DocBook-Based Literate Programming

1.4

Mark Wroth

April 12, 2001

Contents

Chapter 1

Functional Description

The DocBook-based Literate Programming system provides a mechanism to write literate programs using a minor extension of the Standard Generalized Markup Language (SGML) DocBook Document Type Definition, Document Type Declaration (DTD).

The system consists of two main parts:

- A DTD that extends DocBook to add the logic needed for literate programming. These are relatively minor extensions to the basic DTD. The details are discussed in Chapter 3.
- Document Style Semantics Specification Language (DSSSL) style sheets that, together with a DSSSL engine that implements some of James Clark's extensions, serve as "weave" and "tangle" processors. These style sheets are discussed in Chapters 5 and 4, respectively.

This document also discusses the design considerations behind the implementation, and provides a short sample document that serves as an example of how the DTD is used (and serves as a simple test case).

1.1 Purpose of the Functional Description

This functional description for "DocBook-based Literate Programming" is written to provide:

1. The system requirements to be satisfied which will serve as a basis for the system design.
2. Information on performance requirements, preliminary design considerations, and user impacts including fixed and continuing costs.
3. A basis for development of system tests.

1.2 Project References

Documents significant to this project are discussed in the bibliography.

Chapter 2

System Summary

2.1 Background

2.1.1 Literate Programming

Literate programming is a style of computer programming pioneered by Professor Donald Knuth in the early 1980's (the defining paper was published in *The Computer Journal* in May 1984, at which point the earliest literate programming system, WEB, was already functional).

The key tenent of literate programming is that computer programs are written to be read and understood by *human beings* as well as computers—and the organization of the program's source code should allow the program author to explain the purpose and implementation of the code to the human audience. In Knuth's own words, "Instead of imagining that our main 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."[1]

To this end, a literate programming system typically supports several functions:

- Mechanisms to extract or otherwise make available to the computer the code instructions making up the computer program in a form usable by the computer (e.g. in a form suitable for submission to the compiler), and to translate the source code and documentation into appropriately rendered documentation.
- The ability to write code documentation using the full range of typographic features used in normal typesetting.
- The ability to arbitrarily intermingle code and documentation.

[1] "Literate Programming," originally published in *The Computer Journal*, May 1984, quoted in [2, p. 99]

- The ability to arbitrarily reorder code fragments so that the exposition of the program design may precede in an order appropriate for the human reader, while retaining the instruction order required for the computer to correctly execute the program.

The process of making the code instructions available to the computer is called the "tangle" phase of processing, while the process of rendering the documentation is called the "weave" phase. These names derive from the names of the two programs making up the original WEB system that performed these functions.

Some literate programming processors, including Knuth's original WEB system, additionally "pretty print" the programming language instructions (for example, by setting the reserved words of the language in boldface type). This pretty print capability is not universal, and in fact is not desired by some programmers. Another capability sometimes found is the ability to define macros in the literate programming system, usually to supplement the capabilities of the underlying programming language.

Since the introduction of Knuth's WEB system, which was used for the TEX and METAFONT programs, a variety of other literate programming systems have appeared, as have a number of other mechanisms for improved commenting of source code[2]. A complete review of the available systems is beyond the scope of this discussion. However, experience with the original WEB system, John Krommes' FWEB, and Preston Brigg's NUWEB literate programming systems, and the DOC LATEX 2_ε documentation system and Normal Walsh's DocBook-based DSSSL style sheet documentation systems (both of which may be characterized as improved commenting systems) has been a significant factor in defining the characteristics desired in this system.

2.1.2 SGML and XML

The Standard Generalized Markup Language is defined in International Organization for Standardization (ISO) Standard ISO 8879:1986, and defines a mechanism for *defining* markup languages and enforcing certain relationships among the data contained in appropriately marked up documents. Among other things, SGML provides a clear mechanism for making explicit what parts of a document have what role, and does so in a way that encourages the construction of tools able to parse such documents—and hopefully do useful things with the parsed information.

In the late 1990's SGML was supplemented by Extensible Markup Language (XML), which is a simplified SGML application designed to make it easier to construct parsers and other processing tools. In general, XML has the same basic functionality as SGML, with some of the lesser-used features of the language omitted. Tools that can process SGML documents can also usually

[2]The boundary between "literate programming" systems and "improved commenting" mechanisms is somewhat subjective. However, for the purposes of this discussion, a system is considered a literate programming system if it offers the capabilities listed above.

process XML documents; it is not generally the case than an XML tool can process a general SGML document.

2.1.3 DocBook

One of the useful features of SGML is the ability to create DTDs that describe the structure of documents and the markup that makes that structure explicit. This permits the creation of general document types—and the tools to process them.

One such document type is DocBook, a document type created for computer-oriented technical books. DocBook is maintained by Organization for the Advancement of Structured Information Standards (OASIS), and has evolved into a flexible and robust document definition, supported by a variety of tools. DocBook exists in both SGML and XML versions.

In particular, Norman Walsh has defined a set of DSSSL style sheets that process DocBook documents and produce printed (or Hypertext Markup Language (HTML)) output renderings. These style sheets are both extensible and customizable, and serve as a significant base for computer-oriented documentation—such as a literate program.

2.2 Objectives

This project creates a set of extensions to the DocBook SGML DTD to allow its use for literate programming markup. The resulting system shall

- Provide a mechanism to extract program files from the literate programming source in appropriate forms for their use as source code in the intended programming language or languages.
- Permit the use of existing DocBook-based tools with only minor modifications (ideally none) to produce documentation of software projects.

2.3 Existing Methods and Procedures

There are a variety of literate programming systems in use at the current time. In general, they fall into three main categories:

- Language-aware systems. These systems are designed to support a single computer programming language, and are marked by the ability to do limited parsing of code sections, usually accompanied by "pretty printing" of the computer source code. Knuth's original WEB system falls into this category. Most language aware systems use TEX as their typesetting system.

- Language-independent systems. These systems attempt no parsing of the code sections. Most language-independent systems use TEX as their typesetting systems, although there is some move towards HTML as a documentation language.

- Comment-based systems. These systems extend the comment structures of the supported language in an attempt to provide usable documentation. Examples of such systems are the "doc" system used to document LATEX 2_ε packages, and—I believe—the "Javadoc" system.

Most or all of these existing systems target a specific output format, usually the printed page, rendered via the TEX typesetting system. In part, this is a historical accident; the first literate programming system used TEX. However, attempts to implement other documentation languages (notably an attempt to write a 'C' language literate programming system with TROFF as the documentation processor) indicate that the demands on the documentation branch of a literate programming system are relatively taxing. This has undoubtedly contributed to the limited number of output forms supported, although some attempts to support HTML have been made.

However, I am not aware of any literate programming systems based on SGML markup of the source code. This omission seems unfortunate, given the obvious applicability of SGML markup to the process of defining a literate program.

2.4 Proposed Methods and Procedures

This project implements an SGML markup-based literate programming system. Literate computer programs are written using some form of text editor—preferably, but not necessarily an SGML-aware editor. The documentation and programming language code is marked up using an extension of the DocBook DTD. Once the literate program is written (partially or completely), it is processed using a variant of the Jade DSSSL engine with either a "tangle" style-sheet (which is a stand-alone style sheet provided by this project), or a "weave" style sheet (which extends the DocBook Modular Style Sheets in several small but important ways).

2.4.1 Summary of Improvements

Creating an SGML-based literate programming system makes it possible to exploit the wide variety of SGML (or, with minor variations, XML) tools. In particular, this makes it straightforward to produce different output formats, such as a printed version or an HTML version.

The use of SGML also separates the definition of the document markup from the definition of the processing tools. In principle, this allows many different tools to be used with literate programs written with this system. This advantage is largely theoretical at this point, however.

2.4.2 Summary of Impacts

2.4.3 Assumptions and Constraints

It is assumed that the user of this system is familiar with the use of SGML-based tools, and the DocBook DTD.

The processing DSSSL style sheets assume the presence of selected DSSSL extensions implmented in James Clark's Jade engine (specifically the "entity" and "processing-instruction" flow objects). This capability is necessary to use the specific style sheets provided by this project, but the use of the DTD is not affected.

2.5 Detailed Characteristics

2.6 Design Considerations

- Minimize changes to the DocBook DTD to allow processing using existing output tools.
- Keep the existing `<programlisting>` as the basis for a scrap (again to minimize changes needed in the output processors).
- Use added attributes for file output, definition and reference to continued and continuation scraps. Use of both continued and continuation markup is semantically redundant, but will make processing of the "weave" branch easier—I think.
- Use `<xref>` for reference to definition scraps, and the `xreflabel` attribute for the definition scrap title. Again, this is driven by a desire to minimize changes that would impact the output processing tools.
- Make maintaining, modifying, and adding this functionality to other document types as easy as possible (see Section 2.6.3).

2.6.1 System Description

2.6.2 System Functions

This system performs three basic functions:

- Provide a DTD that allows the markup of literate programs, including a flexible system for describing the purpose and implementation of the computer program (based on DocBook) and markup of the program code itself to allow the literate program to produce the computer instructions.
- A tangle mechanism that actually produces the computer instructions from the literate programming source code. This implementation, `SGMLTangle.dsl` is a DSSSL style sheet using extensions to the DSSSL standard as implemented in James Clark's Jade DSSSL engine.

- A WEAVE implementation that renders the literate programming source into useful documentation. This style specification, `SGMLWeave.dsl`, extends Norman Walsh's *Modular DocBook Style Sheets*. It provides both print and HTML output, in the style sheets `print` and `html` respectively.

2.6.3 Flexibility

It is the intention of this system to:

- Maintain the ability to update the extensions to new versions of DocBook as they are published.
- Make the extensions as easy as possible to to move between the SGML and XML versions.
- Make it as simple as possible to add the literate programming functionality to other DocBook-based DTDs.
- Provide a basis on which other implementations of the TANGLE and WEAVE functions could be built to support other tool chains.

2.7 Environment

2.7.1 Equipment Environment

The hardware required to use this system is defined by the selected software tools (in particular the SGML processor). Development and testing was accomplished on 32-bit Windows-based computers.

2.7.2 Support Software Environment

Effective use of this system requires three major classes of supporting software. Except as noted with regard to the TANGLE application, it is not necessary that the actual supporting software used in the development of this system be available.

Programming Editor

Some form of text editor is needed to write the literate program. The minimum necessary functionality is the ability to write a plain text output file.[3]

The authoring process will be much easier, however, if the editor supports both SGML markup and the programming language or languages in which the code is being written. A customizable editor such as EMACS is probably a useful choice.

[3] Actually, even this is an overstatement: the editor has to be able to produce SGML input files compatible with the SGML and DSSSL processors being used. While this is *usually* a plain text file, there are other ways to implment SGML systems.

SGML Processors

The literate programming DTD extends the DocBook DTD, and therefore requires the underlying DTD. This implementation specifically uses the SGML DTD, Version 4.1 as the basis for its extension.

To use this system for literate programming without additional customization, a DSSSL engine that implements James Clark's `entity` and `processing-instruction` extensions to the DSSSL standard is needed. The Jade or OpenJade engines meet this requirement.

The weave style sheet is based on Norman Walsh's *Modular DocBook Style Sheets* [5], which must be available if the `SGMLWeave.dsl` style specification is used.

As an SGML application, of course, documents marked up with this DTD can, in principle, be used by any SGML-compliant tool.

While this DTD and the associated DSSSL style sheets were written for the SGML version of the DocBook DTD, it should require only minor changes to re-implement this system in XML. This has not, however, been tested.

2.8 System Development Plan

This system has been implemented using the Nuweb literate programming processor, which is a TEX-based system. The choice to implement the system in this way was a "bootstrapping" choice; until this system achieved basic functionality, it would be difficult to implement the system in an SGML-based system.

Because the original implementation was written using Nuweb, basic maintenance of the system is currently intended to continue in that system. Eventually, the system may be translated into itself, but this is not currently planned.

Once basic functionality has been demonstrated, this system will be made publically available by publication on the World-Wide Web. Future modifications and extensions may or may not be made, depending on the author's use of the system and feedback from other users (if any).

Chapter 3

DTD Implementation

3.1 Purpose

3.2 Top Level Organization

"dblp.dtd" 13a ≡

```
<!--
DBLP.DTD; a literate programming DTD based on DocBook
PUBLIC
"-//Mark Wroth//DTD DocBook V4.1-Based Extension Literate Programming 1.0//EN"
-->
⟨Add <programlisting> attributes 14a⟩
<!ENTITY % programlisting.element "IGNORE">
<!ENTITY % docbook PUBLIC "-//OASIS//DTD DocBook V4.1//EN">
%docbook;
⟨Redefine the <programlisting> element 13b⟩
⟨The 'literalchar' element 14b⟩
```

3.3 The programlisting Customization

We will set up the <programlisting> element to ignore the DocBook defined definition and substitute our own.

⟨Redefine the <programlisting> element 13b⟩ ≡

```
<!ELEMENT programlisting - -
  ((CO | LineAnnotation | literalchar | %para.char.mix;)+)>
```

Macro referenced in scrap 13a.

To the attribute list, we add the attributes that we will use for literate programming:

file The file name the code is to be written to. This attribute is required for the scraps which begin output files.

continuedfrom The ID of the scrap this scrap continues.

continuedin The ID of the scrap this file continues.

We also make use of several of the attributes already defined for programlisting:

ID Unique identifier for this scrap; required for scraps which are either continued or continue others.

xreflabel The title of the scrap; used for the head of a definition scrap.

⟨Add `<programlisting>` attributes 14a⟩ ≡

```
<!ENTITY % local.programlisting.attrib "
 file          CDATA  #IMPLIED -- file name for output file --
 continuedfrom IDREF  #IMPLIED
 continuedin   IDREF  #IMPLIED ">
```

Macro referenced in scrap 13a.

The choice to use both a `continuedfrom` and a `continuedin` attribute allows the creation of a doubly-linked list for each code section. This permits the programmer to order the code scraps in any desired sequence, while permitting the processing system to easily traverse the scraps that make up the section.

While one or the other direction of the links between scraps is sematically redundant in an absolute sense (with adequate effort, the backward links could be constructed from the forward set, and vice-versa), including both sets of links makes construction of the processing tools much simpler.

3.3.1 The 'literalchar' element

The following definitions are used to provide a workaround to get an actual "less than" character into the SGML output. Since the character has syntactic meaning to the SGML parser, by default it is 'escaped' when placed in the SGML output as character data.

By defining an element to contain the required information, we let the DSSSL processor have access to it. Defining entity references to it simplifies the actual data entry. If particular combinations seem appropriate for a specific programming language it would make sense to define entities which make syntactic sense. This would allow one to use, for example `&logicaland;` instead of `&amp;amp;`[1].

⟨The 'literalchar' element 14b⟩ ≡

[1] The basic suggestion to use a formatting-instruction to address the problem came from David Carlisle `davidc@nag.co.uk` in a post to the DSSSList, Vol 3, Number 241, although the actual implementation does not follow his suggestions precisely.

```
<!ELEMENT literalchar  - o EMPTY
       -- literal data, to be handled in the DSSSL -->
<!ATTLIST literalchar data CDATA #REQUIRED>
<!ENTITY  lessthan  "<literalchar data='&#60;'>"
       -- ``less than'' sign-->
<!ENTITY  greaterthan  "<literalchar data='>'>"
       -- ``greater than'' sign-->
<!ENTITY  ampersand "<literalchar data='&#38;'>"
       -- ``ampersand'' sign-->
```

Macro referenced in scrap 13a.

Chapter 4

SGML Tangle

4.1 Purpose

The SGMLTangle style sheet performs the "tangle" phase of literate programming. In other words, it takes the literate programming code and rearranges it so that it is acceptable to the computer as a computer program (assuming, of course, that the programmer has correctly written the program!)

4.2 Implementation

In order to make it as simple as possible to bring this system up on a new machine, this style sheet is presented as a single scrap. Notes on the program follow the implementation scrap.

"SGMLTangle.dsl" 16 ≡

```
<!DOCTYPE style-sheet
  PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN">
<style-sheet>
<style-specification
 id = "tangle">
<style-specification-body>

 (declare-flow-object-class entity
   "UNREGISTERED::James Clark//Flow Object Class::entity")
 (declare-flow-object-class formatting-instruction
   "UNREGISTERED::James Clark//Flow Object Class::formatting-instruction")

 (default (process-node-list (element-children)))

 (element programlisting
  (make sequence
   (if (attribute-string "file")
```

```
    (make entity
      system-id: (attribute-string "file")
      (make sequence
        (process-children)
        (if (attribute-string "continuedin")
          (with-mode continuation
            (process-element-with-id
              (attribute-string "continuedin")))
          (empty-sosofo))))
     (empty-sosofo))))

(element (programlisting xref)
  (with-mode definition
    (process-element-with-id
      (attribute-string "linkend"))))

(mode definition
  (element programlisting
    (make sequence
      (process-children)
      (if (attribute-string "continuedin")
          (with-mode continuation
            (process-element-with-id
              (attribute-string "continuedin")))
          (empty-sosofo)))))

(mode continuation
  (element programlisting
    (make sequence
      (process-children)
      (if (attribute-string "continuedin")
          (with-mode continuation
            (process-element-with-id
              (attribute-string "continuedin")))
          (empty-sosofo)))))

(element literalchar
 (make sequence
  (make formatting-instruction
    data: (attribute-string "data"))))
```

⟨Define element-children function 18⟩

```
</style-specification-body>
</style-specification>
</style-sheet>
```

`(element-children snl)` finds all of the elements that are direct children of singleton nodelist "snl". This is useful for filtering, processing, and counting,

when you're not interested in any text, PI, or comment nodes. This function (and the default processing rule) were provided by Christopher R. Maden <crism@maden.org> in a message on the DSSSList dated Mon, 02 Apr 2001 22:36:33 -0700, and titled "Re: (dsssl) "Default" processing rule?".

⟨Define element-children function 18⟩ ≡

```
(define (element-children #!optional (snl (current-node)))
  (select-by-class (children snl)
                   'element))
```

Macro referenced in scrap 16.

4.3 Implementation Notes

Chapter 5

SGML Weave

5.1 Purpose

The "SGMLWeave" program (`SGMLWeave.dsl`) is a relatively minor customization of Norman Walsh's DSSSL stylesheets for DocBook. In fact, for minimum functionality, the *only* necessary change to the stylesheets is the addition of a processing rule for the `<literalchar>` element, and that processing rule (shown in Section 5.3 is relatively simple.

This is in fact a design goal of the DocBook-based literate programming system, as it exploits the significant efforts of a number of people to develop tools for DocBook.

5.2 Minimum DocBook Customization Layer

"SGMLWeave.dsl" 19 ≡

```
<!--
PUBLIC "-//Mark Wroth//DOCUMENT DBLP Weave Print Rules 1.0//EN"

This document is intended to be a minimum DocBook style-sheet
customization layer.
-->
<!DOCTYPE style-sheet
  PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN" [
 <!ENTITY dbprint.dsl PUBLIC
   "-//Norman Walsh//DOCUMENT DocBook Print Stylesheet//EN"
   CDATA DSSSL>
 <!ENTITY dbhtml.dsl PUBLIC
   "-//Norman Walsh//DOCUMENT DocBook HTML Stylesheet//EN"
   CDATA DSSSL>
]>
<style-sheet>
<style-specification id="print" use="dbprint">
<style-specification-body>
```

⟨literalchar print processing rule 20a⟩
⟨programlisting print customizations 20b⟩

```
</style-specification-body>
</style-specification>
<style-specification id="html" use="dbhtml">
<style-specification-body>
```

⟨literalchar print processing rule 20a⟩
⟨programlisting HTML customizations 25a⟩

```
</style-specification-body>
</style-specification>
<external-specification id="dbprint" document="dbprint.dsl">
<external-specification id="dbhtml" document="dbhtml.dsl">
</style-sheet>
```

5.3 The literalchar Processing Rules

⟨literalchar print processing rule 20a⟩ ≡

```
(element literalchar
  (make sequence
    (literal (attribute-string "data"))))
```

Macro referenced in scrap 19.

5.4 The programlisting Customizations

The basis of our customization is the existing `<programlisting>` code from [5, dbverb.dsl]. We will modify this by adding information ahead of the basic element (by identifying the scrap by either the file name or the definition name, depending on the type of scrap), and adding information after the program listing (if the scrap is continued).

5.4.1 Print Customizations

Almost everything about the print stylesheets will remain untouched. However, there are a few changes that need to be made to the processing of the `printlisting` element. The `printlisting` element itself gets some additional information added to it based on its place in the literate programming web, and the special use of `<xref>` means that we should wrap the cross reference text in some punctuation to indicate that it is a code section reference rather than actual code.

⟨programlisting print customizations 20b⟩ ≡

⟨Customize the programlisting proper 21a⟩
⟨Customize the (programlisting xref) 22b⟩
⟨Print auxiliary definitions 23a⟩

Macro referenced in scrap 19.

The printlisting Proper

There are essentially two sets of changes we want to make to the processing of the `programlisting` element: adding header information identifying the code section and the particular scrap, and adding continuation information if the scrap has a continuation.

Mechanically, we do this by wrapping the original code (taken from [5, `dbverb.dsl`]) inside a `make sequence` and adding the code to produce the header and continuation text before and after the original code.

⟨Customize the programlisting proper 21a⟩ ≡

```
(element programlisting (make sequence
  ⟨Provide scrap header information 21b⟩
  ⟨Original programlisting print code 23c⟩
  ⟨Provide optional continuation info 22a⟩ ))
```

Macro referenced in scrap 20b.

The scrap header information is straight forward except for the actual title of the code section. That title is either the file name of the disk file the section will be written to, or the `xreflabel` of the section. In either case, the title is found in the first scrap of the section, which may or may not be the scrap we are currently processing.

We deal with this as a set of nested `if` statements to grab the title if we are in the first scrap of the section, or processing the `continuedfrom` scrap with a special mode that recursively hunts back up the linked list until it gets to the first scrap.[1]

The other refinement is that we will set the header and continuation information in a specific font and size intended to be distinct from the general font and size used in the document. For ease of reference, these are defined in the auxiliary information.

⟨Provide scrap header information 21b⟩ ≡

```
(make paragraph
  (make sequence
    font-family-name: scrap-header-font
    font-size:        scrap-header-size
```

[1] It is exactly this application that caused us to define the programlisting scraps as a doubly-linked list.

```
(literal "\left-pointing-angle-bracket")
(if (attribute-string "file")
  (literal (attribute-string "file"))
  (if (attribute-string "xreflabel")
    (literal (attribute-string "xreflabel"))
    (with-mode scrap-title-mode
      (make sequence
        (process-element-with-id
          (attribute-string "continuedfrom"))
        (literal " "))))))
 (make sequence
   font-size: (* scrap-header-size 0.75)
   (literal " (ID: ")
   (literal (attribute-string "id")))
 (literal ")\right-pointing-angle-bracket")
 (if (attribute-string "continuedfrom")
     (literal "+")
     (empty-sosofo))
 (literal "\identical-to")))
```

Macro referenced in scrap 21a.

⟨Provide optional continuation info 22a⟩ ≡

```
(if (attribute-string "continuedin")
    (make paragraph
      font-family-name: scrap-header-font
      font-size:        scrap-header-size
      (make sequence
        (literal "Continued in ")
        (literal (attribute-string "continuedin"))))
    (empty-sosofo))
```

Macro referenced in scrap 21a.

The printlisting xref Element

The main customization we want for the `xref` element is to put the cross-reference text in a running text font, and enclose it in angle brackets. In both cases, this is to distinguish a code section reference from actual code.

For consistency, we will put the reference in the same font and size as the scrap header and continuation information.

⟨Customize the (programlisting xref) 22b⟩ ≡

```
(element (programlisting xref) (make sequence
  font-family-name: scrap-header-font
  font-size:        scrap-header-size
 (literal "\left-pointing-angle-bracket")
 ⟨Original xref print code 24⟩
```

```
 (literal "\right-pointing-angle-bracket")
))
```

Macro referenced in scrap 20b.

Print Auxiliary Definitions

Here we define the common information such as the size and font used for the header and continuation. Additionally, we define the special mode used to find the code section title here.

⟨Print auxiliary definitions 23a⟩ ≡

```
(define scrap-header-size 8pt)
(define scrap-header-font "Georgia")
⟨Define scrap-title-mode 23b⟩
```

Macro referenced in scrap 20b.

The `scrap-title-mode` is defined to process `programlisting` elements and either extract the title (found in the `file` or `xreflabel` attributes, or to continue up the linked list if neither attribute is present.

⟨Define scrap-title-mode 23b⟩ ≡

```
(mode scrap-title-mode
  (element programlisting
    (make sequence
      (if (attribute-string "file")
          (literal (attribute-string "file"))
          (if (attribute-string "xreflabel")
              (literal (attribute-string "xreflabel"))
              (process-element-with-id
                (attribute-string "continuedfrom")))))))
```

Macro referenced in scraps 23a, 26c.

Original Modular Style Sheet Print Code

This is the complete DSSSL code for the `programlisting` element, found in `dbverb.dsl`.

⟨Original programlisting print code 23c⟩ ≡

```
($verbatim-display$
 %indent-programlisting-lines%
 %number-programlisting-lines%)
```

Macro referenced in scrap 21a.

This is the complete DSSSL code found in `dblink.dsl` for the `xref` element. It is probably *much* more complex than is actually needed for the rather specialized use we are making of it, but it is easier to just reproduce it than to try to simplify it.

⟨Original xref print code 24⟩ ≡

```
(let* ((endterm (attribute-string (normalize "endterm")))
       (linkend (attribute-string (normalize "linkend")))
       (target  (element-with-id linkend))
       (xreflabel (if (node-list-empty? target)
                      #f
                      (attribute-string (normalize "xreflabel") target))))
  (if (node-list-empty? target)
      (error (string-append "XRef LinkEnd to missing ID '" linkend "'"))
      (if xreflabel
          (make link
            destination: (node-list-address target)
            (literal xreflabel))
          (if endterm
              (if (node-list-empty? (element-with-id endterm))
                  (error (string-append "XRef EndTerm to missing ID '"
                                        endterm "'"))
                  (make link
                    destination: (node-list-address (element-with-id endterm))
                    (with-mode xref-endterm-mode
                      (process-element-with-id endterm))))
              (cond
               ((or (equal? (gi target) (normalize "biblioentry"))
                    (equal? (gi target) (normalize "bibliomixed")))
                ;; xref to the bibliography is a special case
                (xref-biblioentry target))
               ((equal? (gi target) (normalize "co"))
                ;; callouts are a special case
                (xref-callout target))
               ((equal? (gi target) (normalize "listitem"))
                (xref-listitem target))
               ((equal? (gi target) (normalize "question"))
                (xref-question target))
               ((equal? (gi target) (normalize "answer"))
                (xref-answer target))
               ((equal? (gi target) (normalize "refentry"))
                (xref-refentry target))
               ((equal? (gi target) (normalize "glossentry"))
                ;; as are glossentrys
                (xref-glossentry target))
               ((equal? (gi target) (normalize "author"))
                ;; and authors
                (xref-author target))
               ((equal? (gi target) (normalize "authorgroup"))
```

```
 ;; and authorgroups
 (xref-authorgroup target))
(else
 (xref-general target)))))))
```

Macro referenced in scrap 22b.

5.4.2 HTML Customizations

In the HTML style sheet, we need to make the same kinds of customization we did with the print style sheet. The details of the actual customizations differ slightly.

⟨programlisting HTML customizations 25a⟩ ≡

⟨Programlisting element HTML customization 25b⟩
⟨xref element HTML customization 26b⟩
⟨HTML auxiliary definitions 26c⟩

Macro referenced in scrap 19.

The programlisting Proper

In the `programlisting`, we make the same kinds of additions (header and continuation information) as used in the print style sheet.

⟨Programlisting element HTML customization 25b⟩ ≡

```
(element programlisting (make sequence
 ⟨Provide HTML scrap header information 25c⟩
 ⟨Original programlisting HTML code 27a⟩
 ⟨Provide optional HTML scrap continuation info 26a⟩
))
```

Macro referenced in scrap 25a.

The code is also largely common, with some minor changes for the HTML output and its limited character repertoire.

⟨Provide HTML scrap header information 25c⟩ ≡

```
(make element gi: "P"
  (make sequence
    (literal "<")
    (if (attribute-string "file")
      (literal (attribute-string "file"))
      (if (attribute-string "xreflabel")
        (literal (attribute-string "xreflabel"))
        (with-mode scrap-title-mode
          (make sequence
            (process-element-with-id
              (attribute-string "continuedfrom"))
```

```
        (literal " "))))))
  (make sequence
    (literal " (ID: ")
    (literal (attribute-string "id")))
  (literal ")>")
  (if (attribute-string "continuedfrom")
      (literal "+")
      (empty-sosofo))
  (literal "=")))
```

Macro referenced in scrap 25b.

⟨Provide optional HTML scrap continuation info 26a⟩ ≡

```
(if (attribute-string "continuedin")
    (make element gi: "P"
      (make sequence
        (literal "Continued in ")
        (literal (attribute-string "continuedin"))))
    (empty-sosofo))
```

Macro referenced in scrap 25b.

Customizing the xref Element

⟨xref element HTML customization 26b⟩ ≡

```
(element (programlisting xref) (make sequence
(literal "<")
⟨Original xref HTML code 27b⟩
(literal ">")
))
```

Macro referenced in scrap 25a.

HTML Auxiliary Definitions

Here we employ the same basic structure as with the print style sheet, although there is less to define; this primarily is intended to allow room for future elaboration of the style sheet.

The DSSSL code for the `scrap-title-mode` is identical with that used in the print style sheet, so we simply re-use it.

⟨HTML auxiliary definitions 26c⟩ ≡

⟨Define scrap-title-mode 23b⟩

Macro referenced in scrap 25a.

Original programlisting Code

⟨Original programlisting HTML code 27a⟩ ≡

```
($verbatim-display$
%indent-programlisting-lines%
 %number-programlisting-lines%)
```

Macro referenced in scrap 25b.

Original programlisting Code

⟨Original xref HTML code 27b⟩ ≡

```
(let* ((endterm   (attribute-string (normalize "endterm")))
         (linkend   (attribute-string (normalize "linkend")))
         (target    (element-with-id linkend))
         (xreflabel (if (node-list-empty? target)
                        #f
                        (attribute-string (normalize "xreflabel") target))))
    (if (node-list-empty? target)
        (error (string-append "XRef LinkEnd to missing ID '" linkend "'"))
        (make element gi: "A"
              attributes: (list
                           (list "HREF" (href-to target)))
              (if xreflabel
                  (literal xreflabel)
                  (if endterm
                      (if (node-list-empty? (element-with-id endterm))
                          (error (string-append
                                  "XRef EndTerm to missing ID '"
                                  endterm "'"))
                          (with-mode xref-endterm-mode
                            (process-node-list (element-with-id endterm))))
                      (cond
                       ((or (equal? (gi target) (normalize "biblioentry"))
                            (equal? (gi target) (normalize "bibliomixed")))
                        ;; xref to the bibliography is a special case
                        (xref-biblioentry target))
                       ((equal? (gi target) (normalize "co"))
                        ;; callouts are a special case
                        ($callout-mark$ target #f))
                       ((equal? (gi target) (normalize "listitem"))
                        ;; listitems are a special case
                        (if (equal? (gi (parent target)) (normalize "orderedlist"))
                            (literal (orderedlist-listitem-label-recursive target))
                            (error (string-append "XRef to LISTITEM only supported in ORDEREDLIS
                       ((equal? (gi target) (normalize "question"))
                        ;; questions and answers are (yet another) special case
                        (make sequence
```

```
                              (literal (gentext-element-name target))
                              (literal (gentext-label-title-sep target))
                              (literal (question-answer-label target))))
                            ((equal? (gi target) (normalize "answer"))
                             ;; questions and answers are (yet another) special case
                             (make sequence
                              (literal (gentext-element-name target))
                              (literal (gentext-label-title-sep target))
                              (literal (question-answer-label target))))
                            ((equal? (gi target) (normalize "refentry"))
                             ;; so are refentrys
                             (xref-refentry target))
                            ((equal? (gi target) (normalize "glossentry"))
                             ;; as are glossentrys
                             (xref-glossentry target))
                            ((equal? (gi target) (normalize "author"))
                             ;; and authors
                             (xref-author target))
                            ((equal? (gi target) (normalize "authorgroup"))
                             ;; and authorgroups
                             (xref-authorgroup target))
      ; this doesn't really work very well yet
      ;                     ((equal? (gi target) (normalize "substeps"))
      ;                      ;; and substeps
      ;                      (xref-substeps target))
                            (else
                             (xref-general target))))))))
```

Macro referenced in scrap 26b.

Chapter 6

Sample Literate Program

6.1 The Sample Document

"sample.sgm" 29 ≡

```
<!DOCTYPE article
 PUBLIC "-//Mark Wroth//DTD DocBook V4.1-Based Extension Literate Programming 1.0//EN">
<article id="sample-lp">
  <title>A Sample DocBook-Based Literate Program</title>
  <section>
    <title>Introduction</title>
    <para>This is a sample document illustrating basic use of the
    DocBook-based literate programming tool.</para>
    <para>This document combines human-readable documentation of
    a computer program with the actual computer-readable source
    code.  Depending on how it is processed, it becomes either
    printed (on-line) documentation of the program, or the actual
    source submitted to the computer for compilation (or
    interpretation, depending on the language).</para>
  </section>
  <section>
   <title>Source Code</title>

   <para>The source code is divided into a number of
   <quote>scraps</quote>, each containing a discrete fragment of
   code.  These scraps are assembled into code sections by
   concatenating the header scrap with the various continuation
   scraps, in an order defined by the programmer. File sections are
   written to the file indicated by the programmer, while definition
   sections are inserted at a place or places defined by the
   programmer. </para>

   <para>The first code scrap defines a file output, specifically
   to <filename>sample.code</filename>.</para>
```

```
   <programlisting
     id="scrap1"
     file="sample.code"
     continuedin="scrap2"
     >
-- This is sample code in an imaginary language
-- Taken from the first scrap
if a &lessthan; b then
  <xref linkend="scrap3">
fi
   </programlisting>

   <para>The next code scrap is a continuation of the first
   scrap.</para>

   <programlisting
    id="scrap2"
    continuedfrom="scrap1"
    >
-- This is continued code, taken from the second scrap
--
set c = a &ampersand; b
 greater than: &greaterthan;
   </programlisting>

   <para>The following code section is an example of a definition
   scrap, and will be included in a file output scrap.</para>
   <programlisting
    id="scrap3"
    xreflabel="The Third Scrap"
    continuedin="scrap4">
-- Yet more program code from the third scrap
   </programlisting>

   <para>Finally, we have a continuation scrap continuing
   a definition scrap.</para>

   <programlisting
     id="scrap4"
     continuedfrom="scrap3"
   >
-- This is scrap 4, which continues scrap 3
-- It should appear where scrap 3 was inserted.
   </programlisting>
 </section>
</article>
```

Chapter 7

System Performance

Generally speaking, verifying the functionality of the literate programming system consists of checking that the TANGLE branch correctly produces the intended code, and the WEAVE branch produces appropriate human-readable documentation.

7.1 Sample Code Output

The code output file `sample.code` produced from the sample document looks like this:

```
-- This is sample code in an imaginary language
-- Taken from the first scrap
if a < b then
  -- Yet more program code from the third scrap
   -- This is scrap 4, which continues scrap 3
-- It should appear where scrap 3 was inserted.

fi
   -- This is continued code, taken from the second scrap
--
set c = a & b
 greater than: >
```

This demonstrates the key functions needed in the TANGLE branch, namely that:

- Code sections are correctly assembled from the separate scraps identified in the source code.
- Definition sections are inserted into code sections at the location identified by the `<xref>` tag pointing to the head of the definition section.

- File output sections are written to disk in with the desired file names.
- Syntactically significant characters (to SGML) including `<`, `>`, and `&`, are written correctly in the output file.

The handling of whitespace in the code scraps is not quite what I expected, but appears to be consistent and reasonable. It appears that whitespace is (correctly) transcribed to the output file, except that an SGML record end (`CR-LF` pair, under WINDOWS) following the `<programlisting>` start tag is *not* transcribed to the output.

7.2 Sample Woven Output

The WEAVE DSSSL style sheets produce two sets of human-readable documentation from `sample.sgm`: `sample.rtf` and a set of HTML files. The HTML files are divided and named by the conventions of the HTML *Modular DocBook Style Sheets*, which (without customization) produce multiple small files.

In both cases, the WEAVE outputs reasonably produce verbatim program listings, with header and continuation information that matches the actual structure defined by the sample file.

The typographic treatment of the header and continuation is acceptable, although the amount of vertical space introduced in both the print and HTML versions between the header and the body of the programlisting is larger than I would prefer.

7.3 Evaluation

Overall, these implementations of a DTD, TANGLE, and WEAVE are functional. They do not produce really high-quality typographical output, but they do reasonably display the structure of the code sections. Pending further experience with the system, this seems to be an acceptable implementation.

Acronyms

DSSSL Document Style Semantics Specification Language DSSSL is an ISO standard defining how to specify transformations from SGML documents to page-oriented output renderings.

DTD Document Type Definition, Document Type Declaration

HTML Hypertext Markup Language

ISO International Organization for Standardization

OASIS Organization for the Advancement of Structured Information Standards

OSNL Optional Singleton Node List

SGML Standard Generalized Markup Language

SNL Singleton Node List

XML Extensible Markup Language

Bibliography

[1] Carlisle, David. "Re: Issues with literate programming DSSSL Script", in *DSSSList Digest* Vol 3, Number 241 (Thu, 16 Dec 1999 16:27:27 GMT). `<davidc@nag.co.uk>`.

[2] Knuth, Donald. *Literate Programming*. Stanford, CA: Center for the Study of Language and Information, 1992.

[3] Maden, Christopher R.. "Re: (dsssl) "Default" processing rule?", in *DSSSList Digest* Vol 4, Number 143 (Mon, 02 Apr 2001 22:36:33 (-0700)). `<crism@maden.org>`.

[4] OASIS. *DocBook*. `http://www.oasis-open.org/docbook`.

[5] Norman Walsh. *The Modular DocBook Stylesheets*. `http://www.nwalsh.com/docbook/dsssl/index.html`.