An Experiment in Literate Programming Using SGML and DSSSL

Revision 0.109

Mark B. Wroth

December 31, 1999

Contents

Chapter 1

Purpose

1.1 Background

Literate programming is a style of computer programming in which priority is given to the exposition of the program *to the human reader*, rather than for the convenience of the computer which will execute the program. Since computers are notoriously intolerant of changes in the way their inputs are structured, and most computer programming languages have, at best, limited facilities for any but the simplest comments, this style requires a set of tools which allow the author to both explain the program for the human audience, and give precise instructions on how the program is to be read by the computer.

The original literate programming system, WEB, was developed by Profesor Donald Knuth, who used it in the production of TEX and other programs. This system combined the TEX typesetting system with the PASCAL programming language. Subsequent developments have largely been confined to adding to the choice of programming languages. Various "language-aware" WEB variants have appeared since the original WEB system, covering a variety of programming languages. The difficulty of converting WEB systems to other languages prompted Norman Ramsey to continue SIlvio Levy's work to develop a SPIDER system to assist in generating new WEBs for different languages, but even this system does not make it trivial to add a new language-aware WEB. Krommes, with his development of FWEB, appears to have developed the concept of a "current language", allowing the same program web to contain program text in multiple languages while retaining the language awareness and specialized typesetting of the original WEB.

A second branch of literate programming development addressed the difficulties of developing WEB systems in new languages by ignoring the language characteristics entirely. This branch, typified by such programs as NOWEB and NUWEB, makes no attempt to "pretty print" the program text or take advantage of syntactic knowledge of the program. Instead, the program text is reproduced verbatim in the typeset documentation. In addition to simplifying the use of a new programming language—since there are no language dependencies—some programmers prefer seeing the source code reproduced more or less as they would in the editor in which it was written. Since a language independent WEB can be dramatically simpler than a language aware versio, this approach has some obvious advantages which offset its inability to take advantage of the target language.

But despite all of the activity centered around adapting WEB to various programming languages, relatively little effort has been devoted to changing the documentation language used. With the exception of one early effort combining the TROFF typesetting system (Thimbleby's system is confusingly called CWEB although it is unrelated to the more commonly known literate programming system named CWEB, by Silvio Levy), almost all of the literate programming systems use TEX as their documentation language. This may be due to the difficulty of the typesetting task—Sewell reports that Thimbleby (the author of the TROFF version of CWEB) estimated that 95 percent of the effort involved in that system was in this area [5, p. 144]. Recently, a few systems have emerged which relax this—NUWEB [1] for example, can be used to produce HTML with some effort, and FUNNELWEB [7] attempts to provide some formatter independence.

I am not aware, however, of any released systems which have used SGML (or its cousin XML) markup to define a literate program, despite the apparent easy fit of the concept[1]. This seems odd, since the structured nature of SGML would seem to lend itself to the natural intermingling of code and documentation that is at the heart of literate programming. Additionally, a variety of powerful tools to author and manipulate SGML-marked up documents have emerged; such tools would appear to greatly simplify the cre-

[1] As I asked for (and got) assistance with the DSSSL code for the `"tangle"` style sheet used in this paper from the `DSSSList`, an email list devoted to the DSSSL programming language, Christopher R. Madden (`chrism@exemplary.net`) commented that he was in the process of such a project using the XML variant of DOCBOOK. Additionally, C.M. Sperberg-McQueen has written a tag set for literate programming called SWEB [6]. While he appears to have implemented at least part of the necessary processing software using LEX and YACC, and has made papers discussing the system available on the web, he labels the work unfinished and unpublished. It is nonetheless a very interesting system, and shows considerable thought.

ation of an SGML-based WEB system. Among the readily available tools that would seem applicable are PERL (with freely available SGML libraries), OMNIMARK, and DSSSL.

DSSSL, the Document Style Semantics and Specification Language, is an ISO standard [3] language aimed at producing output documents from SGML-marked up input files. Frustratingly, the release of the DSSSL standard in 1996 appears not to have been accompanied by any programs implementing the defined language, nor have any complete implementations appeared since. James Clark's DSSSL implementation, JADE [2], was used for this paper. It is free, readily available, and implements a significant fraction of the ISO-defined style language. In addition, it implements a number of extensions for SGML-to-SGML transformations which make it quite effective for that purpose despite not implementing the ISO-defined DSSSL transformation language.

This paper is an experiment in creating a "proof of concept" literate programming system using SGML markup for documentation and code scrap delimination and DSSSL in the form of James Clark [2] processor for the implementation language.

1.2 Design

A literate programming system has two basic processing branches, which we will call the `"tangle"` and `"weave"` branches after the original programs defined by Dr. Knuth [4]. The `"weave"` branch produces the form that is converted into a human-readable program listing (traditionally in hard-copy but more recently in on-line forms as well). The `"tangle"` branch produces the source files as they are used by the computer itself

The `"weave"` branch is straightforward, at least in principle. It amounts to using SGML to mark up a document for printing, and this is an area where a great deal of effort has been expended. Including "pretty printing" of the source code appears straightforward, if not necessarily trival, if we are willing to mark up the source code. It is probably doable even if we are not. However, for the first cut, we will simply assume that no pretty printing is needed and very simple documentation is used. Basically, we are not going to spend much effort here because we think that this branch is clearly within the capabilities of SGML and SGML-based processing systems. The simplified `"weave"` processing script is shown in Section 3.3, and the complete script is reproduced in Appendix B.

The `"tangle"` branch is more challenging. The goal of the experiment

is to demonstrate:

- Assembly of code scraps;
- Insertion of assembled code scraps into other scraps;
- Output of assembled scraps to disk file.

The `"tangle"` script is discussed in Section 3.2, and the complete script is in Appendix B.

In essence, we have two kinds of *header* scraps: scraps which will be written to a file, and *definition* scraps which are included in other scraps as part of the definition of the top level program. Either kind of scrap may be continued by other scrap definitions, which shall be assembled in the order they appear in the input file.

Other possible functionality to consider:

- Macro definition. Deferred for future continuation. Experience with NUWEB indicates that this functionality may not be necessary. Additionally, SGML itself allows for a primitive macro facility in the form of entity definitions. While this has some disadvantages from the perspective of clear elucidation of the concepts (the entity definitions are hidden from the reader), the fact that some literate programming systems omit the macro capability while retaining significant functionality, combined with the SGML entity facility, persuades me to defer this capability—perhaps forever.
- Scrap numbering
- Scrap usage listing
- List of files output

Chapter 2

The Source sgml Document

In order to test the concepts, we need a sample document. This provides the valid SGML document as a test case.

2.1 The Document Type Description

The basic DTD is very simple.

⟨Test DTD 6⟩ ≡

```
<!DOCTYPE document [
<!ELEMENT document o o (p|scrap|continuation)*>
<!ELEMENT p        - o (#PCDATA|scrapref)*>
⟨The 'scrap' element 7⟩
⟨The 'continuation' element 8a⟩
⟨The 'scrapref' element 8b⟩
⟨The 'literal' element 9a⟩
]>
```

Macro referenced in scrap 9b.

2.1.1 The 'scrap' element

There are, in fact, a number of syntactic uses for code scrap elements:

- Beginning of an output file definition (the "unnamed section" in the original WEB system);
- Continuation of an output file definition;

- Beginning of a "defined" section—one which will eventually be inserted into an output file section;
- Continuation of a "defined" section;
- Reference to a scrap within a scrap, intended to be result in the referenced scrap being inserted in the code in place of the reference;
- Reference to a scrap in documentation, where is should be treated as a citation.

All of these might be handled with a single element type. In our initial implementation we used two types, a ⟨scrap⟩ for all of the code definitions, and a ⟨scrapref⟩ for the references to a scrap. The initial implementation ran into trouble with nested, continued scraps, and so we split out ⟨continuation⟩s.

The ⟨scrap⟩ is the key element of the literate programming setup. It contains program code, which may be either inserted into another scrap or output to a file. Scraps are not necessarily defined at a single point in the literate program; following Knuth's convention, they may be arbitrarily continued over many parts of the input file, and are assembled in the order in which they appear.

It's not clear at this point if this is the right approach, but for now we will define the initial scrap to have an `id` attribute and possibly a `file` attribute indicating the output file. Continuations use the ⟨continuation⟩ element, with the scrap being continued identified by the `continues` attribute, with its value equal to the `id` of the beginning scrap.

⟨The 'scrap' element 7⟩ ≡

```
<!ELEMENT scrap    - o (title, code)>
<!ATTLIST scrap    file      CDATA #IMPLIED
                   id        ID    #REQUIRED
>
<!ELEMENT title    o o (#PCDATA) >
<!ELEMENT code     o o (#PCDATA|scrapref|literal)* >
```

Macro referenced in scrap 6.

2.1.2 The 'continuation' element

The ⟨continuation⟩ element continues a scrap previously opened.

Because of difficulties with mixing modes associated with having continuation scraps and a desire to clarify the syntax, we add a ⟨continuation⟩

element. This also significantly simplifies the handling of nested scraps and their continuations.

⟨The 'continuation' element 8a⟩ ≡

```
<!ELEMENT continuation - o (code)>
<!ATTLIST continuation
                    continues IDREF #REQUIRED
>
```

Macro referenced in scrap 6.

2.1.3 The 'scrapref' element

The ⟨scrapref⟩ element is to be used to insert a scrap into a code section; the `id` attribute specifies the (head of) the scrap to be inserted. It will also be used in documentation in a similar manner, except that there only a cross reference will be used.

⟨The 'scrapref' element 8b⟩ ≡

```
<!ELEMENT scrapref - o EMPTY>
<!ATTLIST scrapref id IDREF #REQUIRED >
```

Macro referenced in scrap 6.

2.1.4 The 'literal' 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 (for example the `&&` used below, which acts lie a logical and), it would make sense to define entities which make syntactic sense. This would allow one to use, for example `&and;` instead of `&&`[1].

[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.

⟨The 'literal' element 9a⟩ ≡

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

Macro referenced in scrap 6.

2.2 The Document Instance

And here is the actual document.

"test.sgm" 9b ≡

```
⟨Test DTD 6⟩
<document>
<p>This is some sample documentation text. It is entirely
unremarkable. The included code conforms to no particular programming
language. It is chosen just to provide examples that can be examined
to see if it is being reproduced properly.  Becaus of this, it
includes punctuation marks that are likely to be syntactically
significant to the various processors. This particular scrap includes
a "less than" character, "<" which is the SGML element
start character.</p>
<scrap file="scrap1.out" id="scrap1">The main code
<code>
-- scrap1 head
  for i = 1 to 10
    write i
  rof
  if a &lt; b fi
-- include scrap2 by reference
<scrapref id="scrap2">

</scrap>
```

File defined by scraps 9b, 10abc.

We split the desired output file into multiple scraps to test how the output entity is formed. Unfortunately, if we just give all of the scraps the same file id, only one scrap is in the result. While expected, this means we're going to have to be more canny in the `tangle` script so that we can get the desired concatenation.

"test.sgm" 10a ≡

```
<p>This is documentation of a continuation scrap, specifically the
first continuation of the first scrap. It is entirely unremarkable.</p>
<continuation continues="scrap1">
<code>
-- first continuation of scrap1
  if (i &lt; 10)
    call iout
  fi

</continuation>
```

File defined by scraps 9b, 10abc.

"test.sgm" 10b ≡

```
<p>This scrap is another continuation.  It is unremarkable, except that
it contains two other characters likely to be an issue for the SGML
tools, specifically the "greater than" and ampersand characters (">"
and "&").</p>

<continuation continues="scrap1">
<code>
  -- Second continuation of scrap1
  if (i &lt; 10) &amp;&amp; (j &gt; 12)
    call iout
  fi
</continuation>
```

File defined by scraps 9b, 10abc.

Now we write another scrap which will be included in an output file. Since this is a header scrap, it is defined with the ⟨scrap⟩ element.

"test.sgm" 10c ≡

```
<p>This is a header scrap, which is intended to be included in another
scrap in order to finally be included in an output file. The scrap
documentation is entirely unremarkable.</p>
```

```
<scrap id="scrap2">
<title>An included scrap (scrap2)
<code>
-- included scrap2 head
while a % b &lt; c
  incr(a)
end

</scrap>

<p>Some documentation of the next scrap. It is unremarkable in every way.</p>
<continuation continues="scrap2">
-- included scrap2 continuation 1
some more code

-- include scrap3 by reference
<scrapref id="scrap3">
</continuation>

<p> And finally, documentation of the third scrap. It is entirely
unremarkable, except that it includes a reference to the scrap that
it is included in, which is <scrapref id="scrap2">.</p>
<scrap id="scrap3">A nested scrap
<code>
-- contents of scrap3
-- scrap 3 should have continuation 1

</scrap>

<p>The third scrap is continued. The documentation is entirely
unremarkable, and is extended only to provide some reasonable text in
the woven file.</p>
<continuation continues="scrap3">
-- continuation 1 of scrap 3
</continuation>
</document>
```

File defined by scraps 9b, 10abc.

Chapter 3

Processing Scripts

We create the two desired processing scripts as simple shells to begin with. While in some ways it would be convenient to use a third *style-sheet* to contain common code, by using defined scraps (in NUWEB, the literate programming tool being used fir this experiment) we can define the code once and use it as needed with little trouble.

"test.dsl" 12 ≡

```
<!-- $Id: Experiment.w,v 0.109 1999/12/31 19:35:12 penny Exp penny $ -->
<!DOCTYPE style-sheet
  PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN">
<style-sheet>
<style-specification
 id = "tangle">
 ⟨DSSSL Tangle 14⟩
 ⟨Function to find the document element 13a⟩
</style-specification>
<style-specification
 id = "weave">
 ⟨Weave declarations 20b⟩
 ⟨DSSSL Weave 17b, ... ⟩
 ⟨Function to find the document element 13a⟩
</style-specification>
</style-sheet>
```

3.1 Supporting Functions

This section describes several DSSSL functions which are needed at various spots in the DSSSL code.

3.1.1 Finding the 'document' Element

The following function definition is from Norman Gray, `norman@astro.gla.ac.uk`, and returns a singleton node list consisting of the root "document-element" of the current document (or #f if there is no such element).

⟨Function to find the document element 13a⟩ ≡

```
(define (document-element #!optional (node (current-node)))
  (let ((gr (node-property 'grove-root node)))
   (if gr  ; gr is the grove root
      (node-property 'document-element gr default: #f)
      ;; else we're in the root rule now
      (node-property 'document-element node default: #f))))
```

Macro referenced in scrap 12.

3.1.2 Passing Literal Data

This processing rule is used to pass literal data specified in a ⟨literal⟩ element through to the SGML output.

⟨Output literal data 13b⟩ ≡

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

Macro referenced in scrap 14.

This requires a non standard flow object, which must be declared before use.

⟨Tangle non-standard flow objects 13c⟩ ≡

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

Macro defined by scraps 13c, 15b.
Macro referenced in scrap 14.

3.2 The dsssl 'Tangle' Script

Like the original TANGLE, the `"tangle"` style-specification is intended to output the code parts of the input file in the order specified by the author. This is probably *not* the order they appear in the document; the document is organized for human comprehension, while the output file must satisfy the needs of the computer.

In essence, we have three major tasks to perform:

- Assemble a scrap from its defining pieces, which may include a header and any number of continuation pieces;
- Insert an assembled scrap into another scrap;
- Write an assembled scrap (including any inserted scraps) to a specified data file.

Both scraps which are to be directly written to file and those which are used internally share the need to assemble all of their continuation scraps. This common need is discussed in Section 3.2.2. The insertion of defined scraps into other scraps is covered in Section 3.2.3, and the top level output to file is the subject of Section 3.2.1.

⟨DSSSL Tangle 14⟩ ≡

⟨Tangle non-standard flow objects 13c, ... ⟩
⟨Process a file output scrap 15a⟩
⟨Insert a scrap via 'scrapref' 17a⟩
⟨Output literal data 13b⟩

Macro referenced in scrap 12.

3.2.1 File Output Scraps

The file output scrap is in a sense the basic element of a literate program. It provides the "top level" output to a file—which is the ultimate purpose of the `"tangle"` routine!

While we're processing the input file, we will ignore all scraps except those which produce file output. There should be exactly one scrap with a `file` attribute refering to each output file. (Perhaps we could enforce this by defining the `file` attribute to be of type `ID`?)

⟨Process a file output scrap 15a⟩ ≡

```
(element scrap
 (make sequence
  (if (attribute-string "file")
    (make entity
      system-id: (attribute-string "file")
      (make sequence
        (process-matching-children 'code)
        ⟨Find and process all scraps that refer to this one 16⟩))
    (empty-sosofo))))
```

Macro referenced in scrap 14.

This requires a non-standard flow object, the `entity`, which must be declared before use.

⟨Tangle non-standard flow objects 15b⟩ ≡

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

Macro defined by scraps 13c, 15b.
Macro referenced in scrap 14.

3.2.2 File Output Continuations

The point of this scrap is to find and process all of the ⟨continuation⟩ scraps of the current node. We do this by selecting from all of the descendants of the document node (i.e. the whole document instance) the nodes which have GI "continuation" and a `continues` attribute with value equal to the `id` attribute of the current node.

The implementation of this raises an interesting question regarding the DSSSL language. While the processing order of a `process-node-list` is defined to be that of the list order [3, Section 12.4.3], it is less clear that the `select-elements` and the `descendants` will provide the node list in the correct order. It appears from Chapter 10, and specifically [3, Section 10.2.5] that `select-elements` will preserve the order existing in the node list that is its argument, although this is not explicitly stated[1]. The nodelist provided to `select-elements` is created by the `descendants` procedure [3, 10.2.3];

[1] My thanks to Brandon Ibach (`bibach@infomansol.com`), in discussion on the DSSSList, for his assistance in clarifying this point.

here again the implication is that the document order is preserved, but this is not explicit.

The expression needed to parse the attribute is somewhat tricky (at least for the author, who found this to be an instructive example on the difference between quotation and quasi-quotation in DSSSL). Quoting Brandon Ibach (`bibach@infomansol.com`)[2], who resolved the problem in a post to the DSSSList:

> The problem here is that the single quote in your version quoted the *entire* expression, meaning that the "attribute-string" symbol and the "id" string got passed in as part of the pattern, rather than being evaluated and replaced with the value of the `"ID"` attribute. The backquote, above, introduces a "quasi-quote" expression, which is similar to a regular quoted expression, except that you can "unquote" certain parts of it, so that they will be evaluated. In this case, we're unquoting the (attribute-string) call, such that the final result of this would be a structure like:
>
> `(scrap (continues "ABC"))`
>
> if the current node was an element with an `ID` of `"ABC"`, that is.
> `:)`

We will reuse this code to process the continuation scraps for a scrap reference, as well.

⟨Find and process all scraps that refer to this one 16⟩ ≡

```
(make sequence
  (process-node-list
    (select-elements
      (descendants
        (document-element (current-node)))
        `(continuation
           (continues ,(attribute-string "id")))))))
```

Macro referenced in scraps 15a, 17a, 18b.

3.2.3 Scrap References

A ⟨scrapref⟩ in program code indicates that we should insert the complete scrap referenced at this point in the program. The basic strategy is the

[2] in the DSSSList Digest Vol. 3, Number 242

same as with a file output scrap, except that we need to start by finding the scrap head, and we need the other processing branch (when the `file` is *not* specified).

⟨Insert a scrap via 'scrapref' 17a⟩ ≡

```
(element scrapref
  (with-mode scrapreference
    (make sequence
      (process-element-with-id
        (attribute-string "id")))))
(mode scrapreference
  (element scrap
    (make sequence
      (if (attribute-string "file")
          (empty-sosofo)
          (make sequence
            (process-matching-children 'code)
            ⟨Find and process all scraps that refer to this one 16⟩)))))
)
```

Macro referenced in scrap 14.

3.3 The dsssl 'Weave' Script

In contrast to the `"tangle"` specification, the `"weave"` style-specification produces the human readable documentation.

The only element we are using for general documentation is the ⟨p⟩ element for general paragraphs.

⟨DSSSL Weave 17b⟩ ≡

```
(element p
  (make paragraph
    (process-children)))
```

Macro defined by scraps 17bc, 18ab, 19ab, 20a.
Macro referenced in scrap 12.

A ⟨scrapref⟩ appearing in running text is set using the *scraptitle* mode, which we will reuse at the beginning of each defined scrap.

⟨DSSSL Weave 17c⟩ ≡

```
(element scrapref
  (make sequence
    (with-mode scraptitle
      (process-element-with-id
        (attribute-string "id")))))
```

Macro defined by scraps 17bc, 18ab, 19ab, 20a.
Macro referenced in scrap 12.

For header scraps, we show the name of the scrap followed by an equivalence sign, followed by the text of the scrap itself

⟨DSSSL Weave 18a⟩ ≡

```
(element scrap
  (make sequence
    (make paragraph
      (make sequence
        (with-mode scraptitle
          (process-matching-children 'title))
      (literal "\identical-to")))
    (make paragraph
      lines: 'asis
      font-family-name: "Courier New"
      (process-matching-children 'code))))
```

Macro defined by scraps 17bc, 18ab, 19ab, 20a.
Macro referenced in scrap 12.

⟨DSSSL Weave 18b⟩ ≡

```
(element (code scrapref)
  (make sequence
     lines: 'asis
     font-family-name: "Courier New"
     (process-children)
     ⟨Find and process all scraps that refer to this one 16⟩
))
```

Macro defined by scraps 17bc, 18ab, 19ab, 20a.
Macro referenced in scrap 12.

For continuation scraps, we do the same as with header scraps, adding at plus-sign to indicate that this is a continuation.

⟨DSSSL Weave 19a⟩ ≡

```
(element continuation
  (make sequence
    (make paragraph
      (make sequence
        (with-mode scraptitle
          (process-element-with-id
            (attribute-string "continues")))
        (literal "\identical-to +")))
    (make paragraph
      lines: 'asis
      font-family-name: "Courier New"
      (process-matching-children 'code))))
```

Macro defined by scraps 17bc, 18ab, 19ab, 20a.
Macro referenced in scrap 12.

The *scraptitle* mode sets the title of the referenced scrap. We also include a section number indicating the section being written or continued, and, in the case of a file output scrap, the name of the file.

This code refers to the *current-node*, which is the *header* scrap, not the continuation. This is not exactly the desired behavior; we need a way to number each scrap. The "traditional" way to do this is to number each scrap sequentially; NUWEB numbers the scraps with the page number and a suffix if there is more than one scrap on a page.

⟨DSSSL Weave 19b⟩ ≡

```
(mode scraptitle
  (element scrap
    (process-matching-children 'title))
  (element title
    (make sequence
      (literal "\left-pointing-angle-bracket")
      (process-children-trim)
      (literal " (\section-sign")
      (literal
        (format-number
          (element-number
            (parent (current-node))) "1"))
      (if (attribute-string "file"
            (parent (current-node)))
          (make sequence
```

```
              font-family-name: "Courier New"
              (literal "’")
              (literal
                (attribute-string "file"
                  (parent (current-node))))
              (literal "’"))
            (empty-sosofo))
          (literal ")")
          (literal "\right-pointing-angle-bracket")))
)
```

Macro defined by scraps 17bc, 18ab, 19ab, 20a.
Macro referenced in scrap 12.

Finally, we need a similar mechanism for passing literal data through to the back end as in the `"tangle"` script.

⟨DSSSL Weave 20a⟩ ≡

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

Macro defined by scraps 17bc, 18ab, 19ab, 20a.
Macro referenced in scrap 12.

The literal output requires the JADE extension *formatting-instruction*, which must be declared.

⟨Weave declarations 20b⟩ ≡

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

Macro referenced in scrap 12.

Chapter 4

Results and Analysis

The processing scripts shown here appear to work as advertised: `"weave"` produces a (simplified) version of printed documentation, and `"tangle"` produces an output file which concatentates the defined scraps as we expect.

4.1 'Tangle' Output

The test source file, `test.sgm`, is designed to provide examples of the assembly of program texts from sequences of scraps, and scraps inserted into other scraps.

The resulting file does in fact assemble the scraps in the intended order, as shown below:

```
-- scrap1 head
  for i = 1 to 10
    write i
  rof
  if a < b fi
-- include scrap2 by reference

-- included scrap2 head
while a % b < c
  incr(a)
end
 -- included scrap2 continuation 1
some more code

-- include scrap3 by reference
```

```
-- contents of scrap3
-- scrap 3 should have continuation 1
-- continuation 1 of scrap 3

-- first continuation of scrap1
  if (i < 10)
    call iout
  fi

  -- Second continuation of scrap1
  if (i < 10) && (j > 12)
    call iout
  fi
```

The inserted scrap is assembled from the two scraps intended, and the basic file is assembled from it and the designated file output scrap..

Appendix A

The Assembled SGML Input File

The complete SGML input file (sample document) is included here for reference.

```
<!DOCTYPE document [
<!ELEMENT document o o (p|scrap|continuation)*>
<!ELEMENT p        - o (#PCDATA|scrapref)*>

<!ELEMENT scrap    - o (title, code)>
<!ATTLIST scrap    file      CDATA #IMPLIED
                   id        ID    #REQUIRED
>
<!ELEMENT title    o o (#PCDATA) >
<!ELEMENT code     o o (#PCDATA|scrapref|literal)* >


<!ELEMENT continuation - o (code)>
<!ATTLIST continuation
                   continues IDREF #REQUIRED
>


<!ELEMENT scrapref - o EMPTY>
<!ATTLIST scrapref id IDREF #REQUIRED >
```

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

]>

<document>
<p>This is some sample documentation text. It is entirely
unremarkable. The included code conforms to no particular programming
language. It is chosen just to provide examples that can be examined
to see if it is being reproduced properly.  Becaus of this, it
includes punctuation marks that are likely to be syntactically
significant to the various processors. This particular scrap includes
a "less than" character, "<" which is the SGML element
start character.</p>
<scrap file="scrap1.out" id="scrap1">The main code
<code>
-- scrap1 head
  for i = 1 to 10
    write i
  rof
  if a &lt; b fi
-- include scrap2 by reference
<scrapref id="scrap2">

</scrap>

<p>This is documentation of a continuation scrap, specifically the
first continuation of the first scrap. It is entirely unremarkable.</p>
<continuation continues="scrap1">
<code>
-- first continuation of scrap1
  if (i &lt; 10)
    call iout
```

```
  fi

</continuation>
<p>This scrap is another continuation.  It is unremarkable, except that
it contains two other characters likely to be an issue for the SGML
tools, specifically the "greater than" and ampersand characters (">"
and "&").</p>

<continuation continues="scrap1">
<code>
  -- Second continuation of scrap1
  if (i &lt; 10) &amp;&amp; (j &gt; 12)
    call iout
  fi
</continuation>
<p>This is a header scrap, which is intended to be included in another
scrap in order to finally be included in an output file. The scrap
documentation is entirely unremarkable.</p>
<scrap id="scrap2">
<title>An included scrap (scrap2)
<code>
-- included scrap2 head
while a % b &lt; c
  incr(a)
end

</scrap>

<p>Some documentation of the next scrap. It is unremarkable in every way.</p>
<continuation continues="scrap2">
-- included scrap2 continuation 1
some more code

-- include scrap3 by reference
<scrapref id="scrap3">
</continuation>

<p> And finally, documentation of the third scrap. It is entirely
unremarkable, except that it includes a reference to the scrap that
it is included in, which is <scrapref id="scrap2">.</p>
```

```
<scrap id="scrap3">A nested scrap
<code>
-- contents of scrap3
-- scrap 3 should have continuation 1

</scrap>

<p>The third scrap is continued. The documentation is entirely
unremarkable, and is extended only to provide some reasonable text in
the woven file.</p>
<continuation continues="scrap3">
-- continuation 1 of scrap 3
</continuation>
</document>
```

Appendix B

The Assembled DSSSL Script File

The complete DSSSL script file is included here for reference.

```
<!-- $Id: Experiment.w,v 0.109 1999/12/31 19:35:12 penny Exp penny $ -->
<!DOCTYPE style-sheet
  PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN">
<style-sheet>
<style-specification
 id = "tangle">


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

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


 (element scrap
  (make sequence
   (if (attribute-string "file")
     (make entity
       system-id: (attribute-string "file")
       (make sequence
         (process-matching-children 'code)
```

```
          (make sequence
            (process-node-list
              (select-elements
                (descendants
                  (document-element (current-node)))
                  `(continuation
                     (continues ,(attribute-string "id"))))))
        ))
    (empty-sosofo))))


(element scrapref
  (with-mode scrapreference
    (make sequence
      (process-element-with-id
        (attribute-string "id")))))
(mode scrapreference
  (element scrap
    (make sequence
      (if (attribute-string "file")
          (empty-sosofo)
          (make sequence
            (process-matching-children 'code)

              (make sequence
                (process-node-list
                  (select-elements
                    (descendants
                      (document-element (current-node)))
                      `(continuation
                         (continues ,(attribute-string "id"))))))
            ))))
)


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

```
 (define (document-element #!optional (node (current-node)))
   (let ((gr (node-property 'grove-root node)))
    (if gr  ; gr is the grove root
       (node-property 'document-element gr default: #f)
       ;; else we're in the root rule now
       (node-property 'document-element node default: #f))))

</style-specification>
<style-specification
 id = "weave">

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


 (element p
   (make paragraph
     (process-children)))

 (element scrapref
   (make sequence
     (with-mode scraptitle
       (process-element-with-id
         (attribute-string "id")))))

 (element scrap
   (make sequence
     (make paragraph
       (make sequence
         (with-mode scraptitle
           (process-matching-children 'title))
       (literal "\identical-to")))
     (make paragraph
       lines: 'asis
       font-family-name: "Courier New"
```

```
      (process-matching-children 'code))))

(element (code scrapref)
  (make sequence
     lines: 'asis
     font-family-name: "Courier New"
     (process-children)

       (make sequence
         (process-node-list
           (select-elements
             (descendants
               (document-element (current-node)))
               `(continuation
                  (continues ,(attribute-string "id"))))))

))

(element continuation
  (make sequence
    (make paragraph
      (make sequence
        (with-mode scraptitle
          (process-element-with-id
            (attribute-string "continues")))
        (literal "\identical-to +")))
    (make paragraph
      lines: 'asis
      font-family-name: "Courier New"
      (process-matching-children 'code))))

(mode scraptitle
  (element scrap
    (process-matching-children 'title))
  (element title
    (make sequence
      (literal "\left-pointing-angle-bracket")
      (process-children-trim)
      (literal " (\section-sign")
      (literal
```

```
          (format-number
            (element-number
              (parent (current-node))) "1"))
        (if (attribute-string "file"
              (parent (current-node)))
            (make sequence
              font-family-name: "Courier New"
              (literal "'")
              (literal
                (attribute-string "file"
                  (parent (current-node))))
              (literal "'"))
            (empty-sosofo))
        (literal ")")
        (literal "\right-pointing-angle-bracket")))
 )

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


 (define (document-element #!optional (node (current-node)))
   (let ((gr (node-property 'grove-root node)))
    (if gr  ; gr is the grove root
       (node-property 'document-element gr default: #f)
       ;; else we're in the root rule now
       (node-property 'document-element node default: #f))))

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

Bibliography and Indices

Bibliography

[1] Preston Briggs. *Nuweb: A Literate Programming System*. CTAN, CTAN, 1998. With local modifications and compilation for Windows NT.

[2] James Clark. *JADE: James' DSSSL Engine*. http://www.jclark.com, 1998. See `http://www.jclark.com`.

[3] International Organization for Standardization and the International Electrotechnical Commission. *ISO/IEC 10179:1996 Document Style Semantics and Specification Language (DSSSL)*, 1996.

[4] Donald Ervin Knuth. *Literate Programming*. Center for the Study of Language and Information, Leland Stanford University, Stanford, California, 1991.

[5] Wayne Sewell. *Weaving a Program: Literate Programming in WEB*. Van Nostrand Reinhold, New York, New York, 1989.

[6] C. M. Sperberg-McQueen. SWEB: An SGML tag set for literate programming. "This incomplete, unpublished document is distributed privately for comment by friends and colleagues; it is not now a formal publication and should not be quoted in published material.", 1992–1996.

[7] Ross N. Williams. *FunnelWeb User's Manual*. Adelaide, Australia, v1.0 for FunnelWeb v3.0 edition, May 1992.

Cross References

Identifiers

Files

Scraps

Index

Colophon

This paper was written primarily with EMACS as the text editor, and NUWEB as the literate programming system. PDFLATEX (with the HYPERREF package) was the primary document compiler, and JADE was the DSSSL processor.

This is some sample documentation text. It is entirely unremarkable. The included code conforms to no particular programming language. It is chosen just to provide examples that can be examined to see if it is being reproduced properly. Becaus of this, it includes punctuation marks that are likely to be syntactically significant to the various processors. This particular scrap includes a "less than" character, "<" which is the SGML element start character.

⟨The main code (§1'scrap1.out')⟩≡

```
-- scrap1 head
  for i = 1 to 10
    write i
  rof
  if a < b fi
-- include scrap2 by reference

⟨An included scrap (scrap2) (§2)⟩≡ +
-- included scrap2 continuation 1
some more code

-- include scrap3 by reference

⟨A nested scrap (§3)⟩≡ +
-- continuation 1 of scrap 3
```

This is documentation of a continuation scrap, specifically the first continuation of the first scrap. It is entirely unremarkable.

⟨The main code (§1'scrap1.out')⟩≡ +

```
-- first continuation of scrap1
  if (i < 10)
    call iout
  fi
```

This scrap is another continuation. It is unremarkable, except that it contains two other characters likely to be an issue for the SGML tools, specifically the "greater than" and ampersand characters (">" and "&").

⟨The main code (§1'scrap1.out')⟩≡ +

```
  -- Second continuation of scrap1
  if (i < 10) && (j > 12)
    call iout
  fi
```

This is a header scrap, which is intended to be included in another scrap in order to finally be included in an output file. The scrap documentation is entirely unremarkable.

⟨An included scrap (scrap2) (§2)⟩≡

```
-- included scrap2 head
while a % b < c
  incr(a)
end
```

Some documentation of the next scrap. It is unremarkable in every way.

⟨An included scrap (scrap2) (§2)⟩≡ +

```
-- included scrap2 continuation 1
some more code
```

```
-- include scrap3 by reference
```

⟨A nested scrap (§3)⟩≡ +
```
-- continuation 1 of scrap 3
```
And finally, documentation of the third scrap. It is entirely unremarkable, except that it includes a reference to the scrap that it is included in, which is ⟨An included scrap (scrap2) (§2)⟩.

⟨A nested scrap (§3)⟩≡
```
-- contents of scrap3
-- scrap 3 should have continuation 1
```

The third scrap is continued. The documentation is entirely unremarkable, and is extended only to provide some reasonable text in the woven file.

⟨A nested scrap (§3)⟩≡ +
```
-- continuation 1 of scrap 3
```