A Meta-Model for Literate Programming

10th November 1999

Tim Evans

Abstract

Literate programming is a program development method based on combining source code and documentation in a single file. The separate documents are then automatically extracted by a literate programming tool. This report presents the development of a meta-model for literate programming, which is a system for specifying literate programming tools. The meta-model was created as an abstraction based on a review of existing tools.

Use of the meta-model for specifying a simple but usable tool is demonstrated, along with the implementation of that tool. The implementation was well specified, extensible and easy to implement.

Contents

Chapter 1

Introduction

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.
>
> —Donald Knuth, *Literate Programming*, 1984

Literate programming is a technique, invented by Donald Knuth in 1984, that combines the source code and documentation of a program into a single file and allows both the be written at the same time. A literate program is written in a way that concentrates on describing the program to people rather than to computers. Making the code easier to understand has several benefits, including easing maintenance and reuse. The separate code and documentation files are extracted from the combined file using a literate programming tool.

Explanation of the code uses two techniques. Firstly, documentation is written using a mark-up language like LaTeX or HTML, allowing for the use of figures, tables, mathematical equations, sectioning, and other devices. The code is presented as part of the output documentation, with pretty printing and cross-referencing. Secondly, literate programming allows the programmer to present the program in whatever order is best for explaining it, which will usually be quite different from the order that the compiler expects to see it. Sections of code that are complex can be replaced by a short description of what the code does, and then the full explanation (and source code) is left till later.

One of the problems with literate programming is the fragmentation of tools. A large number exist, each of which uses a slightly different model of processing. Also, the model, in most cases, is not explicitly defined but must be inferred from the behaviour of the tool. The construction of a meta-model for literate programming could form part of a solution to this problem by providing a means of explicitly describing each processing model.

A meta-model is a model for specifying other models. Each literate programming tool uses some kind of model for its input, whether stated or implied. The model will specify what it expects its input to look like, what structures are implied from the input and how it extracts the output. The meta-model described in this report is an general purpose representation of the models used by many current literate programming tools. The meta-model is also a powerful exploratory tool, as it allows the rapid specification and implementation of new literate programming tools.

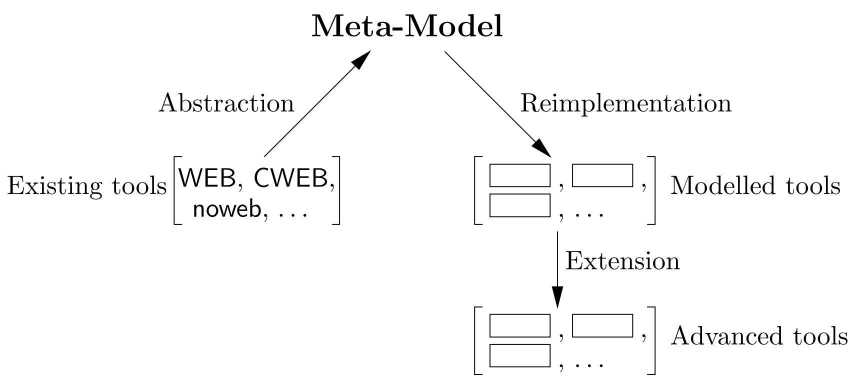


Figure 1.1: The process of implementing and using the meta-model.

The steps involved in this project are displayed in Figure 1.1. First, a survey of current literate programming tools is needed to determine what literate programming is and what properties are required for the meta-model. These current literate programming tools will then be abstracted into a meta-model. The next step is testing the completeness of the meta-model by ensuring that it is capable of describing tools of similar complexity to current tools. Tools implemented by the meta-model will hopefully be easier to understand, easier to modify, and capable of exchanging features between themselves because of their explicit and consistently defined models.

If this step is successful, the meta-model can then be used to extend these tools, possibly producing the next generation of literate programming tools. This report goes as far through this process as implementing the first modelled tool.

A detailed description of literate programming, and a review of current tools is presented in Chapter 2. The meta-model is defined in Chapter 3. A presentation of an example model and its implementation is given in Chapter 4.

Acknowledgements

Thanks go to Dr. Neville Churcher for supervising this project and providing many helpful comments, and to Jane McKenzie for proof reading the first drafts of this report.

Chapter 2

What is Literate Programming?

The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.

—Fred Brooks, Jr.

2.1 Basics

The concept of literate programming was invented by Donald Knuth[1]. Knuth used the phrase "literate programming" because he considered programs and their documentation created in this way to be works of literature that describe the program to humans as well as to computers.

A complex software system, like any complex system, can be understood by understanding the small parts of the system and the relationships between them. Literate programming divides a plain program into parts, presents these parts in any order and documents them. Documentation is usually done using a markup language or word processor, allowing it to more complete and descriptive than can be achieved via plain text comments.

A literate program has two outputs, the source code and the documentation. The source code is included in the documentation; generally in a pretty-printed and cross-referenced form.

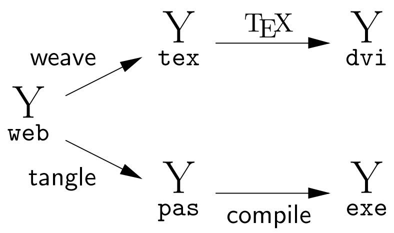
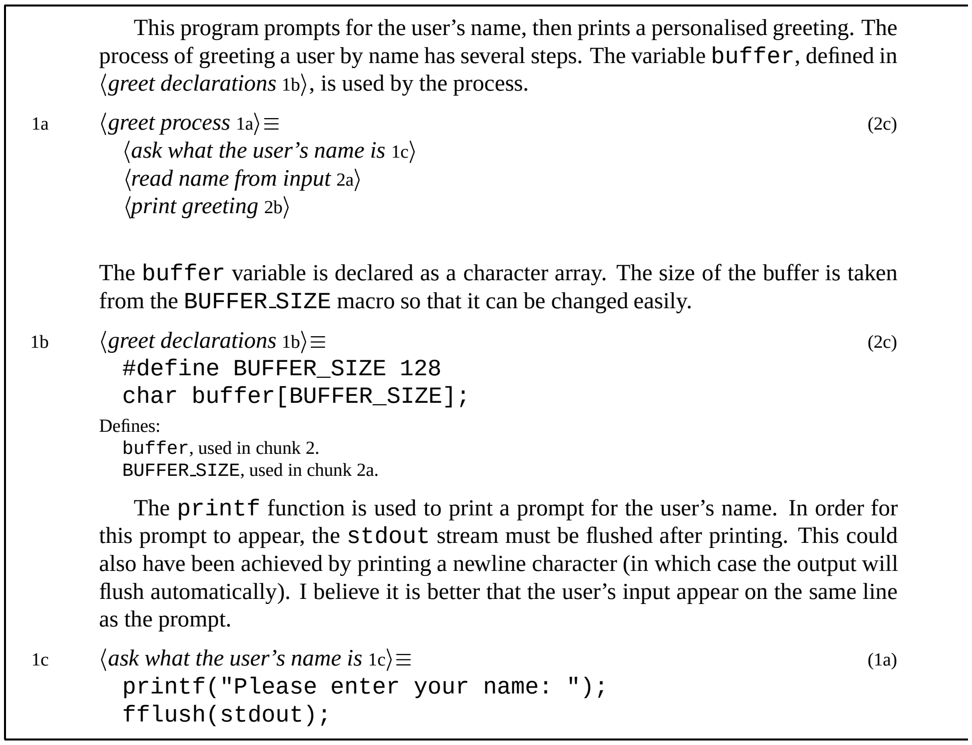


Figure 2.1: The processing steps for a WEB literate program. Other literate programming tools follow a similar pattern.



This program prompts for the user's name, then prints a personalised greeting. The process of greeting a user by name has several steps. The variable `buffer`, defined in ⟨*greet declarations* 1b⟩, is used by the process.

1a ⟨*greet process* 1a⟩≡ (2c)

```
⟨ask what the user's name is 1c⟩
⟨read name from input 2a⟩
⟨print greeting 2b⟩
```

The `buffer` variable is declared as a character array. The size of the buffer is taken from the `BUFFER_SIZE` macro so that it can be changed easily.

1b ⟨*greet declarations* 1b⟩≡ (2c)

```
#define BUFFER_SIZE 128
char buffer[BUFFER_SIZE];
```

Defines:
`buffer`, used in chunk 2.
`BUFFER_SIZE`, used in chunk 2a.

The `printf` function is used to print a prompt for the user's name. In order for this prompt to appear, the `stdout` stream must be flushed after printing. This could also have been achieved by printing a newline character (in which case the output will flush automatically). I believe it is better that the user's input appear on the same line as the prompt.

1c ⟨*ask what the user's name is* 1c⟩≡ (1a)

```
printf("Please enter your name: ");
fflush(stdout);
```

Figure 2.2: A literate programming example

The first literate programming tool was WEB[2]. It documented Pascal programs using the TEX mark-up language. Knuth used the WEB system to create several programs, most notably METAFONT[3] and TEX[4]. The WEB system consisted of two tools, one to extract the code, which was called tangle and one to extract the documentation, called weave. The operation of these two tools is shown in Figure 2.1.

Since WEB was created there have been many other literate programming tools developed, most based on WEB. Many of the new tools were different only in the languages they supported, covering almost every popular programming language of the time, from C to Lisp, and also many different mark-up languages including LATEX, HTML and Troff. Several tools have also been created that are language independent, in both programming and/or mark-up language.

2.2 Details

As stated above, a literate program is divided into many parts. Each of these parts is traditionally called a 'chunk'. Chunks can be either source code or documentation. Source code chunks are each given a unique name by the programmer. This name should contain a very brief description of what the piece of code does, such as "parse the command line arguments". Following this short description is number of lines of source code. This code may include other code chunks by making reference to their names. In the documentation output, ref-

The `fgets` function is used to read the user's name. `fgets` was chosen over `gets` because `gets` does not check for buffer overflows.

2a ⟨*read name from input* 2a⟩≡ (1a)
```
fgets(buffer, BUFFER_SIZE, stdin);
```

Uses `buffer` 1b and `BUFFER_SIZE` 1b.

The greeting is printed using the `printf` function again, this time with a `%s` format to insert the name.

2b ⟨*print greeting* 2b⟩≡ (1a)
```
printf("Hello %s", buffer);
```

Uses `buffer` 1b.

The root chunk wraps the greeting routine in the C `main` function so that it will run when the program is executed.

2c ⟨* 2c⟩≡
```
#include <stdio.h>
int main(void) {
  ⟨greet declarations 1b⟩
  ⟨greet process 1a⟩
  return 0;
}
```
Defines:
`main`, never used.

erences to other code chunks are printed as the names of those chunks, while in the code output the complete text is substituted.

Each code chunk should be documented by the preceding documentation chunk. What is contained in the documentation chunk is up the programmer, but will normally include a description of what the code does, how it does it and any design decisions made by the programmer. Documentation chunks can contain commands of whatever mark-up language is used by the literate programming tool.

The first reaction of many people is to ask why literate programming is better than correct use of comments, and there are two main reasons. The first is the ability to use the power of a mark-up language for describing code. This allows the programmer to use diagrams, mathematical equations, references and indexing in their descriptions. The second advantage (and more important according to many people) is the ability to describe a complex piece of code in terms of its components. For example, consider a complex C function that is constructed out of three major parts: data gathering, data processing and reporting of results.

One solution would be to reduce this into three functions, but this will make it less efficient and will add a number of complicated interfaces in the form of arguments lists. A literate program would simply present something like:

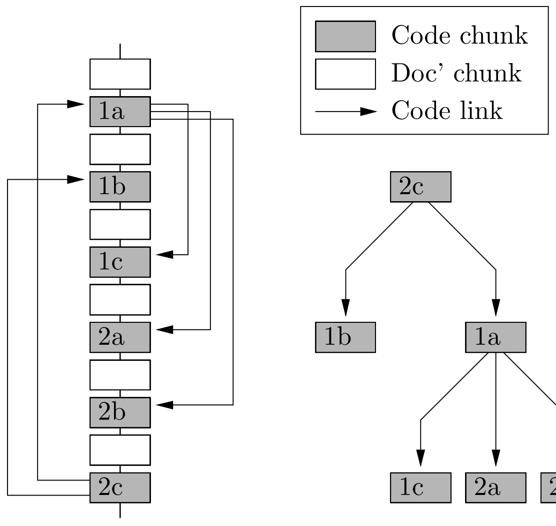
```
void read_complex_file(FILE *input) {
  ⟨get data from the file⟩
  ⟨search for interesting parts of the data⟩
  ⟨dislay the results in a table⟩
}
```

Even without any documentation, this allows the reader to easier understand what the function does and what its major steps are, by removing the need to wade through what is probably a lot of very rather complex code. The complex code is simply described later, probably by dividing it yet again into its major steps.

The order that the steps are presented can also be changed (it is legal to use a chunk before it is defined), and will depend a lot on the design methods used by the programmer. For example, a program that is designed in a top-down manner will probably be documented that way as well. The code chunk that represents the top level of the module would be presented first, followed by each of its parts, with the actual implementation details left to the end. Code that is designed bottom-up, on the other hand, will be documented in the opposite order. The first chunks will describe the low level parts, and the following chunks will slowly combine them into a module. Real programs are more likely to use a hybrid order, both for design and presentation.

It is possible to represent a literate program as graph of connected nodes, with each node representing one chunk. Inside this graph there are really two different structures. The first structure is the order of chunks in the input, a linear list. This ordering is also maintained in the output documentation. The second structure is the hierarchy created by the inclusion links in the code chunks. These links will normally form a tree, but can form an acyclic directed graph in some rare situations. In most cases the two structures will be almost completely unrelated.

To give an idea of what the documentation output of a literate program looks like, a simple example is shown in Figure 2.2. The two structures of the example are shown in Figure 2.3. This example was produced by the noweb tool, described in Section 2.4.4. Some of the more important aspects of the example program are listed below.



Linear structure. Hierarchical structure.

Figure 2.3: The two structures of the example literate program, linear and hierarchical

- The important code is described first, with unimportant but necessary parts of the code placed later.
- The function was broken down into definitions and execution, then the execution was further broken down into individual steps.
- Each section of code is described by the preceding documentation.
- The first section of documentation contains a reference to a chunk, inserted by the programmer, which appears as "⟨*greet declarations* 1b⟩".
- The noweb tool has not pretty printed the code, but it has automatically included cross-referencing of identifiers. Each code chunk is followed by a list of what it defines and uses.
- Cross-referencing is also done for chunks. Each chunk is given an number such as 2a, which includes the page number of which the chunk was defined. Other references to the chunk include this number. Also, when a chunk is defined the numbers of the chunks that include it are printed to the right.
- The code chunk labelled "*" is the root chunk, it defines what code will be produced by the tangling process. The tangling part of noweb can also start extraction at any other chunk name, allowing multiple source file to be documented in one literate program.

2.3 Features

The many different literate programming systems have, along with the same basic functionality, a number of different features. The following sections describe some of the more common features.

2.3.1 Pretty Printing

How source code is displayed in the documentation output is the most visually obvious difference between one tool and another. Simple tools, particular those that are language independent, present code without pretty printing. Most tools perform some type of pretty printing, but the degree and the style used differs greatly. Most people have their own ideas about how pretty printing should be done and there is certainly no consensus in the literate programming field.

The simplest form is highlighting basic language constructs such as reserved words, built-in types, string constants and comments by changing the typeface or text colour. This type of pretty printing should only require a regular expression based scanner, not a parser, and is common in both literate programming tools and other tools, such as a2ps[5] and lgrind[6] programs. An example of the output of lgrind is shown in Figure 2.4, and its output is analogous to many literate programming tools.

A more complex form of pretty printing requires the literate programming tool to parse the input file. This complexity allows semantic constructs to be highlighted. For example identifiers can be highlighted differently depending on whether they represent types, classes, variables or functions. This technique

```
/*
 * hello.c
 *
 * The first program written in any language.  It prints ‘hello world’
 * to its output when run.
 */

#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("hello world\n");
}
```

Figure 2.4: Sample output of the lgrind tool. Many literate programming systems produce similar output.

is used by many literate programming tools but is not common among other tools, as it is very language dependent. One major problem with this approach is if the pretty printing parser does not support *exactly* the same language as the compiler (possibly due to non-standard compiler extensions) as the pretty printer can then encounter parse errors or produce incorrect output.

Another popular extension for either of the above types of pretty printing is to use the richer symbol set available in whatever mark-up language is in use. In C, it is common to replace symbols such as "<=" with "$\leq$", or "==" with "$\equiv$". This can be useful in some cases, but it is possible readability will be reduced by having the printed code look like a completely different language.

There are many different ideas about how pretty printing should be done. A system that allows optional or configurable pretty printing (as some do) can be a better, but more difficult, way to satisfy more people.

2.3.2 Cross-Referencing

An important part of the output of a literate programming tool is automatic cross-referencing of the definition and use of various program artifacts, e.g. functions and variables. For language dependent systems it is relatively easy to do, although it may require parsing the source code to identify where artifacts are defined and used.

Cross-referencing usually comes in two related forms. The first is in-place cross-referencing, where the tool will insert a list of artifacts used in each part of the source code and a reference to the other part of the code that defines that artifact. The second form is the creation of a separate index of all program artifacts, listing where they are defined and used.

For programming language independent tools cross-referencing is much more difficult. In some cases extensions are used to parse and cross-reference a particular language, but it often comes down to programmers having to specify themselves which variables were defined and used in each part of the code.

2.3.3 Programming Language Dependence

Most of the early literate programming tools supported only a single programming language. This was mostly so they could implement pretty printing, cross referencing and macro facilities for that language. The result of this was several slightly different tools that supported the different programming languages. The advantage of the method was that each tool could specialize for its own programming language, producing better results in both output code and documentation.

The problem with this situation is that many projects do not involve only a single programming language. A project might, for example, be written mostly in C but also have some complex shell scripts that also need to be documented. Ideally, you could document both programs without having to learn to use two separate tools.

There exists a set of tools that have been created to be able to support any programming language. This ability generally comes at the cost of a reduction in the complexity or removal of the pretty printing and cross referencing features.

2.3.4 Mark-up Language Dependence

The first literate programming tools were also dependent on a single mark-up language. The majority of tools used either TEX or LATEX, with smaller numbers using Troff, SGML and later HTML. A few tools have actually been written directly in LATEX or SGML.

Several approaches have been used to remove mark-up language independence. The easiest is for the tool to able to create its own mark-up commands in one of a set of different languages. The user must then choose between one of these languages at the start of the project. This approach allows the user to start using a new mark-up without having to change to a different literate programming system.

Another approach used to create mark-up language independent tools was to create a new mark-up language specific to that literate programming tool. The tool then produces documentation by converting its internal mark-up language into any other mark-up language that it knows about. The main problem of this approach is that the mark-up language is generally an intersection of features of something like LATEX and HTML, giving none of the benefits and all of the deficiencies of both languages. The advantage is that the tool can generate output that is useful for *both* an online hypertext database and for printed documentation.

2.3.5 Macros

Many literate programming tools provide a macro facility, similar to that provided by the C preprocessor. Initially this feature was part of WEB, mostly to overcome deficiencies in the Pascal language. In almost all cases the same functionality could be provided by using another macro processor as a second step of program extraction, but it is more efficient to combine the two steps. Unfortunately this gain in efficiency comes at the cost of increased complexity. Tools striving to be simple will normally drop this feature, assuming users will add a separate macro tool if and when they require it.

2.4 Tools

This section presents some examples of literate programming tools. The examples given attempt to cover a broad range of the different types of tool available.

	programming language	mark-up language	pretty print	cross-ref	macros
CLiP	any	any	–	–	–
CWEB	C, C++	LaTeX, TeX	yes	yes	yes
FunnelWeb	any	LaTeX, HTML	–	–	yes
Noweb	any	LaTeX, HTML	ext.	manual/ext.	–
Nuweb	any	LaTeX	–	manual	–
Spider	any	TeX	yes	yes	yes
WEB	Pascal	TeX	yes	yes	yes

Table 2.1: A comparison of literate programming tools

2.4.1 WEB

WEB[2] was the first literate programming tool. Written in part as an experimental system WEB was reasonably successful in both establishing the literate programming paradigm and in some real programming projects. Several examples of WEB programs have been published by Knuth and others[7, 8, 9, 10], and the WEB sources for TeX and METAFONThave been published as books[4, 3].

WEB was created to work exclusively with Pascal and TeX but it was always intended that related tools would be developed for other languages. Many of the features of WEB were created in part to assist the Pascal compiler in the same way that the C preprocessor assists a C compiler. For example WEB would automatically evaluate constant expression like “2 + 3”, allowing them to be used where the Pascal compiler is expecting a single value.

WEB is a complex tool, with advanced features, a complex syntax and many different permutations and implementations. Despite not being the best tool for usability WEB remains as the definition of literate programming.

Many literate programming tools simply reimplement WEB to support different programming and mark-up languages, the first and probably most notable of which is CWEB[11]. CWEB supports the C and recently C++ languages. A notable piece of example code is the Stanford GraphBase[12, 13], a set programs and data for combinatorial algorithms and data structures.

2.4.2 CLiP

CLiP[14] is one of the few literate programming tools that does not claim to be descended from WEB. CLiP uses different method for literate programming that makes it completely independent of programming language and mark-up language, even supporting WYSIWYG word processors. It is also programming language independent.

Rather than extracting both documentation and code from one source file, the programmer writes the documentation (with embedded code chunks) first and CLiP extracts the code from it. This requires a word processor that is capable of producing plain text output, but that includes almost all such applications. The code chunks can be presented in any form, but must be delimited by special strings that CLiP uses to identify them later.

The advantage of this process is that it can use any document preparation system that can output as plain text, including almost all word processors as well as mark-up languages like LaTeX and HTML. Several disadvantages stem from the fact that CLiP cannot alter the documentation in any way. This means that automatic pretty printing, cross-referencing and indexing are impossible. Such features could be added to the documentation manually be the programmer, but this will be difficult and time consuming in most word processors.

2.4.3 Nuweb

Nuweb[15] was designed to be a very simple literate programming tool. The advantages of this simplicity are easy of use, speed, and programming language independence. There is no support for any mark-up language other than LaTeX.

Simplification down to nuweb removes pretty-printing and restricts cross-referencing. Rather than the tool automatically finding variable and function definitions, the programmer must specify which identifiers are defined in each code chunk. The tool will find all places where this identifier is later 'used' by searching for matching parts of other code chunks.

Rather than having two separate tools to 'weave' and 'tangle' the input files, nuweb can output multiple code files and the documentation in one step.

2.4.4 Noweb

Noweb[16] is another simplified literate programming tool. Despite its simplicity, and in contrast to some other modern tools, it is quite similar WEB in structure, syntax and processing. The programmer can choose to use LaTeX or HTML for their documentation, or noweb can try to convert simple LaTeX commands into HTML, similar to the latex2html program.

The major advantage of noweb is the ability to extend the standard weave and tangle processes with customised extensions that operate via pipelines. These extensions can be used to provide the higher level abilities of the more complicated literate programming tools. Extensions exist for automatic generation of definition directives for several languages, replacing the need for manual directives. Extensions have also been developed for pretty printing various languages, and the Pretzel[17] program can actually generate such pretty printers based on a description of the language.

2.4.5 FunnelWeb

FunnelWeb[18] is another of the more modern tools, but is more complex than the very simplest tools like nuweb and noweb. Macros are an important part of the tool, and they are used to define most behaviour.

Rather than force a user to choose either LaTeX or HTML, FunnelWeb has its own internal mark-up language (defined via its macro facilities) that it can convert to either mark-up language on output. If the programmer needs the specific features of either mark-up language then language-specific documentation can also be used.

2.4.6 Spider

Spider[19] is a tool that generates programming language dependent weave and tangle pairs based on a description of the programming language. The language description specifies how the language should be pretty-printed and how variable and function definitions and uses appear. The spider tool works from the language description and generates a literate programming tool specific to that language. The behaviour of the generated tools is similar to that of WEB.

2.5 Problems and Extensions

2.5.1 Editing

Editing literate program sources poses a problem. The literate sources contain both documentation and source code, which often have very different editing semantics (compare a word processor and an integrated development environment). The literate programmer is often reduced to using a plain text editor, that gives no help for code development or documentation.

One alternative is to use a literate programming tool that can support a word processor (CLiP, WinWordWeb and others). This presents the documentation well, but can be worse for code editing than a text editor. There are also the approach where the literate program is editing using a program editor (such as Emacs) and the documentation appears as mark-up language commands. This also fails to work well, as the documentation can only really be easily read after it has been processed, and traditional program editors do not adapt well to the different structure of literate programs.

The only way to effectively solve these problems is to have an editor designed for editing literate program. Such editors have been developed as research projects[20, 21, 22], but none have yet reached the sophistication of modern word processors and program editors. No work appears to have been made into developing such an editor commercially, as part of some kind of integrated development environment.

2.5.2 Debugging

A serious problem with traditional methods of literate programming is line number mangling. This occurs when the lines of the actual code file (that the compiler reads) do not match in any way with the line numbers of inside the literate program. When the compiler produces:

```
line 753: parse error
```

The error might be in anywhere at all in the literate program, because of the reordering of the code chunks. In C, this problem can be solved by emitting "`#line`" directives that are normally used by the C preprocessor. Other languages have no preprocessor, so have no way of understanding this type of directive.

The best place to solve this problem is in an editing program, just adapt the "goto line" action so that it can map the line in the output code back to the correct line in the literate program.

2.5.3 Object Orientation

Literate programming was designed and developed around mostly procedural languages. The current object oriented literate programming tools are often little more than adapted procedural tools.

The most difficult problem seems to be in cross-referencing. If a code chunk contains "`foo.toString()`", the class of the variable "`foo`" must be identified before the correct "`toString`" method can be found. In statically typed languages, like Java, it is possible to determine the class of a variable from variable declarations. This will require parsing the code, but full parsers are not uncommon in literate programming tools. In a dynamic language, like Smalltalk or Python, the class of a variable can only be determined at run-time, making an exact cross-reference impossible. One possible solution to such cross-referencing is to link a call of a "`toString`" method to all classes that define such a method (which could be a very large set).

There are also possible problems in the basic methodology of most literate programming tools. One of the principles of object orientation is that objects have a very well defined interface, which is separate from their implementation. This separation would suggest that the interface and implementation should also be documented separately, and a standard literate programming tool is unable to do this.

2.5.4 The Three Syntax Problem

When editing a literate program, the programmer must deal with three different syntaxes: the programming language, the mark-up language and the commands of the literate programming tool itself. Syntax errors can occur in any of the three syntaxes, or because of a combination of them. Many people find that dealing with the three syntaxes can be confusing, and increases the difficulty of learning and using a literate programming tool. Three different methods for solving the three syntax problem are given below.

1. Implement the mark-up language as part of the literate programming tool, generally by extending the macro concept, as used by FunnelWeb. This approach both reduces the power of the mark-up language and makes the syntax of the literate programming tool more complex, and harder to learn.

2. Make the literate programming tool part of the mark-up language. This is possible with both LaTeX (a basic implementation is ProTEX[23]) and SGML based tools, as both mark-up languages are fully extensible. This approach is better, but has the disadvantage that the tool is forever tied to that mark-up language.

3. Implement an editor that hides the syntax of the literate programming tool behind a graphical user interface (GUI). If the editor also incorporates WYSIWIG documentation abilities then the syntax of the mark-up language is also hidden, reducing the three syntax problem to one syntax (the programming language) and a user interface.

2.6 Summary

There are currently a large number of different literate programming tools. Although conceptually similar, they all have slightly different behaviours, feature sets, syntaxes and results. Despite the differences they share a set of features that define literate programming. These defining features are:

- The ability to edit source code and documentation at the same time, using the same tool.
- The ability to describe a complex piece of code as being made up of several simpler pieces of code, and to present these pieces in any order.
- The production of a document, describing the program, that is designed for humans rather than computers.
- The ability to use some type of mark-up language to typeset the documentation.

There are also a set of features that are optional but common:

- Cross-referencing of code artifacts (variables, functions) and of each code chunk in the human readable document.
- Pretty printing of code in the human readable document.
- Macros to make it easier to produce different output code under different circumstances and to reduce the need to repeat code. This is the same functionality that the C preprocessor implements.
- Various features to help when creating documentation that extend the mark-up language being used. For example, consistent formatting of variable names in documentation chunks.

The large number of different tools creates a challenge when designing a meta-model for literate programming. The meta-model must be able to model the current tools, focusing on the common functionality that defines literate programming, and at the same time be able to go beyond the current literate programming tools to explore new areas.

Chapter 3

The Meta-Model

If there are epigrams, there must be meta-epigrams.

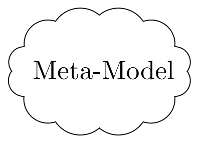
—*fortune*

3.1 Motivation

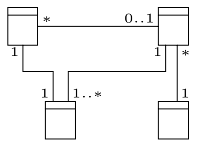
3.1.1 Three Layers

The meta-model exists in a three layer architecture, as shown below. Mapping between the layers occurs when one layer is used to create an artifact in the next layer down. This can be either using the meta-model to define a new model (and therefor a new literate programming tool), or using a model to create a literate program designed for a particular tool.

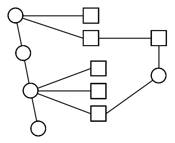
1. The top layer is the meta-model itself.



2. The second layer is the set of all literate programming tools. Each tool implements its own model of what a literate program is. The layer 1 meta-model should be general enough to describe these models. This layer can be represented as a entity-relationship diagram.



3. The third level is the literate programs themselves. Each literate program will conform to the model in layer 2. This layer can be represented as a graph of connected chunks, the structure of which obeys the layer 2 model.



3.1.2 Tool Fragmentation

One of the problems in literate programming is the fragmentation of tools. A large number of tools have been developed, and each has a slightly different set of features and behaviours. The meta-model should provide a way to build literate programming tools that are easy to modify and extend, and make it easy for these modifications and extensions to be applied to other tools that are also constructed using the meta-model.

3.1.3 Extending the Models

Current literate programming tools use a very simple structure, based on only two different types of chunks with only simple connections between them. In some cases this structure is simpler than what is desirable. For example, consider the separate interface and implementation documentation for object orientation as described in Section 2.5.3. The structure of such a system would be more complex than most literate programming systems support. The meta-model should be able to easily describe and implement these structures.

3.1.4 Adaptive Tools

The highest goal for a meta-model for literate programming is the construction of adaptive tools. Such tools would be able to dynamically change their behaviour based on a a particular literate programming model. The meta-model will need to satisfy a number of goals before this becomes possible.

- Models described by the meta-model must be represented in a computer readable format.
- The model, as represented, must include all the behaviour of the model (no special cases).
- The meta-model must describe structure, processing and the input format. For a command line tool, this will require specifying the syntax of the input file.

Adaptive tools could take several forms. The simplest would be batch processing command line tool, similar to most current tools, while a more complex tool would adapt a user interface to the model.

3.2 Definition

The following sections describe how the meta-model is used to define a literate programming tool. Along with the definition of the model, a running example of a very simple literate programming tool is presented in the boxed paragraphs. The example model supports LaTeX output, is programming language independent and does not do automatic pretty printing or cross-referencing. A UML (Unified Modelling Language)[24] diagram of the example model is shown in Figure 3.1.

The structure of a literate program can be represented as a node-edge graph, as in Figure 2.3. Each chunk is a node, and each link between chunks is an edge. The structure required of the graph by the literate programming tool is represented in the meta-model as an adapted entity-relationship model (as used in the development of a relational database). Each different type of chunk is an entity and each type of link between chunks is a relation.

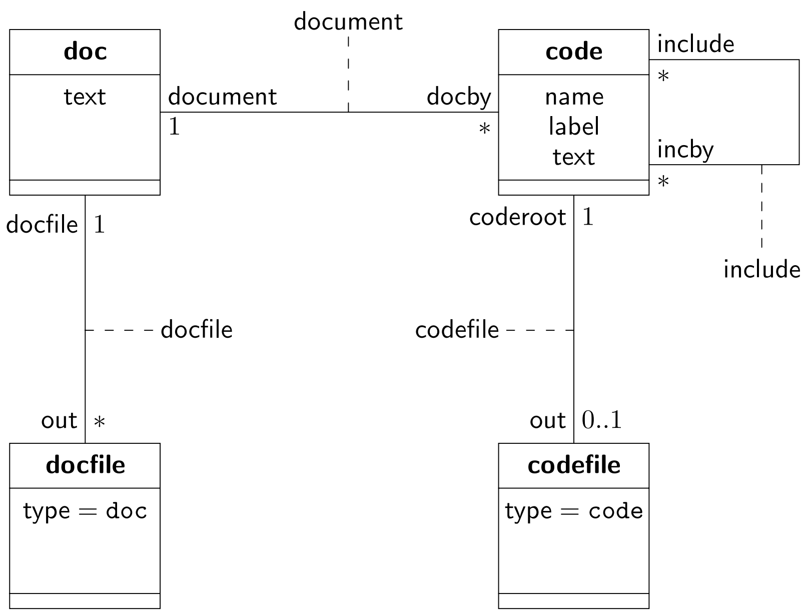


Figure 3.1: UML diagram of the example model presented in this chapter. The names of edges are attached to them via dashed lines.

3.2.1 Chunks

Chunks are the building block of a literate program. The normal types of chunk for a literate program are documentation and code. As an example of where another type of chunk could be added consider how common it is to find comments that start something like "`/* FIXME:` ... " to indicate bugs and missing features. A literate programming tool could store these comments in a separate type of chunk, and either insert them into the documentation or output a list of all such chunks to form a "`TODO`" or "`BUGS`" file. Chunks are also used for storing more abstract entities like variables and functions, or in fact anything else that requires data to be stored.

The meta-model allows for the definition of an arbitrary number of different types of chunks. For each type of chunk, there can be an arbitrary number of instances of it in any literate program. Each type of chunk is given a unique identifier.

For the simplest literate programming model, only two types of chunks are required, code and doc.

3.2.2 Edges

Edges are used to link chunks together. Edges should be added where two different types of chunk are related in some way, for example a documentation chunk relates to the following code chunks because it describes them, and two code chunks can be related because one includes the other. Edges will also connect to the stranger types of chunks, for example a code chunk would be connected to all the variables that it defines and uses (with different edge types for definition and use).

The meta-model allows an arbitrary number of different types of edges, each uniquely named. Each type of edge defines one or more 'ends' that must attach to particular type of chunk and specify a multiplicity for that connection. The multiplicity defines the number of ends of that type that may attach to each chunk. Multiplicity values are specified using the syntax of the UML as described in Table 3.1. Each end will also specify an attribute of the chunk that can be used to access the end from within the chunk. Chunk attributes are described further in Section 3.2.3.

value	meaning
$*$	Any number of connections, including zero.
n	Exactly n connections.
$n..*$	Greater than or equal to n connections.
$n..m$	Between n and m connections, inclusive.

Table 3.1: The meanings of the different multiplicity values

The example requires several different types of edges. The document edge connects a doc chunk with the zero or more code chunks that it describes. The include edge links a code chunk to all the code chunk that it includes.

The alldoc edge is created to link to all the doc chunks, so that they can all be accessed and printed in one step. The coderoot edge is used to identify the code chunk that is the root of the hierarchical structure of code chunks. These edges link to the file output nodes defined in Section 3.2.4.

edge	end	chunk	attribute	multiplicity
document	src	doc	document	$*$
	dst	code	docby	1
include	src	code	include	$*$
	dst	code	incby	$*$
alldoc	src	docfile	out	$*$
	dst	doc	docfile	1
coderoot	src	codefile	out	1
	dst	code	coderoot	0..1

3.2.3 Attributes

The meta-model stores data by adding attributes to chunks. There are three types of attributes: simple attributes, end attributes and complex attributes. Each type of chunk may have any number of each of these attributes, and each attribute must have a unique name within that type of chunk.

Simple attributes store a single textual value. Some of these attributes may be required to have a unique value amongst all chunks of that type, so that, for example, code chunks can have a unique name.

End attributes provide a access point for the edge ends that connect to each chunk. Depending on the multiplicity of the end, these attributes may be single values or lists. These attributes are defined as part of the edges, in Section 3.2.2.

Complex attributes store a piece of text with embedded references to ends of a particular type. This is used, for example, for the text of code chunks, which contain embedded links to other code chunks. The meta-model can allow complex attributes to embed any number of different types of ends, and the specification of the complex attribute will list all such end types.

The simple and complex attributes of each chunk are listed below. The 'end' value will the end that is embedded in a complex attribute or empty for simple attributes. Edge end attributes are described as part of the edges. This example has only a few attributes, including one complex attribute. The label attribute of the code chunk type is used to store a LaTeX cross-reference identifier, which can be any value as long as it is unique.

chunk	attribute	end	unique
code	name	–	✓
	label	–	✓
	text	include	–
doc	text	–	–

3.2.4 Files

A literate program is an interweaving of two or more different files. Each of these files has a type, the most common of which are code and documentation. These output files are represented in the meta-model by file chunks. A file chunk is a particular type of chunk, with two required attributes: type and out. The type attribute stores the user-defined output type for that file, which controls the processing This value will be the same for all instances of the file chunk type. The end attribute is an end attribute whose attached process generates the output file. The multiplicity of this end must be "1".

The simple example model has only two output files, the source code and the documentation. These files are detailed below.

file chunk	type
codefile	code
docfile	doc

3.2.5 Filters

When producing output for most mark-up languages, there will be a number of special characters that are used as directives to the language. These characters will need to be quoted in some way in the output of the literate programming tool, for example in HTML, the characters "`&<>`" need to be quoted as "`&amp;`", "`&lt;`" and "`&gt;`" respectively.

The meta-model provides filters as a generic method for adding these quoting strings, and other simple text transformations. The meta-model allows the for the definition of an arbitrary number of such filters, each with a unique name. Filters are implemented as a set of regular expression patterns and substitution strings. Each regular expression is applied in order on the result of the previous regular expression.

The expression and substitution string pairs are specified in the format used by the `re` module[25] of the Python programming language. The substitution string may contain back-references of the form "`\1`" that are replaced by parts of the string that the parenthesised parts of the regular expression. For example, if the expression "`x(y+)x`" matches the string "`xxxyyx`", the result of "`\1`" would be "`yy`", the part of the string that matched inside the first set of parentheses.

The example model will produce its source code inside a LaTeX `alltt` environment. The only special characters inside such a environment are braces and backslash, i.e. "`{`", "`}`" and "`\`". These characters can be quoted by placing them inside a `verb` command, such as `\verb|\|`.

The filter also provides line wrapping at 70 characters, by inserting a call to the LPwrapline macro. This addition was inspired by noweb, where the lack of line wrapping is an annoying problem. This filter, similar to the rest of the example tool, relies on a custom LaTeX style file to define the macros used.

filter	**pattern**	**substitution**
alltt	`([{}\]+)`	`\verb=\1=`
	`((\verb=..|.){80})`	`\1\LPwrapline{}`

3.2.6 Processes

Processes are used to control the extraction of individual documents from the literate program. A process operates in a hierarchical name-space created from the structural parts of the model. Each chunk has a local symbol table constructed from its attribute names, in which end attributes access the symbol table of the edge that they connect to. The symbol table of an edge contains all of the ends defined for that edge, with each end accessing the chunk it is connected to.

Processes are attached to one end of an edge, handle a single output type and have their initial local name-space set to the symbol table of the edge. When activated, a process will return a string value, which it constructs based on the surrounding chunks and edges.

The behaviour of a process is represented by a block of text. This text may contain references to variables by prefixing them with dollar signs, similar to the syntax used by many programming languages and command shells. These variable references may contain dots to access variables in nested symbol tables, for example the string "`$foo.bar` will access the variable "`bar`" in the nested symbol table of the local variable "`foo`". What a variable access produces will depends on the type of the variable, as described below.

Chunks are illegal to access.

Simple attributes return the value of the attribute.

Ends when accessed from within an edge return the result of any process attached to that end and handling the current output type. If no such process exists, an empty string is returned.

End attributes can have multiple values for one attribute, depending on the multiplicity of the end. Each end will be accessed as described above, and the results concatenated.

Complex attributes have the embedded links accessed as above, and the results joined with the textual parts of the attribute and returned.

Processes may also access filters by name in the same way as local variables. The filter should be followed by a string in parentheses that contains the text to be filtered. The filtered text can contain further substitutions, which will be performed before filtering. For example `$barfilter($foo)` will substitute that value of "`foo`", then filter the result through the filter called "`barfilter`".

Filters can also be used in a slightly different way when accessing complex attributes, by the syntax "`$chunk.complex(filter=barfilter)`". This will cause the filter to be applied only to the plain text parts of the complex attribute, not to the results of the embedded links.

The processes for the simple model are described below. All of these processes are attached to the 'src' end of the edges, implying that edges, at least in the simple model, are only navigated in their 'natural' order. This would not be the case in a more complex model.

The include edges appear different depending on which output is being used. In code output, the complex attribute text is accessed, which will recursively flatten. In documentation output, a string will be inserted that displays the name of the code chunk that is linked to.

When the complex attribute text of the code chunk is printed in the documentation the alternative filtering method is used. This avoids quoting the results of the embedded links.

The dst.document variable used by the alldoc edge has a multiplicity of "$*$". The result of the access is the concatenation of the results of accessing each separate edge end in order.

edge	end	type	process

document	src	doc	`\begin{LPcode}{$dst.name}{$dst.label}`
			`  $dst.text(filter=alltt)`
			`\end{LPcode}`

include	src	doc	`\LPcoderef{$dst.name}{$dst.label}`
		code	`$dst.text`
alldoc	src	doc	`$dst.text` `$dst.document`
coderoot	src	code	`$dst.text`

3.3 Deficiencies and Possible Additions

3.3.1 Process Attachments

The choice of attaching processes to edges rather than chunks was made to enhance the separation of data (stored by chunks) and processing (attached to edges). It appears that this choice may complicate the object oriented implementation of the models, as discussed in Section 4.4.

The aspect that complicates the implementation in particular is that processes are not attached directly to the edge, but rather to one end of that edge. This requires that, to simplify access, each end actually exist as a separate object. If processes were the same for all ends of an edge then separate edge objects would be unnecessary. This simplification will not work, however, for edges that are navigated in both directions. Consider an extension to the example model described earlier that will, for each code chunk, print all the code chunks that include it. This will require following the include edge from dst to src, opposite to the existing processing direction, and producing a different output than the existing process. The easiest way to support different outputs for following an edge in each direction is to have the processes attached to the ends of the edge.

Attaching processes to chunks could simplify the model by allowing edge name-spaces to be removed; chunks would be accessed directly from other chunks. This simplification works well for edges with only two ends, but falls apart when an edge has three ends. With three or more ends, the edge (or some other structure) must remain as an access point to all of the chunks attached to the other ends. This situation has an analogy in database creation, where separate tables are required for relationships that link more then two entities. This does not necessarily preclude having processes attached chunks instead of edges, but it does mean that the static model can not be simplified much because of this change.

3.3.2 Syntax Specification

The meta-model described above does not specify the syntax of the literate program input. Also, the models generated can be so complex that automatically generating a syntax would be very difficult. This problem will also affect tools that attempt to use the meta-model as the basis for a user interface.

This problem could be solved by inserting some type of ‘hints’ into the model, that specify what the file syntax will be. These hints might have to be specific to whatever interface form was required, so that there are different hints for the file format and for the user interface.

3.3.3 Scripts

The process specification as described above can be used for simple tools, but more complex tools will need a more powerful type of process. The mostly likely way for this to be added is via embedded scripts, such as the Javascript used in HTML. The name-space structure of the chunks and edges is already suitable for an easily embedded object oriented language like Javascript or Python. Such a script would be able to adapt its output based on the text, or to perform more complex string manipulations than the processes currently supported by the meta-model.

With the extra processing power of such scripts, it would be useful to be able to pass extra values between scripts. This could be used for passing local control options to other processes, or to implement macros with arguments. One useful variable to be passed to processes would be the input file name, which could be used, for example, to generate LaTeX labels that will not clash with labels generated from different source files.

Scripts could also be used to implement more complicated filters than can be constructed with regular expressions. This could, for example, be used to implement a complex pretty-printing filter that requires a full parser. Filters could also be extended by providing a pipeline interface similar to noweb, so that external programs could perform filtering. The pretty-printing could then be done using a separate program such as lgrind.

3.3.4 Output Options

The only way to produce two different output files in the current meta-model is to have two completely separate output types. Most literate programming tools are not as restricted, they have multiple command line arguments to their tangling and weaving programs, each of which changes the output slightly.

One way of achieving this in the meta-model would be by creating a hierarchy of different output types. This would involve creating a specialised output type code.option1 that inherits from the code output type. While processing, processes that handle code.option1 would be searched for first, but if they are not found, the code process would be used instead. This would allow a model to specify two related outputs without replicating processes.

A related addition that could be useful would be the ability to temporarily change the output document type part way through the extraction of another document. This could be used to include the full text of one output as part of another output. Using this feature might, for example, attach the ‘read-me’ file for a system to its technical documentation without requiring new edges or processes.

Chapter 4

An Application of the Meta-Model

In theory, there is no difference between theory and practice; In practice, there is.

—Chuck Reid

4.1 Steps for Using the Meta-Model

'Using' the meta-model involves moving from layer 1 to layer 2, according to the three layer architecture described in Section 3.1.1. The meta-model is used to define a new model, which in turn is implemented as a new literate programming tool. Instances of the new model are literate programs for the new tool.

Correct use of the meta-model is important for creating a literate programming tool that is consistent, complete and understandable. The following steps describe the method used for creating a new literate programming model. Many of these steps can be either simplified or skipped if the tool is being based on an existing tool. This tool could be meta-model based, but could also be any other tool whose underlying model is easy to extract. The example model, described in Section 3.2 and in the following section of this chapter, was based on a simplification of noweb.

1. Before starting, construct a basic model of what you want your literate programming tool to do.

2. Work out what data your tool will deal with. Each data item will become an attribute, and related data will be grouped into a chunk. Remember that chunks are often closely related to user's input, but that chunks will also be needed for more abstract entities like variables that are referenced from multiple places.

3. Determine exactly what files will be produced by your tool, and create file chunk definitions for them.

4. Identify the static relationships between your chunks, and create edges for them. In most cases edges will have only two ends, but more ends

might be useful in some situations. Determining multiplicities can be complicated, and should you should try not to restrict the structure unnecessarily. Don't forget to link to your file chunks.

5. Create an concrete example of what each output will look like. This could include creating LaTeX style files or similar constructs if appropriate.

6. Specify the filters that your outputs will use. Make sure that each stage of the filter will not overwrite substitutions made by the previous stages.

7. Add processes to your ends. This is the most complex step, and may require shuffling processes between edges to get the correct results. If required, add edges that have only a single end, which can be used to attach processes indirectly to a chunk. Make sure that you deal with multiplicities correctly.

8. Create an implementation of your model, making it as flexible as possible in case you need to modify pats of the model later. This step could theoretically be done by an automated tool, or could be ignored completely if an adaptive tool as described in Section 3.1.4 was available.

9. Test the model, and assess its usability, adapting it until the correct output is produced. It should be possible to add extra features later without trouble.

4.2 Implementation

As a basic test of the quality and applicability of the meta-model, the simple model presented as an example in Chapter 3 was implemented. While this model is particularly simple, it serves to demonstrate the usefulness or otherwise of the meta-model. The test implementation was made in Python, a dynamic object oriented language with an emphasis on simplicity. During implementation, an effort was made to follow the structure of the example model and the meta-model as closely as possible.

Several of the classes that implement the model have a `__call__` method. This method has a special meaning in the Python language, such that for instances of classes defining it, "`foo( ... )`" is translated to "`foo.__call__( ... )`". This is used to make parts of the code appear simpler. For example, filters use this method, so that while they are actually objects they look like functions when they are called.

A description of the implementation details is given in the following sections. Each section starts with a repetition of the appropriate part of the example model presented in Chapter 3. Also, refer back to the UML diagram of this model in Figure 3.1.

4.2.1 Behaviour of the Tool

The example model implements a literate programming tool which is based on noweb, but simplified.

As in traditional literate programming, there will be two types of chunks, documentation and code. The tool does not support variable and function cross-referencing so no extra chunks will be needed for these artifacts. The set of edges is also based on standard literate programming, a documentation chunk is linked to all the code chunks it documents and a code chunk is linked to the other code chunks that it includes. The documentation chunk will need only one attribute, to store the documentation text. The code chunk will need two unique attributes, the name as chosen by the programmer and an internally generated label for use by LaTeX, as well as a complex attribute that will store source code along with links to other code chunks.

Other edges are required to link the file chunks for documentation and code files. The code file chunk will link to a single root chunk (the chunk named "*" in noweb), and the documentation file chunk will link to all documentation chunks, in order.

The appearance of the output will also follow closely to noweb (see Figure 2.2), although the details will be controlled by a LaTeX style file rather than directly by the tool. The model is able to support multiple code output files, but the initial implementation will support having only one.

4.2.2 Chunks and Attributes

chunk	attribute	end	unique
code	name	–	✓
	label	–	✓
	text	include	–
doc	text	–	–

Chunks are very simple to implement as they are data stores only, rather than active elements. Attributes are stored in instance variables, with simple attributes initialised to a null value, and end attributes initialised to either a null value or to an empty list depending on the multiplicity of the end. The code to perform the initialisation was placed in a `Chunk` superclass, which reads the attribute names and multiplicities set by each inheriting class. Each type of chunk is then created as a subclass of `Chunk`, which needs only to set the `_attributes` and `_ends` class variables. The class declarations for the two chunk types are shown below.

Complex attributes were implemented as a separate class. This class includes a `__call__` method to produce the output of of that attribute for a particular output type, and with an optional filter. The class is reasonably simple but not simple enough to be initialised automatically, so chunks using these attributes will have them added directly by the parser.

```
class Doc(Chunk):
    _attributes = ('text',)
    _ends = (('document', 0, '*'), ('docfile', 1, 1))

class Code(Chunk):
   _attributes = ('name', 'label', 'text')
   _ends = (('include', 0, '*'), ('docby', 1, 1),
            ('coderoot', 0, 1), ('incby', 0, '*'))
```

4.2.3 Files

file chunk	type
codefile	code
docfile	doc

Files are created as subclasses of a `File` class that is itself a subclass of `Chunk`. The file superclass expects its subclasses to define a class variable `type` to the output type, and to define an `out` attribute. It also provides a method that will start the extraction of the file by accessing the `out` attribute. The class declarations for the two types of file chunk are shown below.

```
class Codefile(File):
    _extractend = 'root'
    _ends = (('out', 1, 1),)
    type = 'code'

class Docfile(File):
    _extractend = 'alldoc'
    _ends = (('out', 0, '*'),)
    type = 'doc'
```

4.2.4 Filters

filter	pattern	substitution
alltt	`([{}\]+)`	`\verb=\1=`
	`((\verb=..|.){80})`	`\1\LPwrapline{}`

Filters are implemented by a `Filter` class that uses Python's `re` regular expression module. Each filter is created as an instance of this class, and will specify its own list of expression/substitution pairs, the same as the meta-model specification. The code that creates the alltt filter is shown below.

```
alltt = Filter([
    (r'([{}\\]+)', r'\\verb=\1='),
    (r'((\\verb=..|.){70})', r'\1\\LPwrapline{}'),
    ])
```

4.2.5 Edges and Processes

edge	end	chunk	attribute	multiplicity
document	src	doc	document	$*$
	dst	code	docby	1
include	src	code	include	$*$
	dst	code	incby	$*$
alldoc	src	docfile	out	$*$
	dst	doc	docfile	1
coderoot	src	codefile	out	1
	dst	code	coderoot	0..1

edge	end	type	process

document	src	doc	`\begin{LPcode}{$dst.name}{$dst.label}`
			`  $dst.text(filter=alltt)`
			`\end{LPcode}`

include	src	doc	`\LPcoderef{$dst.name}{$dst.label}`
		code	`$dst.text`
alldoc	src	doc	`$dst.text` `$dst.document`
coderoot	src	code	`$dst.text`

The edge/end/process conglomeration is the most complex part of the meta-model, and not surprising it generates the most complex code. Unlike the simpler parts of the meta-model, it was not realistic to have the major processing of this part hidden behind a single class that conforms exactly to the meta-model. Instead, similar but specialised class were written for each of the edge types defined in the example model.

The name-space system used by the meta-model was implemented using Python's name-spaces. This allows the code used in processes to be quite close to the meta-model specification. As processes run in the name-space of the edge, it made sense for them to be methods of each edge. To connect processes to the ends of an edge, a stub class End was created that passes attribute access back to the containing edge, but when called will choose the correct method of the containing edge to run as the 'process' that is attached to that end. Each process takes a single argument that specifies the current output document type, and will return its result as a string.

```
class Document(Edge):
   def __init__(self, src, dst):
      self.src = src
      self.dst = dst
      self.src.document.append(End(self, self.process_src))
      self.dst.docby = End(self, None)

   def process_src(self, type):
      if type == 'doc':
         return '\\begin{LPcode}{%s}{%s}\n%s\n\\end{LPcode}' \
                % (self.dst.name, self.dst.label,
                   self.dst.text(type, filter=filters.alltt))
```

The code above implements the document edge for example model. When created the edge takes its source and destination chunks as arguments, and is responsible for attaching itself to them. The attached chunks are stored as attributes of the instance in accordance with the name-spaces of the meta-model.

The chunk instances do not contain references directly, they store instances of the End class. The End instance is initialised with two arguments, the first is the edge instance that the end is part of, and the second argument specifies a function that is the process to be attached to that end, or None if there is no process attached to that edge. When attributes of the end are accessed it will pass these accesses back to its edge, so that they will appear to be the same object. The only method that the end implements itself is the `__call__` method, which will activate the process attached to that end, returning its output. This allows the end to provide a transparent interface to the edge instance while still activating the correct process when called.

The method `process_src` of the above class implements the process attached to the src end of the document edge. This process first checks what output document edge is active, and proceeds only for the doc output type. The string returned is constructed according to definition provided by the meta-model. The inserted values are the name and label attributes of the dst code chunk, and the output of the text complex attribute for the current output type and using the alltt filter.

4.2.6 The Parser

As discussed in Section 3.3.2, the meta-model does not provide a description of the syntax of the literate program. This makes it impossible to implement a parser that conforms to the meta-model. Instead, the parser is a simplification of the syntax of noweb. The structures generated by the parser do, however, conform to the example model.

"<<*str*>>=" starts a code chunk with the name *str*, which includes all the text up to the start of the next chunk, excluding leading and trailing blank lines. The label attribute of the code chunk is generated automatically.

"<<*str*>>", when used inside a code chunk, will be replaced by a reference to the chunk with name *str*.

"@", at the start of a line and followed by whitespace, starts of documentation chunk. The text attribute of the chunk will contain all the text up to the start of the next chunk, excluding leading and trailing blank lines.

4.3 An Example of Use

To demonstrate the basic behaviour of the new tool, a trivial C program was written using it. The input and outputs of the new tool for this example program are in Figure 4.2, and a diagram of the structure graph is shown in Figure 4.1.

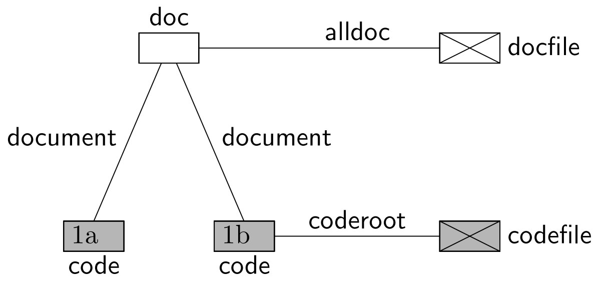


Figure 4.1: The chunk and edge structure of the trivial example program.

```
This file produces a trivial demonstration of the new literate
programming tool.  The code is a simple hello world program.
The program is made out of two chunks, the chunk that actually
does the printing and the standard C main function.

<<body of program>>=
printf("Hello World!\n");

<<*>>=
int main(void) {
  <<body of program>>
  return 0;
}
```

(a) Input

```
int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

(b) Code output

This file produces a trivial demonstration of the new literate programming tool. The code is a simple hello world program. The program is made out of two chunks, thebit that actaully does the printning and the standard C mai function.

1A ⟨*body of program*⟩≡

```
printf("Hello World!\n");
```

1B ⟨***⟩≡

```
int main(void) {
  ⟨body of program 1A⟩
  return 0;
}
```

(c) Documentation output

Figure 4.2: The input and output files of a trivial literate program with the new tool.

4.4 Problems

4.4.1 Unnecessary Objects

Creating the implementation to be as close to the meta-model as possible resulted in a system that has more objects than necessary. This is most obvious problem is requiring that a separate object be used for each end of each edge. If these objects were discarded somehow the total amount of memory and time used could probably be reduced significantly.

The problem with removing these objects is that the implementation moves further from the meta-model, increasing the complexity of actually implementing each new model. To go back to the above example, discarding the end objects would complicate the implementation of the processes. Currently, where the model specifies a process something like "`$dst.outend`" the implementation of the process will contain "`self.dst.outend(type)`", which only works because the end object runs the correct method of the edge. If the end objects were removed, the process implementation would have the correct method hard-coded, producing something like "`self.dst.outend.fromsrc(type)`".

Basically, simplifying one aspect seems to only make another more complex. That leaves two choices, accept the extra complexity at some point or modify the meta-model so that it produces models that are easier to implement.

4.4.2 Language Portability

Implementing the example model using Python was not particularly difficult, even when the simplicity of the model was taken into account. An implementation in a different programming language may not be as simple or as elegant.

The implementation made use of the dynamic features of Python, for example by having the `Chunk` superclass add attributes based on a list of strings defined by its subclasses. The use of these features makes it easier to adapt the implementation to a different model, as the amount of model-specific code is kept to a minimum. However, this dynamic behaviour is not available on other common languages, such as C and Java, where the implementation is likely to be more complex to create, but will run faster and use less memory.

4.5 Comparison

The example model that was implemented was always designed to be very simple, so will never compare favourably to a real literate programming tool. In basic structure, the model matches noweb but there are a few of the missing features, as described below.

4.5.1 Processing of Documentation Chunks

Most literate programming tools also apply filters to the documentation chunks. In noweb, text between double square brackets (`[[ ... ]]`) is typeset as if it was in a code chunk, quoting all the LaTeX special characters. This behaviour could be added quite easily by defining a new filter and apply it when outputting the text attribute of the doc chunks.

The other feature provided by many literate programming tools is the ability to embed references to code chunks inside the documentation. This would require making the text attribute of the doc chunk type a complex attribute, and defining a new edge (something like refer) to link doc chunks to code chunks.

4.5.2 Pretty Printing

Pretty printing does not exist in standard noweb, and can be thought of as an optional part of a literate programming tool. A basic pretty printer could probably be implemented using the current filter system, but with extreme difficulty due to the difficulty of creating such complex regular expressions. If pretty printing is to be well supported by the meta-model a new, more powerful, filtering mechanism will be required.

4.5.3 Cross-Referencing

The current tool has no support for program variable cross-references. This could be added by creating a chunk type to represent the variable, and edges to represent variable definition and use. In the current scheme, these links would have to be added by the parser; the meta-model would not specify how the variables were identified. The way to incorporate this information into the meta-model would be to create 'active' filters, that, as well as altering text can actually create new chunks and edges. These active filters could even be used to construct a complete parser that would be part of the meta-model.

Chapter 5

Conclusion

Absolutely nothing should be concluded from these figures except that no conclusion can be drawn from them.

—Joseph L. Brothers, *Linux/PowerPC Project*

A review of current literate programming tools was used to gain an understanding of literate programming, and of the data and processes models involved. This review found a number of required and optional features for literate programming tools.

Based on the structures and processes identified a meta-model system was created that can be used to specify models for literate programming. This system views a literate program as a number of different types of chunks with various connections between them. A literate programming model is a set of constraint rules for this graph and a set of simple processes that attach to the edges of the graph. The meta-model consists of a system for specifying these constraints and processes. A methodology was outlined that gives a set of steps to follow when constructing a new model.

A simple literate programming tool was implemented to test the meta-model. This tool conformed to the minimum rules for a literate programming tool as discovered in the review. The Python implementation of this new tool was reasonably straightforward, and was written in such a way as to make it easy to modify if the model was changed. Importantly, the implementation was also a functioning and usable literate programming tool. This demonstrates that the meta-model functions as a way of specifying and implementing new literate programming tools.

5.1 Further Work

The models created by the meta-model do not include a specification of input file syntax. This is partly so that they are just as applicable to a system that does not work from input files. For example, a graphical editing tool will be able to supply reconstructed object to the model. What is required is a set of accompanying systems that can generate the model sets that the literate programming models use, either from an input file or other sources.

Although quite functional, the meta-model is still only a prototype version, and it presents many opportunities for improvement. The process specification is currently very simple. More powerful processes could be added using scripts that work the same way as Javascript functions within HTML. Another way to add more powerful processes would be via a pipeline interface to external formatting programs.

The meta-model can be used to explore the limits of the literate programming concept. By allowing fast implementation of extended tools,the meta-model can reduce the time required for testing new ideas.

Bibliography

[1] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[2] Donald E. Knuth. The WEB system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.

[3] Donald E. Knuth. *METAFONT: The Program*, volume D of *Computers & Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[4] Donald E. Knuth. *TEX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[5] Akim Demaille and Miguel Santana. a2ps version 4.12c. Computer program, available online from `http://www.inf.enst.fr/~demaille/a2ps/`, September 1999.

[6] Michael Piefel. lgrind version 3.6. Online available from CTAN: `support/lgrind/`, May 1999.

[7] Jon Bentley. Programming pearls—literate programming. *Communications of the Association for Computing Machinery*, 29(5):364–369, May 1986.

[8] Jon Bentley, Donald E. Knuth, and Doug McIlroy. Programming pearls—A literate program. *Communications of the Association for Computing Machinery*, 29(6):471–483, June 1986.

[9] David R. Hanson. Literate programming—printing common words. *Communications of the Association for Computing Machinery*, 30(7):594–599, July 1987.

[10] Jon Bentley and David Gries. Programming pearls—abstract data types. *Communications of the Association for Computing Machinery*, 30(4):284–290, April 1987.

[11] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0.* Addison-Wesley, Reading, MA, USA, 1993.

[12] Donald E. Knuth. Stanford GraphBase: A platform for combinatorial algorithms. In ACM-SIAM-DA4 [26], pages 41–43.

[13] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press, New York, NY 10036, USA, 1993.

[14] Eric W. van Ammers and Mark R. Kramer. The CLiP style of literate programming. Online available from CTAN: `/web/clip/clip_style.ps`. Version 4.074, 26-feb-93.

[15] Preston Briggs. Nuweb, A simple literate programming tool. `cs.rice.edu:/public/preston`, Rice University, Houston, TX, USA, 1993.

[16] Norman Ramsey. Literate programming tools need not be complex. Technical report CS-TR-351-91, Princeton University, Dept. of Computer Science, Princeton, NJ, USA, October 1991.

[17] Felix Gaertner. Pretzel version 2.0. Computer program, available online from `http://www.iti.informatik.th-darmstadt.de/~gaertner/pretzel/`, December 1996.

[18] Ross Williams. FunnelWeb user's manual. `ftp.adelaide.edu.au` in `/pub/compression` and `/pub/funnelweb`, University of Adelaide, Adelaide, South Australia, Australia, 1992.

[19] Norman Ramsey. Weaving a language-independent WEB. *Communications of the Association for Computing Machinery*, 32(9):1051–1055, September 1989.

[20] M. Knasmueller. Reverse literate programming. In Samson et al. [27], pages 97–105.

[21] Trygve Reenskaug and Anne Lise Skaar. An environment for literate Smalltalk programming. *ACM SIGPLAN Notices*, 24(10):337–345, October 1989.

[22] Marcus E. Brown. *An Interactive Environment for Literate Programming.* Thesis (ph.d.), Texas A&M University, College Station, TX, USA, August 1988.

[23] Eitan Gurari. *TEXand LATEX: Drawing and Literate Programming.* McGraw-Hill, 1994.

[24] James Rumbaugh, Grady Booch, and Ivar Jacobson. *Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, 1999.

[25] Guido van Rossum. *Python Library Reference.* Corporation for National Research Initiatives (CNRI), 1895 Preston White Drive, Reston, Va 20191, USA, 1.5.2 edition, july 1999. Section 4.2, `re` module.

[26] *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, New York, NY 10036, USA, 1993. ACM Press.

[27] W. B. Samson, I. M. Marshall, and D. G. Edgar-Nevill, editors. *Proceedings of the 5th Software Quality Conference: 9 and 10 July 1996, Dudhope Castle, University of Abertay Dundee, Business School, Dundee, Scotland, UK*, Dundee, Scotland, 1996. University of Abertay Dundee.