Lake - a Lua-based Build Engine

Introduction

Hello, World!

Building large systems is a complicated task already, without factoring in the difficulty of being cross-platform. And the definition of 'portable' here means more than 'builds on most POSIX systems"!

`make` is a fine tool that does one thing well, managing dependencies. For everything else it relies on a Unix-like host system, which must be effectively be replicated for it to work elsewhere. There are a number of well-known solutions to this problem, for instance CMake, which will generate the makefiles or solution files needed. Lake is more like Scons in that it is a direct interpreter of build rules.

The goal of Lake is to make simple things trivial and complicated things manageable. In its simplest use, you may compile and run a C program directly with parameters:

```
$> lake hello.c one two "three 3"
gcc -c -O1 -Wall -MMD  hello.c -o hello.o
gcc hello.o  -Wl,-s -o hello.exe
hello.exe one two three
hello, world!
0 hello.exe
1 one
2 two
3 three 3
```

Subsequently it will only rebuild `hello.exe` when `hello.c` changes. If the Microsoft compiler `cl.exe` was on the path, it will use that in preference on Windows. For instance if you type this command in the Visual Studio Command prompt, we get:

```
D:\dev\lua\Lake\scratch>lake hello.c one two "three 3"
cl /nologo -c /O1 /WX /DNDEBUG /showIncludes  hello.c /Fohello.obj
link /nologo hello.obj  /OUT:hello.exe
hello.exe one two "three 3"
hello, world!
0 hello.exe
1 one
2 two
3 three 3
```

Lake understands how to drive these very different compilers, and will let you express builds in a platform-agnostic way.

It also (naturally) knows about Lua. To build LuaFileSystem, use the 'lua' flag (build as Lua extension):

```
D:\dev\lua\Lake\examples\errors\msvc\lfs>lake -lua lfs.c
cl /nologo -c /O1 /WX /I"C:/lua/include" /MD /DNDEBUG /showIncludes  lfs.c /Folfs.obj
link /nologo lfs.obj /DEF:lfs.def  /LIBPATH:"C:/lua/lib"  lua5.1.lib  /DLL /OUT:lfs.dll
   Creating library lfs.lib and object lfs.exp
```

(Note that it will use a .def file if present)

And on Ubuntu:

```
$ lake -lua lfs.c
gcc -c -O1 -Wall -I/usr/include/lua5.1 -fpic -MMD lfs.c -o lfs.o
gcc lfs.o -Wl,-s -shared -o lfs.so
```

There is also the 'p' flag for just building a program, and the 'l' flag for linking as a shared library.

As a special case `lake prog.lua` will run the Lua script in the Lake library environment. This makes it a useful general utility, particularly if it is packaged as a standalone executable.

We'll see in the section on defining new languages how other languages can be compiled/interpreted directly in this way.

Beyond One File

TThese simple command-line invocations are convenient and useful, but most programs have more than one source file.

Say we have two files, `main.c` and `utils.c`, with a shared header file `utils.h`, that uses the floating point libraries. Lake describes builds with *lakefiles*, which are Lua scripts executed within a special context:

```
-- lakefile
c.program{'hello',src='main utils',libs='m'}
```

The source files are specified here as a string (with or without commas) but could also be specified as a Lua table `{'main','utils'}`. No extensions are specified, since that is determined by the language and the platform.

This simple one-liner already does what we want; it automatically tracks the shared dependency on `utils.h` - if you modify that, both C files are recompiled. These dependencies are in the .d files, and are generated by the `-MMD` flag for gcc. (So the most tedious aspect of writing correct makefiles becomes automatic.)

It also provides a default 'clean' target, so `lake clean` does the expected thing. And the '-g' flag will generate a debug build, although you will have to do a clean first. We'll see later how to keep debug and release builds separate.

It is not yet truly portable - it assumes that the program must explicitly link against the math library, which is certainly true for Linux, but not for Windows. Lake handles details like this with the idea of *needs*. A portable lakefile looks like this, using the pre-defined need 'math':

```
c.program{'hello',src='main utils',needs='math'}
```

Other pre-defined needs are 'dl' (for programs that need to dynamically open shared libraries), 'readline' (for programs that need command-line history) and 'sockets' (which does require linking to an external library for Windows).

There is also the need 'lua' for programs and libraries that link against Lua itself, so a lakefile to build LuaFileSystem would be:

```
c.shared{'lfs',needs='lua'}
```

If there's no explicit 'src', it is deduced from the output name.

C++ is peculiar in that there is no 'canonical' file extension. In Lake, my prejudice means that C++ files have a default '.cpp' extension, but tool makers cannot be too dogmatic. So if a C++ program uses '.cxx' then:

```
cpp.program{'tester',ext='.cxx',src='main util database'}
```

Note that 'src' remains a list of files without their extension.

'src' may contain wildcards, which is convenient but grabs everything, hence 'exclude':

```
c.library{'lua',src='*',defines='LUA_USE_LINUX',exclude='lua luac print'}
```

Please note that 'list' here means both a string separated by commas or spaces, or a Lua table; all lists are converted internally into Lua tables. Filenames with spaces can be double-quoted in strings.

There are also some need aliases - for instance, 'gtk' is short for 'gtk+-2.0'. If a need is not known, then Lake tries to use `pkg-config`. So simply adding the need 'gtk' to a program will make it build against the GTK+ libraries.

The example in `examples/gtk' shows the complexity that a simple need can provide:

```
$ cat lakefile
c.program{'hello',needs='gtk'}
$ lake
gcc -c -O1 -Wall   -pthread -D_REENTRANT -I/usr/include/gtk-2.0
-I/usr/lib/gtk-2.0/include -I/usr/include/atk-1.0 -I/usr/include/cairo
-I/usr/include/pango-1.0 -I/usr/include/gio-unix-2.0/ -I/usr/include/glib-2.0
-I/usr/lib/glib-2.0/include -I/usr/include/pixman-1 -I/usr/include/freetype2
-I/usr/include/directfb -I/usr/include/libpng12   -MMD  hello.c -o hello.o
gcc hello.o  -pthread -lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgio-2.0
-lpangoft2-1.0 -lgdk_pixbuf-2.0 -lm -lpangocairo-1.0 -lcairo -lpango-1.0
-lfreetype -lfontconfig -lgobject-2.0 -lgmodule-2.0 -lgthread-2.0
-lrt -lglib-2.0   -Wl,-s -o hello
```

These compile commands are rather verbose for routine purposes; the '-b' flag just shows the build results:

```
$ lake -b
built hello.o
built hello
```

Configuration

Actually, we could do the last compilation all on the command-line:

```
$ lake NEEDS='gtk' hello.c
```

Global variables can be set on the Lake command-line, just like `make`, and some of these have specific meanings. To make them available for all compilations in a directory, then create a lakeconfig file:

```
-- lakeconfig.lua
NEEDS='gtk'
```

Another way for asking for a debug build is by setting `DEBUG=1`.

The order of configuration files is as follows: first try load `~/.lake/config.lua`, and then `./lakeconfig.lua`.

If there is an environment variable `LAKE_PARMS`, then it's assumed to be a list of name/value assignments seprated by a semi-colon. (This is currently the only specific environment variable used by Lake)

Another option is our old friend `require`. Lake modifies `package.path` so that modules are first found in `~/.lake`. This allows Lake-specific code to be separated out and easily updated without administrator privileges on Unix systems. There are some conventions; any imported new needs are 'lake.needs.NEED' and any new languages are `lake.lang.LANG`.

Building is a kind of Programming

One source of difficulty with building software is not recognizing that is a different kind of programming, which is dependency-oriented. Imperative thinking would result in a build environment where the functions directly execute the tools. We do sometimes write shell scripts like that, but tracking dependencies becomes hard.

these Lake one-liners involve a *language* (one of 'c','cpp', 'c99', 'cp11', 'f') and a *target kind*:: 'program','shared' (for DLLs) and 'library' (for static libraries). They do not execute the tools directly, but create a list of dependencies which is examined for changes; a target which is older than any of its dependencies is re-generated using the appropriate tool.

As with `make`, there must be a target which depends on all other targets ultimately; the root of the tree. Lake also chooses the first target generated as the default.

This is clearer with more complicated builds. Say we build a static library, and then build some executables using it. The Lua build on Linux works like this:

```
defs = 'LUA_USE_LINUX'
lualib = c.library{'lua51',src='*',defines=defs,exclude='lua luac print'}
lua = c.program{'lua',src='lua',defines=defs,deps=lualib,needs='math readline'}
luac = c.program{'luac',src='luac print',defines=defs,deps=lualib,needs='math'}
default {lua,luac}
```

'default' is the explicit way of specifying a default target and its immediate dependencies. (Technically, it's a 'dummy' target because it does not actually correspond to a file). So we depend on the target `lua`, which depends on the target `lualib`, which in term depends on all the object files, and so on. Unlike `make`, the targets have to be defined before they can be used, which is why we need an explicit 'default' - if you leave it out, this lakefile will happily build the Lua static library, and then stop.

'deps' serves two purposes here; it explicitly specifies a dependency, and implicitly provides something to link against. If (say) `loadlib.c` changes, then the output `liblua51.a` must be rebuilt, and since `lua` depends on that, it will also be rebuilt and link against it.

Unlike `make`, flags such as 'defines' are not global. This gives us great flexibility, but it can be more verbose. This is where 'defaults' is useful:

```
defs = 'LUA_USE_LINUX'
c.defaults { defines = defs }
lualib = c.library{'lua51',src='*',exclude='lua luac print'}
lua = c.program{'lua',src='lua',deps=lualib,needs='math readline'}
luac = c.program{'luac',src='luac print',deps=lualib,needs='math'}
default {lua,luac}
```

then all subsequent C target types will use these defines. Defaults are used if the corresponding field has not been explicitly defined; nothing clever like merging values occurs.

This does work, but it is not yet cross-platform. Usually on Windows we build a DLL rather than a static library, and link 'lua.exe' against that - except for 'luac.exe' which is always statically linked. And it's often useful to have Lua as a shared library on Unix.

There are several globals available to lakefiles which are useful for making platform decisions. `PLAT` is either 'Windows' or the result of `uname -a`; `WINDOWS` is true if `PLAT=='Windows'` and `CC` is the actual compiler used.

Later I will discuss a more complete build for Lua that uses this information to give a result more appropriate to the platform.

Basics

Targets and Dependencies

Building software and preparing websites both involve tools which take input files and convert them to output files. For example, a task may involve resizing original images and converting Markdown files into HTML. It's easy enough to write scripts which explicitly apply the desired tool to given files, but this can involve extra work for the user and potentially much redundant processing by the computer. Hundreds of images take a while to be processed, and it's irritating and unnecessary to do this everytime a new image is added.

You only want to convert files which have changed, and this is the role of dependency-tracking tools like `make` . The output files are called the *targets*, and each target depends on one or more input files, which are called *prerequisites* in `make` terminology, or simply *dependencies* in Lake.

Just as the instructions for `make` are contained inside *makefiles*, the equivalent files for Lake are called *lakefiles*. When Lake is run without any parameters, it will look for `lakefile` or `lakefile.lua`. Lakefiles are Lua scripts which can use the full power of the language, but typically a lakefile is organized around explicit targets and dependencies.

The basic function `target` connects an output fille, the required files (or dependencies) and the command or function needed to produce that output.

```
target('sgm.bak','sgm.c','copy $(DEPENDS) $(TARGET)')
```

Given a called `lakefile` with this line, the Lake command gives the following output when executed twice:

```
D:\dev\app>lake
copy sgm.c sgm.bak
        1 file(s) copied.

D:\dev\app>lake
lake: up to date
```

The copy command is only executed the first time, because after copying the file `sgm.bak` will be more recent than `sgm.c`. Lake will only re-copy `sgm.c` when it has changed, and becomes more recent than `sgm.bak` (or if `sgm.bak` has been deleted.)

The command argument contains the variables `DEPENDS` and `TARGET` which will be replaced by their actual values when the target is generated. In this case, you could use the explicit names, but it's better to only have to mention the names once. It's then possible to make a number of similar target actions:

```
ccmd = 'copy $(DEPENDS) $(TARGET)'
target('sgm.bak','sgm.c',ccmd)
target('test.bak','test.c',ccmd)
```

This will not work as expected. Lake cannot guess what all the targets are and chooses to run the first-defined target, like `make`. So here is a `make`-like solution - define a target upfront which depends on the two copy targets:

```
ccmd = 'copy $(DEPENDS) $(TARGET)'
target('all','sgm.bak, test.bak')
target('sgm.bak','sgm.c',ccmd)
target('test.bak','test.c',ccmd)
```

Here the second argument to target is now a list of files, and the third argument is not given, since this target isn't really a file and merely exists to ensure that the dependencies are checked. So Lake sees that 'all' requires both `sgm.bak` and `test.bak`, and then examines their dependencies in turn. This is the central point to understand; a target depends on other targets, which depend on others, and so on. Lake will follow the dependencies until it finds the files, or finds a rule that generates that file.

Lists of files are common in Lake and can be space/comma separated strings, or as tables. So the 'sgm.bak, test.bak' could also be written as 'sgm.bak test.bak' or `{'sgm.bak','test.bak'}`.

A more Lake-ish way of writing the same lakefile is:

```
ccmd = 'copy $(DEPENDS) $(TARGET)'
t1 = target('sgm.bak','sgm.c',ccmd)
t2 = target('test.bak','test.c',ccmd)
default {t1,t2}
```

The `default` function creates a target that depends on the list of targets provided, and forces itself to the top of the list of all targets. This fits in better with the way Lua works and also doesn't require re-specifying filenames (Lua programmers tend to assume that the action starts at the end of a file;)).

Rules

Consider the problem of working with an arbitrary set of `.c` files. A programmer-friendly solution is:

```
ccmd = 'copy $(DEPENDS) $(TARGET)'
targets = {}
for file in path.mask '*.c' do
    local bak = path.change_extension(file,'.bak')
    table.insert(targets,target(bak,file,ccmd))
end
default (targets)
```

Again, `default` takes a list of target objects, which have been explicitly generated in a loop over all files matching the file mask `*.c`. Lake provides functions like `mask` and `change_extension` to make working with files and directories easier but there is a more elegant way of solving the problem using `rule`:

```
crule = rule('.bak','.c','copy $(INPUT) $(TARGET)')
crule '*.c'
default (crule)
```

A Lake rule is constructed by `rule`, and the arguments are output extension, input extension, and command (as passed to `target`) - that is, in the same order as `target`. (earlier versions of Lake had this the other way around.) A rule object is a factory for creating targets, and it is callable; it can be passed a target name, or a file mask.

Note the `INPUT` variable; this is more specific than `DEPENDS` - generally a target may depend on many files, but the rule defines the input precisely as `NAME.in_ext`. This little lakefile shows the difference; here the target depends on two files, and `$(DEPENDS)` is always the dependencies separated by spaces.

```
target('arb','sgm.c test.c','echo $DEPENDS')
```

The output is:

```
echo sgm.c test.c
sgm.c test.c
```

(Again, the second argument could be written `{'sgm.c','test.c'}`)

The rule object has associated targets, and functions expecting a list of dependencies will treat it as a list of targets. Since calling a rule object returns the object itself, the last two lines can be simply expressed as `default {crule '*.c'}`.

As it stands, this rule is very platform-dependent. But a lakefile is just a Lua script, so it is easy to define a new global and have it substituted:

```
if WINDOWS then
    COPY = 'copy'
else
    COPY = 'cp'
end
crule = rule('.bak','.c','$(COPY) $(INPUT) $(TARGET)')
default (crule '*.c')
```

There is an important difference between an ordinary global like `COPY` and basic variables like `INPUT`. Basic variables are only substituted when the target action 'fires'; the initial set is `INPUT,TARGET,DEPENDS,LIBS,CFLAGS`.

Another example is converting image files using ImageMagick, which provides `convert`, the Swiss Army Knife of image file converters.

```
to_png = rule('.png','.jpg',
  'convert $(INPUT) $(TARGET)'
)
```

```
default(to_png '*')
```

This lakefile will convert all the JPEG files in the current directory to PNG, and thereafter will only update PNG files if any of the JPEGs change.

It is possible to construct a rule which can work on all extensions, but you do have to be careful that the target files are not in the same directory as the input files.

```
crule = rule('*','*','$(COPY) $(INPUT) $(TARGET)')
crule.output_dir = 'temp'
```

Having a way to copy groups of files is sufficiently useful that Lake defines `copy.group`, which works like any `group` function.

Actions may be Functions

Up to now the action specified explicitly for a target or indirectly by a rule has been a shell command. This action may also be a function:

```
-- test.lake
target('out.c','out.tmpl',function(t)
   dump(t,'target fields')
   dump(t.deps,'dependencies')
end)
```

Lake provides a simple table dumper, so we can see exactly what the *target object* `t` contains:

```
$ lake -f test.lake
<<< target fields
deps    table: 0x9878188
cmd function: 0x988cc58
time    -1
target  out.c
>>
<<< dependencies
1   out.tmpl
>>
```

Armed with this information, a simple source translation would look as follows:

```
target('out.c','out.tmpl',function(t)
    local tmpl = file.read (t.deps[1])
    file.write(t.target,tmpl:format(os.date()))
end)
```

Here a source file has been generated from a template, using a trivial transformation which replaces the first %s in the template with a timestamp. If you wanted `out.c` re-created for *every* build, then specify `nil` for the dependencies and use 'out.tmpl' instead of `t.deps[1]`.

Using a full-featured template library like Cosmo gives you much more control over the generated code. As a simple alternative, Lake provides `utils.substitute`:

```
> =  utils.substitute('$(dog) likes $(cat)',{dog='Bonzo',cat='Felix'})
Bonzo likes Felix
```

There is some syntactical sugar for some common target usages. `target.fred 'one two'` is equivalent to `target('fred','one two')`.

`action` is an alias for creating *unconditional* targets where the action is always a function.

An application of function actions is *rule-based programming*.

Martin Fowler has an article on using Rake for managing tasks with dependencies. Here is his first rakefile:

```
task :codeGen do
  # do the code generation
end

task :compile => :codeGen do
  #do the compilation
end

task :dataLoad => :codeGen do
  # load the test data
end

task :test => [:compile, :dataLoad] do
  # run the tests
end
```

This lakefile is equivalent:

```
task = target

task('codeGen',nil,function()
  print 'codeGen'
end)

task('compile','codeGen',function()
  print 'compile'
end)

task('dataLoad','codeGen',function()
  print 'dataLoad'
end)

task('test','compile dataLoad',function()
  print 'test'
end)
```

Try various commands like 'lake compile' and 'lake test' to see how the actions are called. The default target here would be 'codeGen' since it was the first target defined. (see the `examples/fowler` directory.)

You may find Lua's anonymous function syntax a little noisy. But there's nearly always another way to do things in Lua. This style is probably more natural for Lua programmers:

```
-- fun.lua
actions,deps = {},{}

function actions.codeGen ()
  print 'codeGen'
end

deps.compile = 'codeGen'
function actions.compile ()
    print 'compile'
end

deps.dataLoad = 'codeGen'
function actions.dataLoad ()
    print 'dataLoad'
end

deps.test = 'compile dataLoad'
function actions.test ()
    print 'test'
end

for name,fun in pairs(actions) do
```

```
    target(name,deps[name],fun)
end

default 'test'
```

An interesting aspect of this style of programming is that the order of the dependencies firing is fairly arbitrary (except that the sub-dependencies must fire first) so that they could be done in parallel.

As a fun exercise, consider Moonscript as a way of generating makefiles

```
-- alternative.moon
task = target

task.codeGen nil, ->
    print 'codeGen'

task.compile 'codeGen',->
    print 'compile'

task.dataLoad 'codeGen',->
    print 'dataLoad'

task.test 'compile dataLoad',->
    print 'test'

default 'test'
```

That looks even cleaner than the original Ruby example, due to the lightweight function syntax:

```
$ moonc alternative.moon
Built   ./alternative.moon
$ lake -f alternative.lua
codeGen
compile
codeGen
dataLoad
test
lake: 'build' took  0.00 sec
lake: up to date
```

You can name the Moonscript file `lakefile.moon`, and then the output will be `lakefile.lua` and be accepted directly by Lake.

Dependency-Based Programming with Objects

New with 1.4 is the capability to use *objects* as targets. To be acceptable as a target, an object must be a table with a `time` field which behaves like a timestamp, and have no array items.

Here is a suitable 'class' definition for such an object:

```
-- dobject.lua
-- classic Lua OOP boilerplate
local TO = {}
TO.__index = TO
TO._NOW = 1
TO._objects = {} -- keep a list of these guys

-- objects can show themselves as a string
function TO:__tostring()
    return "["..self.name..':'..self.time.."]"
end

-- they may have a method for updating - basically this is 'touch'
function TO:update()
    print('updating '..tostring(self))
    self.time = TO._NOW
end

-- by default, objects have time 0!
local function T(name,time)
    local obj = setmetatable({name=name,time=time or 0},TO)
    table.insert(TO._objects, obj)
    return obj
end
```

Given this class, we can do dependency-based programming.

```
-- lazy global object generation - unknown uppercase vars become target objects
setmetatable(_G,{
    __index = function(self,key)
        if key:match '^%u+$' then
            local obj = T(key)
            rawset(_G,key,obj)
            return obj
        end
    end
})

local function touch(t)
    t.target:update()
end

-- B is younger than A, so A is updated
-- (comment this out and nothing happens)
B.time = 1

tA = target(A, {B, C},touch)

-- which in turn forces action on D (but it is not updated)
tB = target(D, A, function(t)  -- could also have tA as dep..
    print('D action!')
    for o in list(TO._objects) do print(o) end
end)

default{ tB }
```

How Lake is Configured

The command Lake will load configuration files, if it can find them. It will first try load `~/.lake/config` as a Lua script. (In Windows, `~` means something like `c:\Users\Name`) It will then try to load `lakeconfig` in the current directory, so that local configuration takes precedence. These files may have a `.lua` extension.

You can then define custom rules in the user or local configuration file and use them as prepackaged functionality.

The Lua package path is modified so that Lake first looks in the `~/.lake` directory, so that `require 'mymod'` will match `~/.lake/mymod.lua`.

(You can also use `require` to bring in Lake configuration files from the usual Lua package path - this is the recommended way to configure Lake for all users. For instance, you can use `require 'lake.global`. For a Unix system this script would have a path like `/usr/local/share/lua/5.1/lake/global.lua`.)

Next, any arguments to Lake of the form `VAR=STRING` set the global variable `VAR` to the value `STRING`.

If there is an environment variable `LAKE_PARMS`, it is assumed to consist of variable-value pairs separated by semicolons; such arguments on the Lake command-line take precendence.

Finally, if the global `LAKE_CONFIG_FILE` is defined, then it is assumed to be a configuration file and loaded.

Building Programs and Libraries with Lake

Usual Pattern for Build Tools

The usual pattern for compilers is this: source files are *compiled* into object files, which are *linked* together to make programs or libraries. Generally the compilation phase is the time-consuming part, so we wish to only re-compile files which have changed, or which *depend* on files that have changed. This is important for languages (like C/C++) where the extra dependencies comes from include or header files. These dependencies can come from a header file itself including other header files, and so forth, and has traditionally been an awkward part of organizing the efficient building of large systems. You do not want to rebuild files unnecessarily, but you definitely do not want to miss out rebuilding something, since the symptoms can be subtle and hard to track down.

Schematically, these tools work like this:

source files, compilation flags -> **COMPILER** -> object files, dependency information.

object files, linker flags -> **LINKER** -> program, shared library, static library.

One of the things that Lake can do for you is auto-generate dependency information using facilities provided by the supported compilers. In this way, a complex build can be specified with a compact lakefile and you can expect the right thing to happen.

Building a Simple Program

Lake organizes its functionality in language objects. To build a simple C program is easy:

```
c.program 'hello'
```

(Lua conveniently allows the parentheses for function calls to be left out if the single argument is a string or a table, and we will follow that convention here.)

The name of the program given, and the source file is assumed to be the name with the appropriate extension.

```
c.program {'hello',src = 'hello utils'}
```

This version has two source files specified explicitly. The value of `src` follows the usual Lake convention for lists (a table or a string of separated names) or a wildcard. Please note that it is easy to forget to use *curly braces* here; the argument is a Lua table.

`src` can contain wildcards. So `src = '*'` can be used to specify the source files, and `exclude` can be used to filter the result:

```
c.program{'hello',src='*',exclude='test'}
```

`exclude` uses the same rules as `src`, so you could exclude any source file that began with `test-` with a wilcard, etc.

Often the language does not fully specify the extension. C++ files have a number of common extensions (including upper-case C). My preference is `.cpp`; but it's easy to override this with `ext`:

```
cpp.program{'hello',ext='.cxx',src='*'}
```

At this point, it's useful to step back and examine what Lake is providing with these simple recipes, starting with the simplest lakefile:

```
$> cat lakefile
c.program{'hello'}

$> lake
gcc -c -O1 -MMD  hello.c
gcc hello.o  -o hello.exe
```

This lakefile automatically understands a 'clean' target, and the `-g` option forces a debug build:

```
$> lake clean
```

```
removing        hello.exe
removing        hello.o
$> lake -g
gcc -c -g -MMD  hello.c
gcc hello.o  -o hello.exe
```

(You can achieve the same effect as `-g` by passing `DEBUG=true` on the command-line)

If running Windows, and the MS compiler cl.exe is on your path, then:

```
$> lake -g
cl /nologo -c /Zi /showIncludes  hello.c
link /nologo hello.obj  /OUT:hello.exe
```

Lake knows the common flags that these compilers use to achieve common goals - in this case, a debug build. This places less stress on human memory (which is not a renewable resource) especially if you are working with a compiler which is foreign to you.

Now, what if `hello.c` had a call to a math function? No problem with Windows (it's part of the C runtime) but on Unix it is a separate library. A program target that has this *need* would be:

```
c.program{'hello',needs='math'}
```

On Unix, we will now get the necessary `-lm`. All this can be done with a makefile, but it would already be an irritating mess, even if it just handled `GCC` alone. The purpose of Lake is to express build rules in a high-level, cross-platform way.

Dependency Checking

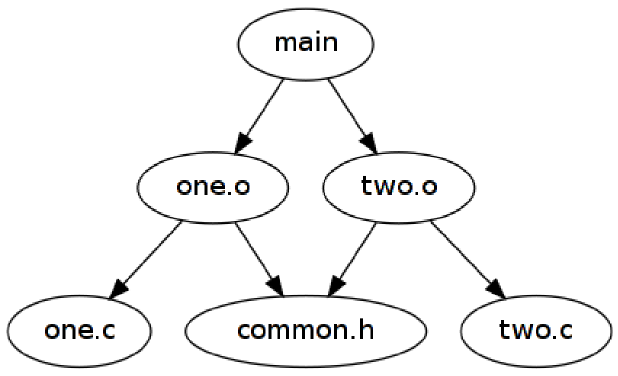
Looking at `examples/first`:

```
$> cat lakefile
c.program{'first',src='one,two',needs='math'}

$> lake
gcc -c -O1 -MMD  one.c
gcc -c -O1 -MMD  two.c
gcc one.o two.o   -o first.exe
```

This simple lakefile does dependency checking; if a source file changes, then it is recompiled, and the program is relinked since it depends on the output of the compilation. We don't need to rebuild files that have not changed.

```
$> touch one.c
$> lake
gcc -c -O1 -MMD  one
gcc one.o two.o   -o first.exe
```

Actually, Lake goes further than this. Both `one.c` and `two.c` depend on `common.h`; if you modify this common dependency, then both source files are rebuilt.



Lake knows about the `GCC` `-MMD` flag, which generates a file containing the non-system header files encountered during compilation:

```
$> cat one.d
one.o: one.c common.h
```

This also works for the `CL` compiler using the somewhat obscure `/showIncludes` flag.

So the lakefiles for even fairly large code bases can be short and sweet. In `examples/big1` there are a hundred generated .c files, with randomly assigned header dependencies:

```
$> cd examples/big1
$> cat lakefile
c.program {'name',src='*'}
```

The initial build takes some time, but thereafter rebuilding is quick.

By default, Lake compiles one file at a time. If you set the global `COMBINE` then it will try to compile as many files as possible with one invocation. Both `GCC` and `CL` support this, but *not* if you have explicitly specified an output directory.

With modern multi-core processors, a better optimization is to use the `-j` ('jobs') flag which works like the equivalent `make` flag; run tools in parallel processes if possible.

Building Lua Extensions

Lake has special support for building Lua C/C++ extensions. In `examples/lua` there is this lakefile:

```
c.shared{'mylib',needs='lua'}
```

And the build is:

```
gcc -c -O1 -MMD -Ic:/lua/include   mylib.c
gcc mylib.o mylib.def  -Lc:/lua/lib  -llua5.1  -shared -o mylib.dll
```

Lake will attempt to auto-detect your Lua installation, which can be a little hit-and-miss on Windows if you are not using Lua for Windows. It may be necessary to set `LUA_INCLUDE_DIR` and `LUA_LIB_DIR` explicitly in a local `lakeconfig` or user `~/.lake/config`.

On Unix with a 'canonical' Lua install, things are simpler:

```
gcc -c -O1 -MMD -fPIC mylib.c
gcc mylib.o   -shared -o mylib.so
```

On Debian/Ubuntu, the `liblua5.1-dev` package puts the include files in its own directory:

```
gcc -c -O1 -MMD -I/usr/include/lua5.1 -fPIC mylib.c
gcc mylib.o   -shared -o mylib.so
```

With Lua for Windows, you have to be a little careful about the runtime dependency for non-trivial extensions. LfW uses the VC2005 compiler, so either get this, or use `GCC` with LIBS='-lmsvcr80'. The situation you are trying to avoid is having multiple run-tiime dependencies, since this will bite you because of imcompatible heap allocators.

The 'lua' need also applies to programs embedding Lua. It is recommended to link such programs against the shared library across all platforms, to ensure that the whole Lua API is available.

The Concept of Needs

Compiling and linking a target often requires platform-specific libraries. A Unix program needs `libm.a` if it wants to link to `fabs` and `sin` etc, but a Windows program does not. We express this as the *need* 'math' and let Lake sort it out.

Other common Unix needs are 'dl' if you want to load dynamic libraries directly using `dlopen`. On the other side, Windows programs need to link against `wsock32` to do standard Berkerly-style sockets programming; the need 'sockets' expresses this portably. The need 'readline' is superfluous on Windows, since the shell provides most of this functionality; on Linux it also implies linking against `ncurses` and `history`; on OS X linking against `readline` is sufficient.

The built-in needs are currently: 'math','readline','dl','sockets' and 'lua'.

There are also two predefined needs for GTK+ programming: 'gtk' and 'gthread'. These are implemented using `pkg-config` which returns the include directories and libraries necessary to build against these packages.

If a need is unknown, then Lake will try to use `pkg-config`.

For instance, installing the computer vision library OpenCV updates the package database:

```
$ pkg-config --cflags --libs opencv
-I/usr/local/include/opencv  -L/usr/local/lib -lcxcore -lcv -lhighgui -lcvaux -lml
```

so `needs='opencv'` will work with a standard install of OpenCV.

Not resolving the package with `pkg-config` is only an error if the need has been *explicitly* defined as requiring it. Lake defines 'gtk' like this:

```
lake.define_pkg_need('gtk','gtk+-2.0')
```

which provides a convenient alias, but also insists that `pkg-config` be available and aware of the package. So for example, a lakefile for an OpenCV program may also insist on this behaviour with `lake.define_pkg_need('opencv','opencv')`.

Finally, Lake assumes that the need has been manually specified, and it will complain if these are wrong. It tries to make constructive criticism. Say I have:

```
c.program{'bar',needs = 'foo baz'}
```

then we will get:

```
$ lake
--- variables for package foo
FOO_INCLUDE_DIR = 'NIL' --> please set!
FOO_LIB_DIR = 'NIL' --> please set!
FOO_LIBS = 'foo' --> please set!
----
--- variables for package baz
BAZ_INCLUDE_DIR = 'NIL' --> please set!
BAZ_LIB_DIR = 'NIL' --> please set!
BAZ_LIBS = 'baz' --> please set!
----
use -w to write skeleton needs files
lake: unsatisfied needs
```

This is in a form that can be directly used in a configuration file; you can then say `lake -w` to put this into needs files, which you can then edit to your satisfaction:

```
$ lake -w
writing baz.need.lua
writing foo.need.lua
please edit the needs files!
lake: unsatisfied needs
$ cat *.need.lua
--- variables for package baz
BAZ_INCLUDE_DIR = 'NIL' --> please set!
BAZ_LIB_DIR = 'NIL' --> please set!
BAZ_LIBS = 'baz' --> please set!
----
--- variables for package foo
FOO_INCLUDE_DIR = 'NIL' --> please set!
```

```
FOO_LIB_DIR = 'NIL' --> please set!
FOO_LIBS = 'boo' --> please set!
----
```

Once this works, you can then install these into your Lake home:

```
$ lake -install baz.need
$ lake -install foo.need
```

The most common way to install a package in Windows is to put it into its own directory. If you specify `FOO_DIR` then Lake will try to find `include` and `lib` subdirectories.

Another way of seeing this is that Lake expects global variables of this form in order to satisfy a need. So you simply might have in your lakefile:

```
FOO_LIBS = 'foo3'
if WINDOWS then
    FOO_DIR = 'c:\\foolib'
else
    FOO_INCLUDE_DIR = '/usr/include/foo3'
end
```

If there is a Lua module of the form 'lake.needs.NAME', then it will be loaded. Here 'NAME' can be a simple name or be 'PACKAGE-SUB'. The module is assumed to return a function, which will be passed the 'SUB' name if present.

For example, a module that satisfies a simple 'foo' need would be called 'lake.needs.foo' and could simply look like this:

```
return function()
    FOO_INCLUDE_DIR = '/usr/include/foo3'
    FOO_LIBS = 'foo3'
end
```

Now imagine that this module does some more sophisticated, OS-dependent checking, and we have a mechanism that can do arbitrary work to satisfy a need. Plus, `luarocks` can be then used to deliver a particular need to all users.

Additional needs can also be specified by the `NEEDS` global variable. If I wanted to build a program with OpenCV, I can either say:

```
$ lake NEEDS=opencv camera.c
```

or I can make all programs in a directory build with this need by creating a file `lakeconfig` with the single line:

```
NEEDS = 'opencv'
```

and then `lake camera.c` will work properly.

Release, Debug and Cross-Compile Builds

If `program` has a field setting `odir=true` then it will put output files into a directory `release` or `debug` depending if this is was a release or debug build (`-g` or `DEBUG=true`.)

This is obviously useful when switching between build versions, and can be used to build multiple versions at once. See `examples/releases' - the lakefile is

```
-- maintaining separate release & debug builds
PROG={'main',src='../hello',odir=true}
release = c.program(PROG)
lake.set_flags {DEBUG=true}
debug = c.program(PROG)
default{release,debug}
```

Please note that global variables affecting the build should be changed using `set_flags()`

This feature naturally interacts with cross-compilation. If the global `PREFIX` was set to `arm-linux` then the compiler becomes `arm-linux-gcc` etc. The release directory would become `arm-linux-release`.

`odir` (alias `output_directory`) can explicitly be set to a directory name.

Shared Libraries

Unix shared libraries and Windows DLLs are similar, in the sense that both orcas and sharks are efficient underwater predators but are still very different animals.

Consider `lib1.c` in `examples/lib1`; the lakefile is simply:

```
c.shared {'lib1'}
```

which results in the following compilation:

```
gcc -c -O1 -MMD  lib1.c
gcc lib1.o  -shared -o lib1.dll
```

(Naturally, the result will be `lib1.so` on Unix.)

By default, `GCC` exports symbols; using the MS tool `dumpbin` on Windows reveals that the function `answer` is exported. However, `CL` does not. You need to specify exports explicitly, either by using the `__declspec(dllexport)` decoration, or with a DEF file:

```
$> cat lib1.def
LIBRARY lib1.dll
EXPORTS
        answer

$> lake
cl /nologo -c /O1 /showIncludes  lib1.c
link /nologo lib1.obj /DEF:lib1.def  /DLL /OUT:lib1.dll
   Creating library lib1.lib and object lib1.exp
```

So on Windows, if there is a file with the same name as the DLL with extension .def, then it will be used in the link stage automatically.

(Most cross-platform code tends to conditionally define `EXPORT` as `__declspec(dllexport)` which is also understood by `GCC` on Windows.)

There is a C program `needs-lib.c` which links dynamically against `lib1.dll`. The lakefile that expresses this dependency is:

```
lib = c.shared {'lib1'}
c.program{'needs-lib1',lib}
```

Which results in:

```
gcc -c -O1 -MMD  needs-lib1.c
gcc -c -O1 -MMD  lib1.c
gcc lib1.o lib1.def  -shared -o lib1.dll
gcc needs-lib1.o lib1.dll  -o needs-lib1.exe
```

In this lakefile, the result of compiling the DLL (its *target*) is added as an explicit dependency to the C program target. `GCC` can happily link against the DLL itself (the recommended practice) but `CL` needs to link against the 'import library'. Again, the job of Lake is to know this kind of thing.

Linking against the C Runtime

This is an example where different compilers behave in different ways, and is a story of awkward over-complication. On Unix, programs link dynamically against the C runtime (libc) unless explicitly asked not to, whereas `CL` links statically. To link a Unix program statically, add `static=true` to your program options; to link a Windows `CL` program dynamically, add `dynamic=true`.

It is tempting to force consistent operation, and always link dynamically, but this is not a wise consistency, because `CL` will then link against `msvcr80.dll`, `msvcr90.dll` and so on; you will have to redistribute the runtime with your application anyway, either as a private side-by-side assembly or via `VCDist`.

Here is the straight `CL` link versus the dynamic build for comparison:

```
link /nologo test1.obj  /OUT:test1.exe

link /nologo test1.obj msvcrt.lib /OUT:test1.exe && mt -nologo -manifest test1.e
xe.manifest -outputresource:test1.exe;1
```

The first link gives a filesize of 48K, versus 6K for the second. But the dynamically linked executable has an embedded manifest which is only satisfied by the *particular* version of the runtime for that version of `CL` (and it is picky about sub-versions as well.) - so you have to copy that exact DLL (msvcr80.dll, msvcr90.dll, depending) into the same directory as your executable, and redistribute it alongside. So the size savings are only worth it for larger programs which ship with a fair number of DLLs. This is (for instance) the strategy adopted by Lua for Windows.

Partitioning the Build

Consider the case where there are several distinct groups of source files, with different defines, include directories, etc. For instance, some files may be C, some C++, for instance the project in `examples/main`. One perfectly good approach is to build static libraries for distinct groups:

```
lib = c.library{'lib'}
cpp.program{'main',lib}
```

(It may seem silly to have a library containing exactly one object file, but you are asked to imagine that there are dozens or maybe even hundreds of files.)

This lakefile shows how this can also modelled with *groups*;

```
main = cpp.group{'main'}
lib = c.group{'lib'}
cpp.program{'main',inputs={main,lib}}
```

There is main.cpp and lib.c, and they are to compiled separately and linked together.

`program` normally constructs a compile rule and populates it using the source, even if it is just inferred from the program name. Any options that only make sense to the compile rule get passed on, like `incdir` or `defines`. But if `inputs` is specified directly, then `program` just does linking. `group`, on the other hand, never does any linking, and can only understand options for the compile stage.

Compile-time Dependencies

'deps' is a way to make a program/library target become dependent on other targets. But we need another way to introduce dependencies into the compilation stage.

Consider the case where a header file is copied into a another directory. That is done with a file group; we want the program to rebuild when the header changes.

```
f = file.group{src=path.join('common','common.h'),odir='include'}

prog = c.program{'first',
    src = 'one common/common',
    incdir = 'include',
    compile_deps =  f
}

default {prog}
```

Here 'compile_deps' expresses the fact that all the source files depend on `include/common.h`, which in turn depends on `common/common.h`. This is a useful pattern when headers are generated by some code.

A More Realistic Example

Lua is not a difficult language to build from source, but there are a number of subtleties involved. For instance, it is built as a standalone executable with exported symbols on Unix, and as a stub program linked against a DLL on Windows. Here is the lakefile, section by section:

```
LUA='lua'
LUAC='luac print'

as_dll = WINDOWS
if as_dll then
  defs = 'LUA_BUILD_AS_DLL'
end
if not WINDOWS then
  defs = 'LUA_USE_LINUX'
end
```

The first point (which should not come as too much of a suprise) is that this is actually a Lua program. All the power of the language is available in lakefiles. Lake sets some standard globals like WINDOWS and PLAT.

```
-- build the static library
lib,ll=c.library{'lua',src='*',exclude={LUA,LUAC},defines=defs}
```

The Lua static library (`.a` or `.lib`) is built from all the C files in the directory, *except* for the files corresponding to the programs `lua` and `luac`. Depending on our platform, we also have to set some preprocessor defines.

```
-- build the shared library
if as_dll then
  libl = c.shared{'lua',inputs=ll,dynamic=true}
else
  libl = lib
end
```

On Windows (or Unix *if* we wanted) a DLL is built as well as a static library. This DLL shares the same *inputs* as the static library - these are the second thing returned by the first `library` call. The `dynamic` option forces the DLL to be dynamically linked against the runtime (this is not true by default for `CL`.)

```
-- build the executables
lua = c.program{'lua',libl,src=LUA,needs='dl math readline',export=not as_dll,dynamic=true}
luac = c.program{'luac',lib,src=LUAC,needs='math'}

default {lua,luac}
```

The `lua` program either links against the static or the dynamic library; if statically linked, then it has to export its symbols (otherwise Lua C extensions could not find the Lua API symbols). Again, always link against the runtime (`dynamic`).

This executable needs to load symbols from shared libraries ('dl'), to support interactive command-line editing ('readline') and needs the maths libraries ('math'). Expressing as needs simplifies things enormously, because Lake knows that a program on Linux that needs 'readline' will also need to link against 'history' and 'ncurses', whereas on OS X it just needs to link against 'readline'. On Windows, equivalent functionality is part of the OS.

The `luac` program always links statically.

Finally, we create a target with name 'default' which depends on the both of these programs, so that typing 'lake' will build everything.

Expressing the Lua build as a lakefile makes the build *intents* and *strategies* clear, whereas it would take you a while to work these out from the makefile itself It also is inherently more flexible; it works for both `CL` and `GCC`, a debug build just requires `-g` and it can be persuaded easily to give a `.so` library on Unix.

Precompiled Headers

Many of C++'s evils are inherited from C. In particular, it uses the very same separate compilation model, with 'dumb' object files and heavy use of the preprocessor. A simple 'Hello, World' C++ program with iostreams involves including over 18K lines of headers. So `g++` is *not* a slow compiler, but it has to get through a lot of headers, mostly involving tricky-to-parse template code.

One common solution, available for both our reference compilers, is precompiled headers. You isolate the big headers and compile them in a special binary form that subsequent compilations can more easily digest.

'examples/precomp' shows the strategy. `std.h` has all the heavy headers used globally:

```
// std.h
#ifndef STD_H
#define STD_H
#include <iostream>
#include <string>
#include <list>
#include <map>
#endif
```

And the lakefile looks like this:

```
cpp.program {'hello',precompiled_headers='std.h'}
```

To give an indication of the sheer amount of information in these headers, the size of `std.gch` generated by `g++` is over 7 megabytes! In this case we don't gain much, but in a project with many files, this can significantly speed up compilation. This is particularly true for the Microsoft compiler, which has had precompiled headers for much longer.

As before, Lake captures the basic pattern and implements it in a compiler-specific way so that builds can be more portable.

Parallel Building

Most computers (even the ones sneaking into your pocket) have multiple cores. `make` has a 'j' flag for specifying the number of jobs that can be run in parallel, and it can make a big difference, especially for full rebuilds. I felt that this was a feature that Lake needed as well, and the invocation with 'j' is the same.

It does require extra library support - on Windows, winapi is used, elsewhere luaposix. On Linux, luaposix is available through the package manager, and the .deb for Lake will bring it in as a dependency. On Windows the standalone executable comes with winapi. If these libraries are missing, then Lake will bug you if you use 'j'.

On an AMD 4-core Linux server, Lake was able to do a full rebuild of Lua in 2 seconds, just under six times faster than a single-core build. I got a similar build time on Windows 7 64bit (i3) with MSVC 2010, although gcc did not respond so dramatically to multiple jobs. (Both of these machines are a few years behind the curve, by the way, and my younger colleagues would be somewhat scornful of them.)

The moral of the story: do some experiments to find the optimal value to give to 'j'. A rule of thumb is twice the number of available cores. The function `lake.concurrent_jobs` can be also used to set the number of threads in lakefiles.

The assumption used is that the targets generated by any particular rule may be safely compiled in parallel. We could do better with dependency analysis, but it's good enough for now, and it's properly conservative.

Massaging Tool Output

Although in many ways an easier language to learn initially than C, C++ is sometimes its own worst enemy. The extensive use of templates in Boost and the standard library can make error messages painful to understand at first.

Consider the following silly C++ program (and remember that we start by writing silly programs):

```
// errors.cpp
```

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
  list<string> ls;
  ls.append("hello");
  cout << "that's all!" << endl;
  return 0;
}
```

Actually, this program is more than half-competent, for a beginner who doesn't know the standard libraries well.

The original error message is:

```
errors.cpp:9: error: 'class std::list<std::basic_string<char, std::char_traits<char>,std::allocator<char> >,   std::alloc
```

Seasoned C++ programmers learn to filter their error messages mentally. Lake provides the ability to filter the output of a compiler, and reduce irrelevant noise. Here is the lakefile:

```
if CC ~= 'g++' then quit 'this filter is g++ specific' end
lake.output_filter(cpp,function(line)
  return line:gsub('std::',''):
    gsub('basic_string%b<>','string'):
    gsub(',%s+allocator%b<>',''):
    gsub('class ',''):gsub('struct ','')
end)

cpp.program {'errors'}
```

And now the error is reduced to:

```
errors.cpp:9: error: 'list<string >' has no member named 'append'
```

We have thrown away information, true, but it is implementation-specific stuff which is likely to confuse and irritate the newcomer.

Such an output filter can be added to `~/.lake/config.lua` or brought explicitly in with `require 'cpp-error'` and becomes available to all of your C++ projects.

Currently, only one such filter can be in-place for a given language object. (Well, maybe two; but the `CL` compiler output has to be filtered for dependency information to be extracted. `lake.output_filter` is a bonus that came from that basic functionality.)

Look at the example lakefile in `examples/errors` for a version that handles both compilers. This organizes the filter as an installable plugin which can be fetched remotely with:

```
$ lake -install get:cpp.filter.lua
```

In that more elaborate C++ example, I get 125 lines of raw output using mingw, which is filtered to 9 much shorter lines.

Adding a New Language

Lake mostly knows about C/C++ and has a fair bit of insider knowledge about the GCC and MSVC compilers. It is naturally easier to add a new compiler if it follows the same pattern.

- there are separate compile and link steps
- the link step takes the intermediate *object* files and combines them into a program or shared library, and finds external symbols in libraries.
- such external libraries are specified by optional library search paths and are included one by one.

The Lake way of defining language objects is higher-level than defining the compile rules directly and can be very straightforward. Consider `luac` which compiles Lua source files into bytecode files with extension `.luac`.

```
lua = {ext='.lua',obj_ext='.luac'}
lua.compile = 'luac -o $(TARGET) $(INPUT)'
lake.add_group(lua)
```

I can now compile a group of Lua files like so:

```
lc = lua.group{src='test/*'}
default(lc)
```

(Note that `group` returns a *rule* populated with targets, not a target itself. So for this to work properly you need to make a target that depends on this group of targets - hence `default`)

So at a miminum, Lake needs to know the input and output extensions and the command for converting the input into the output - which is precisely what defines a rule. But doing it this way makes some standard features automatically available, like specifying `odir` for the output directory, `exclude` to exclude files from `src` and `recurse` to find files in a directory tree.

A common strategy with new (or specialized) languages is to use C as an intermediate 'high-level assembler'. Say we have a new language T, and it compiles to C.

```
T = {ext='.t',obj_ext='.c'}
T.compile = 'tc $(INPUT)'
lake.add_group(T)

tr = T.group{src='*'}
c.program{'first',src=tr:get_targets(),libs='T'}
```

Here the output of the group - which is a rule with C targets - is fed as input into the C program step.

So `T.program` would look like this:

```
function T.program(args)
    local tr = T.group(args)
    args.src = tr:get_targets()
    args.libs = choose(args.libs,args.libs..' T','T')
    return c.program(args)
end
```

Java, like Lua, lacks an explicit link step, but requires a classpath to be set for resolving symbols at compile time. `javac` will also by default generate class files in the same directory as the source file. It is a good idea to try to compile as many source files at once, since `javac` is slow to get started.

```
java = {ext='.java', obj_ext = '.class'}
java.output_in_same_dir = true
java.compile = 'javac $(CFLAGS) $(INPUT)'
java.compile_combine = java.compile
```

`compile_combine` indicates to Lake that the compiler can accept multiple source files, and also what command to use. In this case `INPUT` becomes a space-separated list of input files.

The standard `group` function is not quite right, so `java.group` is extended to do some custom preprocessing of options and pass them as `args.flags`; this option will set `CFLAGS` in the compile command. Also, Lake is strict about checking program/group option flags, so it must be told about new options.

```
lake.add_group(java)
local java_group = java.group

function java.group(args)
  local flags=''
  if args.classpath then
```

```
    libs = args.classpath
    libs = deps_arg(libs)
    if libs[1] ~= '.' then table.insert(libs,1,'.') end
    flags = '-classpath "'..table.concat(libs,';')..'"'
  end
  if args.version_source then
    flags = flags..' -source '..args.version_source
  end
  if args.version_target then
    flags = flags..' -target '..args.version_target
  end
  args.flags = flags
  return java_group(args)
end

lake.add_program_option 'classpath version_source version_target'
```

And then things work as expected:

```
corba = java.group{src = 'org/csir/azisa/corba/*', classpath='libs',recurse=true}
```

The closest equivalent to linking for Java would be building a jarfile, which is fairly straightforward to express as well - the involved bit is setting the main class in a manifest for an executable jarfile.

Lake has provision for extensions which add new languages. Once a language is registered, it is directly available from lakefiles, plus it can choose to register its source extension so that `lake file.ext` will work as expected.

A language which does not fit the usual compile-link C/C++ pattern is C#. Multiple source files are compiled together into a single program or library.

```
clr = {ext = '.cs',obj_ext='.?'}
clr.link = '$(CSC) -nologo  $(LIBS) -out:$(TARGET) $(SRC)'
-- do this because the extensions are the same on Unix
clr.EXE_EXT = '.exe'
clr.DLL_EXT = '.dll'
clr.LINK_DLL = '-t:library'
clr.LIBPOST = '.dll'
clr.DEFDEF = '-d:'
clr.LIBPARM = '-r:'
```

This is a case where the usual platform conventions for program and library names do not apply!

When in doubt, hook into the flags handler. This is only called during the 'link' phase. (We're hijacking the 'link' phase to do the actual compilation - `clr.compile` is not set.)

```
clr.flags_handler = function(self,args,compile)
  local flags
  if args.debug or DEBUG then
    flags = '-debug'
  elseif args.optimize or OPTIMIZE then
    flags = '-optimize'
  end
  local subsystem = args.subsystem
  if subsystem then
    if subsystem == 'windows' then subsystem = 'winexe' end
    flags = flags..' -t:'..subsystem
    -- clear it so that default logic doesn't kick in
    args.subsystem = nil
  end
  ...
```

The next thing comes from the two roles that 'deps' serves: apart from specifying a dependency, it (usually) implicitly provides a library to link against. This is a semi-accidental feature that comes from how C/C++ linkers work. In C# we have to massage any dependencies on other assemblies so that they come out as references:

```
  if args.deps then -- may be passed referenced assemblies as dependencies
     local deps_libs = {}
     for d in list(args.deps) do
        if istarget(d) and d.ptype == 'dll' then
            local target = path.splitext(d.target)
            table.insert(deps_libs,target)
        end
     end
     if #deps_libs > 0 then
        args.libs = args.libs and lake.deps_arg(args.libs) or {}
        list.extend(args.libs,deps_libs)
     end
  end
  return flags
end
```

If 'deps' contains targets which are assemblies, then we add them to 'libs', taking away the original extension because `clr.LIBPOST` will add this again.

To complete the support, we specify how to run the results of a successful compilation, define `clr.program/shared` and register the '.cs' extension.

```
if not WINDOWS then
    clr.runner = function(prog,args)
        exec('mono '..prog..args)
    end
end

lake.add_prog(clr)
lake.add_shared(clr)
lake.register(clr,clr.ext)
```

In 'examples/csharp', this code is found in `clr.lang.lua`. The first part of this file does compiler detection, which is a simple yes/no on Unix - do we have either `gmcs` or `mcs`? On Windows, if `csc.exe` is not on the path, we look in the .NET framework directory and set the appropriate path. By default, it will pick the latest .NET version, but the global DOTNET can be used to specify a version exactly. In this way, a Windows machine can build C# programs as long as it has the framework installed - no SDK is required. (A rare example of Microsoft shipping useful programming tools with their operating systems.)

To install C# support:

```
$ lake -install clr.lang.lua
```

and this file is copied to `./lake/lua/lake/lang/clr.lua` and `require 'lake.lang.clr'` is added to the global configuration file.

You can now run a C# file directly using `lake hello.cs`!

OS X Support

OS X's version of `GCC` (and recently `clang`) has the concept of 'frameworks' which allow the compiler to resolve both include and library paths:

```
$ cat lakefile
c.program{'prog',framework='Carbon OpenGL'}
$ lake
gcc -c -O1 -Wall  -MMD  prog.c
gcc prog.o  -framework Carbon -framework OpenGL -o prog
```

Defining Objective-C as a new language is straightforward. It 'inherits' most behaviour from C, except that the extension is now '.m'. We can hook into the compile and link phases with `flags_handler` - in this case to ensure that the `Foundation` framework is present if not specified.

```
objc = lake.new_lang(c,{ext='.m'})
```

```
objc.flags_handler = function(lang,args,compile)
 if not compile and not args.framework then
       args.framework = 'Foundation'
 end
 return c:flags_handler(args,compile)
end

lake.add_prog(objc)
lake.add_shared(objc)

obj.program{'first',src='main car'}
```

Running Tests

This is an important activity, and it's useful to have some tool support.

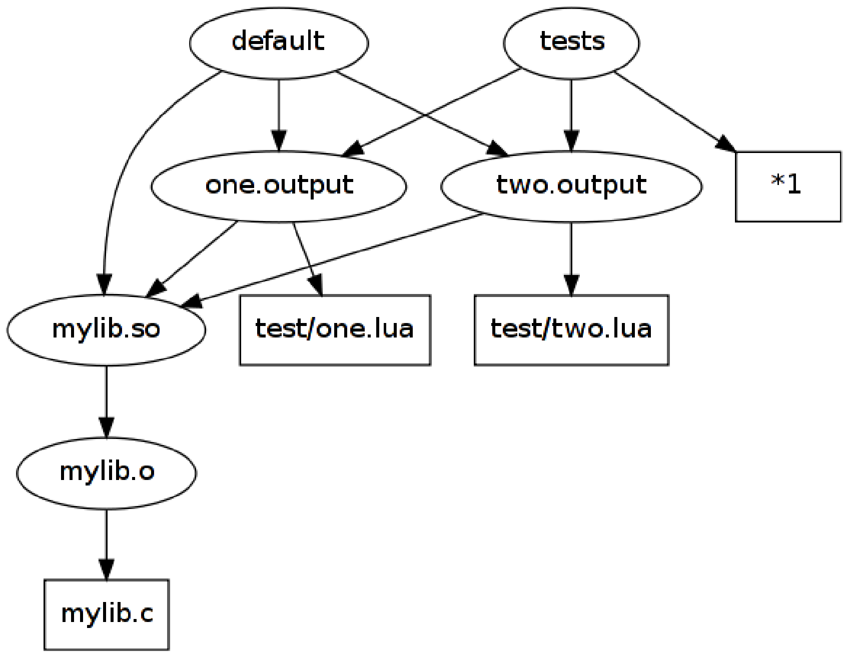
Consider `examples/lua`. We want to run some Lua scripts against the result `mylib`. They must all run if `mylib` changes, and individual tests must run if updated or created. The idea is to construct a rule which makes up a fake target for each test run, and then populate the rule from the `test` directory; this is made explicitly dependent on `mylib`

```
lt = rule('.output','.lua','lua $(INPUT) > $(TARGET)')

lt ('test/*',mylib)

default{mylib,lt}
```

Now, maybe there is also a requirement that tests can always be run directly using `lake tests`. So we have to create a target dependent on the test targets, which first resets the tests by deleting the fake targets:

```
target.tests {
  action(utils.remove, '*.output'),
  lt
 }
```

Depending on an unconditional action does the job. (However, this is not entirely satisfactory, since in an ideal world the order of dependencies being resolved should not matter, but this will do for now.)



From 1.4, there is more direct support. Consider this lakefile, which builds a library consisting of one file, and compiles the single test file twice, once against the DLL and once against the static lib.

```
if PLAT ~= 'Windows' then
    ENV.LD_LIBRARY_PATH='.'
end
dll = c.shared {'lib1'}
lib = c.library {'lib1'}

default {
    c.program{'with_dll',src='needs-lib1',dll}:run(),
    c.program{'with_lib',src='needs-lib1',lib}:run()
}
```

The `run` method of a target generates another target which depends on it. The new target's action is to run the program, if the program has changed.

```
d:\test> lake
gcc -c -O2 -Wall -MMD  needs-lib1.c -o needs-lib1.o
gcc -c -O2 -Wall -MMD  lib1.c -o lib1.o
gcc lib1.o lib1.def  -Wl,-s -shared -o lib1.dll
gcc needs-lib1.o lib1.dll  -Wl,-s -o with_dll.exe
with_dll.exe >with_dll-output
ar rcu liblib1.a lib1.o && ranlib liblib1.a
gcc needs-lib1.o liblib1.a  -Wl,-s -o with_lib.exe
with_lib.exe >with_lib-output
```

Lake as a Lua Library

I have a feeling that there is a small, compact dependencies library buried inside `lake.lua` in the same way that there is a thin athletic person inside every fat couch potato. To do its job without external dependencies, Lake defines a lot of useful functionality which can be used for other purposes. Also, these facilities are very useful within more elaborate lakefiles.

we can load 'lake' as a module. Here `lake.expand_args` is a file grabber which recursively looks into directories, if the third parameter is `true`.

```
$ lua
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> dofile '/path/to/lake'
> t = lake.expand_args('*','.c',true)
> = #t
112
> for i = 1,10 do print(t[i]) end
examples/hello.c
examples/test1/src/test1.c
examples/first/one.c
examples/first/two.c
examples/lib1/needs-lib1.c
examples/lib1/lib1.c
examples/lua/mylib.c
examples/big1/c087.c
examples/big1/c014.c
examples/big1/c007.c
```

Note that all of these libraries are available when a script is invoked with `lake script.lua`.

You may use `lake` as a regular Lua library using `require` if a copy (or preferrably a symlink) called `lake.lua` is on your Lua module path. The default operation using `lakefile` is equivalent to the following script:

```
require 'lake' --> assuming it's on the module path
dofile 'lakefile'
lake.go()
```

By using Lake in this way, you can control when the dependencies are resolved. Have a look at `examples/objects.lua`, which is a script version of `examples/objects/lakefile`. It ends with these lines:

```
print 'go 1'
lake.go()
print 'go 2'
B.time = 11
lake.go()
```

`utils.sleep` is available in a full Lake installation, so you can pause your script. It's possible to recheck on file changes. On Windows, you can use winapi.watch_for_file_changes; there is no POSIX equivalent, but on Linux you can use linotify which is available through LuaRocks.

The `list` table provides some useful functions for operating on array-like tables. It is callable, and acts as an iterator:

```
> for s in list {'one','two','three'} do print(s) end
one
two
three
> -- `list` can also be passed a space-or-comma separated string.
> for s in list 'ein zwei' do print(s) end
ein
zwei
> ls = L{'one',nil,'two "three 3"',{'four','five'}}
> for s in list(ls) do print(s) end
one
two
three 3
four
five
```

The 'list constructor' `L` makes a *flattened* Lua table from a source containing strings, tables or `nil`. It removes the 'holes', expands the strings, and copies the tables. Note that you can double-quote a string with spaces, which can happen if you genuinely cannot avoid such a file path.

There are other useful functions for working with lists and tables:

```
> ls = {1,2}
> list.extend(ls,{3,4})
> utils.forall(ls,print)
1
2
3
4
> = list.index({10,20,30},20)
2
> t = {ONE=1}
> utils.update(t,{TWO=2,THREE=3})
> for k,v in pairs(t) do print(k,v) end
THREE   3
TWO     2
ONE     1
```

There are cross-platform functions for doing common things with paths and files

```
> = file.temp()
/tmp/lua_KZSFkZ
> f = file.temp_copy 'hello dolly\n'
> = f
/tmp/lua_07J5r8
> file.read(f)
hello dolly
```

These work as expected on the other side of the fence (please note that `os.tmpname()` is *not* safe on Windows since it doesn't prepend the temp directory!).

```
> = path.expanduser '~/.lake'
C:\Documents and Settings\SJDonova/.lake
> = file.temp()
C:\DOCUME~1\SJDonova\LOCALS~1\Temp\s3uk.
> = utils.which 'ls'
d:\utils\bin\ls.exe
....
> = path.expanduser '~/.lake'
/home/steve/.lake
> = path.join('bonzo','dog','.txt')
bonzo/dog.txt
> = path.basename 'billy.boy'
billy.boy
> = path.extension_of 'billy.boy'
.boy
> =  path.basename '/tmp/billy.boy'
billy.boy
> = path.replace_extension('billy.boy','.girl')
billy.girl
> for d in path.dirs '.' do print(d) end
./doc
./examples
```

There is a subsitution function which replaces any global variables, unless they are in an exclusion list:

```
> FRED = 'ok'
> = utils.subst('$(FRED) $(DEBUG)')
ok
> = utils.subst('$(FRED) $(DEBUG)',{DEBUG=true})
ok $(DEBUG)
```

Much of Lake's magic is done using this very useful function. It's used to expand compile strings while still leaving some parameters for later expansion.

Future Directions

Naturally, this is not a new idea in the Lua universe. PrimeMover is similar in concept. There are a number of Lua-to-makefile generators, like premake and hamster - the former can also generate SCons output.

`PrimeMover` can operate as a completely self-contained package, with embedded Lua interpreter. This would be a useful thing to emulate.

There is a need for a compact dependency-driven programming framework in Lua; see for instance this stackoverflow question. A refactoring of Lake would make it easier to include only this functionality as a library. The general cross-platform utilities could be extracted and perhaps contribute to a proposed project for a general scripting support library.

lake`has got too large to be a single-file script, and modularization will make it easier to maintain. My initial feeling was to make Lake as easy as possible to install, but this is not really a very strong argument for bad practice, particularly as tools like`squish`and`soar` make generating standalone archives containing many Lua modules.

There are some common patterns which are not supported, for instance *installation* and *running tests*. The former is awkward to do well in a cross-platform way, but the latter is definitely a good candidate. As Lake becomes more modular, it becomes easier to write extensions, rather than burdening the core with every possible scenario.