Fubsy Documentation

Release 0.0.1

Greg Ward

January 06, 2013

CONTENTS

CHAPTER

ONE

USER GUIDE

1.1 Introduction

Fubsy is a tool for efficiently building software. Roughly speaking, Fubsy lets you (re)build target files from source files with minimal effort based on which source files have changed. More generically, Fubsy is an engine for the *conditional execution* of *actions* based on the *dependencies* between a collection of related *resources*.

Let's unpack that generic description and see how it relates to a concrete example. Typically, resources are files: source code that you maintain plus output files created by compilers, linkers, packagers, etc. For example, consider a simple C project that consists of four source files:

```
mytool.c
util.c
util.h
README.txt
```

Initially, your goal is simply to build the executable `mytool` by compiling `mytool.c` and `util.c`, and then linking the two object files together. More importantly, you want to perform the minimum necessary work whenever source files change: if you modify `mytool.c`, then recompile `mytool.o` and relink the executable. But if you modify `util.h`, you may need to recompile both `util.o` and `main.o` before relinking. This is exactly the sort of problem that Fubsy is designed for.

(At this point, outraged Windows programmers might point out that they build `mytool.obj` and `mytool.exe` This platform variation is a quirk of C/C++ that Fubsy's C/C++ plugins handle, but which core Fubsy knows nothing about.)

1.1.1 Disclaimer

Note: Currently, this document is more of a specification than a description of actual software. Many of the features described here have barely even been thought through, never mind implemented and tested. Mentally add a "not implemented yet" footnote to every sentence in this guide, and you won't be too far off from the truth. I've tried to help by adding explicit "this actually works" and "not implemented yet" notes here and there, but don't be surprised if Fubsy doesn't behave quite as the document promises.

Furthermore, it's quite likely that the final product will differ considerably from the description in this document; that's just the nature of software. Consider this an invitation to join in the development of Fubsy and influence how it will turn out.

1.1.2 Similar tools

Of course, Fubsy is hardly the first piece of software that attempts to tackle this problem. Most C programmers are familiar with Make, which does a reasonable job for small-to-medium C/C++ projects on Unix-like systems. However, Make has awkward syntax, confusing semantics, is hard to extend, and is largely limited to Unix. These limitations have led many people over the years to paper over its difficulties by writing programs that generate Makefiles, without actually tackling the real problems with Make.

Similarly, most Java programmers are familiar with Ant, which attempts to solve the problem in a radically different way. Ant doesn't provide much in the way of dependency management (which is surprisingly difficult for Java) [1], but it is extensible in a real programming language (Java). As a result, it works the same across platforms, which is more than Make can say. Unfortunately, Ant takes "awkward syntax" to a whole new level by using XML rather than a custom language. And it's useless for programmers outside the Java ecosystem.

Some C/C++ programmers are familiar with SCons, which brought a new level of rigour, consistency, and extensibility to build tools. SCons puts the graph of dependencies front and centre. It requires developers to get their dependencies right, guaranteeing a correct build in exchange for the effort. Additionally, SCons ships with excellent support for C and C++ which makes many build scripts trivial. Unfortunately, SCons suffers from poor performance, and its dependency engine is incapable of handling weird languages like Java where target filenames are not easily predicted from source filenames.

Fubsy finally promises to be the build tool you've wanted all along. Like Make, Fubsy has a simple custom language designed specifically for writing build scripts, which makes most build scripts quite concise. Unlike Make, Fubsy uses a familiar syntax, has local variables, and distinguishes strings from lists. Like Ant, Fubsy has a small core with most interesting stuff happening in plugins. Unlike Ant, plugins are trivial to implement: you can write small "inline" plugins right in your build script for simple cases, and you can extend Fubsy in any high-level language that it supports: e.g. Python, Lua, Ruby, JavaScript, ... as long as someone has implemented a Fubsy "meta-plugin" for a given language, you can implement plugins in that language. Finally, like SCons, Fubsy puts the graph of dependencies in the foreground. But unlike SCons, Fubsy has minimal runtime overhead, and allows you to modify the graph of dependencies even while the build is running.

1.2 A simple C example

Enough with the vague promises; let's see some code.

1.2.1 C the naive way

Note: As of Fubsy 0.0.1, this mostly works! (Fubsy currently has no memory of previous builds, so it always thinks that all files have changed.)

Here is a naive build script for that simple C project above:

```
# the NAIVE WAY to build a C program; in reality, you should
# use the 'c' plugin!
main {
    CC = "/usr/bin/gcc"
    headers = <*.h>
    source = <*.c>
```

[1] Terminology note: most Java programmers understand "dependency" in the sense of "my application depends on `commons-lang.jar`", so "dependency management" in the Java world typically means "figure out a way to get `commons-lang.jar` into the build environment". C/C++ programmers, however, usually speak of dependencies at the file level: "`foo.c` includes `util.h`, so `foo.o` depends on `util.h`". That is, if `util.h` changes, we need to rebuild `foo.o`. In Fubsy, "dependency" takes the C/C++ programmer's meaning: for example, `MyApp.class` depends on `MyApp.java` as well as `commons-lang.jar`.

```
    # rebuild mytool when any source or header file changes
    "mytool": headers + source {
        "$CC -o $TARGET $source"
    }
}
```

The first thing you notice is that all the code is in the *main* phase. A Fubsy script can contain multiple phases, corresponding to different phases in Fubsy's execution. The only phase that must be present in every build script is *main*, whose purpose is to describe the graph of dependencies that drives everything Fubsy does. We'll see the other phases in a little while.

Next we see some variable assignments:

```
CC = "/usr/bin/gcc"
headers = <*.h>
source = <*.c>
```

Since finding files is very common in build scripts, Fubsy has special syntax for it: angle brackets <> contain a space-separated list of wildcards. (The wildcard syntax is the same as Ant's, e.g. `<**/*.c>` finds all `*.c` files in your tree, including the top directory.)

Fubsy wildcards are evaluated as late as possible. At this point, `headers` simply contains a reference to a "filefinder" object that will expand `*.h` when needed. Also, wildcard expansion uses both the filesystem and the dependency graph. If you have a build rule somewhere in your script that generates a new `*.h` file, the expansion of `<*.h>` will include it.

Finally, the whole point of a build tool is to build something, which you do in Fubsy with *build rules* like

```
"mytool": headers + source {
    "cc -o $TARGET $source"
}
```

The generic syntax for a build rule is

```
TARGETS : SOURCES {
    ACTIONS
}
```

which means that `TARGETS` depend on `SOURCES`, and can be rebuilt by executing `ACTIONS`. `TARGETS` and `SOURCES` can each take on various forms:

- bare string (presumed to be a filename)
- list of strings (presumed filenames)
- filefinder object, e.g. `<*.c>` (effectively a lazy list of filenames)
- node object (for resources other than files)
- list of node objects
- variable referencing any of the above
- concatenation of any of the above (hence `headers + source`)

`ACTIONS` is a newline-separated list of actions, which can be any of:

- string containing a shell command
- function call (e.g. `remove(FILE)`)
- local variable assignment

All actions happen later, during the *build* phase. Calling `remove()` in a build rule doesn't remove anything while the *main* phase is running, it just tells Fubsy to call `remove()` later, when executing the actions in this build rule. However, if you call `remove()` outside of a build rule, it will go ahead and remove the specified files when the *main* phase is running–probably not what you want.

In any event, Fubsy only executes the actions in a build rule when it determines that at least one target is out-of-date, i.e. any of the source files have changed since the targets were last built.

You're probably wondering why that shell command uses uppercase `$TARGET` but lowercase `$source`. `$source` is easy: it's just a reference to the variable `source` defined earlier in the *main* phase. If we had instead called that variable `files`, then the command would use `$files`. `$TARGET` is special: it expands to the build rule's first target file. Other special variables that are only available in build rule actions are `$TARGETS` (all targets), `$SOURCE`, and `$SOURCES`. We don't use `$SOURCES` in this case because it includes `*.h` as well as `*.c`, and you don't pass header files to the C compiler.

So what's wrong with this example? Why is this the naive way to build C programs with Fubsy? There are several problems:

- it's not portable: `mytool` is the wrong filename on Windows, and `cc` is a Unix convention
- it won't scale: for a 3-file project, it's no big deal to recompile the world on every change. But if you have 300 source files, then this build script will cause Fubsy to recompile all of them every time you change one of them. Not good. You want an *incremental build*, where Fubsy rebuilds the bare minimum based on your actual source dependencies and which files have changed.

Incidentally, this build script isn't really *wrong*, as long as you only care about building on Unix. It will do the job, and it illustrates an important feature of Fubsy: you can throw together a quick and dirty build script that gets the job done with simple core features. The vast majority of `Makefiles` ever written are quick and dirty hacks, and Fubsy aims to provide the same relaxed, do-whatever-it-takes experience for those use cases. But when your build script needs to grow up and get professional, Fubsy's plugin architecture and default plugins will make life much easier than it ever was with Make.

So what is the right way to build a C program with Fubsy?

1.2.2 C the right way

Note: Not implemented yet. First we need to figure out the architecture for plugins, then start implementing useful plugins.

The right way is to use Fubsy's builtin plugin for analyzing, compiling, and linking C libraries and programs, unsurprisingly called `c`. Here's the complete build script:

```
import c

main {
    c.binary("myapp", <*.c>)
}
```

`c.binary()` is a *builder*, a function that defines build rules. In this case, the rule is "build binary executable `myapp` from `*.c`". There's a lot going on behind the scenes here.

- `"myapp"` isn't a filename, it's the name of a binary executable. On Unix, it expands to filename `myapp`, on Windows to `myapp.exe`. Similar tricks apply to object files (`foo.o` vs. `foo.obj`), static libraries (`libfoo.a` vs. `foo.lib`), and shared libraries (`libfoo.so` on Linux, `libfoo.dylib` on OS X, `foo.dll` on Windows).

- There are actually multiple build rules defined here: for example, one to compile `myapp.c` to `myapp.o`, another to compile `util.c` to `util.o`, and a third to link the two object files together.
- The build rules respect header file dependencies: the `c` plugin actually reads your `*.c` source files to find who includes which header files. For example, if `myapp.c` includes `<util.h>`, then Fubsy will ensure that `myapp.o` depends on `util.h`. You don't have to do anything; Fubsy just automatically takes care of C (and C++) header dependencies for you. Note that this is a feature of the C/C++ plugins, and other language plugins might not be as clever. For example, determining compile-time dependencies for Java is surprisingly difficult, so the Java plugin takes a completely different approach to dependency analysis.

In case you're wondering, Fubsy also has excellent built-in C++ support, but the plugin is called `cxx`. More details later.

1.3 Phases

I mentioned above that a Fubsy build script can contain multiple phases, corresponding to the phases of Fubsy's own execution. Those phases are:

options add command-line options and variables that the user can pass to `fubsy` (executes very early, so Fubsy can tell if the command line is valid without churning through lots of expensive dependency analysis)

configure examine the build system to figure out which compilers, tools, libraries, etc. are present in order to influence later phases

main specifies the resources (files) involved in your build, constructing the graph of dependencies that will drive everything

build follow the graph of dependencies to rebuild out-of-date files (i.e. conditionally execute actions based on dependencies between related resources)

clean remove some or all build products (typically used in a separate invocation of `fubsy`: running *build* and *clean* in the same invocation would be pointless)

All phases except `main` are optional; a build script with no `main` phase would have an empty dependency graph, so nothing to build.

Currently, Fubsy always runs the *main* phase to define the graph of dependencies, followed by the *build* phase to walk the graph and build stale or missing targets. When *options* is implemented, it will always run first, since *main* will depend on user-defined command-line options and variables.

The other phases depend on user actions. For example, the *clean* phase will run if and only if the user executes

```
fubsy clean
```

The *configure* phase will run if the user executes

```
fubsy configure
```

However, there will probably be circumstances under which Fubsy runs *configure* automatically, e.g. in a fresh working dir that has never been configured. This is all to be sorted out in the future.

Note: So far, only *main* and *build* are implemented. The *main* phase must be explicitly provided in every build script, and the *build* phase is implicit. It's unclear what it would mean if a build script provided an explicit *build* phase. It's entirely possible that using the same mechanism to describe both explicitly coded phases like *main* and the implicit, behind-the-scenes *build* phase is a bad idea.

1.4 A simple Java example

Since C and Java are two of the most widely-used programming languages in the world, it's surprising that so few build tools even try to support both [2]. So before the Java programmers start to feel left out, let's go through a similar exercise for a simple Java project.

First, here's how the project is laid out:

```
src/
  main/
    com/
      example/
        mylib/*.java
        myapp/*.java
  test/
    com/
      example/
        mylib/*.java
```

(This is a simplified variation on a common Java convention: test code goes in `src/test/`, and production, or non-test, code in `src/main/`.)

The goal is to build all the production code into `example.jar`, then the test code to `example-tests.jar`. Compiling in this order is nice because it means you can't accidentally make production code depend on test code: the build will fail if you do. (Eventually we want to run the tests too, but that'll come later.)

1.4.1 Java the naive way

Note: As of Fubsy 0.0.1, this mostly works! (Fubsy currently has no memory of previous builds, so it always thinks that all files have changed.)

First, here's the naive way to do it, using only core Fubsy features (no plugins):

```
main {
    # assume a Debian-ish system with junit4 installed
    junit = "/usr/share/java/junit4.jar"

    mainsrc = <src/main/**/*.java>
    testsrc = <src/test/**/*.java>
    mainjar = "example.jar"
    testjar = "example-test.jar"

    # recompile all production code and rebuild the production
    # jar file when any production source file changes
    mainjar: mainsrc {
        classdir = "classes/main"
        mkdir(classdir)
        "javac -d $classdir $mainsrc"
        "jar -cf $TARGET -C $classdir ."
        remove(classdir)
    }

    # similar for the test code, but make it depend on the
    # production jar too -- i.e. recompile test code when the
```

[2] In fact, I'm aware of only one other build tool that supports Java and C/C++: Gradle.

```
    # production *bytecode* changes, not necessarily the source
    # code
    testjar: testsrc + mainjar {
        classdir = "classes/test"
        mkdir(classdir)
        "javac -d $classdir -classpath $mainjar:$junit $testsrc"
        "jar -cf $TARGET -C $classdir ."
        remove(classdir)
    }
}
```

Like the naive C example above, this works, but it could be better. Here's how it works: first, we assign four variables with filenames of interest. `mainsrc` and `testsrc` are filefinder objects as above, which use wildcards to find files as late as possible. Note the use of recursive patterns here, since Java programmers are prone to deep package hierarchies. `mainjar` and `testjar` are just string variables, the names of the two files we're building. Fubsy has no idea that these are filenames; they're just strings. It's only once a string is used as a target or source in a build rule that it is interpreted as a filename.

Building the main `example.jar` is straightforward: compile a bunch of `.java` files to `.class` files, then archive those `.class` files into a `.jar` file. We remove the intermediate directory because it's bad form to leave around intermediate results that Fubsy doesn't know about: those `.class` files are not part of Fubsy's dependency graph, so it cannot make any use of them or even clean them up. (Actually, it's bad form to even *create* intermediate files that Fubsy doesn't know about; things work out better when Fubsy knows all of your source and target files. That's tricky to do with Java, though, so we'll hold off on doing it right until we meet the `java` plugin, below.)

Also, using `remove()` illustrates an action that is not a shell command: you can't do this portably (`rm -rf` on Unix, `rmdir /s /q` on Windows), so instead Fubsy provides built-in support for it.

This example demonstrates variables local to a build rule: those two `classdir` variables are in fact distinct and not visible outside of the build rules. They exist only while the actions for each build rule are running. (And, by the way, those actions run later, during the *build* phase. That's the whole point of build rules, after all: to specify actions that might run in the build phase, if at least one source has changed.)

Building `example-test.jar` is a bit more troublesome, and illustrates most of the problems with this naive approach to building Java. For starters, it largely repeats the build rule for `example.jar`, and every programmer should know and respect the DRY principle: Don't Repeat Yourself. Build scripts are programs too, and should follow the same standards as your main code. But repetition is hard to avoid when you're using core Fubsy, since the language omits subroutines, macros, and other constructs seen in "real" programming languages. That's deliberate: all the interesting stuff belongs in plugins, which you can implement in a variety of "real" languages.

Another problem is that the dependency on `example.jar` is expressed twice: first, we have to tell Fubsy that `example-test.jar` depends on `example.jar`, and then we have to tell `javac` the same thing by putting it in the compile-time classpath. That's a Java-specific convention, though, so of course it doesn't belong in core Fubsy. That sort of knowledge belongs in the `java` plugin.

1.4.2 Java the right way

Note: Not implemented yet. First we need to figure out the architecture for plugins, then start implementing useful plugins.

As with C, the right way to build your Java code is to use Fubsy's built-in `java` plugin:

```
import java

main {
    mainjar = "example.jar"
```

```
    testjar = "example-test.jar"

    classdir = "classes/main"
    java.classes(classdir, <src/main/**/*.java>)
    java.jar(mainjar, classdir)

    classdir = "classes/test"
    java.classes(classdir, <src/test/**/*.java>, CLASSPATH=mainjar)
    java.jar(testjar, classdir)
}
```

We're using two builders provided by the `java` plugin: `classes()` and `jar()`. Note that builders are conventionally named after *what* they build, not *how* they build it – hence `classes()` rather than the more obvious `javac()`. This is largely motivated by C/C++: if `c.binary()` was instead named `c.link()`, what would you call the builder that links shared libraries? By using *what* rather than *how*, Fubsy easily distinguishes `c.binary()` from `c.sharedlibrary()`. For consistency, that convention carries over to other plugins. It makes sense even for Java: if you're using `javac` to generate annotations rather than compile to bytecode, it's cleaner to have a separate `annotations()` builder than to abuse a generic `javac()` builder with a clever hack that tricks it into generating annotations.

The second use of `java.classes()` shows our first explicit use of a *build variable*, which is a special type of variable defined by plugins and used by build actions. In this case, rather than having a single value of `CLASSPATH`, we override it for one particular builder (and thus for all build rules defined by that builder). As usual, Fubsy is relaxed about the distinction between lists and atomic values: normally `CLASSPATH` is a list of filenames and directories, but if you just pass a lone filename, that's OK.

1.5 Fubsy, the scripting language

By now you've probably noticed that Fubsy is a simple scripting language as well as a build tool. It has constructs that are familiar from other programming languages, like variables, strings, lists, and function calls. It's missing lots of functionality that you would expect in a general-purpose language, like numbers, arithmetic, loops, and user-defined functions. And it has some features that are vaguely like subroutines, but geared towards the particular needs of a build tool like Fubsy: phases and build rules.

The formal specification for Fubsy's scripting language doesn't exist yet, so here's an informal guide.

1.5.1 Top-level elements

At the top level, a Fubsy script contains three elements: import statements, inline plugins, and phases:

```
import PLUGIN

plugin LANG {{{
    CONTENT
}}}

PHASE {
    STATEMENT
    ...
}
```

There can be any number of each element, and they can be intermixed in any order:

```
import java

plugin python {{{
    def pyhello():
        # println() is a Fubsy function exposed to plugins
        println("hello from a python plugin")
}}}

import c

clean {
    jshello()
    remove("junkfile")
}

plugin javascript {{{
    func jshello() {
        println("hello from a javascript plugin")
    }
}}}

main {
    pyhello()
    c.binary("bin/myapp", <src/*.c>)
    java.classes("classes/foo", <src/foo/**/*.java>)
    touch("junkfile")
}
```

However, good taste suggests that imports should come first, followed by inline plugins, followed by phases in a logical order. Here's an equivalent build script:

```
import java
import c

plugin python {{{
    def pyhello():
        # println() is a Fubsy function exposed to plugins
        println("hello from a python plugin")
}}}

plugin javascript {{{
    func jshello() {
        println("hello from a javascript plugin");
    }
}}}

main {
    pyhello()
    c.binary("bin/myapp", <src/*.c>)
    java.classes("classes/foo", <src/foo/**/*.java>)
    touch("junkfile")
}

clean {
    jshello()
    remove("junkfile")
}
```

Note: Don't expect this to work with Fubsy 0.0.1. The parser supports all of the syntax shown here, but almost none of the required backend code has been implemented yet.

Note: There's no provision for builds with multiple build scripts (aka "hierarchical builds") yet. I'm leaning towards a simple `include FILENAME` syntax (like Make), with automatic reinterpretation of filenames in child scripts (like SCons). Join the fubsydev mailing list if you want to help shape the design of Fubsy!

1.5.2 (Some) whitespace is significant

End-of-line is syntactically significant: it's the delimiter between statements. The only difference between

```
# invalid syntax
import a import b
```

and

```
# valid syntax
import a
import b
```

is the newline after `import a`. The same applies to statements inside phases and build rules.

Less obviously, the relationship between curly braces and newlines is fixed by the grammar:

```
# valid syntax
main {
    a = "a"
}

# invalid syntax
main
{
    a = "a"
}

# invalid syntax
main {
    a = "a" }

# valid syntax (special case for empty phases)
main { }
```

Build rules have similar syntax.

Newline is the *only* type of whitespace that is significant, though: Fubsy does not care how you indent your code. You can use tabs or spaces or both or (shudder) no indentation at all. (But *I* care: please indent with four spaces, so all Fubsy scripts will be consistent.)

Note: I like the idea of consistent style being enforced by the grammar, but clearly I didn't have the guts to go as far as enforcing indentation. If you strongly disagree with this design choice, one way or the other, please join the fubsydev mailing list and discuss!

1.5.3 Imports

Fubsy will support *external* and *inline* plugins. One import statement loads one external plugin using a dot-delimited name:

```
import c
import java.eclipse
import foo.bar.baz.wing.ding
```

The effect of each `import` is to add one name to the local namespace of the current script: in this case, `c`, `eclipse`, and `ding`. Each plugin provides values that you can use in your build script:

```
main {
    c.binary("app", "main.c")
    println(ding.TOOLNAME)
}
```

Precisely what a plugin provides is entirely up to the plugin.

Note: Apart from syntactic support, this is completely unimplemented in Fubsy 0.0.1.

1.5.4 Inline plugins

Fubsy deliberately does not provide general programming features such as numbers, arithmetic, loops, or user-defined functions. That's what inline plugins are for. There are plenty of good high-level general-purpose languages out there already, so it seems silly to design and implement yet another general-purpose language for a special-purpose build tool. (And Fubsy deliberately does not use an existing general-purpose language for its syntax, because that would limit it to fans of that particular language. Fubsy aims to be a *universal* build tool, and inline plugins are a key part of achieving that goal. If you want SCons/Rake/Waf/Gradle, you know where to find them.)

The syntax for an inline plugin is

```
plugin LANGUAGE {{{CONTENT}}}
```

where `LANGUAGE` is a short identifier like "python" or "javascript" and `CONTENT` is any sequence of bytes, except for `}}}`.

The language tells Fubsy how to interpret the content. If you put JavaScript code in a plugin marked `python`, then Fubsy will happily fire up a Python interpreter, ask Python to parse your code, and fail.

Whitespace inside the triple-brace delimiters is ignored and passed verbatim to the plugin interpreter, *except* that common leading whitespace is trimmed. That is, if every line of `CONTENT` starts with (at least) four spaces, then four spaces will be trimmed from `CONTENT` before attempting to parse it. That lets you indent your inline plugin content without angering indentation-sensitive languages like Python.

Functions and values defined by inline plugins will be available to the build script directly. See the example above, under "Top-level elements".

Note: Apart from syntactic support, this is completely unimplemented in Fubsy 0.0.1.

1.5.5 Phases

A phase is just a sequence of statements:

```
NAME {
    STATEMENT
    ...
}
```

where `NAME` is an identifier like `main`, `clean`, `options`, etc.

A statement can be one of the following:

- a variable assignment, like

  ```
  src = <src/main/**/*.java>
  java.JAVAC = "/usr/bin/javac"
  java.CLASSPATH = ["lib/util.jar", "lib/stuff.jar"]
  ```

- an expression, like

  ```
  src.exclude("**/Stub*.java")
  pyhello()
  mkdir(builddir + "/" + "bin")
  ```

- a build rule, like

  ```
  "app.jar": <classes/app/**/*.class> {
      "jar -cf ../../$TARGET -C classes/app .
  }
  ```

Every Fubsy build script must contain a *main* phase, which defines sources and targets and the relationships between them. See *Phases* for more information on the phases that Fubsy will eventually implement and the relationships between them.

1.5.6 Local and global variables

By default, variables are local to the current script, and available to all phases in it:

```
main {
    junkfile = "tmp/junk.dat"
    touch(junkfile)
}

clean {
    remove(junkfile)
}
```

Thus, while phases look like a scoping mechanism, they aren't. They're really a mechanism for specifying what happens at different times in the process of a build. That's why they are *sort of* like subroutines, but not really. (They also don't have parameters or return values, and you don't have much control over when they run.)

Variables can also be defined in a build rule, in which case they are local to that build rule only:

```
main {
    "outfile": "infile" {
        tmpfile = "$TARGET.tmp"
        "./process $SOURCE > $tmpfile"
        rename(tmpfile, TARGET)
    }

    # runtime error: 'tmpfile' not defined
```

```
    println(tmpfile)
}
```

Thus, build rules *are* a scoping mechanism. But they are primarily a means for you to write code that isn't run until the *build* phase, and only runs if any of the rule's targets are stale or missing.

A future version of Fubsy will support hierarchical builds where a top-level build script includes child scripts for building code in subdirectories. When that happens, Fubsy will also grow support for global variables that are visible to all scripts in the same process. Until that point, there's not much point in implementing global variables.

1.5.7 Value expansion

All values in Fubsy – strings, lists, and filefinders – are subject to *expansion*. The precise meaning of expansion varies according to the data type, but in general it means converting a value from the form initially seen in the build script to the form that will be needed in order to actually build targets.

For example, the filefinder value `<*.c>` might expand to a list like

```
["main.c", "util.c", "stuff.c"]
```

and the string

```
"$CC -o $TARGET $sources"
```

might expand to

```
"/usr/bin/cc -o app main.c util.c stuff.c"
```

Expanding a list just means expanding its member values recursively, and flattening the result. For example, the list

```
[<*.c>, "hello $audience", <include/*.h>]
```

consists of three values which might respectively expand to

```
["main.c", "util.c", "stuff.c"]
"hello world"
["include/util.h", "include/stuff.h"]
```

But list expansion results in a flattened value

```
["main.c",
 "util.c",
 "stuff.c",
 "hello world",
 "include/util.h",
 "include/stuff.h"]
```

Fubsy is perfectly capable of representing deeply nested data structures, but it generally flattens lists whenever it can. Fubsy is not a general-purpose programming language, and flat lists tend to be more convenient in build scripts.

In the absence of explicit expansion, by you or by plugin code that you call, values are expanded in the *build* phase. Values that are nodes in the dependency graph (a common use of filefinders) are expanded early in the build phase, when Fubsy converts the initial dependency graph to its final form. Other values (e.g. command strings) are not expanded until right before the command is executed. Consider this build script:

```
main {
    flags = "-O2"
    "myapp": <*.c> {
        flags = "-O0 -Wall"
```

```
        "cc $flags $SOURCES -o $TARGET"
    }
}
```

Expanding command strings at the last possible moment means `$flags` expands to `-O0 -Wall`, as you would expect. It's also essential for automatic variables like `SOURCES` and `TARGET` to work.

1.5.8 Summary

Fubsy's scripting language provides the following familiar features, which should be familiar from most general-purpose programming languages:

- variables
- data types: strings, lists
- expressions, including function calls

The scoping rules for variables are a bit odd:

- most variables are local to current script
- but phases are not scopes: a variable defined in *main* is visible in *build*, *clean*, etc.
- each build rule is a scope and has local variables

Fubsy also has some distinctive features:

- filefinder objects for wildcards (`<src/*.c>`)
- expansion of variables embedded in strings (`"$CC -o $TARGET"`)

These may look familiar from Unix shell programming, but there's a key difference: in Fubsy, wildcards and strings are expanded as late as possible.

Finally, Fubsy deliberately omits a number of features found in any general-purpose programming language:

- numbers
- arithmetic
- loops
- user-defined functions
- logic ("a or b and not c")
- conditionals (if/then/else)

Fubsy is not a general-purpose language. If you need those things, you'll have to write an inline plugin in an existing language (when Fubsy grows support for inline plugins!).

(Actually, I suspect Fubsy will have to provide conditionals and logic eventually. The point of the *options* and *configure* phases will be to make the build vary according to user wishes and the state of the build system. User-defined options won't be very useful if they don't provide a way for you to enable/disable parts of your build, and explicit conditional constructs are the obvious answer there.)

1.6 How it works

Now that we've seen four small but realistic examples, this is a good time to delve into how Fubsy really works: what exactly is going on behind the scenes in these build scripts?

Note: This section is mostly accurate as of Fubsy 0.0.1. Fubsy currently has no memory of previous builds, so it always thinks that all files have changed. That means that incremental builds aren't actually incremental. All of the code *except* for determining what has changed is written and tested, though, so things are looking pretty good here.

1.6.1 The dependency graph

The central data structure that drives everything in Fubsy is the *dependency graph*, which describes how your source and target files are related. The purpose of the *main* phase of a Fubsy build script is to construct the dependency graph by describing the relationships between source and target files. This is your job, hopefully aided by plugins like `c` or `java`.

Let's revisit our naive build script for `myapp.c` and friends. The core of that script was a single build rule:

```
"myapp": ["myapp.c", "util.c", "util.h"] {
    "cc -o $TARGET myapp.c util.c"
}
```

(I've dropped all variables and filefinders here to make things more explicit.)

This build rule specifies a simple dependency graph:

[diagram: myapp depends on myapp.c, util.c, util.h]

The list of actions ("cc ...") is attached to the target file(s). That is, Fubsy knows exactly one way to build every target file in a particular invocation. (The actions can change across runs, e.g. if you override the `CFLAGS` build variable on the command line.) This graph makes visible the weakness of the naive build script: changing *any* source file means recompiling *all* of them.

The smarter build script for `myapp` invokes the `c` plugin:

```
c.binary("myapp", ["myapp.c", "util.c"])
```

which effectively adds several build rules:

```
"myapp": ["myapp.o", "util.o"] {
    "$LINKER -o $TARGET $SOURCES"
}
"myapp.o": ["myapp.c"] {
    "$CC -o $TARGET $SOURCES"
}
"util.o": ["util.c"] {
    "$CC -o $TARGET $SOURCES"
}
depends("myapp.o", "util.h")
depends("util.o", "util.h")
```

You can see here the two types of dependencies: *direct* dependencies spelled out directly in the build rule, like "myapp.o depends on myapp.c"; and *indirect* dependencies added with a call to `depends()` outside of the build rule, like "myapp.o depends on util.h". The reason for the distinction is that we don't want `util.h` to be included in `$SOURCES` in the build rule, but we do want it to affect Fubsy's decisions about what to rebuild.

The dependency graph resulting from this build script is more complex, but will result in a much more scalable build:

[diagram: myapp depends on 2 .o files, which depend on .c and .h]

Everything you do in the *main* phase of your build script is there to construct the dependency graph. Fubsy then uses that data in the *build* phase.

1.6.2 The build phase

You'll notice that we haven't included an explicit *build* phase, like

```
build {
    ...
}
```

in any of our sample scripts. That's because the *build* phase is where Fubsy takes over and conditionally executes the actions it finds in your dependency graph based on the state of the nodes in the graph (source and target files in your working directory).

(At this point, I'm going to stop talking about files and talk about nodes in the dependency graph instead. Nodes are *usually* files, but can be any resource involved in a dependency relationship. For example, unit tests typically don't generate any output: they run and either pass or fail. If a unit test passed in the last build, there's no need to re-run it unless something that it depends on has changed. So it's useful to have a node in your dependency graph that records the successful execution of a unit test. Similar reasoning applies to static analysis tools.)

Here's how it works. First, Fubsy figures out the set of *goal nodes*, i.e. which targets to build. By default, the goal is the set of all *final targets*: targets that are not themselves the source for some later target. Alternately, you can specify which targets to build on the command line–e.g., if you're having trouble compiling `myapp.c` and just want to concentrate on it for the moment:

```
fubsy myapp.o
```

(Incidentally, *source* and *target* are relative terms: `myapp.o` is a target derived from `myapp.c`, but a source for `myapp`. Any node that is both a source and a target is also called an *intermediate target*. Any node that is not built from something else is called an *original source*. Original sources are what you modify and keep in source control; everything else is temporary and disposable. And final targets, as already described, are nodes that are not the source to any other node–typically deployable executables, packages, or installers.)

Let's cook up a slightly more complex example to illustrate: now we're going to build two binaries, `tool1` and `tool2`, from the following source files:

```
tool1.c
tool2.c
util.c + util.h
misc.c + misc.h
```

`tool1` depends on both `util.c` and `misc.c`, but `tool2` depends only on `util.c`. Here is the build script:

```
import c

main {
    c.binary("tool1", ["tool1.c", "util.c", "misc.c"])
    c.binary("tool2", ["tool2.c", "util.c"])
}
```

And here is the dependency graph described by that build script:

> [diagram: tool1 -> tool1.o -> tool1.c, util.h, misc.h tool1 -> util.o -> util.c, util.h tool1 -> misc.o -> misc.c, misc.h tool2 -> tool2.o -> tool2.c, util.h tool2 -> util.o -> util.c, util.h]

Once Fubsy has determined the targets that it's trying to build–the goal nodes–it constructs a second dependency graph containing only the goal nodes and their ancestors. This step is also used to expand any filefinder nodes that have survived this far: e.g. if there is a node like `<src/**/*.java>` in the graph, it is replaced with nodes for every matching file. We'll call this second graph the *build graph*.

Then, Fubsy walks the new dependency graph in *topological order*: that is, if node *B* depends on (is a child of) node *A*, it will visit *A* before visiting *B*. In fact, it will visit all nodes that *B* depends on before visiting *B*. As it visits each node, Fubsy performs the following steps:

1. if the node is an original source node (it depends on nothing else), skip to the next node in topological order
2. if the node is *tainted* because one of its ancestors failed to build, skip to the next node
3. if the node is missing or *stale* (one of its parents has changed since the last build), build it

Once those three tests have been applied to every node in the goal set, then the build is finished. If there were any failures, the whole build is a failure.

1.6.3 Example: initial build

An example should clarify things. Let's continue with the case above, building `tool1` and `tool2`. By default, the goal consists of all final targets. To make things interesting, let's suppose you specify a different goal: `fubsy tool2`, which means the build graph contains only ancestors of `tool2`:

[diagram: same as above, with non-ancestors of tool2 removed]

Let's assume that Fubsy's topological graph walk visits all of the original source nodes first.

[diagram: same as above, with tool2.c, util.c, util.h "skipped"]

When it visits `tool2.o`, Fubsy looks in the filesystem and sees that that node is missing, so builds it:

```
cc -o tool2.o tool2.c
```

Now the graph looks like this:

[diagram: same as above, with tool2.o marked "built"]

Next in line is `util.o`, which is also missing:

```
cc -o util.o util.c
```

Finally we visit and build `tool2`:

```
cc -o tool2 tool2.o util.o
```

We're done; every node in the graph has been visited:

[diagram: same as above, but now util.o and tool2 are "built"]

1.6.4 Example: incremental rebuild

Of course, if all you want to do is build everything, you don't need a fancy build tool like Fubsy. A shell script will work just fine. The real value of Fubsy becomes apparent when you modify your source code. To make things interesting, let's say we've made a real change in `tool.c`, i.e. one that affects the object code. Again, we'll assume the goal node is just `tool2`.

The initial build graph is the same as in the previous example, and the first couple of steps are the same. Things change slightly when Fubsy reaches `tool2.o`: this time the target node exists, but one of its parents (`tool2.c`) has changed since the last build. So Fubsy has to rebuild the target:

```
cc -o tool2.o tool2.c
```

The graph looks the same as it did at this point in the previous example:

[diagram: as above, tool.o marked "built"]

Next we visit `util.o`. But none of its parents have changed, so no rebuild is required.

[diagram: as above, util.o marked "skipped"]

Finally we visit the `tool2` node. One of its parents, `tool2.o`, has changed, so we have to rebuild the final target:

```
cc -o tool2 tool2.o util.o
```

Because none of the ancestors of `util.o` changed, we didn't have to rebuild it, and used the pre-existing version of `util.o` to link `tool2`.

1.6.5 Example: short-circuit rebuild

Now let's say you edit a comment in `util.h`. Assuming this does not affect the object code, this should avoid unnecessary downstream rebuilds: a short-circuit rebuild.

When Fubsy reaches `tool2.o`, it will inspect its parents and realize that `util.h` has changed; likewise for `util.o`. So those two files must be rebuilt:

```
cc -o tool2.o tool2.c
cc -o util.o util.c
```

But because you only changed a comment, the object code in both files is unchanged. So when Fubsy visits `myapp`, none of that node's parents are changed, and it can skip rebuilding. The final graph:

> [diagram: as above, with tool2.o, util.o "built" and tool2 "skipped"]

We've saved the cost of linking one binary. In this trivial example, that's not much. But it can make a difference in larger builds, and Fubsy is designed to scale up to very large builds indeed.

1.7 Indices and tables

- *genindex*
- *search*