Getting Started with Jam/MR – A Tutorial

Laura Wingerd
Perforce Software
Perforce User Conference 2001

Abstract

Jam/MR ("Jam – make(1) *Redux") is a software build tool. Its command language and processing logic are enough unlike* make *that new users often have trouble learning how to use it to its best advantage. In this talk, I'll demonstrate some of Jam's most useful features in a series of progressively interesting examples. The basic knowledge gleaned from these examples will give you the insight you need to incorporate very sophisticated Jam features into a powerful build system. You'll find it helpful to have the Jam documentation at hand as you follow along.*

1 Background

Jam was written by Christopher Seiwald. It has been freely available as C++ source for many years. It is widely used to build commercial and academic software, and has evolved into a few variants.[1] It's even shipped with the Macintosh OS/X operating system. Nevertheless, it hasn't supplanted *make*, and the O'Reilly book on Jam has yet to be written.

Jam's distinguishing strengths are its portability and speed. Its portability comes from the way the Jam command language segregates platform-dependent specifications, like compile and link commands, from platform-independent specifications, like dependencies. (More on this in a bit.) Its speed comes in part from its single-invocation processing logic. The Jam executable is invoked once -- unlike *make*, which must invoke itself recursively -- to gather, analyze, and act on its input.

2 How Jam Works

Jam is a stand-alone executable program. When you invoke it, it does three things. First, it reads in "Jamfiles" (akin to "Makefiles") containing your instructions. Next, it scans directories (and sometimes files) looking for "targets" defined by your instructions; a Jam "target" is a file that is used, created or updated during a build. Finally, it executes system commands to create or update targets. These three phases of Jam processing are called parsing, binding, and updating. At the end of this presentation you'll understand why it's important to know about the three phases.

2.1 Running Jam

If you have a "jam" executable available, you can try any of the Jam examples shown here yourself. Simply put the example text in a file and run:

```
jam -fyourfile
```

(Normally you'd put your Jam commands in a file called "Jamfile". That's the file Jam looks for by default when you don't use the "-f" flag. However, by default, Jam also invokes a setup file called the Jambase, which would add unecessary complexity to the examples here.)

You can use a number of Jam command line flags to show revealing diagnostic information:

`-d2` Diagnostic output level two: shows the OS commands used to create or update files being built.

[1] The official Jam/MR source is available from the Perforce Public Depot at www.perforce.com.

`-d5` Diagnostic output level five: shows you how your Jamfiles are being interpreted (which procedures are being run, which variables are being set, etc.). Try using this flag with the simple examples discussed below.

`-n` Run without actually executing any update commands.

`-a` Rebuild all targets, whether they need it or not.

You can combine these flags, e.g.:

```
jam -nd5a -fyourfile
```

3 The Jam Language

Before we get started with example Jamfiles, let's take a look at some interesting aspects of Jam's simple yet unintuitive language.

3.1 Syntax

The Jam language has a very straightforward syntax. Even so, it manages to confound new users. The most misunderstood rules of the language are:

- Case is significant.
- Statement elements (called "tokens") *must* be separated by whitespace.
- Every statement ends with a semicolon – in its own token!

Here are some examples:

```
X = foo.c ;
```

This is a single Jam statement that sets the value of "X" to "foo.c".

```
X = foo.c ; x = bar.c ;
```

This pair of statements sets two Jam variables. First it sets a variable named "X" to "foo.c", then it sets a variable named "x" to "bar.c".

```
X = foo.c;
```

This is an incomplete statement that will cause a syntax error, or worse, unexplainable results. When Jam reads this it will set the value of X to "foo.c;" plus whatever happens to be on the next line. Why? Because there is no whitespace before the semicolon that is meant to end the statement.

Each statement in the Jam language either sets a variable or invokes a "rule". (A "rule" is basically a procedure.) You can tell whether a statement is setting a variable or invoking a rule by the presence of an equal sign. For example, all the statements above set variables. Here are some statements that invoke rules:

```
X foo.c ;
```

This invokes a rule called "X" and passes it one argument, "foo.c".

```
X=foo.c ;
```

This is a perfectly acceptable Jam statement, but it doesn't do what you think it's going to do. This statement invokes a rule named "X=foo.c" with no arguments. Why is this not treated as an assignment statement? Because there is no whitespace delimiting the "=" sign.

Actually, Jam recognizes a third type of statement as well. Statements that begin with certain keywords are "flow-of-control" statements. These keywords are fairly easy to recognize; they include "if", "for", "switch", and "include".

Flow-of-control statements and additional Jam language syntax will be introduced in later examples. By the end of this presentation you'll be familiar enough with how Jam works that any syntax or functionality not shown here will make perfect sense to you when you read about it in the Jam documentation.

3.2 Literals

Literals don't have to be quoted in the Jam language. Anything that is not a variable name, a rule name, or an operator is assumed to be a literal. And every literal is a string. (There is no other data type in Jam!) For example:

```
X = foo.c ;
foo.c = X ;
```

This assigns the literal value "foo.c" to a variable named X, then assigns the literal value "X" to a variable named "foo.c".

Quotes *are* necessary, however, to refer to a literal that contains spaces, "=", or other characters that Jam would interpret differently. For example:

```
X = "this ; and ; this" ;
```

This assigns the string value "this ; and ; this" to variable X.

3.3 Variables

Jam variable values are strings. A single Jam variable can hold a list of values, each value being a string:

```
X = a b 1 "2 3" ;
```

Assigns a list of string values, "a", "b", "1" and "2 3" to variable X.

Jam variable names are also strings. You can name a variable almost anything you want:

```
"My dog has fleas!" = yes ;
```

The name *of the variable set here is "My dog has fleas!" and its* value *is a single-element list containing the string "yes".*

You can also set more than one variable at once:

```
My dog has fleas! = yes ;
```

The one-element list containing the string "yes" is assigned to variables named "My", "dog", "has" and "fleas!".

3.3.1 Variable Expansion

Referring to a variable's value is called *expansion* in Jam. In its simplest form, you use "$(name)" to expand a variable. For example:

```
X = This is a message. ;
Echo $(X) ;
```

Invokes Jam's built-in "Echo" rule to output the list of strings assigned to X. In other words, this outputs: This is a message.

You can use variable expansion on the left-hand side of an assignment statement as well:

```
X = Hello ;
$(X) = Bye ;
```

Assigns "Hello" to X, then assigns "Bye" to a variable named Hello.

When a variable list contains more than one item, its expansion results in a list:

```
X = A B C ;
$(X) = Hi there ;
```

Assigns strings "A", "B", and "C" to variable X. Then assigns the list of strings "Hi" and "there" to each of variables A, B, and C.

You can use a "subscript" to refer to specific list items, with the syntax "$(name[subscript])":

```
X = A B C ;
Echo $(X[2]) ;
```

Outputs "B".

You can add elements to a list with the "+=" operator:

```
X = A B C ;
X += $(X) ;
```

Now X contains this list: A B C A B C

3.3.2 Variable Expansion Products

When variable expansions are combined with each other or with literals in a single Jam token they are expanded to a *product*. That product is itself a list. Here are some examples:

```
X = A B C ;
Y = E F ;
Echo $(X)$(Y) ;
```

Outputs: AE AF BE BF CE CF

```
X = A B C ;
```

```
Y = test_$(X).result ;
```

Y now contains this list of strings: " test_A.result", " test_B.result", and "test_C.result"

```
X = A B C D E F G H ;
Selected = 3 7 8 ;
Echo $(X[$(Selected)]) ;
```

Outputs: C G H

Remember that a Jam "token" is a statement element delimited by whitespace. To identify a single token that contains a blank, use quotes. Compare these two examples:

```
X = Bob Sue Pat ;
Echo "Hello $(X)!" ;
```

Outputs: Hello Bob! Hello Sue! Hello Pat!

```
X = Bob Sue Pat ;
Echo Hello $(X)! ;
```

Outputs: Hello Bob! Sue! Pat!

When a token in a Jam statement contains a reference to a variable whose value list is empty, the expanded result is an empty list. (Think of it as multiplying by zero.) A variable that has not been set is the same as a variable whose value list is empty. You can also explicitly set a variable to an empty list. Note that an empty list is not the same as a list of one or more null strings. Here are some examples:

```
X = Bob Sue Pat ;
Echo Hello $(X)$(Y) ;
```

Y has not been set, so this outputs simply: Hello

```
X = Bob Sue Pat ;
Y = "" "" ;
Echo Hello $(X)$(Y) ;
```

Y is set to a list of two null strings, so this outputs: Hello Bob Sue Pat Bob Sue Pat

```
Y Z = test ;
X = Bob Sue Pat ;
Y = ;
Z = $(X)$(Y) ;
```

Because Y was unset, the (X)(Y) product is an empty list. As a result, this explicitly unsets Z.

3.3.3 Variable Expansion Modifiers

"Modifiers" in Jam variable expansions can be used to change the resulting values. Modifier syntax in variable expansion is "$(name:modifier)". For example, you can use the "U" and "L" modifiers to force the case of the expanded result:

```
X = This is ;
Y = A TEST ;
Echo $(X:U) $(Y:L) ;
```

Outputs: THIS IS a test

Most Jam modifiers are specifically designed for handling file names and directory paths. Of those, some trim down the expanded result, and some replace parts of the expanded result. For example, the “S=suffix” modifier can be used to replace the filename suffix:

```
X = foo.c ;
Y = $(X:S=.obj) ;
```

Assigns the value “foo.obj” to Y.

You can combine modifiers with each other, with list item subscripts, and with product expansions. Here’s a hideous example:

```
X = foo.c bar.c ola.c ;
Y = .c .obj .exe .dll ;
Echo $(X[2]:S=$(Y):U) ;
```

Outputs: BAR.C BAR.OBJ BAR.EXE BAR.DLL

3.3.4 Variables Are Expanded During Parsing!

Remember the three phases of Jam execution? Variable expansion in your Jamfiles occurs *during the parsing phase*, before Jam scans your filesystem, and before it executes *any* system commands. This means that you can’t assign the output of system commands (e.g. “ls”, or “find”) to Jam variables!

3.4 Rules

Jam “rules” are procedures that are interpreted and executed during the parsing phase. They can be invoked with arguments passed to them. Each argument is a list; arguments are separated by colon tokens. For example:

```
Depends a : b c d ;
```

Invokes the built-in “Depends” rule with two arguments. The first argument is a list of one item, “a”. The second argument is a list of three items, “b”, “c”, and “d”.

Here’s an example of how a rule is defined:

```
rule MyRule
{
   Echo First arg is $(1) ;
   Echo Second arg is $(2) ;
   Echo Third arg is $(3) ;
}
```

And here’s how you might invoke it:

```
MyRule a : b c : d e f ;
```

Outputs:
First arg is a
Second arg is b c
Third arg is d e f

For backward compatibility, $(<) and $(>) are allowed in place of $(1) and $(2) in the body of a rule. In older Jamfiles you may see rule definitions that look like:

```
rule MyRule
{
   Echo First arg is $(<) ;
   Echo Second arg is $(>) ;
}
```

3.5 Actions

An "action" in Jam is a special-purpose rule used to specify system commands that will be run during Jam's updating phase. The "actions" keyword identifies an action definition. The body of an action definition contains system commands, not Jam language statements. However, it *can* contain Jam variables. Here's a simple action definition:

```
actions MyAction
{
      touch $(1)
      cat $(2) >> $(1)
}
```

If this action were invoked thus:

```
MyAction ola : foo bar ;
```

this command sequence would be passed to the OS command shell to update "ola":

```
touch ola
cat foo bar >> ola
```

Actions have these special characteristics which set them apart from rules:

- They accept only two arguments. In other words, you can refer to $(1) and $(2), but not $(3) in the body of an action.
- All arguments passed to an action are assumed to be targets. (See below.)
- Whereas rules are run during Jam's parsing phase, actions are run during its updating phase. Jam variables in an action body are expanded before the action is passed to the OS command shell.

3.6 Targets and Dependencies

When you run Jam, it assumes you want to build one ore more targets. I said earlier that a Jam "target" is a filesystem object, like a file, a directory, or a library member. Jam also recognizes "symbolic targets", which can be used to organize dependencies. Jam provides one built-in symbolic target called "all". If you don't specify a target on the Jam command line, Jam will try to build "all". This is best explained with a demonstration. Put the following commands in a file called "test":

```
rule MyRule
{
   TouchFile $(1) ;
}

actions TouchFile
{
   touch $(1)
}

MyRule test.output1 ;
MyRule test.output2 ;
MyRule test.output3 ;
```

Now run:

```
jam -ftest
```

You'll see that this outputs:

```
don't know how to make all
...found 1 target(s)...
...can't find 1 target(s)...
```

To tell Jam what you really want to build you could specify a target on the command line:

```
jam -ftest test.output2
```

This outputs:

```
...found 1 target(s)...
...updating 1 target(s)...
TouchFile test.output2
...updated 1 target(s)...
```

But the efficient way to tell Jam what to update is to use the built-in "Depends" rule to make all your targets dependencies of "all":

```
rule MyRule
{
   TouchFile $(1) ;
   Depends all : $(1) ;
}

actions TouchFile
{
   touch $(1)
}

MyRule test.output1 ;
MyRule test.output2 ;
MyRule test.output3 ;
```

Now you can build your files without having to specify any targets on the comnmand line:

```
jam -ftest
```

Since one target was built in the previous test, so only two remain to be built. Here's what Jam outputs:

```
...found 4 target(s)...
...updating 2 target(s)...
TouchFile test.output1
TouchFile test.output3
...updated 2 target(s)...
```

Jam also takes it upon itself to tell you when it's building something that isn't in the chain of dependencies. For example, modify your test file so it looks like this:

```
rule MyRule
{
   TouchFile $(1) ;
   Depends all : $(1) ;
}

actions TouchFile
{
   touch $(1)
}

MyRule test.output1 test.output2 test.output3 ;
```

Now run this command to try to rebuild only one of the three files (the "-a" tells Jam to rebuild it even if it already exists):

```
jam -ftest -a test.output2
```

The output shows you the additional targets Jam had to build even though they were not in the dependency chain for your requested target:

```
...found 1 target(s)...
...updating 1 target(s)...
warning: using independent target test.output1
warning: using independent target test.output3
TouchFile test.output1 test.output2 test.output3
...updated 1 target(s)...
```

3.7 Implicity Invoked Actions

When an action and a rule have the same name, Jam implicitly invokes the action with the same arguments that were used in the rule invocation. Here's an example of an implicitly invoked action:

```
rule MyRule
{
   Depends all : $(1) ;
}

actions MyRule
{
   p4 info > $(1)
}
```

```
MyRule info.output ;
```

A single statement invokes "MyRule". The “MyRule” rule will be run during the parsing phase, and the “MyRule” action will be run during the updating phase. Both are passed the same argument, “info.output”.

3.8 Target-specific Variables

Another syntax for setting Jam variables allows you to set values specific to individual targets. Values set this way are expanded only in actions. The syntax is "variable on target =". For example:

```
X on foo = A B C ;
X on bar  = 1 2 3 ;
```

The usefulness of this can be demonstrated by a working example:

```
rule MyRule
{
    CMD on $(1) = $(2) ;
    PORT on $(1) = $(3) ;
    Depends all : $(1) ;
    MyTest $(1) ;
}

actions MyTest
{
    p4 -p$(PORT) $(CMD) > $(1)
}

MyRule test1.output : info ;
MyRule test2.output : info : mars:1666 ;
MyRule test3.output : users : mars:1666 ;
```

Run Jam with this input and you get:

```
...found 4 target(s)...
...updating 3 target(s)...
MyTest test1.output

    p4  info  > test1.output

MyTest test2.output

    p4 -pmars:1666  info  > test2.output

MyTest test3.output

    p4 -pmars:1666  users  > test3.output

...updated 3 target(s)...
```

In this demonstration, two variables, CMD and PORT, were set on each target. The same action updates each target, but when the action body is expanded, the resulting command is different for each.

4 Working Example: Jam as A Test Driver

To build on the Jam language and behavior presented so far, I've put together a series of working Jamfiles implementing a simple test driver. Each example refines the previous one to demonstrate various Jam strengths. You can run these examples yourself if you have a working "jam" and a working "p4".

Note that this sequence of examples has nothing to do with traditional compile-and-link builds. I've chosen these examples because they are small and self-contained. Once you've studied these examples and have an understanding of how Jam works you'll be ready to look at the compile-and-link rules in the Jam-provided Jambase. These are the rules you'd use to implement large build systems. However, they are far too intricate (and boring) to use in a learning example.

4.1 Simple Command Tester

The first example shows a very simple Jamfile that tests "p4" commands. It merely runs each test and captures the output in a file. For each test, the test result file is the target to be updated, and the action that updates it is the "p4" command to be tested.

This example demonstrates how Jam behaves when actions fail. When Jam passes an action to the OS to update a file, it checks the result of the final command. If that result indicates that the command failed, Jam removes the file that was just updated. (This is how Jam normally behaves; it's not anything coded in this particular example.) Thus, the only result files left by this first test driver example will be those left by *successful* tests. With respect to what a test driver should do, that's good, because rerunning Jam will only rerun the tests that fail. But it's also bad, because it leaves no trace of failed tests!

This example also introduces the Jam "local" declaration. It is used to restrict the scope of a variable to the rule in which is declared and any rule invoked from it.

```
rule Test
{
   local f = $(1:S=.out) ;
   Depends all : $(f) ;
   RunTest $(f) ;
   CMD on $(f) = $(1) ;
}

actions RunTest
{
   p4 $(CMD) > $(1)
}

Test info ;
Test users ;
Test clients ;
```

4.2 Capturing Failed Commands

A test driver isn't much use if it doesn't show you the output of failed tests. In this example, the previous version has been modified to capture the error message from the "p4" command being tested. This example introduces the "ignored" action modifier. (See the Jam documentation for other action modifiers.) "Ignored" changes Jam's behavior when an action fails: instead of removing the target file, Jam leaves it, and continues to build any targets dependent on it.

Each test now produces a result file whether or not the test succeeded. However, there's no way to tell whether a test succeeded other than by looking at the result file. Furthermore, once a result file exists, Jam thinks that test is done; rerunning Jam no longer reruns the failed tests.

```
rule Test
{
   local f = $(1:S=.out) ;
   Depends all : $(f) ;
   RunTest $(f) ;
   CMD on $(f) = $(1) ;
}

actions ignore RunTest
{
   p4 $(CMD) > $(1) 2>&1
}

Test info ;
Test clients ;
Test users ;
```

4.3 Comparing Canonical Results

A more useful test driver compares test results to previously stored results, called "canons". Our example has been enhanced to run each test, diff the test output with the canon, and output the diff to a "match" file. After the test run, the presence of match files shows the tester which tests succeeded, and "jam -nd1" shows which tests failed.

Notice the dependencies in this example. "All" is dependent on the match files, and each match file is dependent on its corresponding test result file, which is in turn dependent on its corresponding canon file. With this dependency chain, only the tests with canons will be run, and of those, only the ones that have not previously produced match files will be run.

Also, take a close look at the targets passed to the actions. The match file is a target of both "RunTest", which creates the result file, and "DiffResults", which compares the result to the canon. If a match file is missing, Jam will invoke both actions to create it. The first of the two actions, however, doesn't create it at all -- it only creates the result file. This is a trick to make Jam rerun the test if the match file missing. The "DiffResults" action, on the other hand, is invoked with only the match file as the target. If the diff fails, only the match file will be removed; the result file will remain for the tester to examine.

```
rule Test
{
   local f = $(1:S=.out) ;
   CMD on $(f) = $(1) ;

   local canon = $(1:S=.canon) ;
   local match = $(1:S=.match) ;

   Depends all : $(match) ;
   Depends $(match) : $(f) ;
   Depends $(f) : $(canon) ;

   RunTest $(f) $(match) ;
   DiffResults $(match) : $(f) $(canon) ;
}

actions ignore RunTest
{
   p4 $(CMD) > $(1) 2>&1
}

actions DiffResults
{
   diff $(2) > $(1) 2>&1
}

Test info ;
Test clients ;
Test users ;
```

4.4 Capturing Canonical Results

Another enhancement has been added to the test driver: it can now create or update canon files. This is optional behavior, triggered by the presence of a variable named "CAPTURE". Since nothing in the Jamfile sets CAPTURE, it must be set externally before running Jam. You can set CAPTURE in your environment, or you can simply set it on the Jam command line:

```
jam -sCAPTURE=1 …
```

When CAPTURE is set, Jam follows a completely different logic with different dependencies. "All" now depends on canon files, and if any of those are missing they are copied from corresponding result files. Missing result files are created by simply running the tests.

(This is an admittedly complex alternative to simply writing a script that copies result files into canon files, but it serves to illustrate two things, setting Jam variables externally, and conditional logic.)

```
rule Test
{
   if $(CAPTURE)
   {
      CaptureCanon $(1) ;
   }
   else
   {
      DoTest $(1) ;
   }
}

rule CaptureCanon
{
   local canon  = $(1:S=.canon) ;
   local result = $(1:S=.out) ;
   CMD on $(result) = $(1) ;

   Depends all : $(canon) ;
   Depends $(canon) : $(result) ;

   RunTest $(result) ;
   CopyResult $(canon) : $(result) ;
}

actions CopyResult
{
   cp $(>) $(<)
}

rule DoTest
{
   local f = $(1:S=.out) ;
   CMD on $(f) = $(1) ;

   local canon = $(1:S=.canon) ;
   local match = $(1:S=.match) ;

   Depends all : $(match) ;
   Depends $(match) : $(f) ;
   Depends $(f) : $(canon) ;

   RunTest $(f) $(match) ;
   DiffResults $(match) : $(f) $(canon) ;
}

actions ignore RunTest
{
   p4 $(CMD) > $(1) 2>&1
}

actions DiffResults
{
   diff $(2) > $(1) 2>&1
}

Test info ;
Test clients ;
Test users ;
```

4.5 Removing Old Results and Canons

Now I've enhanced the example to demonstrate a "Clean" rule. This is used to removed built targets from the filesystem. I've added two invocations of the Clean rule, one to clean up result and match files, one to clean up canons. Note the symbolic target, "clean". Nothing depends or is dependent on "clean", thus the only way to activate the Clean action is to use "clean" as an explicit target on the Jam command line. In other words, run:

```
jam clean
```

to remove the result and match files, or:

```
jam -sCAPTURE=1 clean
```

to remove the canon files.

The Clean action has a number of behavior modifiers:

- *ignore* you've seen before. It doesn't really have much purpose in this context other than to suppress "failed to build" messages if file permissions prevent their removal.
- *together* tells Jam this action need only be run once, and that the $(1) that appears in the action definition body can be expanded to all the targets it was invoked with. (Without *together* an

action would be run once per invocation. The end result is the same, but for certain actions, like running a compiler, *together* can make builds much more efficient.)

- *existing* tells Jam to not include any targets not already present in the filesystem when it expands $(1) in the action body.
- *piecemeal* tells Jam that if the size of the expanded action gets too large for the OS command shell, divide the targets up into groups and run the action on each group separately.

```
rule Test
{
   if $(CAPTURE)
   {
      CaptureCanon $(1) ;
   }
   else
   {
      DoTest $(1) ;
   }
}

rule CaptureCanon
{
   local canon  = $(1:S=.canon) ;
   local result = $(1:S=.out) ;
   CMD on $(result) = $(1) ;

   Depends all : $(canon) ;
   Depends $(canon) : $(result) ;

   RunTest $(result) ;
   CopyResult $(canon) : $(result) ;
   Clean clean : $(canon) ;
}

actions CopyResult
{
   cp $(>) $(<)
}

rule DoTest
{
   local f = $(1:S=.out) ;
   CMD on $(f) = $(1) ;

   local canon = $(1:S=.canon) ;
   local match = $(1:S=.match) ;

   Depends all : $(match) ;
   Depends $(match) : $(f) ;
   Depends $(f) : $(canon) ;

   RunTest $(f) $(match) ;
   DiffResults $(match) : $(f) $(canon) ;
   Clean clean : $(f) $(match) ;
}

actions piecemeal together existing Clean
```

```
{
   rm $(2)
}

actions ignore RunTest
{
   p4 $(CMD) > $(1) 2>&1
}

actions DiffResults
{
   diff $(2) > $(1) 2>&1
}

Test info ;
Test clients ;
Test users ;
```

4.6 Writing Portable Actions

The final refinement is this series of examples illustrates how the same action invocations can be used to run commands that vary among OS platforms. Before defining my actions, I've set some variables to OS-specific command names. In the action definitions I use those variables instead of hard-coded commands. In some cases the commands needed to perform equivalent actions will differ so much between platforms that simply substituting command names won't work. For those cases you can actually define actions differently for each platform, as demonstrated here.

```
if $(NT)
{
   REMOVE  = del/f/q ;
   COPY    = copy ;

   actions DiffResults { echo n|comp  $(2) > $(1) 2>&1 }
}

if $(UNIX)
{
   REMOVE  = rm ;
   COPY    = cp ;

   actions DiffResults { diff $(2) > $(1) 2>&1 }
}

rule Test
{
   if $(CAPTURE)
   {
      CaptureCanon $(1) ;
   }
   else
   {
      DoTest $(1) ;
   }
}
```

```
rule CaptureCanon
{
   local canon  = $(1:S=.canon) ;
   local result = $(1:S=.out) ;
   CMD on $(result) = $(1) ;

   Depends all : $(canon) ;
   Depends $(canon) : $(result) ;

   RunTest $(result) ;
   CopyResult $(canon) : $(result) ;
   Clean clean : $(canon) ;
}

actions CopyResult
{
   $(COPY) $(>) $(<)
}

rule DoTest
{
   local f = $(1:S=.out) ;
   CMD on $(f) = $(1) ;

   local canon = $(1:S=.canon) ;
   local match = $(1:S=.match) ;

   Depends all : $(match) ;
   Depends $(match) : $(f) ;
   Depends $(f) : $(canon) ;

   RunTest $(f) $(match) ;
   DiffResults $(match) : $(f) $(canon) ;
   Clean clean : $(f) $(match) ;
}

actions piecemeal together existing Clean
{
   $(REMOVE) $(2)
}

actions ignore RunTest
{
   p4 $(CMD) > $(1) 2>&1
}

Test info ;
Test clients ;
Test users ;
```

5 Conclusion

Although I certainly have not showed you everything Jam can do, by now you probably have a clear enough understanding that you can fill in the blanks by reading the Jam documentation. I'd recommend reading through all of the documents that come with the Jam source:

- *Jam/MR - Make(1) Redux* - describes the Jam language and executable command flags.
- *Using Jamfiles and Jambase* - a chatty overview of how to use the already-written rules that come with the Jam source. (These are the rules you'd use to implement a system that builds objects by compiling, archiving, linking, etc. These rules also provide methods of managing source and generated files in a directory hierarchy.)
- *Jambase Reference* - a terse reference to those rules.

I'd also recommend reading through the Jambase source file itself. It's a little long, but demonstrates a number of techniques you can use in your own Jamfiles.