -nc -nc %.c > %.d;
}

#include-cooked [fromto %.o %.d [obj]]
```

This cookbook fragment shows how include file dependencies are maintained. Notice how the *.d* files have a recipe to construct them, and that they are also included. **Cook** will bring them up-to-date if necessary, and then re-read the cookbook, so that it is always working with the current include dependencies. (The doubly nested quotes are to insulate the spaces and special characters from both **cook** and the shell.)

You could use *gcc -MM* if you prefer (you will need some extra shell script). The *c_incl* program understands absent files better but doesn't understand conditional compilation, and *gcc* understands conditional compilation but gives fatal errors for absent include files. Warning: If you are using *search_list* you **must**

use *c_incl*. Gcc returns complete paths, which will result in **cook** failing to notice when an include file is copied from later in the search list to earlier, and then modified.

7.2.3 include-cooked-nowarn

This directive is almost identical to the one above, but no warning is issued for absent files.

```
#include-cooked-nowarn filename...
```

You may name several filenames on the line, and they may be expressions.

7.2.4 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.

7.2.5 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.

7.2.6 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.

7.2.7 pragma

This is for the addition of extensions.

7.2.7.1 once

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

This directive has the form

```
#pragma once
```

7.2.7.2 unknown extensions

Any extensions not recognized 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 gate compound_statement
          use_clause
        .
```

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 instantiated.

7.4.4.1.1 Recipe Flags

The *flags* are defined as follows.

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

A number of flags may be used

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

noclearstat — Do not clear entries from the last-modified time cache. This is usually the default.

default — If no targets are specified on the command line, the first recipe with the *default* flag will be used. Not meaningful for implicit recipes.

nodefault — If no targets are specified on the command line, and there are no recipes with the *default* flag set, the first recipe **without** the *nodefault* flag will be used. Not meaningful for implicit recipes.

errok — If the *errok* flag is specified, the commands within the actions bound to the recipe must always be successful.

noerrok — Exit status from commands will be ignored. This is usually the default.

fingerprint — File fingerprints are used to supplement last-modified time information about files, which is how *cook* determines if a file is out-of-date and needs to be cooked. If a file appears to have changed, find the last-modified time, it is fingerprinted, and the fingerprint compared with what it was in the past. The file has change if and only if the fingerprint has also changed. A cryptographically strong hash is used, so the chance of a file edit producing an identical fingerprint is less than 1 in 2**200. Fingerprinting is disabled by default.

nofingerprint Do not use file fingerprinting. This is usually the default.

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

noforced If the *noforced* flag is specified, the actions bound to the recipe will be evaluated when the recipe is logically out-of-date. This is usually the default.

mkdir If the *mkdir* flag is specified, the directories of any targets will be created before the actions bound to the recipe are evaluated.

nomkdir If the *nomkdir* flag is specified, the directories of any targets will need to be created by the actions bound to the recipe. This is usually the default.

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

noprecious If the *noprecious* flag is specified, if the actions bound to the recipe fail, the targets of the recipe will be deleted. This is usually the default.

recurse If this flag is specified, recipes will recuse upon themselves if one of their ingredients matches one of their targets. This can cause problems, and so it is not the default.

norecurse If this flag is specified, the recipe will not recurse if one of its ingredients matches one of its targets. This is the default.

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

nosilent Commands will be echoed. This is usually the default.

unlink If the *unlink* flag is specified, of any targets will be unlinked before the actions bound to the recipe are evaluated.

nounlink If the *nounlink* flag is specified, the recipe targets are not removed before the actions bound to the recipe are performed. This is usually the default.

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

nowildpath Leading path will not be prepended to reconstructions. This is usually the default.

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.

nometer Do not meter commands. This is usually the default.

time-adjust This option causes **cook** to check the last-modified time of the targets of recipes, and adjust them if necessary, to make sure they are consistent with (younger than) the last-modified times of the ingredients. This usually adjusts the file time into the (near) future. A warning message will be printed, telling you how many seconds the file was adjusted. This results in more system calls, and can slow things down on some systems.

no-time-adjust Do not adjust the file last-modified times after performing the body of a recipe. This is usually the default.

time-adjust-back This option causes **cook** to force the last-modified time of the targets of recipes to be exactly one (1) second younger than their youngest ingredient. This usualy adjusts the file time into the (recent) past. A warning message will be printed, telling you how many seconds the file was adjusted. This results in more system calls, and can slow things down on some systems. This is primarily useful when some later process is going to compress file modification times; this provides smarter compression.

---

0. This flag was once named the ‘‘update’’ flag. The name was changed to more closely reflect its function. The old name continues to work.

stripdot This option causes **cook** to remove leading "./" prefixes from filenames. This is usually the default.

nostripdot This option causes **cook** to leave leading "./" prefixes on filenames.

Each flag may also be specified in the negative, by adding a "no" prefix, to override any existing positive default setting. There is a strict precedence defined for the various levels of flag setting, see the end of the "How Cook Works" chapter for details.

7.4.4.1.2 Recipe Gate

Each recipe may have a *gate*. The gate is a way of specifying a conditional recipe; if the condition is not true, the recipe is not used. The condition is in addition to the condition that the ingredients are cookable.

```
gate
        = /* empty */
        | 'if' expr
        .
```

7.4.4.1.3 Use Clause

There are times when it is necessary to know that a recipe has been applied, but because the recipe was up-to-date, the recipe body was not run.

```
use_clause
         = /* empty */
         : 'use' compound_statement
```

The use clause is run every time the recipe is applied, even if the recipe is up-to-date. It will be run after the the recipe body, if the recipe body is run. All of the usual percent (%) substitutions and automatic variables will apply.

7.4.4.1.4 Double Colon

Most cookbooks are constructed so that if **cook** finds a suitable recipe for the target it is currently constructing, it will apply the recipe and then conclude that it has finished constructing the target. In some rare cases you will want **cook** to keep going after applying a recipe. To specify this use a ‘‘double colon’’ construction:

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

This operates like a normal explicit recipe, but **cook** will continue on looking for recipes after applying this one. As soon as an applicable ‘‘single colon’’ recipe is found and applied, **cook** will conclude that it has finished constructing the target.

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 gate
      compound_statement use_clause
    .
```

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

Note: if you want the *exprs1* section to be empty you *must* separate the two colons with a space, otherwise **cook** will think this is a ''double colon'' recipe.

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 constructible. 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 constructible, they have been constructed). In this naive form, absent include files result in the recipe not being applied.

7.4.4.2.3 Double Colon
Just as explicit recipes have a ''double colon'' form, so do both types of implicit recipes. The semantics are identical, with **cook** looking for more than one applicable implicit recipe, but stopping if it finds an applicable ''single colon'' implicit recipe.

As stated earlier in this manual, **cook** first scans for explicit recipes before scanning for implicit recipes. If an explicit recipe has been applied, **cook** will not also look for applicable implicit recipes, even if all the applicable explicit recipes were double colon recipes.

7.4.4.3 The Ingredients Recipe Statement
The ingredients recipe has the form

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

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 instantiated.

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. On UNIX this usually means that a shell is invoked to run the command, unless the string contains no shell meta-characters.

The *flags* are those which may be specified in the explicit recipe statement. They have a higher precedence than either the *set* statement or the recipe flags.

Some characters in commands are special both to the shell and to cook. You will need to quote or escape these characters. Each command is executed in a separate process, so the `cd` command will not work, you will need to combine it with the relevant commands, not forgetting to escape the semicolon (`;`) characters.

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 Set Statement
It is possible to override the defaults used by **cook** or even those specified by the *COOK* environment variable, by using the *set* statement.

```
statement
    = 'set' words ';'
    .
```

The flag values are those mentioned in the *flags* clause of the explicit recipe statement. Many command-

line options have equivalent flag settings. There is no ‘‘unset’’ statement, to restore the default settings, but it is possible to set flags the other way, by adding or removing the ‘‘no’’ prefix.

To set flags for individual recipes, use the *flags* clause of the recipe statements.

To set flags for individual commands, use the *flags* clause of the command statements.

7.4.5.3.1 Examples
Fingerprinting is not used by default, because it can cause a few surprises, and takes a little more CPU. To enable fingerprinting for you project, place the statement

```
set fingerprint;
```

somewhere in your *Howto.cook* file. The **-No_FingerPrint** command line option can still override this, but the default behavior will be to use fingerprints.

To prevent echoing of commands as they are executed, place

```
set silent;
```

somewhere in your *Howto.cook* file. The **-NoSilent** command line option can still override this, but the default behavior will be not to echo commands.

7.4.5.4 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 addprefix

The *addprefix"* function is used to add a prefix to a list or words. This function requires at least one argument. The first argument is a prefix to be added to the second and subsequent arguments.

8.1.1 See Also

addsuffix, patsubst, prepost, subst

8.2 addsuffix

The *addsuffix"* function is used to add a suffix to a list or words. This function requires at least one argument. The first argument is a suffix to be added to the second and subsequent arguments.

8.2.1 See Also

addprefix, patsubst, prepost, subst

8.3 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.

8.3.1 Example

The following cookbook fragment shows how to use the [and] function in conditional recipes.

```
#if [and [defined change] [defined baseline]]
...do something...
#endif
```

This fragment will only *do something* if both the *change* and *baseline* variables are defined.

8.3.2 Caveat

This function is rather clumsy, and probably needs to be replaced by a better syntax within the cokbook grammar itself.

This function does not short-circuit evaluation.

8.3.3 See Also

or, not

8.4 basename

The *basename* treats each argument as filenames, and extracts all but the suffix of each filename. If the filename contains a period, the basename is everything up to (but not including) the period. Otherwise, the basename is the entire filename.

8.4.1 Example

Expression	Result
[basename foo.c]	foo
[basename foo/bar.c]	foo/bar
[basename baz]	baz

8.4.2 See Also
dirname, entryname, suffix

8.4.3 Caveat
This function is almost nothing like the unix command of the same name. It operates in this manner for compatibility with other packages.

8.5 cando

This function is used to test whether cook knows how to cook the given targets. For each argument, the result contains either "1" (true) or "" (false).

8.5.1 Caveat
This function is rarely required, since it is inherent in the basic functioning of cook.

8.5.2 See Also
uptodate

8.6 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.

8.6.1 Example

Expression	Result
[catenate a]	a
[catenate a b]	ab
[catenate a " " b]	"a b"

Quotes used in the results for clarity.

8.6.2 See Also
split, unsplit, prepost, join

8.7 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.

The command will not be echoed unless the -No_Silent option is specified on the command line.

8.7.1 Example
Read the date and time and assign it to a variable:

```
now = [collect date];
```

Do not use the collect function to expand a filename wildcard, used the [glob] function instead.

8.7.2 See Also
collect_lines, execute, glob

8.8 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.

8.8.1 Example
To read each line of a file into a variable:

```
files = [collect_lines cat file];
```

Spaces and tabs in the input lines will be preserved in the "words" of the result.

8.8.2 See Also
collect, glob

8.9 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.

8.9.1 Example
This cookbook fragment echoes the number of files, and then the name of the last file:

```
echo There are [count [files]] files.;
echo The last file is [word [count [files]] [files]].;
```

8.9.2 See Also
head, tail, word

8.10 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.

8.10.1 Example
This function is most often seen in conditional portions of cookbooks:

```
if [defined baseline] then
        cc_flags = [cc_flags] -I[baseline];
```

8.11 dir

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

8.11.1 Example

Expression	Result
[dir a]	.
[dir a/b]	a
[dir a/b/c]	a/b

8.11.2 See Also

pathname, entryname, basename, suffix

8.12 dirname

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

8.12.1 Example

Expression	Result
[dirname a]	.
[dirname a/b]	a
[dirname a/b/c]	a/b

8.12.2 See Also

pathname, entryname, basename, suffix

8.13 downcase

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

8.13.1 Example

Expression	Result
[downcase FOO]	foo
[downcase Bar]	bar
[downcase baz]	baz

8.13.2 See Also

upcase

8.14 entryname

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

8.14.1 Example

Expression	Result
[entryname a]	a
[entryname a/b]	b
[entryname a/b/c]	c

8.14.2 See Also
dirname, basename, suffix

8.15 execute

This function requires at least one argument, and executes the command given by the arguments. If the executed command returns non-zero exit status the resulting value is "" (false), otherwise it is "1" (true).

The command will not be echoed unless the -No_Silent option is specified on the command line.

8.15.1 Caveat
This function is not often required as its functionality is available in a more useful for as recipe bodies.

8.15.2 See Also
collect

8.16 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.

8.16.1 Example
To remove the target of a recipe before building it again:

```
%.a: [%_obj]
{
        if [exists [target]] then
                rm [target]
                        set clearstat;
        [ar] qc [target] [%_obj];
}
```

Note: you *must* use the clearstat, because otherwise cook's "stat cache" will be incorrect.

8.16.2 See Also
cando, find_command, uptodate

8.17 filter

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

8.17.1 Example

Expression	Result
[filter %.c a.c a.o]	a.c
[filter %.cc a.c a.o]	

8.17.2 See Also

filter_out

8.18 filter_out

This function requires one or more arguments. The first argument is a pattern, the second and later arguments are strings to match against this pattern. The resulting wordlist contains those arguments which did not match the pattern given as the first argument.

8.18.1 Example

Expression	Result
[filter %.c a.c a.o]	a.o
[filter %.cc a.c a.o]	a.c a.o

8.18.2 See Also

filter

8.19 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.

8.19.1 Example

Some systems require *ranlib*(1) to be run on archives, and some do not. Here is a simple way to test:

```
ranlib = [find_command ranlib];

%.a: [%_obj]
{
        if [exists [target]] then
                rm [target]
                        set clearstat;
        ar qc [target] [%_obj];
        if [ranlib] then
                [ranlib] [target];
}
```

8.19.2 See Also

cando, exists, uptodate

8.20 findstring

The findstring function is used to match a fixed string against a set of strings. This function takes at least one argument. The first argument is the fixed string, the second and subsequent arguments are matched against the first. The result contains one word for each of the second and subsequent arguments, each will either be the empty string (false) or the string to be matched, it a match was found.

8.20.1 Example

Expression	Result
[findstring a a b c]	a "" ""
[findstring a b c]	"" ""

Quotes are for clarity, to emphasize the empty strings. Because the empty string is "false", this can be used in an *if* statement:

```
if [findstring fish [sources]] then
        sources = [sources] hook.c;
```

8.20.2 See Also
match, match_mask, subst, patsubst

8.21 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.

8.21.1 Example
Given a list of C sources files, generate a list of object files as follows:

```
obj = [fromto %.c %.o [src]];
```

8.21.2 See Also
filter, filter_out, subst

8.22 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.

8.22.1 Example
To read the value of the HOME environemnt variable:

```
home = [getenv HOME];
```

Values of variables are not automagically set from the environment, you must set each one explicitly:

```
cc = [getenv CC];
if [not [cc]] then
        cc = gcc;
```

8.22.2 See Also
find_command

8.23 glob

Each argument is treated as a *sh*(1) file name pattern, and expanded accordingly. The resulting list of filenames is sorted lexicographically.

You may need to quote the pattern, to protect square brackets from the meaning cook attaches to them.

Note: The character sequence `/*` is a comment introducer, and is a frequent source of problems when combined with the *glob* function. Remember to quote *glob* arguments which needs this character sequence.

8.23.1 Example
To find the sources in the current directory:

```
src = [glob *.c];
obj = [fromto %.c %.o [src]];
```

8.23.2 See Also
filter, filter_out

8.24 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.

8.24.1 Example
You can iterate along a list using the *loop* statement combined with the *head* and *tail* functions:

```
dirs = a b c d;
src = ;

tmp = [dirs];
loop
{
        tmp_dir = [head [tmp]];
        if [not [tmp_dir]] then
                loopstop;
        tmp = [tail [tmp]];
        src = [src] [glob [tmp_dir]/*.c];
}
```

More efficient ways exist to do this, this an example only.

8.24.2 See Also
count, glob, fromto, prepost, tail, word

8.25 Home

The *home* function is used to find the home directory of the named users. You may name more than one user. If no users are named, it returns the home directory of the current user.

8.26 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.

8.26.1 Caveat

It is often clearer to use the *if statement* than this function.

8.27 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.

This function can also be used for equality testing, just use a single element in the set.

8.27.1 Example

Frequently seen in conditional parts of recipes:

```
%: [%_obj]
{
        [cc] -o [target] [%_obj];
        if [in [target] [private]] then
                chmod og-rwx [target];
}
```

8.27.2 See Also

stringset

8.28 join

The *join* function is used to join two sets of strings together, element by element. The argument list must contain an even number of arguments, with the first have joind pair-wise with the last half. There is no marker of any kind between the lists, so the user needs to ensure they are both the same length.

8.28.1 Example

Expression	Result
[join a b c d]	ac db
[join a b]	ab

8.28.2 See Also

catenate, basename, suffix

8.29 match_mask

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

8.29.1 Example

Expression	Result
[match_mask %.c a.c a.o]	a.c
[match_mask %.cc a.c a.o]	

8.29.2 See Also
filter-out

8.30 matches

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

8.30.1 Example
This function may be used to test for strings which have a particular form:

```
if [matches %1C%2 [version]] then
        cc_flags = [cc_flags] -DDEBUG
```

8.30.2 See Also
filter, filter-out

8.31 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 (sortable) representation of the date and time the files was last modified.

8.31.1 See Also
exists, sort_newest

8.32 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**.

8.32.1 Example
This is often seen in recipes:

```
%1/%0%2.o: %1/%0%2.c
{
        if [not [exists [dirname [target]]]] then
                mkdir -p [dirname [target]]
                        set clearstat;
        [cc] [cc_flags] -I%1 %1/%0%2.;
        mv %2.o [target];
}
```

Note that "`%0`" matches zero or more whole filename portions, including the trailing slash. See the chapter on pattern matching for more information.

8.32.2 See Also
and, or

8.33 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, "system" is assumed. Below is a list of attributes:

system — The name of the operating system **cook** presently being run under. For example: if you were running on SunOS 4.1.3, this would return "`SunOS`".

release — The specific release of operating system, within name, **cook** is presently being run under. For example: if you were running on SunOS 4.1.3, this would return "`4.1.3`".

version — Version information. For SunOS 4.1.3, this would return the kernel build number, for other systems it is often the kernel patch release number.

machine — The name of the hardware **cook** is presently running on. For example: If you were running on SunOS 4.1.3 this would return "`sun4`" or similar.

This function may be abbreviated to "os".

8.33.1 Example
This function is usually used to determine the architecture (either system or machine):

```
arch=[os system]-[os release]-[os machine];
if [matches SunOS-4.1%1-sun4%2 [arch]] then
       arch = sun4;
else if [matches SunOS-5.%1-sun4%2 [arch]] then
       arch = sun5;
else if [matches SunOS-5.%1-i86pc [arch]] then
       arch = sun5pc;
else if [matches ConvexOS-%1-%2 [arch]] then
       arch = convex;
else
       arch = unknown;
```

8.33.2 Caveat
This function is implemented using the *uname*(2) system call. Some systems do not implement this correctly, and therefore this function is less useful than it should be, and needs the pattern match appropach used above.

8.34 Options

This functions takes no arguments. The results is a complete list of *cook* options, exactly describing the current options setting. This intended for use in constructing recursive *cook* invokations.

The option setting generated are a combination of the command line options used to invoke *cook,* the contents of the COOK environment variable, the results of the ‘‘set’’ command and the various ‘‘set’’ clauses.

8.34.1 Example
The top level cookbook for a recursive project structure can be as follows:

```
%:
{
       dirlist = [dirname [glob '*/Howto.cook' ]];
       loop
       {
              dir = [head [dirlist]];
              if [not [dir]] then
              loopstop;
              dirlist = [tail [dirlist]];

              cd [dir]\; cook [options] %;
```

```
        }
}

/*
 * This recipe sets the default.
 * It doesn't actually do anything.
 */
all:;
```

Please note the % hiding on the end of the nested *cook* command, this is how the target is communicated to the nested *cook*.

8.34.2 See Also

The supplied "recursive" cookbook does exactly this. In order to use it, you need a *Howto.cook* file containing the single line

```
#include "recursive"
```

8.35 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.

8.35.1 See Also

and, not

8.36 pathname

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

8.36.1 Example

Relative names are made absolute:

```
pwd = [pathname .];
```

8.36.2 See Also

dirname, entryname

8.37 patsubst

This function requires at least two arguments. Patsubst 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.

8.37.1 Example

Given a list of C sources files, generate a list of object files as follows:

```
obj = [patsubst %.c %.o [src]];
```

8.37.2 See Also

filter, filter_out, subst

See the pattern matching chapter for more information about patterns.

8.38 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.

8.38.1 See Also
addprefix, addsuffix, patsubst, subst

8.39 quote

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

8.39.1 See Also
collect, execute

8.40 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.

8.40.1 Example
This function is used in cookbooks which use the *search_list* functionality:

```
search_list = . baseline;

%.o: %.c
{
        [cc] [cc_flags] [addprefix -I [search_list]] [resolve %.c];
}
```

8.41 shell

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.

The command will not be echoed unless the -No_Silent option is specified on the command line.

8.41.1 Example
Read the date and time and assign it to a variable:

```
now = [shell date];
```

Do not use the shell function to expand a filename wildcard, used the [wildcard] function instead.

8.41.2 See Also
collect_lines, execute, wildcard

8.42 sort

The arguments are sorted lexicographically. Duplicates are *not* removed.

8.42.1 See Also
sort_newest

8.43 sort_newest

The arguments are sorted by file last-modified time, youngest to oldest. File names are resolved first (see the resolve function, below). Absent files will be sorted to the start of the list.

8.43.1 Example
This function is often used to "shorten the wait" when building large project, so that the file you edited most recently is recompiled almost immediately:

```
src = [glob *.c];
obj = [sort_newest [fromto %.c %.o [src]]];
```

8.43.2 See Also
fromto, glob, sort

8.44 Split

The *split* function is used to split strings into multiple strings, given the separator. This function requires at least one argument. The first argument is the separator character, the second and subsequent arguments are to be separated. The result is the separated strings, each as a separate word.

8.44.1 Example

Expression	Result
[split : foo:bar:baz]	foo bar baz
[split " " "New York"]	New York

Each of the words in the result is a separate string.

8.44.2 See Also
unsplit, join, catenate, strip

8.45 stringset

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

8.45.1 Example

Expression	Result
[stringset a b a]	a b
[stringset a b c * a]	a
[stringset a b c - a]	b c

The can be very useful in constructing lists of source files:

```
src = [stringset [glob "*.[cyl]" ] - y.tab.c lex.yy.c];
```

8.45.2 See Also
filter, filter_out, glob, in, patsubst, subst

8.46 Strip

The *strip* function is used to remove leading and trailing white space from words. Internal sequences of white space are replaced by a single space.

8.46.1 Example

Expression	Result
[strip " " "foo " " bar"]	"" foo bar
[strip " really big "]	"really big"

8.46.2 See Also
split

8.46.3 subst
The *subst* function is used to perform string substitutions on its arguments. This functinm requires at least two arguments. The first argument is the "from" string, the second argument is the "to" string. All occurreneces of "from" are replaced with "to" in the third and subsequent arguments.

8.46.4 Example
This is a litteral replacement, not a pattern replacement:

Expression	Result
[subst buffalo cress water.buffalo]	water.cress
[subst .c .o test.c]	test.o
[subst .c .o stat.cache.c]	stat.oache.o

Note that last case: it is not selective.

8.46.5 See Also
patsubst, filter, filter_out

8.47 suffix

The *suffix* function treats each argument as a filename, and extracts the suffix from each. If the filename contains a period, the suffix is everything starting with the last period. Otherwise, the suffix is the empty string (as opposed to nothing at all).

8.47.1 Example

Expression	Result
[suffix a.c foo b.y]	.c "" .y
[suffix stat.cache.c]	.c
[suffix .eric]	""

Quotes used for clarity.

The *suffix* functions in this way to allow sensable results when using the *join* function to re-unite filenames dismembered by the *basename* and *sufix* functions.

8.47.2 See Also
basename, dirname, entryname, join, patsubst

8.48 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.

8.48.1 See Also
count, head, word

8.49 Unsplit

The *unsplit* function is used to glue strings together, using the specified glue. The first argument is the text to go between each of the second and subsequent arguments.

8.49.1 Example

Expression	Result
[unsplit ":" one two three]	"one:two:three"
[unsplit " " four five six]	"four five six"

The quotes are necessary to isolate characters such as colon and space which cook would normally treat differently.

8.49.2 See Also
split, catenate, prepost

8.50 upcase

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

8.51 downcase

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

8.51.1 Example

Expression	Result
[upcase FOO]	FOO
[upcase Bar]	BAR
[upcase baz]	BAZ

8.51.2 See Also
downcase

8.52 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.

8.52.1 See Also
cando

8.53 wildcard

Each argument is treated as a *sh*(1) file name pattern, and expanded accordingly. The resulting list of filenames is sorted lexicographically.

You may need to quote the pattern, to protect square brackets from the meaning cook attaches to them.

8.53.1 Example
To find the sources in the current directory:

```
src = [wildcard *.c];
obj = [patsubst %.c %.o [src]];
```

8.53.2 See Also
filter, filter_out, patsubst

8.54 word

The *word* function is used to extract a specific word from a list of words. The function requires at least one argument. The first argument is the number of the word to extract from the wordlist. The wordlist is the second and subsequent arguments. An empty list will be returned if you ask for an element off the end of the list.

8.54.1 Example

Expression	Result
[word 1 one two three]	one
[word 2 one two three]	two
[word 3 one two three]	three
[word 5 one two three]	

The last element of a list of words may be extracted as:

```
last = [word [count [list]] [list]];
```

8.54.2 See Also
count, head

8.55 words

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.

8.55.1 Example
This cookbook fragment echoes the number of files, and then the name of the last file:

```
echo There are [words [files]] files.;
echo The last file is [word [words [files]] [files]].;
```

8.55.2 See Also
head, tail, word

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 behavior 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 instantiate 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 instantiated 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 instantiated 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 or the target does not yet exist (or the "forced" flag is set in the recipe's *set* clause) the recipe body will be performed. If a recipe has no ingredients, it will not be performed, unless the target does not yet exist, or it is forced.

3. If the target was not in the target list of any explicit recipe, **cook** then scans the instantiated 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.

10.9 File Status

Cook determines the time a file was last modified by asking the operating system. Because this operation tents to be performed frequently, **cook** maintains a cache of this information, rather than make redundant calls to the operating system. Because this information is cached, it is possible for **cook**'s memory of a file's last-modified time to become inconsistent with the file's actual last-modified time. In particular, **cook** doe *not* ask the operating system for the "new" last-modified time of a recipe target once a recipe body is completed. Careful use of the `set clearstat` clause will generally prevent this. For example, the following recipe needs to create a directory when writing its output:

```
bin/%: [%_obj]
{
        if [not [exists bin]] then
                mkdir bin;
        [cc] -o [target] [need];
}
```

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

If there were several programs being cooked, e.g. *bin/foo* and *bin/bar*, the second time **cook** performed the recipe, it would erroneously attempt to make the *bin* directory a second time - contrary to the test. This is because *[exists bin]* used the cache, and nothing tells **cook** that the cache is now wrong. The recipe should have been written

```
bin/%: [%_obj]
{
        if [not [exists bin]] then
                mkdir bin
                        set clearstat;
        [cc] -o [target] [need];
}
```

which tells **cook** that it should remove any files named in the *mkdir* command from the cache.

An alternative way of performing the above example is to the the *mkdir* recipe flag:

```
bin/%: [%_obj]
        set mkdir
{
        [cc] -o [target] [need];
}
```

This flag instructs **cook** to create the directory for the target before running the recipe body. There is a similar *unlink* flag, which unlinks the targets of the recipe before running the recipe body. These two flags take care of most, but not all, uses of the *clearstat* flag.

A second mechanism used by **cook** to determine the last-modified times of files is a file *fingerprint*. This is a cryptographically strong hash of the contents of a file. The chances of two different files having the same fingerprint is less than 1 in 2**200. If **cook** notices that a file has changed, because its last-modified time has changed, a fingerprint is taken of the file and compared with the remembered fingerprint. If the fingerprints differ, the file is considered to be different. If the fingerprints match, the file is considered not to have changed.

This description of fingerprints is somewhat simplified, the actual mechanics depends on remembering two different last-modified times, as well as the fingerprint, in a file called *.cook.fp* in the current directory.

Fingerprinting can cause some surprises. For example, when you use the *touch*(1) command, **cook** will often fail to do anything, and report instead that everything is up-to-date. This is because the fingerprint has not changed. In this situation, either remove the *.cook.fp* file, or use the **-No_FingerPrint** command line option.

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_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).

13.2.3 See Also
The ‘‘library’’ cookbook, for linking C sources into a library.
The ‘‘program’’ cookbook, for linking C sources into a program.

13.3 f77

This cookbook describes how to work with Fortran files.

13.3.1 recipes

%.o: %.f77	Construct object files form Fortran source files.

13.3.2 variables

f77	The Fortran compiler command. Not altered if already defined.
f77_flags	Options to pass to the Fortran compiler command. Not altered if already defined. The default is "-O".
f77_src	Fortran 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.

13.3.3 See Also
The ‘‘library’’ cookbook, for linking Fortran sources into a library.
The ‘‘program’’ cookbook, for linking Fortran sources into a program.

13.4 g77

This cookbook is the same as the ‘‘f77’’ cookbook, but it sets the *f77* variable to the GNU Fortran compiler, `g77`.

13.5 gcc

This cookbook is the same as the ‘‘c’’ cookbook, but it sets the *cc* variable to the GNU C compiler, `gcc`.

13.6 home

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

13.6.1 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.

13.7 lex

This cookbook describes how to work with lex files.

13.7.1 recipes

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

13.7.2 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).

13.8 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.

13.8.1 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.

13.8.2 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.

13.9 print

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

13.9.1 recipes

%.lw: %.ps	Print a PostScript file.
%.lp: %	Print a text file.

13.9.2 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.

13.10 program

This cookbook defines how to construct a program.

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

13.10.1 variables

all	Targets of the all recipe.
install	targets of the install recipe
ld	The name of the linker command. Not altered if already defined. Set to the same as the ‘‘cc’’ variable if set, otherwise set to the same as the ‘‘f77’’ variable if set, otherwise set to ‘‘ld’’.
ld_flags	Not altered if already defined. The default is empty.
ld_libraries	Options passed to the C compiler when linking, these are typically library search paths (`-L`) and libraries (`-l`). Not altered if already defined. The default is empty.

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

13.10.2 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.

13.10.3 See Also
The ‘‘c’’ cookbook, for C sources.
The ‘‘f77’’ cookbook, for Fortran sources.
The ‘‘usr’’ or ‘‘usr.local’’ or ‘‘home’’ cookbooks, for defining install locations.

13.11 rcs

This cookbook is used to extract files from RCS.

13.11.1 recipes

%: RCS/%,v	Extract files from RCS.
%: %,v	Extract files from RCS.

13.11.2 variables

co	The RCS checkout command.
co_flags	Flags for the co command, default to empty.

13.12 recursive

This cookbook may be used to construct recursive cook direwctory structures, where the top-level cookbook only invokes cookbooks in deeper directories.

All largets given to this cookbook result in all sub-directories containing a *Howto.cook* file having **cook** invoked with the same target.

13.12.1 Recipes
The *all* recipe is defined, but it does nothing, it only exists to set the default target name.

13.13 sccs

This cookbook is used to extract files from SCCS.

13.13.1 recipes

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

13.13.2 variables

get	The SCCS get command.
get_flags	Flags for the get command, default to empty.

13.14 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).

13.14.1 recipes

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

13.14.2 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.

13.15 usr

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

13.15.1 variables

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

13.16 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.

13.17 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*.

13.17.1 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].

13.17.2 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).

13.18 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.

fingerprint A cryptographically strong hash of the contents of a file, use to determine if the file contents have changed.

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*. The gate condition is in addition to the requirement that the ingrediaents are cookable.

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