Cook
A File Construction Tool

User Guide

Peter Miller
millerp@canb.auug.org.au

.

This document describes Cook version 2.8
and was prepared 2 February 1999.

1. Introduction

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.

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to see when the prerequisites of a file have changed, implying that the file needs to be recreated as it is logically out of date.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

1.2 Ancient History

Cook was originally developed because I was marooned on an operating system without anything even vaguely resembling *make*(1). This was in 1988. Since I had to write my own, I added a few improvements. When I finally escaped back to UNIX, in 1990, 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, but it comes close. **Cook** has been tested on numerous UNIX flavors. This was made much simpler in 1994 when I started using the GNU Autoconf utility. This means that when you obtain the sources for Cook, all you have to do is run the *configure* script included in the distribution and Cook will be configured for your system. See the `BUILDING` file in the source distribution for more information.

In 1996 Cook had internationalization support added, so that users could have error messages and other warning and informational messages printed in their native language. This was made possible by the GNU Gettext utilities.

In 1997 Cook had a major re-write of significant portions of its inference engine. This enabled the addition of parallel processing support, and simplified adding user-defined functions to the cookbook language.

2. 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.

2.1 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 needs 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.

2.2 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.

2.3 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.

2.4 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).

2.5 The Reference Manual

For more information about the command line arguments and options of the various commands mentioned, you should consult the on-line manual pages. The Cook Reference Manual is also a good source of this information, and is available from the same place as you obtained this manual.

3. 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.

3.1 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.

3.1.1 When is Cook useful?

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

3.1.2 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.

3.2 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*. The information about how to turn **.c** files into **.o** files came from the "`#include "c"`" line, which read in the C recipes distributed with Cook.

3.2.1 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.

3.3 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 many programs, some quite large:

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

As you can see, even for many complex programs, the cookbook is remarkably simple.

4. Cooking in Parallel

Cook is able to use the dependency information in the cookbook to schedule more than one recipe body at once, where they are independent. In large projects this is almost always possible.

Parallel processing is of most use on multi-processor systems. There are cases, however, when running two jobs at once on a workstation can take advantage of disk or network latencies.

Parallel processing requires more resources than the simple case. Because more commands are running, more CPU is required, but also more virtual memory and more temporary file space. You need to be sure that cooking in parallel is a sensible thing to be doing.

4.1 Command Line Option

The `-PARallel` option is used to tell Cook to run the recipe bodies in parallel. By default, 4 jobs run in parallel. You may specify the number of jobs after the option (*e.g.* `--par=2`) if you wish.

4.2 Cookbook Variable

It is also possible to set the number of jobs from within the cookbook by using the `parallel_jobs` variable. This can be used to automate the selection of the number of jobs, based on the current host name:

```
if [not [defined parallel_jobs]] then
{
        host = [os node];
        if [in [host] cerberus] then
                parallel_jobs = 3;
        else if [in [host] zaphod] then
                parallel_jobs = 2;
        else if [in [host] hydra] then
                parallel_jobs = 8;
}
```

In this way, the number of jobs will be set appropriately for each machine, provided the number of jobs was not already set by the command line option.

4.3 Recipe Writing

Most recipes run in parallel without difficulty, however some will require special treatment. The problems arise from conflict for resources – usually temporary files.

The simplest example of this is *yacc*(1). The output filenames are hard-coded, even when you write a more general recipe:

```
%.c: %.y
        single-thread yy.tab.c
{
        [yacc] [yacc_flags] %.y;
        sed "'s/[yY][yY]/%_/g'" yy.tab.c > [target];
        rm yy.tab.c;
}
```

Replacing the `YY` is a common method for getting more than one yacc grammar into a program. We run into trouble with the `yy.tab.c` file because every one of the yacc grammars will need to use the same temporary file name.

The `single-thread` clause tells cook to find something else to do if it discovers that it wants do two of these at the same time.

The temporary file name may not be so evident as in the yacc case. The GNU Autoconf utilities use a number of temporary files in the current directory, but none of them appear in the text of the recipes.

```
%: %.in: config.status
        single-thread conftest.subs
{
        CONFIG_FILES\=[target] CONFIG_HEADERS\= config.status;
}
```

It is common, if your project uses GNU Autoconf, to generate several files in this way. Once the `config.status` script is produced, all of these files will then be candidates for cook to generate – but they can only be done one at a time.

Other resources, such as tape drives, can also be described in the `single-thread` clause. You can do this by device name (*e.g.* `/dev/rmt/0`) or by some descriptive string. The single threading is performed by mutually exclusive string sets, not by inode.

4.3.1 Concurrent Execution Threads

Each recipe, when its actions are executed, is executed within an execution thread. Execution threads share almost everything in common; this includes all of the variables, the state of the "set" statement, the stat cache, *etc*.

If you need to create variable names, or temporary file names, which are unique to a thread, use the `[thread-id]` variable. This variable has a unique value for the life of a thread. No other concurrent thread will have the same value.

Note, however, that the `[thread-id]` values of completed threads will be re-used; this ensures that when it is used to construct variable names, the variables will be re-used. This prevents memory bloat when cooking large projects.

4.4 File Locking

The above discussion applies to utilities which perform no file locking, and thus cannot detect or sequence multiple accesses to a resource. Other programs, such as those which access databases, may have quite capable file locking mechanisms and are able to manage multiple parallel updates on their own, obviating the need for the `single-thread` clause.

4.5 Virtual Machine

It is possible to simulate a parallel machine if you are on a network. Cook is able to distribute tasks to computers on a network, if it is given sufficient information.

The first information Cook requires is the list of machines. This is done using the `parallel_hosts` variable. **Note:** The tasks will be distributed amongst these machines independent of the setting of the `parallel_jobs` variable. *i.e.* even if you are not doing parallel processing.

```
parallel_hosts = larry curly moe;
```

If you want to give one machine more wieghting than the others (say, because it is twice as fast) you simply name it more than once. Cook will use these names in round-robin fashion.

4.5.1 Remote Shell Command

Cook uses the Berkeley *rsh*(1) command to invoke the remote command. You can set the command, or the command and some options, using the `parallel_rsh` variable. The default value is

```
parallel_rsh = rsh;
```

In order to work in a useful way, Cook makes some assumptions about your environment and your account:

- That your system administrators allow *rsh*(1) to be used on your network.
- That your account name is the same on *all* machines (otherwise not even the `rsh -l` *login-name* option will help).

- That the `/etc/hosts.equiv` file, or your `~/.rhosts` file, is set on *all* machines so that you don't need to give a password.
- That all of the necessary files and directories are mounted in exactly the same place on all of the machines; and that they are *the same files* on all machines, via NFS or similar. Automounters can make this especially messy.
- That your account start-up scripts set the necessary environment settings, *e.g.* command search `PATH`, without any intervention required.
- That all of the machines are of the same architecture, or that the architecture doesn't matter.
- That the system time is synchronised on all machines, using *rdate*(1) from *cron*(8), or using NTP, or similar.

4.5.2 Limitations

There are some inherent limitations in the *rsh*(1) protocol.

- Your current environment variable settings are not transferred across. Neither are *ulimit* settings, *etc*. If any are important, you need to write the cookbook to explicitly replicate them.
- The exit status of the remote command is not reported in the exit status of the *rsh*(1) command[1]. There are internal contortions used by Cook to obtain the exit status; error about mysteriously named files usually indicate that one or more of the above assumptions is being broken.

4.5.3 Host Binding

In some cases, such as licensing conditions, some commands will only run on a limited set of hosts. Rather than perform all commands on those hosts, it is possible to bind recipes to specific hosts. This binding overrides the `parallel_hosts` variable.

```
%.c: %.esql
        host-binding shylock
{
        esql %.esql > [target];
}
```

This example says that the embedded SQL preprocessor is only to be run on the database server called "shylock", probably due to usurious licensing fees. However, you may want to perform your other development activities on more lightly loaded machines; this clause only applies to this one recipe, other recipes behave as normal.

The `host-binding` clause may have more than one host named, and they will be used in round-robin fashion. This is a recipe-level variant of the `parallel_hosts` variable.

The `host-binding` clause will apply independent of the setting of the settings `parallel_jobs` and `parallel_hosts` variables.

1. The Berkeley sources certainly don't contain code to do this. Do any other vendors have a more useful implementation?

5. Include File Dependencies

A significant factor in a cookbook accurately describing the dependencies in a program are the include file dependencies. There are three methods for doing this in Cook. The first is easily understandable but is too slow to use on large projects, the second is a little harder to understand, but works well for large projects. The third method is rather convoluted, but works well for projects with many thousands of source files and multiple simultaneous architectures built within the same source tree.

The recipes here are merely examples and starting points; you will almost certainly need to enhance them to suit the needs of your projects. Areas you will need to address include (a) the existence of `cc -I`*path* options, (b) the use of `search_list` variable and the `[resolve]` function, and (c) heterogeneous development. The techniques also apply to other languages, such as Fortran, Pascal and Roff, but each requires a language-specific include scanning program[2].

5.1 The Manual Method

Well, actually there are four methods, if you count maintaining the dependencies manually. This has the serious defect that humans tend to *forget* to update the cookbook. On a large project not all developers are familiar with the workings of Cook, and so they shy away from updating the cookbook. By finding ways to automate include dependency processing, we reduce the risk that a developer will forget to update the cookbook, and we reduce the risk that the cookbook's dependency information is out-of-date.

Automatic include dependency methods described below have flaws, and can never replace a human for flexibility and domain knowledge. On the other hand, humans have better things to do with their time than grope files for include file dependencies (like write neat software).

5.2 Tools

All of the automated include file dependency methods described below use the *c_incl*(1) program included in the Cook distribution. It has a number of options tailored for use with Cook. For exact information about the *c_incl* command, consult the on-line *man*(1) system (it should have been installed) or the Cook Reference Manual.

Other tools are available. The commonest is to use the `gcc -M` option, which produces a list of include files on the standard output. Because the `gcc -M` output is aimed at GNU Make, you will need an *awk*(1) or *sed*(1) script to massage the output into a format suitable for Cook.

5.3 The Small Method

The easiest way to determine a file's include file determine the include dependencies within the recipe's ingredients.

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

Note the second colon – the *second* set of dependencies are only evaluated after Cook has chosen to activate the recipe. This does not guarantee that the file exists yet (it may have to be generated by *lex* or *yacc*), which is why the `--Absent-Program-Ignore` option is required.

This method has the advantage of simplicity. It uses a single recipe which reads the way recipes usually read, and does not contain any unusual constructs.

There are two problems with this method. The first is that it doesn't scale well. When there are only a few source files, the processing burden of running *c_incl* for every **.c** file every time Cook is invoked is hardly noticeable. The *c_incl* program caches the results of its scans, so that is can minimize the length of time taken, and this does help a little. However projects with hundreds or thousands of files find even the cached

2. The *c_incl* program understands Roff, you just need to use the `-r` option.

performance an unreasonable burden; it is constantly re-calculating something which has not changed from one run to the next.

The second problem is that the *c_incl* program is run when the dependency graph is being built, not when it is being walked. This means that the **.c** file (or a subordinate **.h** file) may have been out-of-date at the time. When the graph is walked, it will have been regenerated, and the two sets of include files, those determined by *c_incl* at graph building time, and those seen by *cc* at graph walking time, may not agree – which may result in compile-time errors.

5.4 The Large Method

For projects with large numbers of files, hundreds or even thousands, it is necessary to re-calculate the include file dependencies only when a **.c** file changes, or a subordinate **.h** file. Ideally, Cook should access this information directly, rather than running a program to determine it or to fetch it.

The first task is to move the information which *c_incl* caches into a format that Cook can access directly; Cook can then read in this information as it scans the cookbook. By making a separate ‘‘dependency’’ file for each **.c** file, we can using existing Cook mechanisms to describe how to keep this file up-to-date.

The dependency file is generated and maintained as follows:

```
%.c.d: %.c
{
        c_incl --no-cache %.c
                "--prefix='%.o "[target]": %.c'"
                "--suffix='set nodefault;'"
                -o [target];
}
```

This recipe generates a file which contains a mini-cookbook describing the ingredients of the *object* file. The dependencies are in terms of the object file because if any of the **.h** files change, it is the object file which is out-of-date, not the **.c** file. The mini-cookbook itself is also described, so that if any of the source files change, the mini-cookbook can be brought up-to-date again.

The recipe for the object file is less complicated than in the previous section, because the mini-cookbooks supplement it:

```
%.o: %.c
{
        cc -c %.c;
}
```

The only thing missing is how to get the information in the mini-cookbooks into the main cookbook. This is done with an include directive in the cookbook itself, but a special form of it. The names of the mini-cookbooks can be determined the same way as the names of the object files, and this allows the cookbook fragments such as the following to be written:

```
object_files = [fromto %.c %.o [source_files]];
dependency_files = [fromto %.c %.c.d [source_files]];

#include-cooked [dependency_files]
```

The `#include-cooked` directive says to include the named files (there may be more than one) if the file exist. Once the cookbook (and its includes) have been read in, the files included with this directive are checked to see if they are up-to-date. If they are not, then they are re-cooked, and then Cook starts over again; this time with up-to-date include dependencies.

The advantage of the method is that if the source files don't change, the dependency information is not recalculated, this can result in significant savings. Also, no processes are invoked if nothing has changed, Cook reads the information directly. Because file opens are significantly cheaper than process invocations, this results in a significant performance improvement.

The disadvantage of this method is that it is harder to describe and harder to implement. To the uninitiated the cookbook looks incomplete and overly complex.

Another problem is that if you delete an include file, Cook will complain that it is unable to derive the dependency file because the include file is not present. Simply delete the dependency file and start again. To avoid the problem, remove references to include files, and re-build, before deleting the include files. This problem is seen from time to time, but does not present a huge problem in normal practice.

5.5 The Cascade Method

When large numbers of files are involved, it becomes clear that the more popular include files are being scanned repeatedly. This can be un-necessarily time-consuming when a popular include file is touched, as the dependency files of all `.c` files which reference it, even indirectly, must be re-calculated.

There is also a problem when you are attempting to perform heterogenous builds for multiple architectures out of the same sources. This is typically done by inserting the architecture name into the object file path as a directory. This presents another problem: nominating all of the architectures on the left-hand-side of the regenerated dependency recipes. Especially if you add another one after the fact - now all the existing dependency files must be recalculated, merely to add the new architecture.

An alternative is to scan each of the source files and include files once, and request cook to combine them together at build time, rather than at dependence scan time. This is done using `cascade-for` recipes. These recipes nominate additional ingredients (on their right-hand-size) if any of the files on their left-hand-size appears in an ingredients list.

```
cascade-for foo.c = bar.h;
```

This recipe says that any recipe which has *foo.c* for an ingredient, also has *bar.h* for an ingredient.

This takes care of the heterogeneous case, because while the recipes remain specified in a simple manner, *viz:*

```
%1/%0%.o: %0%.c
{
        %1-gcc -o [target] -c %0%.c;
}
```

Any and all of them which compile *foo.c* will depend on *bar.h* from the `cascade-for` recipe. (This example assumes that you are using *gcc*(1) in the usual way, and that your architecture names match the GNU target names.)

The dependency files are generated and maintained in much the same way as before, except that you need two: one for `.c` files and one for `.h` files:

```
%0%.c.d: %0%.c
        set no-cascade
{
        c_incl --no-cache --no-recurs %0%.c
                "--prefix='cascade-for %0%.c ='"
                "--suffix=';'"
                -o [target];
}
%0%.h.d: %0%.h
        set no-cascade
{
        c_incl --no-cache --no-recurs %0%.h
                "--prefix='cascade-for %0%.h ='"
                "--suffix=';'"
                -o [target];
}
```

You will also need to add the `.h.d` files to the `#include-cooked` lines, to ensure they are generated. If there are any generated `.c` or `.h` files, you will need to mention these, too.

5.6 Dependencies on Derived Files

If the relationship between a target and a derived ingredient appears only in a derived cookbook, it is likely that a clean build (solely from primary source files) will fail. It is recommended that relationships such as this be placed in a primary source cookbook. Cook looks for such dependencies, and will warn you about them.

An example of this is commonly seen when using the `-d` option with *yacc*(1). If you have a separate lexical analyzer (the usual reason for using `-d`) it will need to include the generated token definition file.

When you first add the *yacc*(1) grammar definition, Cook will generate both the `.c` and `.h` file from the usual yacc recipes. It is only later, when you have cleaned out all derived files (including the dependency files) that you may have problems. Where is it recorded that Cook needs to regenerate the token definition file before it can determine the include dependencies of the lexical analyzer? (They were in a `.d` file which was ‘‘cleaned’’ away.)

Cook will detect this situation at the first possible moment, and warn you. But placing the dependency in a non-derived cookbook (*e.g.* `Howto.cook`) the warning will go away, and you will be able to do reliable clean builds.

If you are convinced that Cook is *always* wrong in your case, it is possible to suppress this warning. Place the line

```
set no-include-cooked-warning;
```

in your main cookbook, and the warning will not be issued.

Suppressing the warning could lead to problems. It is often better to add the ingredients recipe given in the warning to the cookbook, even if you think it is redundant. This disables a single instance of the warning, rather than all of them – subsequent *valid* instances will still be reported. (Implicit ingredients recipes, rather than explicit ones, are a useful alternative if you have a consistent pattern.)

5.7 Renaming Include Files

A consistent problem when you have automatically generated include dependencies is that when you move an include file, Cook complains that a required ingredient does not exist.

The easiest way to avoid this is to do a few things before you build again after moving the include file.

- Move the include file to the new name.
- Where the include file was *from*, put a file containing the line
  ```
  #error "I'm not here"
  ```
  to make Cook happy (the ingredient will exist), but also have the compiler generate an error if you miss a reference to it.
- Edit all the references to the old include file name to reference the new name. Don't worry if you miss one or two, the previous step will catch it.
- Rebuild the program. Cook will automatically re-calculate all of the include dependences and then recompile.
- If you missed one of the include file references, Cook will not complain, but the compiler will. (This assumes you are using whole-project builds, as described in the *Large Projects* chapter.)
- Once the program builds cleanly, remove the fake old include file, because you know for certain that there are no longer any references.

6. Building Large Projects

This chapter covers some of the issues you may come across in building large projects. It gives a skeleton for how you could use Cook to build a medium-to-large projects, and even covers some heterogenous build issues. It is expected that you will use this chapter as a guide; your development environment, and the shape of each individual project, mean that you will probably change this to suit your own needs.

The material in this chapter uses many, many features of Cook. If you are not familiar with Cook, you may want to read the rest of this User Guide to get a good idea of Cook's features and capabilities. Even if you are familiar with Cook, you may need to refer to the language guide and built-in function descriptions from time to time.

6.1 Whole Project Build

The skeleton given here builds the whole project as a single Cook invocation, even when the project consists of tens thousands of individual source files. This is distinct from a build process which has Cook recursively invoking itself in deeper directories, or a shell script doing much the same. Some of the advantages of doing whole project builds will be discussed in a later section. For now it is sufficient to say that experience has shown repeatedly that this method does scale to significant projects.

The first thing about a single build pass is that it happens relative to a single fixed place. The logical place is the top of the project source tree[3]. This works well the the *search_list* functionality, mentioned below, which simplifies the structure of private work areas.

6.1.1 Project Directory Structure

In the examples use in this chapter, the following directory structure is assumed:

```
Project
├─ Howto.cook
├─ library
│   ├─ source1.c
│   ├─ source2.c
│   └─ etc...
├─ include
│   ├─ api1.h
│   ├─ api2.h
│   └─ etc...
├─ program1
│   ├─ source3.c
│   ├─ source4.c
│   └─ etc...
└─ program2
    ├─ source5.c
    ├─ source6.c
    └─ etc...
```

Below the project directory is a `library` directory, which contains functions common to all of the programs. All source files in this directory are to be compiled, and linked into a library. When the programs are linked, they will all reference this library.

Next to the `library` directiry is the `include` directory. This describes interfaces and data shared by the project. Information which is proivate to the internals of the library or a programs belongs there, not in the shared include space.

The rest of the directories below the project directory are programs to be built. The sources files in each are to be compiled and linked, together with the common library, to form the programs. The name of the

3. If you ever want to use Aegis for configuration management, this is what Aegis expects.

program will be taken from the directory.

This is a common enough picture, repeated for many projects. Your individual projects may vary in the details; you may have more directory levels below the `library` directory, or all of your programs may be below a single `command` directory. With simple changes to the examples given in this chapter, you will be able to cope with just about any project structure.

6.1.2 File Manifest

There are many ways of discovering the source files you are working with. Many configuration management systems are able to give you a liost of them. For example, if you were using Aegis, you would say

```
change_files =
        [collect aegis -l cf -terse -p [project] -c [change]];
project_files =
        [collect aegis -l pf -terse -p [project] -c [change]];
manifest =
        [sort [change_files] [project_files]];
```

If you were using RCS, you could find all the the RCS files, and reconstruct the original filenames from them, *viz:*

```
manifest =
        [fromto ./%0RCS/%,v %0%
                [collect find . -path "*/RCS/*,v" -print]
        ];
```

Or you could simply scan the directory tree:

```
manifest =
        [fromto ./%0% %0%
                [collect find . ! -type d -print]
        ];
```

This is will find too much, but what follows will not be altered by this. If you want to get more advanced, however, it helps to have an accurate primary source file manifest.

6.1.3 Compiling C Sources

Recalling the the build will take place from the top of the source tree, this means that there it is going to have to be directory components in the filenames in the command executed by Cook, and in the recipes Cook is to use.

This chapter uses C examples, but the same techniques work just as will with Fortran or Groff, or anything else. Most of it maps directly; you may need to adjust for your specific compiler behaviour.

This chapter starts with the lowest level of building a project, the individual source files, and works its way upwards, building on the examples until the whole project, including the library and all programs are linked in a single pass.

So, when cooking C sources, you need recipes of the form

```
cc = gcc;
cc_flags = -g -Wall -O;

%0%.o: %0%.c
{
        [cc] [cc_flags] -c %0%.c
                -o [target];
}
```

The ‘‘`%0`’’ part of the patterns matches zero or more directory parts. If your compiler insists on putting the output (`.o`) file into the current directory (the top level one) you will need to move it, after:

```
%0%.o: %0%.c
{
```

```
                [cc] [cc_flags] -c %0%.c;
                mv %.o [target];
        }
```

But, most existing sources will be assuming that most of their include files are in the same directory as the source files. We need include options to indicate this. This is most easily done by using more pattern elements

```
        %1/%0%.o: %1/%0%.c
        {
                [cc] [cc_flags] -I%1 -c %0%.c
                        -o [target];
        }
```

Or by using the dirname of the source file

```
        %0%.o: %0%.c
        {
                [cc] [cc_flags] -I[dirname %0%.c] -c %0%.c
                        -o [target];
        }
```

For structures more than 2 directories deep, these two produce different options. Depending on your project structure, if you have deep directories, one will probably be more suitable than the other. One elegant use for deeper directory structures is to reflect the C++ inheritance hieracy directly in the directory hierachy.

The common include file will also need to be searched. Because of where the command is issued, it is rather simple to add the `include` directory, *viz:*

```
        %0%.o: %0%.c
        {
                [cc] [cc_flags]
                        -I[dirname %0%.c] -Iinclude
                        -c %0%.c -o [target];
        }
```

It is important to note that all of these recipes, and the commands they execute, are independent of the location of the source file. It is possible to customize the `cc-flags` used, based on the target file, or even the directory conating the file, without compromising the generality of the recipe[4].

6.1.4 Tracking Include Dependencies

When it comes to tracking include dependencies using *c_incl*, you need to remember, again, that the Cook happens from a single place. All of the recipes that *c_incl* writes for you must be *relative to that place*.

Continuing our example, and assuming we are using the cascase include method described in the previous chapter, we need include dependency files which look similar to

```
        cascade program1/source3.c =
        include/api1.h
        ;
```

Working backwards, we need to create the dependency file using the following recipe:

```
        %0%.c.d: %0%.c
                set nocascade
        {
                c_incl -nc -ns -nrec
                        -I[dirname %0%.c] -Iinclude
                        %0%.c
                        -prefix "'cascade %0%.c ='"
                        -suffix "';'"
                        -o [target];
```

4. Hint: use a function, and pass `[target]` as the argument.

```
}
```

For other source languages, you will need to use the *c_incl --language* option.

The dependency files need to be included in the magic way so that Cook will build them again if they are out of date. This method needs the source file manifest to know their names.

```
dep-files =
        [addsuffix .d
                [match_mask %0%.c [manifest] ]
                [match_mask %0%.h [manifest] ]
        ];
#include-cooked [dep-files]
```

These files will only be re-calculated if they are out of date; they are small and often zero-length, and so are usually very quick to read, adding little to the time it taks to read the cookbook.

Notice that adding a new source file will automaticaly cause it to be scanned for include dependencies, without modifucation to the cookbook.

6.1.5 Linking Libraries

To link libraries with a generic recipe, you need a generalized way of specifying their contents. A little trickery with constructed variable names does the job:

```
%/lib%.a: [[target]_obj]
        set unlink
{
        ar cq [target] [[target]_obj];
}
```

The right-hand-side of recipes has late binding, and we use the name of the target to tell us the name of the variable which holds all of the object files. Assigning this variable looks bizarre, but it looks more logical as you have more and more of them...

```
library/liblibrary.a_obj =
        [fromto %0%.c %0%.o
                [match_mask "library/%0%.c" [manifest] ]
        ];
```

The great thing about this construct is that you can build a loop, using Cook's loop statement, that assignes a variable for each of your libraries, if you have more than one.

Notice that adding a new library source file will automaticaly cause it to be compiled into the library, without modifucation to the cookbook.

6.1.6 Linking Commands

We'll use a similar trick for each of the programs you want to link... First the link line

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

Then the objects variable. Note how we add a library *filename* here, this will still only use the library portions actually referenced, not the whole library, so it won't bloat your programs.

```
bin/program_obj =
        [fromto %0%.c %0%.o
                [match_mask program/%0%.c [manifest] ]
        ]
        library/liblibrary.a
        ;
```

Notice that adding a new program source file will automaticaly cause it to be compiled and linked into the program, without modification to the cookbook.

The loop construct tends to obscure things, which is why the essential assignment was given first. This next fragment shows the whole loop.

```
programs =
        [fromto %/main.c %
                [match_mask %/main.c [manifest] ]
        ];
program_list = [programs];
loop
{
        program = [head [program_list]];
        if [not [count [program]]] then
                loopstop;
        program_list = [tail [program_list]];

        bin/[program]_obj =
                [fromto %0%.c %0%.o
                        [match_mask [program]/%0%.c
                                [manifest]
                        ]
                ]
                library/liblibrary.a
                ;
}
```

And now tell Cook you actually want it to do something, like build all of the programs...

```
all: [addprefix bin/ [commands]];
```

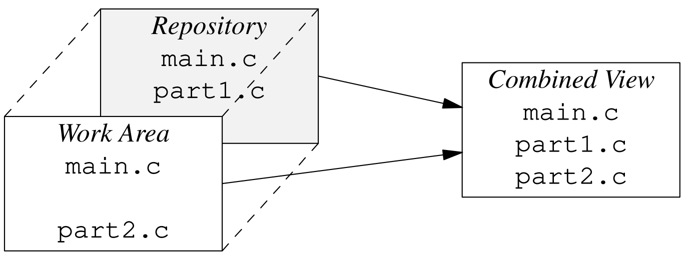
Notice they way the `commands` variable is constructed: just adding a new command (and its `main.c` file) will automatically cause it to be built, without modification to the cookbook.

6.2 Private Work Areas

This chapter is about large projects, but large projects usually means large numbers of developers. The directory structure and cookbook presented so far does not immediately lend itself to use by multiple developers.

6.2.1 Directory Structure

The method suggested here uses Cook's *search_list* functionality, which nominates a search list of directories that Cook looks in to find the files named in the recipes. This can be used to overlay a private work area on top of a master repository.



When recipes are run, the results are written into the work area, which means that the repository can be completely read-only.

It follows from this, that the directory structure of the work area exactly parallels the directory structure of the repository. *Except* you only check out files into your work area that you actually need to change.

6.2.2 Finding the Cookbook

Setting the search list is done with a simple assignment. In your work area, create a simple `Howto.cook` file, containing only 3 lines:

```
set mkdir;
search_list = . /project/repository ;
#include /project/repository/Howto.cook
```

You only use this file if you don't need to modify the cookbook itself. You can make it work always, even if you are modifying the cookbook, by giving the cookbook a different name (`main.cook`), and changing `Howto.cook` to always read

```
set mkdir;
search_list = . /project/repository ;
#include [resolve main.cook]
```

The `[resolve]` function walks the search list, looking for the file[5]. This gives you access to Cook's internal search mechanism. However, we also need to modify each of the recipes to take the search list into account.

The unexplained `mkdir` flag is used to request that directories be automatically created before recipe bodies are run. This may be necessary, for example, if a `.c` file in the repository needs to be recompiled because a `.h` file in the work area has been changed.

6.2.3 File Manifest

The files could be in either of two places. You need to merge them. Most configuration management tools do this for you; in this example we'll scan the directory trees again. Fortunately, Cook comes with a tool to do this efficiently.

```
all_files_in_. = ;
#include manifest.cook
manifest = [all_files_in_.];

/* This reduces re-scanning to a minimum. */
set fingerprint;

%0manifest.cook: ["if" [in "%0" ""] "then" "." "else" "%0"]
        set mkdir
{
        cook_bom /* Bill Of Materials */
                [addprefix '--dir=' [search_list]]
                [need] [target] ;
}
```

At the end of this fragment, the `manifest` variable contains a complete list of all files in the directory tree(s). This variable may then be taken apart with the `match_mask` function to build ingredients lists.

The constructed *manifest.cook* files work for both the top-level directory and individual sub-directories.

6.2.4 Compiling C Sources

The C compilation recipe needs to be changed to read...

```
%0%.o: %0%.c
{
        [cc] [cc_flags]
                [prepost "-I" /[dirname %0%.c] [search_list]]
                [prepost "-I" "/include" [search_list]]
                -c [resolve %0%.c]
                -o [target];
}
```

This ensures that the rights places are searched for include files.

5. The search list defaults to just dot (the current directory) if not set.

6.2.5 Tracking Include Dependencies

A similar change needs to be made to the include dependencies recipe...

```
%0%.c.d: %0%.c
        set nocascade
{
        c_incl -nc -ns -nrec
               [prepost "-I" /[dirname %0%.c] [search_list]]
               [prepost "-I" "/include" [search_list]]
               [resolve %0%.c]
               -prefix "'cascade %0%.c ='"
               -suffix "';'"
               [addsuffix "-rp=" [search_list]]
               -o [target];
}
```

Note that the form of the output of this recipe *does not* change. This means that the recipes it writes work even if you subsequently copy a file from the repository to the work area, or uncopy one.

6.2.6 Linking Libraries

The library recipe needs few modifications.

```
%/lib%.a: [[target]_obj]
        set unlink
{
        ar cq [target] [resolve [[target]_obj]];
}
```

The variable assignment given above requires no modifications.

6.2.7 Linking Commands

The command linking recipe requires few modifications.

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

The variable assignment needs no modifications.

6.3 Whole Project Build Advantages

The advantage of using a whole project build is that the dependency graph is complete, and the order of traversal may be freely determined by Cook. Breaking the build into fractured segments denies Cook access to the whole graph, and dictates the order of traversal to one which, in the light of the entire graph, would be incorrect.

It greatly simplifies the creating of work areas for developers, by using Cook's *search_list* functionality.

A whole project build also permits the *cook -continue* option to work in the presence of a wider range of errors.

The whole project build also permits the *cook -parallel* option to parallelize more operations.

6.4 Heterogenous Build

Large projects frequently involve numerous target architectures. This may be in the form a multiple native compilations, performed in suitable hosts, or it may take the form of cross-compilation.

In this example, we assume that the GNU C Compiler (GCC) is being used. When GCC is installed as a cross compiler, the command names (`cc`, `as`, `ld`, *etc*) are installed with the architecture name as a prefix. For consistency, the native compiler is installed with its own architecture names as a prefix, in addition to the more commonly used `gcc` command. This example will exploit this normal installation practice.

6.4.1 Cross Compiling C Sources

In order to support cross compiling, the C compilation recipe needs to be changed to read...

```
%1/%0%.o: %0%.c
        host-binding [defined-or-null %1-hosts]
{
        %1-gcc [cc_flags]
               [prepost "-I" /[dirname %0%.c] [search_list]]
               [prepost "-I" "/include" [search_list]]
               -c [resolve %0%.c]
               -o [target];
}
```

This uses the first directory element of the *target* to be the architecture name. This allows multiple architectures to be compiled in the same source tree, simultaneously.

Because of the practice of installing a duplicate GCC in the same form as the cross compilers, this same recipe continues to work for native builds.

The *host-binding* line tells Cook to run the command on one of the hosts nominated in a variable named for the architecture (or as a native cross-compiler of no such variable exists). (The `defined-or-null` function is available in the ‘‘functions’’ library distributed with Cook.)

Remembering these architectures follow the GNU convention, these lines could read

```
i386-linux-hosts = fast faster fastest ;
```

This will do two things for you: first, it will always execute linux compiles on linux hosts even when Cook is not executed on one; second, it will use more than one of them whn you use the `--parallel` option.

6.4.2 Tracking Include Dependencies

Because of the cascade form of include dependency, there is no need to do anything different for include dependencies, even if you add another architecture some time in the furture.

6.4.3 Linking Libraries

The library recipe needs few modifications.

```
%1/%/lib%.a: [%/lib%.a_obj]
        set unlink
{
        %1-ar cq [target] [resolve [%/lib%.a_obj]];
}
```

The variable assignment given above requires no modifications.

6.4.4 Linking Commands

The command linking recipe requires few modifications.

```
%1/bin/%: [bin/%_obj]
        set mkdir
{
        %1-gcc -o [target] [resolve [bin/%_obj]];
}
```

The variable assignment needs no modifications.

6.4.5 What to Build

The list of what to build becomes more interesting. You can nominate any and all architectures for which you have cross compilers, or native compilers and native hosts.

```
all:
        [addprefix i386-linux/bin/ [commands]]
        [addprefix sparc-linux/bin/ [commands]]
        [addprefix sparc-solaris2.0/bin/ [commands]]
        [addprefix m68k-sunos4.1.3/bin/ [commands]]
        ;
```

All of these architectures will be built in a single Cook invocation, on appropriate machines if necessary. The use of `--continue` and `--parallel` work over the entire scope of the build.

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 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:

else	host-binding	loopstop	single-thread
fail	if	return	then
function	loop	set	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. If you really want a newline within a string, use the `\n` escape.

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. For example, many small programs can be developed using the following simple cookbook:

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

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

%.c.d: %.c
{
```

```
        c_incl -prefix "'%.o "[target]": %.c'" -suffix "';'"
              -no-cache %.c > [target];
}

#include-cooked [fromto %.o %.c.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.

There are times when you don't want the `#include-cooked` directives to be acted upon. You can override it using the `--no-include-cooked` command line option, but it is often easier to use the `[command-line-goals]` variable, and say something like

```
#if [not [match %1clean%2 [command-line-goals]]]
#include-cooked [fromto %.o %.c.d [obj]]
#endif
```

This construct means that whenever an explicit "`clean`" goal (or similar) is requested, the `#include-cooked` lines will not be performed. This is sensible, because cleaning actions usually remove dependency files; there is no point making sure they are up-to-date first.

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` *variable*`]`

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` *variable*`]]`

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

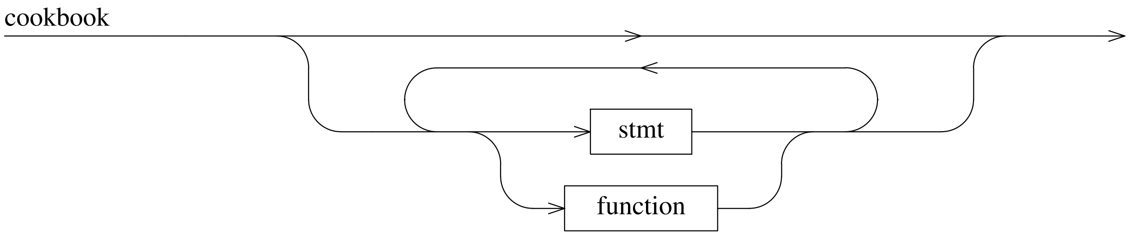
Any pragma extensions not recognized will be ignored.

7.3 Syntax and Semantics

The syntax is described using ‘‘train track’’ diagrams, with prose descriptions of the related semantics.

7.3.1 Overall Structure

The general form of the cookbook is defined as



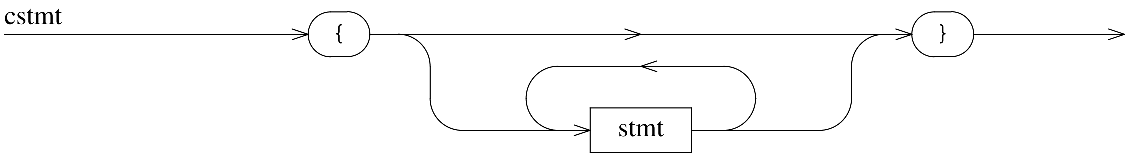
A cookbook is defined as a sequence of statements. Each statement statement is executed. For a definition of what it means when a statement is executed, see the individual statement definitions.

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

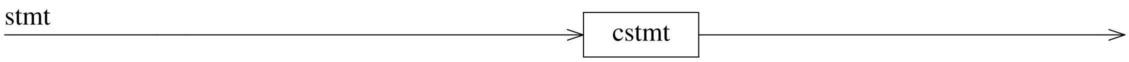
Please note that a statement is not always evaluated when is is read, but at specific, well defined times.

7.3.2 The Compound Statement

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



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

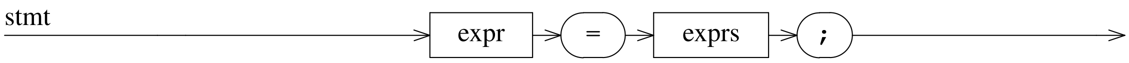


7.3.3 Variables and Expressions

Cook provides variables to the user to simplify things.

7.3.3.1 The Assignment Statement

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

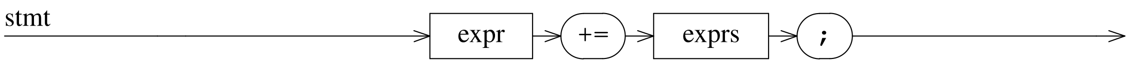


When this statement is executed, the variable whose name the left hand expression evaluates to will be assigned the value that the right hand expression list evaluates to.

Note: It is possible to over-ride the value of built-in functions and variables with this statement. This will not produce an error message, however it is usually not desirable as it will change the meaning of the rest of your cookbook.

7.3.3.2 The Assign-Append Statement

It is possible to append to the value of variables with the following statement.

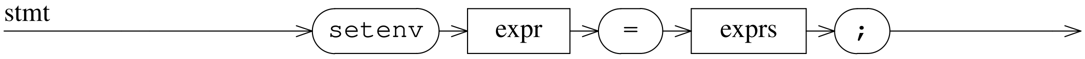


When this statement is executed, the variable whose name the left hand expression evaluates to will have its value appended by the value that the right hand expression list evaluates to. Expression values are lists of words, appending means to append to the word list; it does *not* mean appending to the last string of the value.

Note: It is possible to over-ride the value of built-in functions and variables with this statement. This will not produce an error message (unless evaluating them with no arguments is an error), however it is usually not desirable as it will change the meaning of the rest of your cookbook.

7.3.3.3 The Setenv Statement

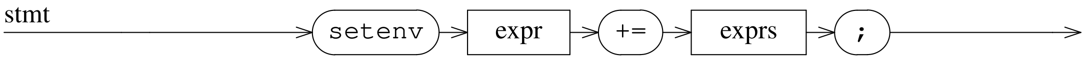
It is possible to assign to environment variables with the following statement.



When this statement is executed, the environment variable whose name the left hand expression evaluates to will be assigned the value that the right hand expression list evaluates to. It is an error if the variable does not already exist.

7.3.3.4 The Setenv-Append Statement

It is possible to append to the value of an environment variables with the following statement.

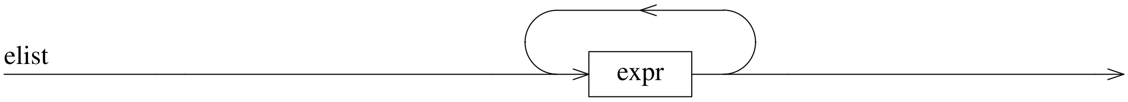


When this statement is executed, the environment variable whose name the left hand expression evaluates to will have its value appended by the value that the right hand expression list evaluates to. Evaluation is analogous to the assign-append statement.

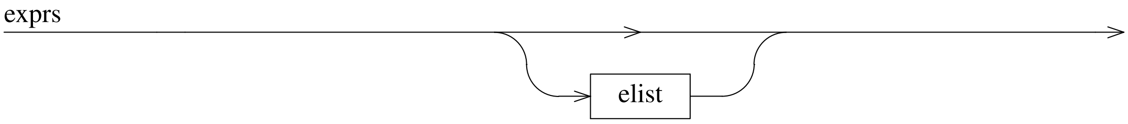
7.3.3.5 Expressions

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

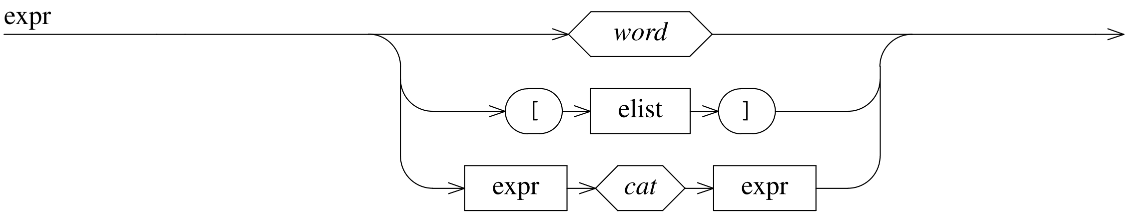
The *elist* is a list of at least one expression,



whereas the *exprs* is a list of zero or more expressions.



An expression is composed of words, variable references, function invocations, 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`".



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.

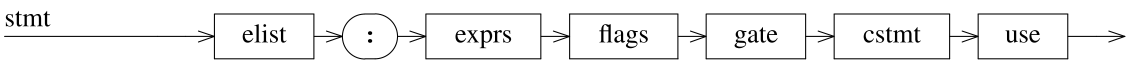
The *cat* operator works as one would expect, joining the last word of the left expression and the first word of the right expression together, and otherwise leaving the order of the expressions alone. One usually uses the trivial case of single word expressions. For more complex concatenations, see the [catenate] and [join] built-in functions.

7.3.4 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.

7.3.5 The Explicit Recipe Statement

The explicit recipe has the form



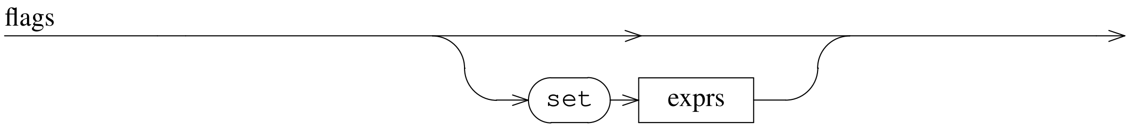
The target(s) of the recipe are to the left of the colon, and the ingredients, if any, are to the right. The statements, usually commands, to perform to cook the target(s) are contained in the compound statement. The expressions are only evaluated into words when the recipe is

The target expressions and recipe flags are evaluated when the recipe is instantiated. The ingredients expressions and the recipe gate are evaluated at graph building time. The body and use statements are executed at graph walking time.

The recipes also take a "*host-binding*" attribute. See the chapter on Cooking in Parallel for how this is attribute is written and used.

7.3.5.1 Recipe Flags

The *flags* are defined as follows.



Recipe flags are evaluated when the recipe targets are evaluated. At this time, *none* of the [target], [targets], [need] or [younger] variables are set, and neither are any of the pattern matches (%, %1, *etc*) available.

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, from the last-modified time, it is fingerprinted, and the fingerprint compared with what it was in the past. The file has changed 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.

gate-after-ingredients — This flags causes the recipe gate to be evaluaged after the ingredients have been evaluated and determined to be cookable. This is usually the default.

gate-before-ingredients — This flag causes the recipe gate to be applied before the ingredients are evaluated and determined to be cookable. This is useful if the ingredients evaluation itself needs to be conditional.

implicit-ingredients — This flag may be used to specify that a recipe's ingredients may be satisfied by implicit recipes. This is usually the default.

no-implicit-ingredients — This flag may be used to specify that a recipe's ingredients may not be satisfied by implicit recipes; this is of most use with utilities such as RCS where the recipe writer knows that the ingredients cannot be constructed.

include-cooked-warning — This flag may be used to enable warnings when the relationship between a target and a derived ingredient appears only in a derived cookbook. This is usually the default. This flag is only meaningful at the cookbook level, it is not meaningful for individial recipes or commands.

no-include-cooked-warning — This flag may be used to disable warnings when the relationship between a target and a derived ingredient appears only in a derived cookbook. This flag is only meaningful at the cookbook level, it is not meaningful for individial recipes or commands.

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.

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, so that erroneous targets will be re-cooked.

recurse — If this flag is specified, recipes will recurse 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 commands within the actions bound to the recipe will not be echoed.

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

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.

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[6].

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 usually 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.

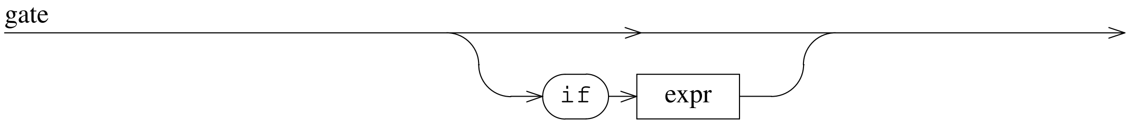
unlink — If the *unlink* flag is specified, of any targets will be unlinked before the actions bound to the recipe are performed.

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.

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.

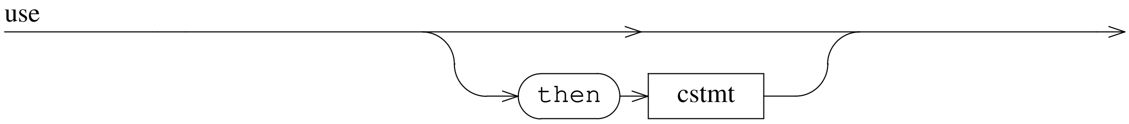
7.3.5.2 Recipe Gate

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.



7.3.5.3 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.



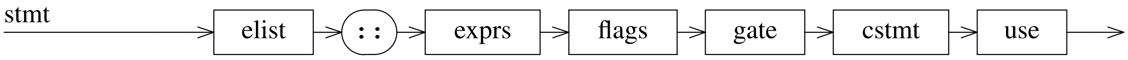
The use clause is run every time the recipe is applied, even if the recipe is up-to-date. It will be run after

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

the the recipe body, if the recipe body is run. All of the usual percent (%) substitutions and automatic variables will apply.

7.3.5.4 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:



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.

7.3.6 The Implicit Recipe Statement

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

7.3.6.1 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

```
%: %.gz
{
        gzcat %.gz > %;
}
```

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

7.3.6.2 Complex Form

The implicit recipe recipe has a second form where there are two sets of ingredients, separated by another colon. 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 -api %.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[7].

The naive form of this recipe

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

7. This is not the recommended way of determining C include dependencies, see the ‘‘Include Dependencies’’ chapter for more information.

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.

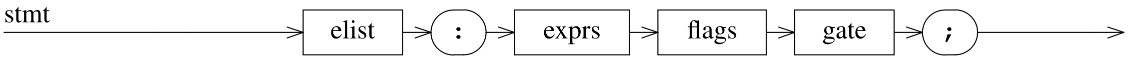
7.3.6.3 Double Colon

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.

7.3.7 The Ingredients Recipe Statement

The ingredients recipe has the form



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.

7.3.8 The Cascade Recipe Statement

The cascade recipe statement has the form

This recipe specifies on its right-hand-side additional ingredients for any recipe which has ingredients mentioned on the left-hand-side of this cascade recipe.

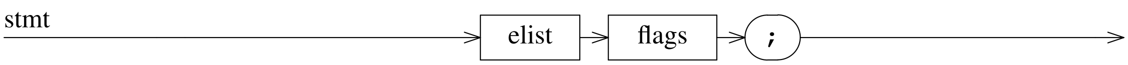
Unlike all other recipe forms, both the left-hand-side *and* the right-hand-side are evaluated when the recipe is instantiated.

7.3.9 Commands

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

7.3.10 The Simple Command Statement

The simplest command form is



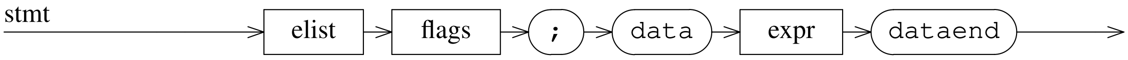
When executed, the *elist* is evaluated into a word list and used as a command to be passed to the operating system. On UNIX this usually means that a shell is invoked to run the command, unless the string contains no shell meta-characters.

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.

7.3.11 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.



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; if it is not alone, it is not a **dataend** keyword and will not terminate the expression.

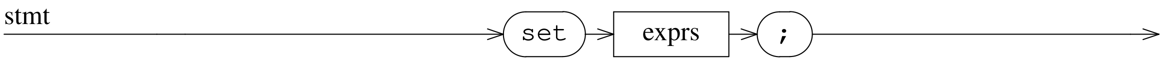
An example of this may be useful.

```
/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.

7.3.12 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.



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.

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

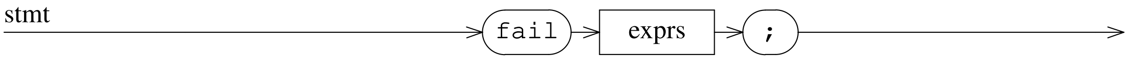
To prevent echoing of commands as they are executed, place

```
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.

7.3.13 The Fail Statement

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



This is hugely useful when programs do not return a useful exit status, but *do* fail. If they have printed an error message, but not produced the output file, you could use the Fail statement without arguments:

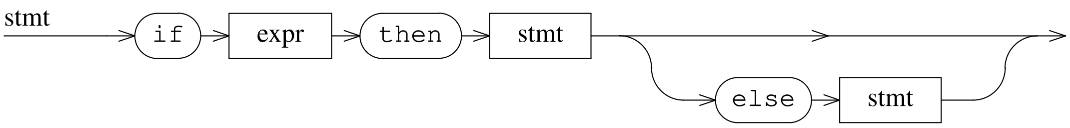
```
fred: other stuff
        set unlink
{
        brain-dead [need] -o [target];
        if [not [exists [target]]] then
                fail;
}
```

If you give the Fail statement any arguments, they will be printed as an error message before the recipe fails:

```
fred: other stuff
        set unlink
{
        brain-dead [need] -o [target];
        if [not [exists [target]]] then
                fail Did not produce [target] file.;
}
```

7.3.14 The If Statement

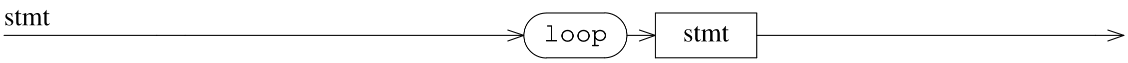
The if statement has one of two forms.



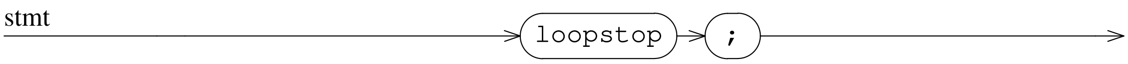
In nested if statements, the **else** will bind to the closest *else*-less *if*. An expression is false if and only if all of its words are null or it has no words.

7.3.15 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.



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

Here is an example of how to use the loop statement:

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

tmp = [dirs];
loop
{
```

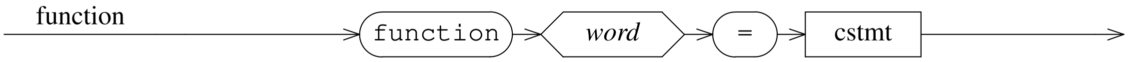
```
            tmp_dir = [head [tmp]];
            if [not [tmp_dir]] then
                    loopstop;
            tmp = [tail [tmp]];
            src = [src] [glob [tmp_dir]"/*.c"];
    }
```

7.3.16 Functions

It is possible to defined your own functions.

7.3.16.1 Function Definition

User-defined functions are specified using something similar to an assignment.

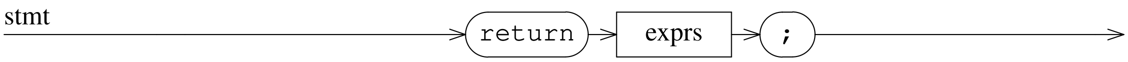


Functions must be defined before they are used.

You need to make sure you do not re-define a built-in-function as this may have dire consequences.

7.3.16.2 The Return Statement

You return values from a function by using the return statement:



Note that return statements are not meaningful outside a function definition.

7.3.16.3 Function Arguments

The arguments to the function are passed in the "arg" variable. Each argument is also separately defined in the "@1" to "@9" variables for direct access. (If there are more than 9, you will need to use "[word *n* [arg]]" for argument 10 and later). These variables are unique for each function invocation, even if they are nested.

You can use the "@1" to "@9" variables as automatic (temporary) variables if you have no need of their values.

7.3.16.4 Example

An example of a function definition is a "capitalize" function:

```
function capitalize =
{
        @1 = [downcase [arg]];
        @2 = ;
        loop
        {
                if [not [count [@1]]] then
                        loopstop;
                @3 = [head [@1]];
                @1 = [tail [@1]];
                @2 = [@2] [upcase [substr 1 1 [@3]]][substr 2 99 [@3]];
        }
        return [@2];
}
```

This function capitalizes the first letter of each of its arguments.

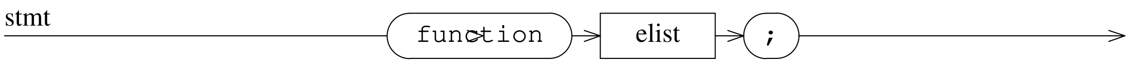
User-defined functions are invoked in the same way a built-in functions.

```
host = [os node];
Host = [capitalize [host]];
```

See the ‘‘Function Library’’ section for additional function examples which are distributed with Cook.

7.3.16.5 Function Call Statement

User defined functions may be invoked in the same way as built-in functions, but they may also be invoked in the same way as commands, providing a form of subroutine.



If the function return value is not zero, it is considered to fail, just as a command would fail. The commonest use of this is to invoke the built-in print function for debugging cookbooks.

```
function print [__FILE__] [__LINE__] hello [getenv USER];
```

These function calls may be used in recipe bodies, or in the general cookbook.

8. Built-In Functions

This chapter defines each of the built-in functions of *cook*.

A built-in 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.

Please note: this is not the same behavior as the Unix *basename*(1) utility. For this, `[basename [notdir` *args*`]]` or `[fromto %0%.c %0%` *args*`]` may be more appropriate.

8.4.1 Example

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

8.4.2 See Also

addsuffix, dirname, entryname, fromto, notdir, 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). This is usually used as a recipe gate.

8.5.1 Caveat

This will use as much of the cookbook as has been read in up to the point where this function is used. This can mean that crucial recipes have yet to be parsed and instanciated. This function works one argument at a time. This is slower than the main cookbook, which will persue all targets simultaniously.

8.5.2 See Also

cook, 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

8.7.3 Also Known As

shell

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

8.8.3 Caveat

You will probably get better performance using the `#include-cooked` directive, and a recipe to create the included file.

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

8.9.3 Also Known As

words

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

basename, entryname, notdir, pathname, suffix

8.11.3 Also Known As

dirname

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]	*‘pwd‘*
[dirname a/b]	a
[dirname a/b/c]	a/b

When the answer would be “.” (the current directory), the result is instead the absolute path of the current directory. This allows repeated [dirname] applications to climb the directory tree, no matter where you start. See *relative_dirname* for one which returns “.” instead.

8.12.2 See Also

basename, entryname, notdir, pathname, relative_dirname, suffix

8.12.3 Also Known As

dir

8.13 dos-path

This function requires one or more arguments, which will be converted from a UNIX path into a DOS path. This is of most use under Windows-NT, to convert Cook's internal pathnames into DOS pathnames. (The UNIX porting layer usually hides this from Cook.)

8.13.1 Example

Expression	Result
[dos-path a/b/c]	a\b\c
[dos-path //c/temp]	c:\temp
[dos-path //server/stuff]	\\server\stuff

8.13.2 See Also

un-dos-path

8.14 downcase

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

8.14.1 Example

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

8.14.2 See Also

upcase

8.15 entryname

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

8.15.1 Example

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

8.15.2 See Also

basename, dir, suffix

8.15.3 Also Known As

notdir

8.16 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.

8.16.1 Caveat

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

8.16.2 See Also

collect

8.17 exists

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

8.17.1 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.

This is only an example. It is better to perform this particular activity using the ‘‘unlink’’ flag. See the [find_command] function, below, for an example.

8.17.2 See Also

cando, find_command, uptodate

8.18 exists-symlink

This function requires one argument, being the name of a file to test for existence. The test will *not* follow symbolic links, so it may be used to test for the existence of symbolic links themselves. The resulting word list is "" (false) if the file does not exist, and "1" (true) if the file does exist.

8.18.1 See Also

exists, readlink

8.19 expr

This function may be used to calculate simple integer arithmetic expressions. The numbers and the operators are expected to each be a separate argument. The result is a string containing the value of the evaluated expression.

8.19.1 Operators

The following operators are understood. They have the same precedence as the equivalent C operators.

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td>( )</td><td>→</td></tr>
<tr><td>! ~ -</td><td>←</td></tr>
<tr><td>* / %</td><td>→</td></tr>
<tr><td>+ -</td><td>→</td></tr>
<tr><td><< >></td><td>→</td></tr>
<tr><td>< <= > >=</td><td>→</td></tr>
<tr><td>== !=</td><td>→</td></tr>
<tr><td>&</td><td>→</td></tr>
<tr><td>^</td><td>→</td></tr>
<tr><td>|</td><td>→</td></tr>
<tr><td>&&</td><td>→</td></tr>
<tr><td>||</td><td>→</td></tr>
<tr><td>?:</td><td>←</td></tr>
</table>

Please note that there is no short-circuit evaluation of the `?:` or `&&` or `||` operators.

You may need to quote some of the operators, to insulate them from their usual Cook interpretation (colon and equals characters in particular).

Numbers may be given in decimal, octal (with a `0` prefix), or hexadecimal (with a `0x` prefix). The result is always decimal.

8.19.2 See Also

count

8.20 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.

8.20.1 Example

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

8.20.2 See Also

filter_out, stringset

8.20.3 Also Known As

match_mask

8.21 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.

8.21.1 Example

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

8.21.2 See Also

filter, stringset

8.22 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.

8.22.1 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]
        set unlink
{
        ar qc [target] [%_obj];
        if [ranlib] then
                [ranlib] [target];
}
```

8.22.2 See Also

cando, exists, uptodate

8.23 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, if a match was found.

8.23.1 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;
```

8.23.2 See Also

filter-out, match, match_mask, patsubst, stringset, subst

8.24 firstword

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 *firstword* and *tail* functions:

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

tmp = [dirs];
loop
{
        tmp_dir = [firstword [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

8.24.3 Also Known As

head

8.25 fromto

This function requires at least two arguments. Fromto gives the user access to the pattern 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.

8.25.1 Example

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

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

8.25.2 See Also

filter, filter_out, subst

See the pattern matching chapter for more information about patterns.

8.25.3 Also Known As

patsubst

8.26 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.

8.26.1 Example

To read the value of the TERM environment variable:

```
term = [getenv tERM];
```

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;
```

8.26.2 See Also

find_command, home

8.27 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 need this character sequence. See the [head] function, below, for an example of this.

8.27.1 Example

To find the sources in the current directory:

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

8.27.2 See Also

filter, filter_out, shell

8.27.3 Also Known As

wildcard

8.28 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.

8.28.1 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.

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

8.28.3 Also Known As
firstword

8.29 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.

8.30 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.

8.30.1 Example
Here is an example of the “if” function. Please note that “if”, “then” and “else” are reserved words, so you need to quote them before they will be recognised on the function context.

```
%: %_obj
        set ["if" [defined all_shallow] "then" shallow]
{
        [cc] -o [target] [%_obj];
}
```

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

The recipe flags are evaluated at the same time as the recipe targets. None of the [target], [targets], [need], [younger] variables or pattern matches (%, %1, *etc*) are set at this time.

8.31 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.

8.31.1 Example
Frequently seen in conditional parts of recipes:

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

8.31.2 See Also
stringset

8.32 interior_files

This function requires zero arguments. The result is the list of files which are interior to the dependency graph. (Files which are constructed by a recipe.) This function is only meaningful within a recipe body.

8.32.1 See Also

interior_files function, graph_interior_file variable, graph_interior_pattern variable

8.33 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 half joined 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.

8.33.1 Example

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

8.33.2 See Also

basename, catenate, suffix

8.34 leaf_files

This function requires zero arguments. The result is the list of files which are leaves of the dependency graph. (Files which are not constructed by a recipe.) This function is only meaningful within a recipe body.

8.34.1 See Also

leaf_files function, graph_leaf_file variable, graph_leaf_pattern variable

8.35 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.

8.35.1 Example

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

8.35.2 See Also

filter-out, findstring, stringset

8.35.3 Also Known As

filter

8.36 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.

8.36.1 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
```

If the version contains a Capital-C character, then turn on debugging.

8.36.2 See Also

filter, filter-out

8.37 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 were last modified.

8.37.1 See Also

exists, sort_newest

8.38 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**.

8.38.1 Example

This is often seen in recipes:

```
%1/%0%2.o: %1/%0%2.c
        single-thread %2.o
{
        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.

This is an example only. The ‘‘mkdir’’ recipe flag creates the directory more efficiently.

8.38.2 See Also

and, or

8.39 notdir

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

8.39.1 Example

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

8.39.2 See Also

dirname, basename, suffix

8.39.3 Also Known As

entryname

8.40 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:

node
: The name of the computer **cook** is presently running on.

system
: The name of the operating system **cook** is 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".

8.40.1 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;
```

8.40.2 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.

8.40.3 See Also

collect

8.40.4 Also Known As

os

8.41 Options

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

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.

8.41.1 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** invocation.

8.41.2 Caveat

Recursive Cook is not recommended, because it segments the dependency graph and forces Cook to walk the graph in (potentially) the wrong order. This introduces a number of significant problems. A single top-level cookbook is recommended.

8.41.3 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"
```

8.42 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.

8.42.1 See Also

and, not

8.43 pathname

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

8.43.1 Example

Relative names are made absolute, and redundant slashes and dots are removed:

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

8.43.2 See Also

basename, dirname, entryname

8.44 patsubst

This function requires at least two arguments. Patsubst gives the user access to the pattern 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.

8.44.1 Example

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

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

8.44.2 See Also

filter, filter_out, subst

See the pattern matching chapter for more information about patterns.

8.44.3 Also Known As

fromto

8.45 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.

8.45.1 Example

Expression	Result
[prepost sun4/ .o a b]	sun4/a.o sun4/b.o
[prepost -I "" . bl]	-I. -Ibl

8.45.2 See Also

addprefix, addsuffix, patsubst, subst

8.46 print

The arguments are printed as an informative message. The usual output wrapping is performed. The function returns the empty list as a result.

This function si frequently use to debug cookbooks.

8.47 quote

Each argument is quoted by double quotes, with shell[8] special characters escaped as necessary.

8.47.1 See Also

collect, execute

8.48 readlink

The arguments are assumed to be symbolic links, and their values are read. It is a fatal error if the files named are not symbolic links.

8.48.1 See Also

collect, exists-symlink

8.49 relative_dirname

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

8.49.1 Example

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

See *dirname* if you want to climb the directory tree with repeated applications, *relative_dirname* will continue to return “.” once the current directory is reached.

8.49.2 See Also

basename, dirname, entryname, notdir, pathname, suffix

8.49.3 Also Known As

reldir

8. See *sh* (1) and *csh*(1) for more information.

8.50 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.

8.50.1 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];
}
```

The cookbooks distributed with Cook contain full support for the `search_list` functionality. They are a good source of examples of how to write recipes which take this into account.

8.51 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.

8.51.1 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.

8.51.2 See Also

collect_lines, execute, wildcard

8.51.3 Also Known As

collect

8.52 sort

The arguments are sorted lexicographically.

Note: Duplicates are *not* removed. Use the *stringset* function if you want to do this.

8.52.1 See Also

sort_newest, stringset

8.53 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.

8.53.1 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]]];
```

This trick does not always work as expected, and can take significant time for little result.

8.53.2 See Also

fromto, glob, sort

8.54 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.

8.54.1 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.

This can be useful in splitting an environment variable into separate words. For example:

```
path = [split ":" [getenv PATH]];
```

8.54.2 See Also

unsplit, join, catenate, strip

8.55 stringset

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

8.55.1 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];
```

8.55.2 See Also

filter, filter_out, glob, in, patsubst, subst

8.56 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.

8.56.1 Example

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

Quotes are used here for clarity, and are not present in the internal representation of strings.

8.56.2 See Also

split

8.57 subst

The *subst* function is used to perform string substitutions on its arguments. This function 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.

8.57.1 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.

8.57.2 See Also

filter, filter_out, patsubst

8.58 substr

The *substr* function is used to perform substring extracton. The first argument is the starting position in the string, starting from one. The second argument is the number of characters to extract. Thirst and subsequent arguments will be processed to extract sub-strings.

8.58.1 Example

Expression	Result
[substr 1 1 Peter]	P
[substr 3 99 Miller]	ller

8.58.2 See Also

subst, patsubst

8.59 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).

8.59.1 Example

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

Quotes are used here for clarity, and are not present in the internal representation of strings.

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

8.59.2 See Also

basename, dirname, entryname, join, patsubst

8.60 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.

8.60.1 See Also

count, head, word

8.61 un-dos-path

This function requires one or more arguments, which will be converted from a DOS path into a UNIX path. This is of most use under Windows-NT, to convert DOS pathnames into Cook's internal pathnames. (The UNIX porting layer usually hides this from Cook.)

8.61.1 Example

Expression	Result
[un-dos-path a\b\c]	a/b/c
[un-dos-path c:\temp]	//c/temp
[un-dos-path \\server\stuff]	//server/stuff

8.61.2 See Also

dos-path

8.62 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.

8.62.1 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.

8.62.2 See Also

catenate, prepost, split

8.63 upcase

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

8.63.1 Example

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

8.63.2 See Also

downcase

8.64 uptodate

This function requires one or more arguments, filenames to be tested to see if they are up-to-date. They are *not* brought up to date if they are not already. The result are true ("1") if the files are up-to-date, or false ("") if they are not.

8.64.1 Caveat

This will use as much of the cookbook as has been read in up to the point where this function is used. This can mean that crucial recipes have yet to be parsed and instanciated. This function works one argument at a time. This is slower than the main cookbook, which will persue all targets simultaniously.

8.64.2 See Also

cando, cook

8.65 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.

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

8.65.1 Example

To find the sources in the current directory:

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

8.65.2 See Also

filter, filter_out, patsubst

8.65.3 Also Known As

glob

8.66 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.

8.66.1 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]];
```

8.66.2 See Also

count, head

8.66.3 Wordlist

This function may be used to extract a list of words from a larger list. The first argument is the starting position, and the second argument is the ending poistion, inclusive. The third and subsequent arguments are the list to be extracted from. Positions are numbered starting from 1. If the start is bigger than the end, they will be quietly swapped. If the start is bigger than the list, the result will be empty.

8.66.3.1 Example

Expression	Result
[wordlist 2 3 foo bar baz]	bar baz
[wordlist 1 1 foo bar baz]	foo
[wordlist 7 3 foo bar baz]	baz

There are a number of functions which are similar

Expression	Similar to
[wordlist 1 1 *list*]	[head *list*]
[wordlist 2 9999 *list*]	[tail *list*]
[wordlist *N N list*]	[word *N list*]

8.66.3.2 See Also

firstword head, tail, word, words

8.67 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.

8.67.1 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]].;
```

8.67.2 See Also

head, tail, word

8.67.3 Also Known As

count

9. Predefined Variables

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

9.1 arg

This is the arguments list for user-defined functions. Individual arguments are split out into "@1" to "@9". These can also be used at automatic variables. Caution: *arg* and the automatic variables are *shared* for parallel execution, causing weird interactions if you execute a command within the function.

9.2 command-line-goals

The value of this variable is the goals specified on the command line, if any. If none were specified, and the default goal is in effect, the value will be empty.

9.3 __FILE__

The value of this variable is the logical name of the file which contains it. In the case of `#include-cooked` files, the physical name may be obtained using the [resolve] function. The logical name may be set using the `#line` directive.

9.4 graph_leaf_file

File names which are listed in this variable could be leaf files of the dependency graph. (See also the *leaf_files* function, for Cook's idea of the leaf files.)

9.5 graph_exterior_file

File names which are listed in this variable cannot be present in any way in the dependency graph.

9.6 graph_interior_file

File names which are listed in this variable could be interior files of the dependency graph. (See also the *interior_files* function, for Cook's idea of the interior files.)

9.7 graph_leaf_pattern

File names which match the patterns in this variable could be leaf files of the dependency graph. (See also the *leaf_files* function, for Cook's idea of the leaf files.)

9.8 graph_exterior_pattern

File names which match the patterns in this variable cannot be present in any way in the dependency graph.

9.9 graph_interior_pattern

File names which match the patterns in this variable could be interior files of the dependency graph. (See also the *interior_files* function, for Cook's idea of the interior files.)

9.10 __LINE__

The value of this variable is the line number within of the file which contains it. The line number may be set using the `#line` directive.

9.11 need

The ingredients of the recipe currently being cooked.

9.12 parallel_hosts

This variable may be set to indicate a list of hosts to use to distribute the execution of recipe bodies.

9.13 parallel_jobs

This variable may be set to the number of parallel execution threads to perform simultaneously. Defaults to 1 if not set.

9.14 parallel_rsh

This variable may be set to the command used to execute commands on remote machines. Assumes to take argument in the same form as the BSD *rsh*(1) command. Defaults to ‘‘*rsh*’’ if not set.

9.15 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* built-in function to determine what file name cook actually found. The targets of recipes are always cooked into the current directory.

The cookbooks distributed with Cook contain full support for the `search_list` functionality. They are a good source of examples of how to write recipes which take this into account.

9.16 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.

9.17 target

The target of the recipe currently being cooked, or the first target if there is more than one.

9.18 targets

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

9.19 thread-id

This variable has a unique value for each execution thread, for the lifetime of that thread. This value may be used to construct thread-unique variable names, thread-unique temporary file names, or anything else that needs to be unique to each execution thread. The thread IDs are re-used, and so several threads in sequence may have the same thread ID; it is only guaranteed that no other simultaneous thread will have the same thread ID. By re-using thread IDs, generated variable names are also re-used, avoiding memory bloat.

9.20 younger

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

9.21 version

The version of **cook** currently executing.

10. Functions Library

There is a file of functions available to you by using a

```
#include "functions"
```

line in your cookbook. The file defines a number of useful functions.

The functions in the file also serve as examples of how you can write your own functions.

10.1 Capitalize

The *capitalize* function maps all of its arguments into lower case, and then the first letter of each argument is mapped to upper case. Zero, one or more arguments may be given.

10.2 Defined-or-null

The *defined-or-null* function may be used to determine if a variable has been set (on the command line, for example) and return its value if so, otherwise return the empty list.

This function should only be given one argument - the name of the variable to look for. Additional arguments will be ignored. Too few arguments will produce a complaint about the "" variable being undefined.

10.3 Defined-or-default

The *defined-or-default* function may be used to determine if a variable has been set (on the command line, for example) and return its value if so, otherwise return the given default value.

The first argument is the name of the variable to look for.

The second and later arguments (if present) are the default value to be used if the named variable is not defined. Optional.

10.4 Repeat

The *repeat* function is used to repeatedly call another function, once for each of the specified arguments. The can be useful when dealing with functions which do not automaticly accept argument lists in the form you require.

There are many instances where the repeat function call be used to elegantly avoid used to the ‘‘loop { loopstop }’’ construct.

The first argument is the name of the function you want called. This function must accept a single argument.

The second and subsequent arguments are argument values to be passed to the named function, one at a time.

The results of the invocations of the function are accumulated in the order in which they were calculated. The accumulated results are returned.

11. 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.

11.1 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.

11.2 Scan the Command Line

The command line is scanned as defined in chapter 3.

11.3 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.

11.4 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.

11.5 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.

11.6 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.

11.7 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.

11.8 Cooking a Target

A derivation graph is formed using all of the targets given. Once the derivation graph is formed, it will be walked, looking for files which are out of date.

To build the derivation graph for a target, each the following steps is performed in the order given:

1. **Cook** exploits knowledge of the derivation graph that the user may provide to it:
   - If the *graph_exterior_file* variable is set, and the file name is listed in it, the file is not a leaf, and the derivation will backtrack and try another alternative.
   - If the *graph_exterior_pattern* variable is set, and the file name matches one of the patterns listed in it, the file is not a leaf, and the derivation will backtrack and try another alternative.
   - If the *graph_leaf_file* variable is set, and the file name is listed in it, the file is a leaf file of the derivation. There is no need to attempt to apply any recipes. It will be an error if the file does not exist.
   - If the *graph_leaf_pattern* variable is set, and the file name matches one of the patterns listed in it, the file is a leaf file of the derivation. There is no need to attempt to apply any recipes. It will be an error if the file does not exist.

   These optimizations require an accurate source file manifest, but can result is substantial performance improvements.

2. **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 also have their derivation graph constructed. When walking the graph, 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.[9]

3. **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 also have their derivation graph constructed. When walking the graph, 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.

4. 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, in two passes. Implicit recipes which not not have pattern elements in the basename of the targets are scanned before implicit recipes which do have patterns in the basename. Usually this has no significant effect, however in heavily heterogeneous builds this method is often used in constructing the dependency files, so that all architectures may use the one implicit dependency recipe, rather than stating every architecture explicitly. Within each pass, the order of scan is the order of definition.

   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 also have their derivation graph constructed. It is an error if a forced ingredient cannot be constructed.

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

5. If the target is not the subject of any ingredients or explicit recipe, and no implicit recipes can be applied, then several derivations are attempted, in the order specified:

---

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

- If the *graph_interior_file* variable is set, and the file name is listed in it, the file is a not leaf file of the derivation. Cook will backtrack and try another alternative.
- If the *graph_interior_pattern* variable is set, and the file name matches one of the patterns listed in it, the file is a not leaf file of the derivation. Cook will backtrack and try another alternative.
- If the *graph_leaf_file* variable is set, and the file name is listed in it, the file is a leaf file of the derivation. It will be an error if the file does not exist.
- If the *graph_leaf_pattern* variable is set, and the file name matches one of the patterns listed in it, the file is a leaf file of the derivation. It will be an error if the file does not exist.
- If either of the *graph_leaf_file* or *graph_leaf_pattern* variables are set, then the file is not a leaf, and the derivation will backtrack and try another alternative.
- If the file exists, then it is up to date, and the file is a leaf file of the derivation.
- If the file does not exist then **Cook** doesn't know how, and the derivation will backtrack and try another alternative.

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.

11.9 The Dependency Graph

The above section describes how Cook derives the dependency graph. Once the dependency graph has been derived, it is then walked. The next section describes a little about how Cook walks the dependency graph.

Cook is a simple kind of expert system. You give it a set of of recipes for how to construct things, and a target to be constructed. The recipes can be decomposed into pair-wise ordered dependencies between files.

Cook determines how to build the target by constructing a *directed acyclic graph*. The vertexes of this graph are the files in the system, the edges in this graph are the inter-file dependencies. The edges of the graph are directed because the pair-wise dependencies are ordered resulting in a *acyclic* graph – things which look like loops are resolved by the direction of the edges.

There are several things that can be done with the graph once it has been derived:
• It can be walked to verify and regenerate the referential integrity of the files (the usual case), or
• it can walked to print the pair-wise dependencies (the `-pairs` option), or
• it can be walked to generate a shell script (the `-script` option) which does something very similar to the first option.

11.10 File Status

Cook determines the time a file was last modified by asking the operating system. Because this operation tends 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];
}
```

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.

12. 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.

13. 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** behavior 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.

13.1 Examples

There are two main places where patterns are used: with the *match_mask* and *fromto* functions, and in recipes.

You can perform file name filtering and rewriting as follows:

```
source_files = [collect cat MANIFEST];
object_files =
        [fromto %0%.c %0%.o [match_mask %0%.c [manifest]]]
        [fromto %0%.y %0%.gen.o [match_mask %0%.y [manifest]]]
        ;
```

The recipes to go with the above files may be

```
%0%.o: %0%.c
        single-thread ["if" %0 "then" %.o]
{
        /* note: no slash before dot */
        cc -c -I%0. %0%.c;
        if %0 then
```

```
                    mv %.o %0%.o;
        }
```

This recipe can compile files in a large project, where source files appear in a number of sub-directories. The "-I%0." ensures that there are locally include-able files in the sub-directories. If the "%0" had been entirely omitted from the recipe, it will only compile files in the current directory.

A common *yacc* recipe, used when there is more than one yacc grammar in a project, looks like this:

```
        %0%.gen.c %0%.gen.h: %0%.y
                single-thread yy.tab.c yy.tab.h
        {
                yacc -d %0%.y
                yy = [collect echo %0% | sed "'s/[^A-Za-z0-9]/_/'"];
                sed "'s/[yY][yY]/"[yy]"_/g'" yy.tab.c > %0%.gen.c;
                sed "'s/[yY][yY]/"[yy]"_/g'" yy.tab.h > %0%.gen.h;
                rm yy.tab.c yy.tab.h;
        }
```

To be more selective about the "%0" portion, use more pattern elements before or after it.

14. 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 "built-in" 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/share/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.

14.1 as

This cookbook defines how to use the assembler.

14.1.1 recipes

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

14.1.2 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.

14.2 c

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

14.2.1 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.

14.2.2 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).

14.2.3 See Also

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

14.3 f77

This cookbook describes how to work with Fortran files.

14.3.1 recipes

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

14.3.2 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.

14.3.3 See Also

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

14.4 g77

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

14.5 gcc

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

14.6 home

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

14.6.1 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.

14.7 lex

This cookbook describes how to work with lex files.

14.7.1 recipes

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

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

14.8 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.

14.8.1 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.

14.8.2 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.

14.9 print

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

14.9.1 recipes

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

14.9.2 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.

14.10 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.

14.10.1 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.

14.10.2 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.

14.10.3 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.

14.11 rcs

This cookbook is used to extract files from RCS.

14.11.1 recipes

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

14.11.2 variables

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

14.12 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.

14.12.1 Recipes

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

14.13 sccs

This cookbook is used to extract files from SCCS.

14.13.1 recipes

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

%: s.% Extract files from SCCS.

14.13.2 variables

get The SCCS get command.

get_flags Flags for the get command, default to empty.

14.14 text

This cookbook is used to process text documents.

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

14.14.1 recipes

%.ps: %.t PostScript for generic *roff source.

%: %.t Straight text from *roff source.

14.14.2 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.

14.15 usr.local

This cookbook defined where certain directories are, and some common uses of those directories, relative to /usr/local.

14.15.1 variables

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

14.16 usr

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

14.16.1 variables

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

14.17 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*.

14.17.1 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].

14.17.2 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).

14.18 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.

15. 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 behavior 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 ingredients 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