Cook

Tutorial

Aryeh M. Friedman
aryeh@m-net.arbornet.org

.

This document describes Cook version 2.34
and was prepared 12 July 2018.

1. Building Programs

If you write simple programs (a few hundred lines of code at most) compiling the program is often no more then something like this:

```
gcc foo.c -o foo
```

If you have a few files in your program you just do:

```
gcc foo.c ack.c -o foo
```

But what happens if some file that is being compiled is the output of an other program (like using yacc/lex to construct a command line parser)? Obviously `foo.c` does not exist before `foo.y` is processed by yacc. Thus you have to do:

```
yacc foo.y
cc foo.c ack.c -o foo
```

What happens if say you modify `ack.c` but do not modify `foo.y`? You can skip the yacc step. For a small program like the one above it is possible to remember what order you need to do stuff in and what needs to be done depending on what file you modify.

Let's add one more complication let's say you have a library that also needs to be "built" before the executable(s) is built. You need to not only remember what steps are needed to construct the library object file but you also need to remember that it needs to be done you make your executables. Now add to this you also need to keep track of different versions as well figuring out how to build different versions for different platforms and/or customers (say you support Windows, Unix and have a Client, Server and trial, desktop and enterprise versions of each and you need to produce any and all combination of things... that's 24 different versions of the same set of executables). It now becomes almost impossible to to remember how each on is built. On top all this if you build it differently every time you need to recompile the program there is no guarantee you will not introduce bugs due to only the order stuff was built in.

And the above example is for a "small" applications (maybe 10 to 20 files) what happens if you have a medium or large project (100s or 1000s of files) and 10+ or 100+ executables with each one having 10+ different configurations. It is clearly the number of possible ways to make this approaches infinity very rapidly (in algorithm designer terms *O(n!)*). There has to be a easier way! Traditionally people have used a tool called *make* to handle this complexity, but make has some major flaws such that it is very hard if not impossible to make know how to build the entire project without some super nasty and flawed "hacks". In the last few years a program called Cook has gained a small but growing popularity as a extremely "intelligent" replacement for make.

2. Dependency Graphs

Clearly, for any build process the build management utility (e.g. *cook* or *make*) needs to know that for event Y to occur event X has to happen first. This knowledge is called a dependency. In simple programs it is possible to just tell the build manager that X depends on Y. This has a few problems:

- You can not define generic dependencies for example you can not say that all `.o` files depend on `.c` files of the same name.
- Often there are intermediate files created during the build process for example `foo.y` → `foo.c` → `foo.o` → `foo`. This means that each intermediate file needs to be made before the final program is built.
- In almost all projects there is no single way of producing any given file type. For example `ack.c` does not need to be created from the `ack.y` file but `foo.c` does need to be created from the `foo.y` file.
- Many times many things depend on event X but X can not happen until Y happens. For example if you need to compile all the `.c` files into `.o` files before you can combine them into a library then once the library is made then and *only* then can you build all the executables that need that library.
- Depending on what variant of an executable you are building you may have a total different set of dependencies for that executable. For example the Microsoft version of your program may be totally different than the Unix one.

Thus one of the most fundamental things any build manager needs to know is create a "graph" of all the dependencies (i.e. what depends on what and what order stuff needs to be built in).

Obviously if you modify only a file or two and rebuild the project you only need to recreate those files that depend on the ones you changed. For

example if I modify `foo.y` but not `ack.c` then `ack.c` does not need to be recompiled but `foo.c` after it is recreated does. All build managers know how to do this.

3. Cook *vs.* Make

Many times the contents of entire directories depend on the building of everything in other directories. Make has traditionally done this with "recursive make". There is a basic flaw with this method though: if you "blindly" make each directory in some preset order you are doing stuff that is either unneeded and/or may cause problems in the build process down the road. For a more complete explanation, see Recursive Make Considered Harmful[1].

Cook takes the opposite approach. It makes a *complete* dependency graph of your entire project then does the entire "cook" at the root directory of your project.

4. Teaching Cook about Dependencies

Each *node* in a dependency graph has two basic attributes. The first is what other nodes (if any) it depends on, and the second is a list of actions needed to be performed to bring the node *up to date* (bring it to a state in which any nodes that depend on it can use it's products safely).

One issue we have right off the bat is which node do we start at. While by convention this node is usually called '`all`' it does not have to be, as we will see later it might not even have a hard coded name at all. Once we know where to start we need someway of linking nodes together in the dependency graph.

In cook all this functionality is handled by *recipes*. In basic terms a recipe is:

- The name of the node so other nodes know how to link to it (this name can be dynamic). This name is usually the name of a file, but not always.
- A list of other recipes that need to be "cooked" before this recipe can be processed. The best way to think of this is to use the metaphor that cook is based on. That being in order to make meal at a fine restaurant you need to make each dish. For each dish you need to combine the ingredients in the right order at the right time. You keep dividing up the task until you get to a task that does not depend on something else like seeing if you have enough eggs to make the bread. A dependency graph for building a software project is almost identical except the *ingredients* are source code not food.
- A list of actions to perform once all the ingredient are ready. Again using the cooking example, in order to make a French cream sauce you gather all the ingredients (in cook's cases the output from other recipes) and then and *only* then put the butter in the pan with the the flour and brown it, then slowly mix the milk in, and finally add in the cheese.

So in summary we have the following parts of a recipe:

- The name of the recipe's node in the graph
- A list of ingredients needed to cook the recipe
- A list of steps performed to cook the recipe

From the top level view in order to make a hypothetical project we do the following recipes:

- We repeatedly process dependency graph nodes until we get a *leaf* node (one that does not have any ingredients). Namely we go from the general to the specific not the other way.
- Visit the `all` recipe which has `program1` and `program2` as its ingredients
- Visit the `program1` node which has `program1.o` and `libutils.a` as its ingredients
- Visit `program1.o` which has `program1.c` and `program1.h` as its ingredients
- Visit `program1.c` to discover that it is a leaf node, because the file already exists we need to do nothing to create it.

1. Miller, P.A. (1998). *Recursive Make Considered Harmful*, AUUGN Journal of AUUG Inc., 19(1), pp. 14-25. http://aegis.sourceforge.net/auug97.pdf

- Visit `program1.h` to discover that it is a leaf node, because the file already exists we need to do nothing to create it.
- Now that we have all the ingredients for `program1.o` we can cook it with a command something like
```
gcc -c program1.c \
    -o program1.o
```
- Visit the `libutils.a` node which has `lib1.o` as its only ingredient.
- Visit `lib1.c` to discover that it is a leaf node, because the file already exists we need to do nothing to create it.
- Now that we have all the ingredients for `lib1.o` we can cook it with a command something like
```
gcc -c lib1.c -o lib1.o
```
- Now that we have all the ingredients for `libutils.a` we can cook it with a command something like
```
rm libutils.a
ar cq libutils.a lib1.o
```
- Now that we have all the ingredients for `program1` we can cook it with a command something like
```
gcc program1.o libutils.a \
    -o program1
```
- Visit the `program2` node which has `program2.o` and `libutils.a` as its ingredients
- Visit `program2.o` which has `program2.c` and `program1.h` as its ingredients
- Visit `program2.c` to discover that it is a leaf node, because the file already exists we need to do nothing to create it.
- Visit `program2.h` to discover that it is a leaf node, because the file already exists we need to do nothing to create it.
- Now that we have all the ingredients for `program2.o` we can cook it with a command something like
```
gcc -c program2.c \
    -o program2.o
```
- There is no need to visit the `libutils.a` node, or any of its ingredient nodes, because Cook remembers that they have been brought up to date already.
- Now that we have all the ingredients for `program2` we can cook it with a command something like
```
gcc program2.o libutils.a \
    -o program2
```
- Return to the `all` recipe and find that we have cooked all the ingredients and there are no other actions for it. We are done and our entire project is built!

Now what happens if I say modify `program2.c` all we have to do is walk to the entire graph from `all` and we find that `program2.c` has changed, and do any node which depends on `program2.c` needs to be brought up to date, and any nodes which depend on *them*, and so on. In this example, this would be `program2.c` → `program2.o` → `program2` → `all`.

5. Recipe Syntax

All statements, recipes and otherwise, are in the form of

statement`;`

Note the terminating simicolon (`;`). An example statement is

```
echo aryeh;
```

The only time the the simicolon (`;`) is not needed is in compound statements surrounded by `{` curly braces `}`. In general the convention is to follow the same general form that C uses, as it is with most modern programming languages. This means that for the main part almost everything you have learned about writing legal statements works just fine in cook. The only exception are the `[` square brackets `]` used instead of `(` parentheses `)` in most cases.

The general form of a recipe, there are some advanced options that do not fit well into this format, is:

name`:` *ingredients*
`{`
    actions
`}`

Note: the actions and ingredients are optional.

Here is a recipe from the above example:

```
program1.o: program1.c program1.h
{
    gcc -c program1.c
        -o program1.o;
}
```

The only thing to remember here is that

`program1.c` either has to exist or Cook needs to know how to cook it. If you reference an ingredient that Cook does not know how to cook you get the following error:

```
cook: program1: don't know how
cook: cookfile: 1: "program1"
    not derived due to errors
    deriving "program1.o"
```

All this says is there is no algorithmic way to build `example1.o` that Cook can find.

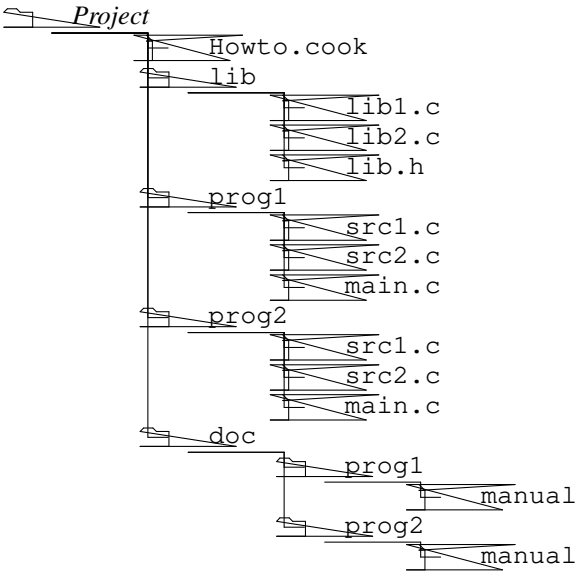
A *cookbook* file can contain zero or more recipes. If there is no *default* recipe (the first recipe whose name is hard coded) you get the following error:

```
cook: no default target
```

Most of the time this just means that Cook cannot figure out what the "concrete" name of a recipe is based solely by reading the cookbook. By default cook looks for the cookbook in "Howto.cook" [note 1].

6. A Sample Project

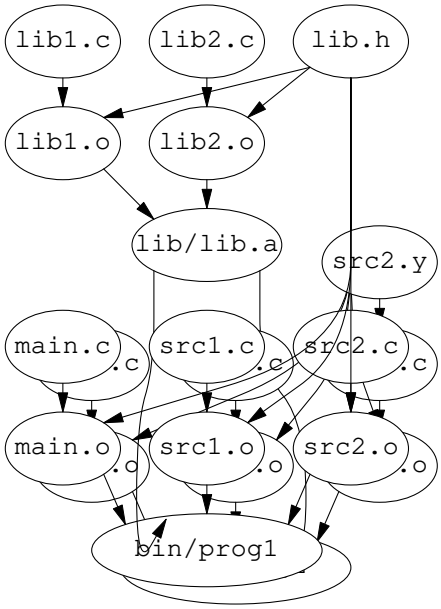
For the remainder of the tutorial we will be using the following sample project source tree:



The final output of the build process will be completely working and installed executables of prog1 and prog2 installed in /usr/local/bin and the documentation being placed in /usr/local/share/doc/myproj.

7. Our First Cookbook

The first step in making a cookbook is to sketch out the decencies in our sample project the graph would be:



Now we know enough to write the first version of our cookbook. The cookbook which follows doesn't actually cook anything, because it contains ingredients and no actions. We will add the actions needed in a later section. Here it is:

```
/* top level target */
all: /usr/local/bin/prog1
    /usr/local/bin/prog2
    /usr/local/share/doc/prog1/manual
    /usr/local/share/doc/prog2/manual
    ;

/* where to install stuff */
/usr/local/bin/prog1:
    bin/prog1 ;
/usr/local/bin/prog2:
    bin/prog2 ;
/usr/local/share/doc/prog1/manual:
    doc/prog1/manual ;
/usr/local/share/doc/prog2/manual:
    doc/prog2/manual ;

/* how to link each program */
bin/prog1:
    prog1/main.o
    prog1/src1.o
    prog1/src2.o
    lib/liblib.a ;
bin/prog2:
    prog2/main.o
    prog2/src1.o
    prog2/src2.o
    lib/liblib.a ;
```

```
/* how to use yacc */
prog2/src2.c: prog2/src2.y ;

/* how to compile sources */
prog1/main.o: prog1/main.c ;
prog1/src1.o: prog1/src1.c ;
prog1/src2.o: prog1/src2.c ;
prog2/main.o: prog2/main.c ;
prog2/src1.o: prog2/src1.c ;
prog2/src2.o: prog2/src2.c ;
lib/src1.o: lib/src1.c ;
lib/src2.o: lib/src2.c ;

/* include file dependencies */
prog1/main.o: lib/lib.h ;
prog1/src1.o: lib/lib.h ;
prog1/src2.o: lib/lib.h ;
prog2/main.o: lib/lib.h ;
prog2/src1.o: lib/lib.h ;
prog2/src2.o: lib/lib.h ;
lib/src1.o: lib/lib.h ;
lib/src2.o: lib/lib.h ;

/* how to build the library */
lib/liblib.a:
    lib/src1.o
    lib/src2.o ;
```

In order to cook this cookbook just type the

```
cook
```

command in the same directory as the cookbook is in.

8. Soft coding Recipes

One of the most glaring problems with this first version of our cookbook is it hard codes everything. This has two problems:

- We have to be super verbose in how we describe stuff since we have to specify every single recipe by hand.
- If we add new files (maybe we add a third executable to the project) we have to rewrite the cookbook for *every* file we add.

Fortunately, Cook has a way of automating the build with implicit recipes. It has a way of saying how to move from any arbitrary `.c` file to its `.o` file.

Cook provides several methods for being able to soft code these relationships. This section discusses file "patterns" that can be used to do pattern matching on what recipe to cook for a given file.

Note on pattern matching notation used in this section:

[string] means the matched pattern.

The first thing to keep in mind about cook's pattern matching is once a pattern is matched it will have the same value for the remainder of the recipe. So for example if we matched prog/[src1].c then any other reference to that pattern will also return src1. For example:

```
prog/[src1].o: prog/[src1].o ;
```

if we matched *src1* on the first match (`prog1/[src1].o`) then we will always match *src1* in this recipe (`prog1/[src1].c`).

Cook uses the percent (`%`) character to denote matches of the relative file name (no path). Thus the above recipe would be written:

```
prog/%.o: prog/%.c ;
```

Cook also lets you match the full path of a file, or parts of the path to a file. This done with `%`*n* where *n* is a part number. For example

```
/usr/local/bin/prog1
```

could match the pattern

```
/%1/%2/%3/%
```

with the parts be assigned

%1	usr
%2	local
%3	bin
%	prog1

Note that the final component of the path has no *n* (there is no `%4` for `prog1`). If we want to reference the whole path, Cook uses %0 as a special pattern to do this.

```
/usr/local/bin/prog1
```

could match the pattern

```
%0%
```

with the parts be assigned

%0	/usr/local/bin/
%	prog1

Patterns are connected together thus `%0%.c` will match any `.c` file in any pattern.

Let's rewrite the cookbook for our sample project using pattern matching. The relevant portions of our cookbook are replaced by

```
/* how to use yacc */
%0%.c: %0%.y;

/* include file dependencies */
%0%.c: lib/lib.h;

/* how to compile sources */
%0%.o: %0%.c;
```

When constructing the dependency graph Cook will match the the first recipe it sees that meets all the requirements to meet a given pattern. I.e. if we have a pattern for `prog1/%.c` and one for `%0%.o` and it needs to find the right recipe for `prog1/src.o` it will match the one that appears first in the cookbook. So if the first one is `%0%.c` then it does that recipe even if we meant for it to match `prog1/%.c`.

9. Arbitrary Statements and Variables

Any statement that is not a recipe, and not a statment inseide a recipe, is executed as soon as it is seen. For example I can have a `Howto.cook` file that only contains the following line:

```
echo Aryeh;
```

and when ever I ise the `cook` command it will print my name.

This in and upon it self is quite pointless but it does give a clue about how we can set some cookbook-wide values. Now the question is how do we symbolically represent those variables.

Cook has only one type of variable and that is a list of string literals, i.e. `"ack"`, `"foo"`, `"bar"`, *etc*. There are no restrictions on how you name variables, except they can not be reserved words, this is pretty close to the restrictions most programming languages have. There is one major difference though: variables can start with numbers and contain punctuation characters. Additionally you can vary variable names, i.e. the name of the actual variable can use a variable expression (this is hard to explain but easy to show which we will do in a few paragraphs).

All variables, when queried for their value, are `[` in square brackets `]` for example if the "name" variable contains "Aryeh" then:

```
echo [name];
```

Has exactly the same result as the previous example. Variables are simply set by using `var = value;` For example:

```
name = Aryeh;
echo [name];
```

Let's say I need to have two variables called 'prog1_obj' and 'prog2_obj' that contain a list of all the .o ingredients in the prog1 and prog2 directories respectively. Obviously the same operation that produces the value of prog1_obj is identical to the one that produces prog2_obj except it operates on a different directories. So why then do we need two different operations to do the same thing, this violates the principle of any given operation it should only occur in one place. In reality all we need to do is have some way of changing the just the variable name and not the values it produces. In cook we do this with something like [[dir_name]_obj]. The actual procedure for getting the list of files will be covered in the "control structures" section.

Let's revise some sections of our sample project's cookbook to take advantage of variables:

```
/* where to install stuff */
prefix = /usr/local;
idoc_dir = [prefix]/share/doc;
ibin_dir = [prefix]/bin;

/* top level target */
all:
    [ibin_dir]/prog1
    [ibin_dir]/prog2
    [idoc_dir]/prog1/manual
    [idoc_dir]/prog2/manual;

/* where to install each program */
[ibin_dir]/%: bin/% ;
[idoc_dir]/%/manual: doc/%/manual ;
```

As you can see we didn't make the cookbook any simpler because we do not know how to intelligently set stuff based on what the actual file structure of our project. The only thing we gain here is the ability to change where we install stuff very quickly be just changing install_dir. We also gain a little flexibility in how we name the directories in our source tree.

10. Using Built-in Functions

If all you could do was set variables to static values and do pattern matching cook would not be very useful, i.e. every time we add a new source file to our project we need to rewrite the cookbook. We need some way to extract useful data from variables and leave out what we do not want. For example if we want to know what all the .c files in the prog1 directory are we just ask for all files that match prog1/%.c. We could use the match_mask built-in function to extract the needed sublist of files. Built-in functions can do many other manipulations of our source tree contents and how to process them. In general I will introduce a given built-in function as we encounter them.

As far as cook is concerned, for the most part, functions and variables are treated identically. This means anywhere where you would use a variable you can use a function. In general a function is called like this:

```
[func arg1 arg2 ... argN]
```

For example:

```
name = [foobar aryeh];
```

11. Source Tree Scanning

The first thing we need to do to automate the process of handling new files is to collect the list of source files. In order to do this we need to ask the operating system to give us a list of all files in a directory and all it's subdirectories. In Unix the best way to do this is with the find(1) command. Thus to get a complete list of all files in say the current directory we do:

```
find . -print
```

or any variation thereof.

Great, now how do we get the output of find into a variable so cook can use it. Well, the `collect` function does this. We then just assign the results of `collect` to a list of files, build experts like to call this the manifest. So here is how we get the manifest:

```
manifest = [stripdot
    [collect find . -print]];
```

That is all nice and well but how do we get the list of source files in `prog1` only, for example. There is a function called `match_mask` that does this. The `match_mask` function returns all "words" that match some pattern in our list. For example to get a list of all `.c` files in our project we do:

```
src = [match_mask %0%.c
    [manifest]];
```

It is fine to know what files are already in our source tree but what we really want to do is find the list of files that need to be cooked. We use the `fromto` function to do this. The `fromto` function takes all the words in our list and transforms all the names which match to some other name. For example to get a list of all the `.o` files we need to cook we do:

```
obj = [fromto %0%.c %0%.o
        [src]];
```

It is rare that we need to know about the existence of `.c` files since in most cases, unless they are derived from cooking something else, they either exist or they do not exist. In the case of them not existing the `.o` target for that source should fail. For this reason we really do not need a `src` variable at all. Remember I mentioned that a function call can be used anywhere a variable can. This means that we can do the `match_mask` call in the same line that we do the fromto. Thus the new statement is:

```
obj = [fromto %0%.c %0%.o
        [match_mask %0%.c
         [manifest]]];
```

Time to update some sections of our sample project's cookbook one more time:

```
/* info about our files */
manifest =
    [collect find . -print];
obj = [fromto %0%.c %0%.o
        [match_mask %0%.c
         [manifest]]];

/* how to build each program */
prog1_obj = [match_mask
    prog1/%.o [obj]];
prog2_obj = [match_mask
    prog2/%.o [obj]];
bin/%: [%_obj] lib/lib.a;

/* how to build the library */
lib_obj = [match_mask lib/%.o
    [obj]];
lib/lib.a: [lib_obj];
```

The important thing to observe here is that it is now possible to add a source file to one of the probram or library directories and Cook will automagically notice, without any need to modify the cookbook. It doesn't matter whether there are 3 files or 300 in these directories, the cookbook is the same.

12. Flow Control

If there was no conditional logic in programming would be rather pointless, who wants to write I program that can only do something once, the same is true in cook. Even though the stuff we need to conditional in a build is often very trivial as far as conditional logic goes, namely there are if statements and the equivalent of while loops and thats all.

If statements are pretty straight forward. If you are used to C, C++, *etc*, the only surprise is the need for the `then` keyword. Here is a example if statement:

```
if [not [count [file]]] then
    echo no file provided;
```

The count function returns the number of words

in the "file" list and the not function is true if the argument is 0. Other then that the if statement works much the way you would expect it to.

Cook has only one type of loop that being the `loop` statement and it takes no conditions. A loop is terminated by the `loopstop` statement (like a C *break* statement). Other then that loops pretty much work the way you expect them to. Here is an example loop:

```
/* set the loop "counter" */
list = [kirk spock 7of9
    janeway worf];

/* do the loop */
loop word = [list]
{
    /* print the word */
    echo [word];
}
```

13. Special Variables

Like most scripting languages Cook has a set of predefined variables. While most of them are used internally by Cook and not by the user, one of them deserves special mention and that is `target`. The `target` variable has no meaning out side of recipes but inside recipes it refers to the current recipe's target's "real" name, i.e. the one that Cook "thinks" it is currently building, not the soft coded name we provided in the cookbook. For example in our sample project's cook book if we where compiling `lib/src1.c` into `lib/src.o` the `%0%.o: %0%.c;` recipe would, as far as Cook is concerned, actually be `lib/src1.o: lib/src1.c;` The recipe name, and thus the `[target]`, of this is set to the `lib/src.o` string.

There are other special variables described in the Cook User Guide. You may want to look them up and use them when you start writing more advanced cookbooks.

14. Super Soft coding

Now we know enough so we can make Cook handle building an arbitrary number of programs in our sample project. Note the following example assumes that all program directories contain a `main.c` file and no other directory contains it. The best way to understand what is needed it to look at the sample cookbook for this line by line. So here are the rewritten sections of our sample cookbook:

```
/* names of the programs */
progs = [fromto %/main.c %
         [match_mask %/main.c
          [manifest]]];

/* top level target */
all:
    [addprefix [ibin_dir]/
        [progs]]
    [prepost [idoc_dir]/ /manual
        [progs]];

/* how to build each program */
loop prog = [progs]
{
    [prog]_obj = [match_mask
        [prog]/%.o [obj]];
}
bin/%: [%_obj] lib/lib.a;
```

The basic idea is that we use a loop to create the list of `.o` files for all programs and then we use variable variable names to reference the right one in the recipe.

15. Scanning for Hidden Decencies

In most real programs most `.c` files have a different set of `#include` lines in them. For example `prog1/src1.c` might include `prog1/hdr1.h` but `prog1/src2.c` does not. So far we have conveniently avoided this fact on the assumption that once made `.h` files don't change. Any experience with a non-trivial project show this is not true. So how do we automatically scan for these dependencies? It would not only defeat the purpose of soft coding but would be a pain in the butt to have to encode this in the cookbook.

One way of doing it is to scan each `.c` for `#include` lines and say any that are found represent "hidden" dependencies. It would be fairly trivial to create a shell script or small C program that does this. Cook though has been nice enough to include program that does this for us in most cases that are not insanely non-trivial. There are several methods of using `c_incl` we will only cover the "trivial" method here, if you need higher performance refer to the Cook User Guide, it has a whole chapter on include dependencies.

The `c_incl` program essentially just prints a list of `#include` files it finds in its argument. To do

this just do:

```
c_incl prog.c
```

Now all we have to do is have Cook `collect` this output on the ingredients list of our recipe and boom we have a list of our hidden dependencies. Here is the rewritten portion of our sample cookbook for that:

```
/* how to build each program and
   include file dependencies */
%0%.o: %0%.c
    [collect c_incl -api %0%.c];
```

The `c_incl -api` option means if the file doesn't exist, just ignore it.

16. Recipe Actions

Now that we have all the decencies soft coded all we have to do actually build our project is to tell each recipe how to actually cook the target from the ingredients. This is done by adding actions to a recipe. The actions are nothing more "simple" statements that are bound to a recipe. This is done by leaving off the trailing semicolon (`;`) on the recipe and putting the actions inside `{` curly braces `}`. This is best shown by example. So here is our final cookbook for our sample project:

```
/* where to install stuff */
prefix = /usr/local;
idoc_dir = [prefix]/share/doc;
ibin_dir = [prefix]/bin;

/* info about our files */
manifest =
    [collect find . -print];
obj = [fromto %0%.c %0%.o
       [match_mask %0%.c
        [manifest]]];

/* names of the programs */
progs = [fromto %/main.c %
         [match_mask %/main.c
          [manifest]]];

/* top level target */
all:
    [addprefix [ibin_dir]/
        [progs]]
    [prepost [idoc_dir]/ /manual
        [progs]];

/* how to build each program */
loop prog = [progs]
{
    [prog]_obj = [match_mask
        [prog]/%.o [obj]];
}
bin/%: [%_obj]
{
    gcc [%_obj] -o [target];
}

/* how to build the library */
lib_obj = [match_mask lib/%.o
    [obj]];
lib/lib.a: [lib_obj]
{
    rm [target];
    ar cq [target] [lib_obj];
}

/* how to "install" stuff */
[ibin_dir]/%: bin/%
{
    cp bin/% [target];
}
[idoc_dir]/%/manual: doc/%/manual
{
    cp doc/%/manual [target];
}

/* how to compile sources*/
%0%.o: %0%.c
    [collect c_incl -api %0%.c]
{
    gcc -c %0%.c -o [target];
}
```

17. Advanced Features

Even though the tutorial part of this document is done, I feel it is important to just mention some advanced features not covered in the tutorial. Except for just stating the basic nature of these features I will not go into detail on any given one.

- Platform polymorphism. This is where Cook can automatically detect what platform you are on and do some file juggling so that you build for that platform.
- Support for private work areas. If you are working within a change management system, Cook knows how to query it for only the files you need to work on. This includes the automatic check-out and in of private copies of those files.
- Parallel builds. For large projects it is possible to spread the build over several processors or machines.

  Conditional recipes. It is possible to execute a recipe one way if certain

conditions are met and an other way if they are not.

Many more that are not directly supported by Cook but can easily be integrated using shell scripts.

18. Contacts

If you find any bugs in this tutorial please send a bug report to Aryeh M. Friedman <aryeh@m-net.arbornet.org>.

The Cook web site is http://miller.emu.id.au/pmiller/cook/

If you want to contact Cook's author, send email to Peter Miller <pmiller@opensource.org.au>.