tensile

A Tool for Literate Programming

Main Program

Taylor Venable
taylor@metasyntax.net

December 30, 2016

Abstract

Documentation for the Tensile literate programming tool.

Contents

Part I
User's Guide

1 Introduction

Tensile is a tool for literate programming. It was inspired by Noweb, a simple and language-agnostic implementation of Donald Knuth's original ideas. Tensile attempts to go beyond Noweb by providing a easily pluggable architecture where external tools can easily tie into the simple native processing capabilities provided by the Tensile core. More on this in a bit. The name "Tensile" was chosen because it is a measure of material strength, viz. how much a material can be stretched before it fails. In particular, spider webs have very high tensile strength and can support incredible loads compared to the cross-section of their strands. This is the connection that makes it a good name choice for a literate programming tool, a nod to Donald Knuth's original `WEB` tool.

Literate programming is a way of writing a computer program so that the documentation is embedded alongside the actual program code. This is a different approach from simply writing code comments, because there is a greater emphasis on the documentation, and writing it in such a way as it clearly explains not only the mechanism of the code, but also the rationale. The objective is to include all facets necessary to completely describe the program, both for humans and for machines, in the same file. From this file then, we can either "tangle" it, producing the program code, or "weave" it to produce the fully typeset documentation.

2 Tensile Syntax

The basic syntax of Tensile is derived from that of Noweb, but contains extensions. Source files are read from top to bottom, and contain a mix of documentation and source code; one could say that when reading the file, Tensile is either reading in "documentation mode" or "source mode." Tensile starts off in documentation mode at the start of the file. When it encounters a line that starts with double left angle-brackets, and ends with double right angle-brackets and an equal sign (i.e. <<⟨*unit-name*⟩>>=, for any *unit-name*) then Tensile enters source mode. While in source mode, every line that is read is considered source code for the program module designated *unit-name*. To get back to writing documentation, insert a solitary at-sign in the line at the first column. The next line will begin documentation again. When a solitary at-sign is encountered in the middle of documentation, it is ignored (it will not appear in the output).

While in source mode, references to other program modules can occur. These start with double left angle-brackets in the first column, and end with double right angle-brackets at the end of the

line (i.e. <<⟨*unit-name*⟩>>). When tangled, the definition of the program module *unit-name* will be inserted at the point it is referenced in another module. In the woven output, however, you will only see a reference to some code to be given later. This splits up the code into smaller units so you can focus on specific elements in turn in your documentation.

When defining program modules, you can use the same name more than once. Modules with the same name are concatenated together in the tangled output in the order they were encountered in the Tensile source. In the woven output they will appear in the order given with any intervening information that appeared in the source, and the second and later definitions are indicated as additions to the existing module definition. The special module name "…" is reserved as a way of indicating that the previously defined module, whatever it was, should be extended. This prevents you from having to type the name again, reducing typing errors.

2.1 Documentation Chunk Threads

As I learnt more about literate programming and began to apply it at work, a friend and colleague asked how to generate API documentation from literate source code. Previously we had been using Doxygen to generate HTML output to describe our functions, especially in Java and Lua code. Thus the problem arose of how to separate the internal documentation from the external documentation. From my research it seems historically that any such distinction (if it was ever implemented, which seems to have been rare) would be up to the person writing the documentation to keep them apart and only typeset what was needed. However, we saw value in keeping both the internal implementation and external interface near each other for the sake of making things easier to understand for engineers working on the internals. Thus, we designed a way to apply a field called a "thread" to each documentation chunk. Threads make it easy to follow a single audience or topic (a "thread" of discussion, if you will)[1] through the document. Or instead you might choose to build a complete document that weaves all threads together into a single analysis. Additionally, and the true objective of using this system, is to establish a single common thread that can be woven through several literate source files, which can then be bundled up and placed into a single document which covers that idea.

There is a single hidden thread which binds all pieces of documentation together: in our code we call it `*` but you should not refer to it by name ever. This thread is in all documentation chunks, even when not referenced directly, and even when no threads at all are identified. Otherwise, to use any thread of your own making, use the following to start a documentation chunk: `@|`⟨*thread-name*⟩`|`. You may include as many thread names, separated by vertical bars, as you like. The effect of such a thread declaration persists until another documentation chunk starts. Not supplying any thread names at all (i.e. just using the standard documentation chunk syntax) has the effect of cancelling any threads currently in play for the upcoming chunk.

```
@|one|two|
This text is in threads one and two, as well as in thread *.
@|one|
This text is only in thread one, as well as in thread *.
@
This text is only in thread *.
```

The final example maintains backward compatibility with Noweb, in addition to just being good sense. To select threads when weaving, just use the `-thread` command line switch.

[1] The original idea floated by my friend was to use the term "strand" which got me thinking about threads, but ultimately the word "thread" had a lot more punning potential and that's why I chose it.

2.2 Source Output: Non-Stop Mode

I have found it occasionally useful, when working on Java code especially, to write at the same location in the literate source code both the declaration and initial definition of class or instance variables. With Tensile certainly this can be done by building up separate units interleaved with one-another, for example:

Example Literate Source

```
public class Foo {
  <<Variables>>
  public Foo() {
    <<Initialize Variables>>
  }
}
@
Here, $i$ represents some important value.
<<Variables>>=
private int i;
<<Initialize Variables>>=
this.i = 42;
@
The variable $j$ is significantly less important.
<<Variables>>=
private int j;
<<Initialize Variables>>=
this.j = 0;
```

This is most useful for reading the code for understanding, but less so when reading for correctness. The woven output of this is exactly as the input: both the *Variables* and the *Initialize Variables* units are split into two segments of output. It becomes more difficult, I find, to correlate these two separate fields in the mind than it is when you simply see them written out fully separately, especially when you're already used to doing so in the normal layout of plain machine source code. Therefore, to enhance the ability to judge correctness in situations like this, we provide a command which looks very similar to the reference command, which prints the source code in what I call "non-stop mode"; it appears in the input like this: <<⟨unit-name⟩>>* and when woven, produces the entire output of the unit without breaks. Given the previous input, using this:

```
<<Variables>>*
<<Initialize Variables>>*
```

produces output like this:

⟨*Variables*⟩≡

```
  private int i;
  private int j;
```

⟨*Initialize Variables*⟩≡

```
  this.i = 42;
  this.j = 0;
```

In this way, a large amount of code can be built up a piece at a time, and then later printed in a manner that can be easily verified. Note that <<⟨unit-name⟩>>* only has meaning inside a documentation chunk, and when it appears inside a source code chunk it has no special meaning (viz. in Noweb-compatible mode it means the definition of *unit-name* followed by a star, and in strict mode means quite literally what it says).

3 Program Options

By default, Tensile will weave a single input file, writing the output to a file with the same name as the input file, but with the TEX file extension `.tex` instead of the extension (if any) used by the input file. It is recommended that input files use the extension `.tnsl` as convention. To disable the automatic weaving, use either the `-no-docs` or the `-dont-weave` flag.

To tangle source code, provide the name of the unit to tangle with the `-R` option, like so: `tensile -Runit input.tnsl`. Alternatively, the `-extract-all` option tangles all toplevel units found in the input file. Tangled output is written to files with the names specified as the unit name.

3.1 Complete Option List

3.1.1 Standard Options

`-help`	Show a help message.

3.1.2 File Handling Options

`-indented-refs`	Allow references to be indented in source.
`-list-tops`	Print all toplevel units and quit.
`-noweb-compat`	Enable Noweb-compatible parsing.
`-show-tops`	Same as `-list-tops`.
`-write-ir`	Write intermediate form to file.

3.1.3 Tangled Output Options

`-extract-all`	Extract all toplevel units.
`-tangle-to ⟨file⟩`	Write the tangled source to *file*. Only works if only one unit is extracted; if more than one unit is tangled then this option is ignored.
`-unit ⟨name⟩`	Tangle unit *name*.

3.1.4 Woven Output Options

`-dont-weave`	Don't produce woven documentation output.
`-hide-margin-tags`	Don't display definition tag number in the margin.
`-hide-defn-page`	Don't show references to first definition.
`-hide-back-refs`	Don't print references to usage location.
`-hide-source-code`	Don't output source code in documentation.
`-no-docs`	Same as `-dont-weave`.
`-thread ⟨thread⟩`	Only weave output for doc chunks in thread *name*.
`-weave-to ⟨file⟩`	Write result to *file*.

3.1.5 Deprecated Options

The following options are available in Noweb and are also honored by Tensile, although I discourage their use.

`-R⟨name⟩`	Same as `-unit ⟨name⟩`. **Clarification:** there is no space between the `-R` and the unit name; example `-RHelloWorld.java`.
`-o ⟨file⟩`	Same as `-weave-to ⟨file⟩`.

4 Hooks

Tensile provides extensibility by facilitating the usage of "hooks" throughout the process of transforming literate code into machine source code and documentation. To set these, simply assign the correct field in the global `hooks` object in your `.tensilerc` file. Some examples can be found in the `hooks.tnsl` file.

4.1 Tangling Hooks

There are currently no hooks available when tangling source code.

4.2 Weaving Hooks

There are several hooks that can be used to modify how TEX code is output when generating documentation.

`hook.doc.text`
: This function takes as its only argument the documentation chunk read by Tensile; what it returns will be used as the output for that documentation chunk.

`hook.doc.preExpand`
: The only argument to this function is the source code chunk before unsafe TEX characters (such as "\" and "{") are turned into their TEX equivalents. What it returns will be expanded unless `g_opts.expand_unsafe_tex` is true. In certain situations, it may be useful to set the expansion option to false from within this function to prevent expansion.

`hook.doc.postExpand`
: Takes a single argument, which is the source text after the unsafe TEX characters in it have been expanded into their TEX equivalents. The return value will be sent directly to the output stream.

5 Intermediate Representation

To facilitate easier addition of new features at various points in the process of operating on literate source, Tensile uses an intermediate representation of the structure of the document after its syntax has been parsed. This representation tracks all source code chunks, the documentation blocks, and all references between the two. Because Lua has a prototype-based object system, the intermediate representation is completely open-ended in what it can support. A hypothetical extension could process this representation and add new fields based on the literate file, then utilize these as part of some output extension.

The specification for the base intermediate representation is an object with three fields: `src`, `doc`, and `ref`.

5.1 Source Code

The `src` field is itself an object. Keys in the object are the names of the modules as defined in the literate file. The values are arrays that contain information on that module in sequence. When tangling, these are processed in order to create the source output.

Each entry in the array that comprises the module definition is an object. Regardless of function, each has a `type` field which indicates its purpose. The three defined types are:

`code` Defines the actual source code to be used. Source code chunks have these fields:

- `text` Contains the source text. For both the tangled and woven outputs, the source text is printed. No facility is currently provided for pretty printing.

`ref` Identifies a reference to another source code chunk. A reference chunk also contains the following fields:

- `name` Indicates the name of that chunk. When tangling, these references are followed when encountered, and the referenced module is printed immediately. For woven output, the name of the reference is displayed in angle brackets.
- `indent` The level of indentation of the contents of the referenced source code unit. This is used when operating in Noweb-compatible mode.
- `followed` When true, indicates that the referenced code unit is followed on the same line by additional code inside the referring definition. This is used to decide when to insert line breaks in the tangled and woven output.

`break` Indicates that the definition was broken up into multiple units in the literate source. A `break` entry has no influence on the tangled output, but it does affect weaving by halting source text output when encountered.

5.1.1 References in Noweb-Compat Mode

When using Noweb-compatibility mode you can have references occur at any point in the definition, not just at the start of the line like you get in strict mode. Here's a good example from a Scheme program:

```
(cond <<beta>>)
<<beta>>=
((integer? n) "integer")
(else "something else")
```

This generates intermediate representation that looks like this:

```
  src = {
    ["alpha"] = {
      {
        ["text"] = "(cond ",
        ["type"] = "code",
      },
      {
        ["indent"] = "6",
        ["type"] = "ref",
        ["name"] = "beta",
        ["followed"] = true,
      },
      {
        ["text"] = ")\
",
        ["type"] = "code",
      },
    },
    ["beta"] = {
      {
        ["text"] = "((integer? n) \"integer\")\
(else \"something else\")\
",
        ["type"] = "code",
      },
    },
  }
```

Reading this, we see there is a unit called *alpha* which contains a reference to *beta* which will be indented 6 spaces (the length of the string which precedes the reference), and which is followed by

a closing parenthesis. Looking at the definition of *beta* we see that it is completely normal, nothing about where it is located appears; this is necessary because it is possible to refer to *beta* from another location in another unit with a different indentation or perhaps no indentation at all. Thus, all the logic of indentation and handling line endings must be carried out by the code which processes the reference. The output of this particular fragment is, of course:

```
(cond ((integer? n) "integer")
      (else "something else"))
```

Astute readers will notice that all definitions which end at the line include the newline in their `text` field. This becomes troublesome when outputting the source: without special handling we end up with too many newlines injected in the output, possibly changing the meaning of the code. This is the scenario that necessitates the use of the `followed` field. When `followed` is true then we suppress the output of the newline at the end of the source text, because there's going to be more code immediately following it in the parent.

Risking further diversion, we must realize however that simply checking the `followed` field is enough to determine if a newline should be written after the unit's source code; we also need to check and see if it is the last element of the parent unit or simply one of a number of units. A simple example is:

```
<<alpha>>=
  alpha
  <<beta>>
  <<delta>>
<<beta>>=
beta
<<gamma>>
<<delta>>=
delta
<<gamma>>=
gamma
```

Obviously the expected output of this is:

```
  alpha
  beta
  gamma
  delta
```

Using only the rule about `followed` we would end up with an extra space between `gamma` and `delta`: the unit *gamma* is not followed, so one newline is inserted (perfectly normal), but *beta* is not followed either so another newline is inserted as well. This leaves an empty line between `gamma` and `delta`: not what we wanted or expected. Thus, we only add newlines at the end of references thus inserted when the unit is both not followed, and last in the list of all units. For this example, *gamma* is the last element in *beta*, so a newline is inserted. However, *beta* is not the last element in *alpha*, so a newline is suppressed there. The result is a single newline between the text `gamma` and the text `delta`: exactly as we expected.

5.2 Documentation

The `doc` field is an array of objects. Each object represents a piece of documentation found in that order in the literate source code. Documentation is not emitted during tangling, but is of course fundamental to weaving. Each object has a `type` field with one of these values:

`text` This element represents a documentation chunk, in TEX code. The following fields must also be present within the object.

`text` Contains the documentation text.

`tags` A list of tags that apply to this documentation chunk; these are used to categorize the documentation chunks.

`def` This element represents the definition of a program module, the code of which will be printed in the woven output. The `name` field indicates the name of the program module, which is looked up from the `src` part of the intermediate representation (covered above). If a field called `start` is present, this indicates the index in the program module entry list where woven output generation starts. This is to facilitate the fact that a single program module can be broken up. The index will indicate the element immediately following a `break` element in the program module definition list.

`def*` This is much like `def` above, but it indicates that the output is to occur in "non-stop" mode, which means that neither breaks nor the starting index are honored: the entire source code comprising the unit is output at once. This is trigged by the use of the `<<⟨unit-name⟩>>*` syntax.

The part about how definitions are broken bears repeating, and an example. If a program module is split up into separate locations in the literate source, it will contain a `break` element in the `src` part of the intermediate representation, and the matching documentation reference will contain a `start` field indicating the index of that element following that break in the program module definition array. Thus, if our literate code contains this:

```
Documentation of alpha.
<<example>>=
alpha
@
Documentation of beta.
<<example>>=
beta
```

then the intermediate representation will look like this:

```
{
  src = {
    ["example"] = {
      { type = "code", text = "alpha" },
      { type = "break" },
      { type = "code", text = "beta" }
    }
  },
  doc = {
    { type = "text", text = "Documentation of alpha." },
    { type = "def", name = "example" },
    { type = "text", text = "Documentation of beta." },
    { type = "def", name = "example", start = 3 }
  }
}
```

Part II
Implementation

10a ⟨*tensile* 10a⟩≡
```
#!/usr/bin/env lua

⟨Gather CVS Information 10b⟩
⟨Generate Intermediate Representation 10c⟩
⟨Write Intermediate Representation 14b⟩
⟨Read Intermediate Representation 15b⟩
⟨Tangle — Create Source 15c⟩
⟨Weave — Create Documentation 17a⟩
⟨Find Toplevel Units 20a⟩
⟨Program Initialization 20b⟩
⟨Option Processing 20c⟩
```

10b ⟨*Gather CVS Information* 10b⟩≡ (10a)
```
version = {}
do
    local rev = "$Revision$"
    version.revision = rev:gsub("%$[^:]+: ", ""):sub(1, -3)
    local date = "$Date$"
    version.date = date:gsub("%$[^:]+: ", ""):sub(1, -3)
end
```

6 Generating Intermediate Representation

Our first function generates the Lua table to represent the literate code.

6.1 Noweb Compatibility

Tensile aims to provide a mode which is compatible with Noweb 2.11b, that which inspired Tensile. There are several differences between what Noweb accepts and what Tensile provides in its default strict mode:

- Noweb allows you to embed code references in the middle of lines which then expand inline. Tensile forces code references to be on their own line. The former is more flexible, unless you're using C++ or doing a lot of bit shifting, in which case it may be better to have that restriction.
- Noweb treats an at-sign in column zero as a documentation block marker, whereas Tensile requires it to be the only thing on the line. There isn't really much pro or con to this, unless you want to use perhaps a row of at-signs as a visual indicator in the file.

10c ⟨*Generate Intermediate Representation* 10c⟩≡ (10a)
```
function generateIR(file)
    local state = "doc"
    local code = nil
    local doc = nil
    local ir = {src = {}, doc = {}, ref = {}}
    local unit = {}
    local unitName = ""
    local threads = nil
    ⟨Create Unit IR Object 11a⟩
    ⟨Create Documentation IR Object 11b⟩
    ⟨Flush Documentation to IR Structure 11c⟩
    ⟨Flush Code to IR Structure 11d⟩
```

This helper function shall be called when we encounter a new program unit definition in the code. If the name is not "…" then we are either reusing an old unit with the same name (in which case we should find it) or we are creating a new unit (in which case we should create it). If there have already been code entries given for that unit, we will insert a break marker before the location where this new code will be entered. This allows the TEX output system to distinguish between the first part of a module definition and any subsequent parts.

11a ⟨*Create Unit IR Object* 11a⟩≡ (10c)
```
local function defineUnit(name)
    if name ~= "..." then
        unitName = name
        ir.src[unitName] = ir.src[unitName] or {}
        unit = ir.src[unitName]
    end
    if #unit ~= 0 then
        unit[#unit + 1] = { type = "break" }
    end
end
```

A bit of a misnomer, this function creates an entry into the documentation part of the intermediate representation that a code definition was given. This is called when we first encounter the beginning of a definition. First we set the properties that all code definitions will have, viz. the type (being "def") and the name of the unit. If the unit was already partially defined and the last item in the definition was a break entry, we indicate in the documentation that it should start listing code at the line that we are about to read, which will become the next entry in the source list of the unit.

11b ⟨*Create Documentation IR Object* 11b⟩≡ (10c)
```
local function createDoc(name)
    ir.doc[#ir.doc + 1] = { type = "def", name = unitName }
    if #unit > 0 and unit[#unit].type == "break" then
        ir.doc[#ir.doc].start = #unit + 1
    end
end
```

When we transition from one state to another, we'll need to flush to the table the documentation or code fragment that we've been reading.

11c ⟨*Flush Documentation to IR Structure* 11c⟩≡ (10c)
```
local function flushDoc()
    if doc then
        ir.doc[#ir.doc + 1] = { type = "text", text = doc }
        if threads and #threads > 0 then
            ir.doc[#ir.doc].threads = threads
        end
        doc = nil
    end
end
```

11d ⟨*Flush Code to IR Structure* 11d⟩≡ (10c)
```
local function flushCode()
    if code then unit[#unit + 1] = { type = "code", text = code } ; code = nil end
end
```

6.2 Parsing the Input File

To begin the actual work, we need only iterate through all the lines in the input file. Tensile works on a single pass to produce an intermediate representation, which is then used for both tangling and weaving; this allows fairly easy extension by manipulation of this intermediate representation. Note that we do nothing TEX-specific in the reading phase, and thus any kind of documentation markup that the user may desire can be used. (Although of course, TEX is the best!)

There are two parser states, “doc” which indicates that the parser is reading documentation, and “code” which indicates that we are processing a code unit definition.

12a ⟨*Generate Intermediate Representation* 10c⟩+≡ (10a)

```
    for line in io.lines(file) do
        if state == "doc" then
            ⟨Process Line in Documentation Mode 12b⟩
        elseif state == "code" then
            ⟨Process Line in Source Mode 13a⟩
        end
    end
    flushCode()
    flushDoc()
    return ir
end
```

6.2.1 Reading: Documentation Mode

If we've found a definition in the middle of a documentation block, then store any documentation we've accumulated into the table. This may potentially be nothing at all if the at-sign were followed immediately by a unit definition. Then we create a new unit for the name (or reuse an existing one), and add the reference to this definition chunk to the documentation object. Finally we transition to the code-reading state.

However, if we haven't found a definition then we simply add the line to the documentation block. If the line consists of merely an at-sign (in other words, the start of another documentation block immediately following a documentation block) then we skip it.

Note: my little trickery here of using percent signs in the pattern is simply a way to be able to run this program in Noweb-compatibility mode without having Tensile think that these are references to a program unit with the name `.*` — we shall see this tactic several times throughout this source, and I hope one day to exorcise it by providing a Noweb-like workaround for explicitly identifying non-referencing << characters.

12b ⟨*Process Line in Documentation Mode* 12b⟩≡ (12a)

```
local name, kind = line:match("^%<%<(.*)%>%>([=*])$")
if name then
    flushDoc()
    if kind == "*" then
        ir.doc[#ir.doc + 1] = { type = "def*", name = name }
    elseif kind == "=" then
        defineUnit(name)
        createDoc(name)
        state = "code"
    else
        error("Internal error: unknown kind " .. kind)
    end
elseif line == "@" or line:match("^@|.+|$") or
        (g_opts["noweb-compat"] and line:match("^@"))then
    flushDoc()
    threads = line:match("^@|.+|")
    if threads then
        local threadList = {}
        for t in threads:gmatch("|[^|]+") do
            threadList[#threadList + 1] = t:sub(2)
        end
        threads = threadList
    end
else
    doc = (doc and doc .. "\n" or "") .. line
end
```

6.2.2 Reading: Source Mode

13a ⟨*Process Line in Source Mode* 13a⟩≡ (12a)

```
if line == "@" or line:match("^@|.+|$") or
        (g_opts["noweb-compat"] and line:match("^@")) then
    flushCode()
    threads = line:match("^@|.+|$")
    if threads then
        local threadList = {}
        for t in threads:gmatch("|[^|]+") do
            threadList[#threadList + 1] = t:sub(2)
        end
        threads = threadList
    end
    state = "doc"
else
    local name = line:match("^%<%<(.*)%>%>=$") or
        (g_opts["noweb-compat"] and line:match("^%s*%<%<(.*)%>%>=%s*$"))
```

When we've found a definition while processing a definition, it's time to dump the existing code that we've been spooling up into the table, and create a new unit. We also add its place to the documentation part of the table.

13b ⟨*Process Line in Source Mode* 13a⟩+≡ (12a)

```
    if name then
        flushCode()
        defineUnit(name)
        createDoc(name)
    else
        local pre = nil
        local post = nil
        local ref = nil
        if g_opts["noweb-compat"] or g_opts["indented-refs"] then
            pre, ref, post = line:match("^(.*)%<%<(.*)%>%>(.*)$")
        else
            ref = line:match("^%<%<(.*)%>%>$")
        end
```

We can also find a reference to another bit of code. If we do, we must first flush the code we've read up until this point to the table. We then create a new entry for this unit list, with the type "ref" and pointing to the name of the reference.

In order to track forward and back references, we use a different part of the intermediate representation. Each unit has a list of links in both directions. For both the reference and the parent (which is the current unit name) we ensure that the reference link entries are present in the intermediate representation. Then we add a forward link from the parent to the child, and a backwards link from the child to the parent. Now we can insert cross-references in the output document showing where each unit is defined and used.

13c ⟨*Process Line in Source Mode* 13a⟩+≡ (12a)

```
        if ref then
            if pre and pre:len() > 0 then
                code = (code or "") .. pre
            end
            flushCode()
            unit[#unit + 1] = { type = "ref", name = ref }

            if pre then
                unit[#unit].indent = pre:len()
            end
            if post and post:len() > 0 then
                unit[#unit].followed = true
            end
```

```
            local r1 = ir.ref[unitName]
            r1 = r1 or { fwd = {}, back = {} }
            r1.fwd[#r1.fwd + 1] = ref
            ir.ref[unitName] = r1

            local r2 = ir.ref[ref]
            r2 = r2 or { fwd = {}, back = {} }
            r2.back[#r2.back + 1] = unitName
            ir.ref[ref] = r2
            if post and post:len() > 0 then
                code = post .. "\n"
            end
        else
```

Otherwise this is just another line of code, and we add it to the list of entries in the unit's definition.

14a ⟨*Process Line in Source Mode* 13a⟩+≡ (12a)

```
            code = (code or "") .. line .. "\n"
        end
    end
end
```

14b ⟨*Write Intermediate Representation* 14b⟩≡ (10a)

```
function writeIR(ir, file)
    local stream = io.open(file, "w")
    stream:write("return {\n")
    stream:write("  src = {\n")
    ⟨Write Source Code Intermediate Representation 14c⟩
    stream:write("  },\n")
    stream:write("  doc = {\n")
    ⟨Write Documentation Intermediate Representation 14d⟩
    stream:write("  },\n")
    stream:write("  ref = {\n")
    ⟨Write Reference Intermediate Representation 15a⟩
    stream:write("  }\n")
    stream:write("}\n")
    stream:close()
end
```

14c ⟨*Write Source Code Intermediate Representation* 14c⟩≡ (14b)

```
    for k,v in pairs(ir.src) do
        stream:write("    [" .. string.format("%q", k) .. "] = {\n")
        for i,v2 in ipairs(v) do
            stream:write("      {\n")
            for k3,v3 in pairs(v2) do
                stream:write("        [\"" .. k3 .. "\"] = ")
                if type(v3) == "boolean" then
                    stream:write(v3 and "true" or "false")
                else
                    stream:write(string.format("%q", v3))
                end
                stream:write(",\n")
            end
            stream:write("      },\n")
        end
        stream:write("    },\n")
    end
```

14d ⟨*Write Documentation Intermediate Representation* 14d⟩≡ (14b)

```
    for i,v in ipairs(ir.doc) do
        stream:write("    {\n")
        for k2,v2 in pairs(v) do
            stream:write("      [" .. string.format("%q", k2) .. "] = ")
            if type(v2) == "table" then
                stream:write("{")
                for i3,v3 in ipairs(v2) do
```

```
                    stream:write(string.format("%q", v3))
                    if i3 < #v2 then stream:write(",") end
                end
                stream:write("}")
            else
                stream:write(string.format("%q", v2))
            end
            stream:write(",\n")
        end
        stream:write("    },\n")
    end
```

15a ⟨*Write Reference Intermediate Representation* 15a⟩≡ (14b)

```
    for k,v in pairs(ir.ref) do
        stream:write("    [" .. string.format("%q", k) .. "] = {\n")
        stream:write("      fwd = {")
        for i2,v2 in ipairs(ir.ref[k].fwd) do
            stream:write(string.format("%q", v2) .. ", ")
        end
        stream:write("},\n")
        stream:write("      back = {")
        for i2,v2 in ipairs(ir.ref[k].back) do
            stream:write(string.format("%q", v2) .. ", ")
        end
        stream:write("}\n")
        stream:write("    },\n")
    end
```

15b ⟨*Read Intermediate Representation* 15b⟩≡ (10a)

```
function readIR(file)
    return dofile(file)
end
```

7 Tangling Source Code

15c ⟨*Tangle — Create Source* 15c⟩≡ (10a)

```
function generateCode(ir, unit, output)
    if not ir.src[unit] then
        error("no such unit: " .. unit)
    end
    output = output or unit

    pcall(os.rename, output, output .. ".bak")

    stream, err = io.open(output, "w")

    if not stream then
        io.stderr:write("! Unable to open output file `" .. output .. "' for tangled code from unit `" .. unit ..
"'.\n")
        io.stderr:write("! " .. err .. ".  Emergency stop.\n")
        os.exit(1)
    end

    local x, y = pcall(GenerateCode, ir, unit, stream, 0)
    if not x then
        pcall(os.rename, output .. ".bak", output)
        io.stderr:write(y .. "\n")
        os.exit(1)
    end
    stream:write("\n")
    stream:close()
    os.remove(output .. ".bak")
end
```

```
function GenerateCode(ir, unit, stream, indent)
    --print("Indenting " .. unit .. " at " .. indent .. " spaces.")
    if not ir.src[unit] then
        error("Program module '" .. unit .. "' was not defined.")
    end
    for i,v in ipairs(ir.src[unit]) do
        if v.type == "code" then
            local s = v.text
            -- Insert indentation at every newline.
            -- Strip off the last indentation (since the code
            -- invariably ends with a newline).
            if indent > 0 then
                local strip = false
                if s:sub(s:len()) == "\n" then
                    strip = true
                end
                s = s:gsub("\n", "\n" .. string.rep(" ", indent))
                if strip then
                    s = s:sub(1, s:len() - indent)
                end
            end
            -- Trim off the newline of the last entry.
            -- Let the referring unit decide if it wants to put something
            -- after it (v.type == ref && v.followed) or not.
            if i == #ir.src[unit] then
                s = s:sub(0, s:len() - 1)
            end
            stream:write(s)
        elseif v.type == "ref" then
            ⟨ Write Referenced Code 16 ⟩
        end
    end
end
```

We have encountered a reference to another piece of code. The first thing we need to do is to indent the current line according to the current indentation level. This is necessary to properly propagate the indentation level to the first line of the referenced code. Without it, we get a problem:

```
<<alpha>>=
  <<beta>>
<<beta>>=
beta
<<gamma>>
<<gamma>>=
gamma
```

will produce output with gamma outdented rather than indented. This is because indentation assumes that the top line starts already indented (with an indented reference, such as *beta* above, the text of *alpha* includes the indentation before *beta* already) and only adds indentation for the second and subsequent lines. Since the text of *beta* does not indent the reference to *gamma* any, the output stream will not contain any indentation spacing, and the text gamma will not be indented at all — it is only a single line. Thus, we must propagate the current indentation level down for the first line of the nested element here.

16 ⟨ *Write Referenced Code* 16 ⟩≡ (15c)

```
stream:write(string.rep(" ", indent))
GenerateCode(ir, v.name, stream, indent + (v.indent or 0))
if i < #ir.src[unit] and not v.followed then
    stream:write("\n")
end
```

8 Weaving Documentation

17a ⟨*Weave — Create Documentation* 17a⟩≡ (10a)

```
function generateDoc(ir, output, start)
    local stream = io.open(output, "w")

    if stream == nil then
        io.stderr:write("! Could not write to location `" .. output .. "'.\n")
        os.exit(1)
    end
```

Here we have the translation table that converts symbols we run across in the source into TEX equivalents. Noweb's code mode is not at all a verbatim mode, so we must escape all these characters which normally have other meanings such as curly braces and the backslash. We also have to handle some cases such as angle brackets and dashes, which when appearing together form other symbols in the output ("geese feet" quotes or en dashes).

17b ⟨*Weave — Create Documentation* 17a⟩+≡ (10a)

```
    local texTrans = { ["{"] = "\\{", ["}"] = "\\}",
                       ["_"] = "\\_", ["-"] = "-{}",
                       ["<"] = "<{}", [">"] = ">{}",
                       ["\\"] = "\\verb+\\+", ["|"] = "\\verb+|+" }
    local counter = 0
    local subcounter = {}
    local threads = nil
    ⟨Output by Thread Match 18b⟩
    ⟨Determine Reference Label 19a⟩
    ⟨Generate Source Output 19b⟩
    for i,v in ipairs(ir.doc) do
        if v.type == "text" then
            threads = v.threads
            if threadMatch() then
                if g_opts["source-code"] then
                    stream:write("\\tnslBeginDoc{" .. counter .. "}\\tnslDocPar\n")
                end
                if hook.doc.text then
                    stream:write(hook.doc.text(v.text))
                else
                    stream:write(v.text)
                end
                if g_opts["source-code"] then
                    stream:write("\n\\tnslEndDoc{}\n")
                end
            end
        elseif v.type:match("^def") and g_opts["source-code"] and threadMatch() then
            local nonstop = false
            if v.type == "def*" then
                nonstop = true
            end

            stream:write("\\tnslBeginCode{" .. counter .. "}")

            local sublabel = nil
            if not nonstop then
                sublabel = getSubLabel(v.name)
                if not sublabel.defined then
                    sublabel.defined = true
                else
                    sublabel.minor = sublabel.minor + 1
                end
                stream:write("\\sublabel{" .. tostring(sublabel) .. "}")

                if g_opts["margin-tags"] then
                    stream:write("\\tnslMarginTag{{\\subpageref{")
```

```
                stream:write(tostring(sublabel))
                stream:write("}}}")
            end
        end

        stream:write("\\tnslBeginUnitDef{" .. v.name)

        if not nonstop and g_opts["defn-page"] then
            stream:write("~{\\subpageref{")
            local s = tostring(sublabel)
            s = v.start and s:gsub("-%d+$", "-0") or s
            stream:write(s .. "}}")
        end

        stream:write("}\\tnslEndUnitDef" .. (v.start and "Plus" or ""))
        stream:write("\\tnslBeginDefLine")
```

When this module has some back references (in other words, this module is used within another) then we try to figure out their sublabels so we can display them. The use of the `getSubLabel()` function here will create the label if it doesn't already exist, but leave it unmodified if it does. This means that back references will refer to the last-defined component of that module, or to the first one if it has not yet appeared.

18a ⟨*Weave — Create Documentation* 17a⟩+≡ (10a)

```
        if g_opts["back-refs"] and ir.ref[v.name] and #ir.ref[v.name].back > 0 then
            stream:write("\\tnslBackRef{\\\\{")
            stream:write(tostring(getSubLabel(ir.ref[v.name].back[1])))
            stream:write("}}")
        end

        stream:write("\\tnslEndDefLine")
        stream:write("\n")

        if nonstop then
            GenerateDoc(ir.src[v.name], stream, v.start, {nonstop=true})
        else
            GenerateDoc(ir.src[v.name], stream, v.start)
        end

        if g_opts["back-refs"] and ir.ref[v.name] and #ir.ref[v.name].back > 0 then
            stream:write("\\tnslUsed{\\\\{")
            stream:write(tostring(getSubLabel(ir.ref[v.name].back[1])))
            stream:write("}}")
        end
        stream:write("\\tnslEndCode{}\n")
    end
    counter = counter + 1
  end
end
```

18b ⟨*Output by Thread Match* 18b⟩≡ (18a)

```
local function threadMatch()
    if g_opts["thread"] == nil then
        return true
    elseif threads == nil then
        return false
    else
        for i,v in ipairs(threads) do
            if g_opts["thread"] == v then
                return true
            end
        end
    end
```

```
    return false
end
```

19a ⟨*Determine Reference Label* 19a⟩≡ (18a)

```
local function getSubLabel(origName)
    local sublabel = subcounter[origName]
    if not sublabel then
        local file = file:match("([^/.]+)%.[^/.]+$")
        if not file then
            error("Could not determine file name.")
        end
        file = file:gsub("%s+", "_")
        local name = origName:gsub("%s+", "_")
        sublabel = { major = counter, minor = 0 }
        setmetatable(sublabel, {
            __tostring = function (e)
                return string.format("tensile:lbl:%s:%s-%s-%s",
                    file, name, e.major, e.minor)
            end
        })
        subcounter[origName] = sublabel
    end
    return sublabel
end
```

8.1 Writing Source

This local function writes the actual source for the given *unit* to the documentation *stream*. To accomodate the fact that units may be broken and continue, it also takes a *start* parameter which tells the function where to begin reading source chunks. We go through every source chunk until we've either read them all or we encounter a break.

If the source chunk found is a code type, then we first pass it through any pre-processing hook if present. Then we perform a translation to expand unsafe characters into ones suitable for TEX output. Then we pass it through any post-processing hook that may exist. Finally we write the result to the documentation output stream.

19b ⟨*Generate Source Output* 19b⟩≡ (18a)

```
local function GenerateDoc(unit, stream, start, ...)
    options = {...}
    nonstop = options[1] and options[1].nonstop
    for i = start or 1, #unit do
        local v = unit[i]
        if v.type == "code" then
            local s = v.text
            if hook.doc.source.preExpand then
                s = hook.doc.source.preExpand(s)
            end
            if g_opts.expand_unsafe_tex then
                s = s:gsub("[{}_%-%|\\<>]", texTrans)
            end
            if hook.doc.source.postExpand then
                s = hook.doc.source.postExpand(s)
            end
            stream:write(s)
        elseif v.type == "ref" then
            stream:write("\\tnslStartUnitName{}" .. v.name)
            if g_opts["defn-page"] then
                stream:write("~{\\rm\\subpageref{")
                local s = tostring(getSubLabel(v.name))
                s = s:gsub("-%d+$", "-0")
                stream:write(s .. "}}")
            end
```

```
            stream:write("\\tnslEndUnitName{}")
            if not v.followed then
                stream:write("\n")
            end
        elseif v.type == "break" and not nonstop then
            break
        end
    end
end
```

20a ⟨*Find Toplevel Units* 20a⟩≡ (10a)

```
function findTops(ir)
    local refs = {}
    local tops = {}
    for k,v in pairs(ir.src) do
        for i2,v2 in ipairs(v) do
            if v2.type == "ref" then refs[v2.name] = true end
        end
    end
    for k,v in pairs(ir.src) do
        if not refs[k] then tops[#tops + 1] = k end
    end
    return tops
end
```

20b ⟨*Program Initialization* 20b⟩≡ (10a)

```
-- Program modules to process.
local units = {}

-- Where to write the TeX output.
local weave_output = nil

g_opts = {
    ["extract-all"] = false     ,
    ["noweb-compat"] = false    ,
    ["indented-refs"] = false   ,
    ["show-tops"] = false       ,
    ["write-ir"] = false        ,
    ["weave"] = true            ,
    ["docs"] = true             ,

    ["margin-tags"] = true      ,
    ["defn-page"] = true        ,
    ["back-refs"] = true        ,
    ["source-code"] = true      ,

    ["expand_unsafe_tex"] = true
}
```

20c ⟨*Option Processing* 20c⟩≡ (10a)

```
local i = 1
while i <= #arg do
    local v = arg[i]
    if v == "-h" or v == "-help" then
        io.stdout:write("This is Tensile, version " .. version.revision .. " (" .. version.date .. " UTC).\n\n")
        io.stdout:write([[
Tensile (c) 2009-2010 Taylor Venable.  All rights reserved.
Provided under the terms of the Simplified (2-Clause) BSD license.
LaTeX support provided under the LaTeX Project Public License, v1.3c or later.

USAGE:
    tensile <options> <literate-file>
```

```
OPTIONS:

  Standard:

    -h / -help          Show this message.

  File Handling:

    -indented-refs      Allow references to be indented in source.
    -list-tops          Print all toplevel units and quit.
    -noweb-compat       Enable Noweb-compatible parsing.
    -show-tops          Same as "-list-tops".
    -write-ir           Write intermediate form to file.

  Tangled Output:

    -extract-all        Extract all toplevel units.
    -tangle-to          Write single a single unit's source to <file>.
                        This option will be ignored if > 1 unit is tangled.
    -unit <name>        Tangle unit <name>.

  Woven Output:

    -dont-weave         Do not produce woven output.
    -hide-margin-tags   Don't display definition tag number in the margin.
    -hide-defn-page     Don't show references to first definition.
    -hide-back-refs     Don't print references to usage location.
    -hide-source-code   Don't output source code in documentation.
    -no-docs            Same as "-dont-weave".
    -thread <name>      Only weave output for doc chunks in thread <name>.
    -weave-to <file>    Write woven output to <file>.

  Deprecated:

    -R<name>            Same as "-unit <name>".
    -o <file>           Same as "-weave-to <file>".

Email bug reports to taylor@metasyntax.net.
]])
        os.exit(0)
    end

    if v:match("^-R") then
        units[#units + 1] = v:sub(3)
    elseif v == "-unit" then
        if i == #arg then error("too few options") end
        units[#units + 1] = arg[i + 1]
        i = i + 1
    elseif v == "-weave-to" or v == "-o" then
        if i == #arg then error("too few options") end
        weave_output = arg[i + 1]
        i = i + 1
    elseif v == "-tangle-to" then
        if i == #arg then error("too few options") end
        g_opts["tangle-to"] = arg[i + 1]
        i = i + 1
    elseif v == "-O" then
        if i == #arg then error("too few options") end
        local s = arg[i + 1]
        local k,v = s:match("^([^=]+)=(.*)$")
        if k and v then
            if     ({FALSE = 1, NO = 1})[v:upper()] then v = false
            elseif ({TRUE = 1, YES = 1})[v:upper()] then v = true
            elseif ({NIL = 1, NULL = 1})[v:upper()] then v = nil end
```

```
            g_opts[k] = v
        end
        i = i + 1
    elseif v == "-thread" then
        if i == #arg then error("too few options") end
        g_opts["thread"] = arg[i + 1]
        i = i + 1
    elseif v:match("^%-") then
            if v:match("^%-hide%-")  then g_opts[v:sub(7)] = false
        elseif v:match("^%-omit%-")  then g_opts[v:sub(7)] = false
        elseif v:match("^%-elide%-") then g_opts[v:sub(8)] = false
        elseif v:match("^%-no%-")    then g_opts[v:sub(5)] = false
        elseif v:match("^%-dont%-")  then g_opts[v:sub(7)] = false
                                     else g_opts[v:sub(2)] = true
        end
    else
        file = v
    end
    i = i + 1
end

if not file then
    io.stderr:write("! No file given.\n")
    os.exit(1)
end

⟨Check File Existence 22⟩
⟨Load Hooks 23a⟩

local ir
⟨Read Literate Source 23b⟩
⟨Determine Toplevel Units in File 23c⟩

if g_opts["list-tops"] or g_opts["show-tops"] then
    ⟨Print List of Toplevel Units 23d⟩
end

if g_opts["extract-all"] then
    units = tops
end

if #units > 1 then
    g_opts["tangle-to"] = nil
end

⟨Tangle Specified Units 23e⟩
⟨Weave Literate Documentation 24⟩
```

Attempt to open the specified file for reading, to make sure it exists. If it does not, try appending the extension `.tnsl` commonly used by Tensile literate programs. If with the extension it still does not exist then an error is raised and the program stops. Otherwise we update the file name to that with the extension and continue. In any case, always close all streams opened in an attempt to check for file existence. One wishes Lua had a function that would do this for me, and not force me to open files just to confirm existence.

22 ⟨*Check File Existence* 22⟩≡ (20c)

```
local x, err = io.open(file, "r")
if not x then
    local y = io.open(file .. ".tnsl", "r")
    if not y then
        io.stderr:write("! Unable to open input file `" .. file .. "' as literate source.\n")
        io.stderr:write("! " .. err .. ".  Emergency stop.\n")
        os.exit(1)
    end
```

```
    y:close()
    file = file .. ".tnsl"
else
    x:close()
end
```

Initialize the hook tables, and load the local Tensile runtime control file. If it is not present, then it is no big deal, just catch the error and continue.

23a ⟨*Load Hooks* 23a⟩≡ (20c)
```
hook = {}
hook.doc = {}
hook.doc.source = {}
hook.src = {}

if os.getenv("HOME") then
    pcall(loadfile(os.getenv("HOME") .. "/.tensilerc"))
end
```

If we're going to write the intermediate representation, we read the file and write it's representation to disk, then read that representation to continue the work of tangling units or whatever. This is a good way to test things in the syntax of Tensile itself, and changes to the reading of the literate source file. If we don't particularly care about getting the intermediate representation then we can just read it and use the object that was created within the system, without serializing it out and back in.

23b ⟨*Read Literate Source* 23b⟩≡ (20c)
```
if g_opts["write-ir"] then
    local tangled = file .. ".tensile"
    writeIR(generateIR(file), tangled)
    ir = readIR(tangled)
else
    ir = generateIR(file)
end
```

Find all the toplevel units in the literate source, and put them into a list. Then, take that list and make it an associative array as well, so that the keys are the names of the units. This is for fast lookup later on, when checking to make sure that the units requested as toplevel units.

23c ⟨*Determine Toplevel Units in File* 23c⟩≡ (20c)
```
local tops = findTops(ir)
for i,v in ipairs(tops) do
    tops[v] = true
end
```

If we were asked to simply print out the list of toplevel units, do that and then stop.

23d ⟨*Print List of Toplevel Units* 23d⟩≡ (20c)
```
for _,unit in ipairs(tops) do
    print(unit)
end
os.exit(0)
```

Go through the unit list, and if it is not a toplevel unit then print a warning message. Tangle the code for that toplevel unit, using the user-supplied output file (this may have been overridden by the system to be nil; if it is then the `generateCode` function will automatically choose a suitable output file).

23e ⟨*Tangle Specified Units* 23e⟩≡ (20c)
```
for i,v in ipairs(units) do
    if not tops[v] then
        print("WARNING: " .. v .. " is not a toplevel module.")
    end
    generateCode(ir, v, g_opts["tangle-to"])
end
```

If an woven output target has not already been specified, then derive it from the literate source file name by replacing the extension on the file with `.tex`.

24 ⟨*Weave Literate Documentation* 24⟩≡ (20c)

```
if g_opts.weave and g_opts.docs then
    weave_output = weave_output or file:gsub("\\.[^.]+$", "") .. ".tex"
    generateDoc(ir, weave_output)
end
```

A Bugs

On 1st December 2010, I started logging all bugs in the system so that I could keep better track of how the quality of the program improves over time.

2010-01-19
- `generateIR`: Threads were not being recognized when the parser was already in source mode. Copy the logic from doc mode to source code for determining threads.

2010-01-09
- `generateDoc`: Using `\text...` in XeLaTeX causes the symbol to be displayed in the symbol font rather than the typewriter font. Use `\verb` to get around this problem.

2010-01-05
- `generateDoc`: Using < or > in XeLaTeX causes the sign to be displayed in the math font rather than the typewriter font. Use `<{}` and `>{}` instead.

Colophon

This document was typeset using XeLaTeX, a direct-to-PDF implementation of Donald Knuth's TeX typesetting system with OpenType font handling and features for printing text in any language. It uses Leslie Lamport's famous LaTeX macro package. Text in the main body of this document uses the Minion typeface by Robert Slimbach, set at 10 / 13 × 36. The sans-serif typeface used in the abstract and for program names is Myriad, a collaboration between the same designer and Carol Twombly. Code listings and other technical matter which appear throughout the document use the Inconsolata typeface by Raph Levien.