This is Scribe!

Manuel Serrano
Inria Sophia-Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Manuel.Serrano@sophia.inria.fr
http://www.inria.fr/mimosa/Manuel.Serrano

Erick Gallesio
Université de Nice - Sophia Antipolis
930 route des Colles, BP 145
F-06903 Sophia Antipolis, Cedex
France
Erick.Gallesio@unice.fr
http://saxo.essi.fr/~gallesio

ABSTRACT

This paper presents Scribe, a functional programming language for authoring documents. Even if it is a general purpose tool, it best suits the writing of technical documents such as web pages or technical reports, API documentations, etc. Executing Scribe programs can produce documents of various formats such as PostScript, PDF, HTML, Texinfo or Unix man pages. That is, the very same program can be used to produce documents in different formats. Scribe is a full featured programming language but it *looks* like a markup language *à la* HTML.

1. INTRODUCTION

Scribe is a functional programming language designed for authoring documentations, such as web pages or technical reports. It is built on top of the Scheme programming language [5]. Its concrete syntax is simple and it sounds familiar to anyone used to markup languages. Authoring a document with Scribe is as simple as with HTML or LaTeX. It is even possible to use it without noticing that it is a programming language because of the conciseness of its original syntax: the ratio *markup/text* is smaller than with the other markup systems we have tested.

Executing a Scribe program with a Scribe evaluator produces a target document. It can be HTML files that suit web browsers, LaTeX files for high-quality printed documents, or a set of *info* pages for on-line documentation.

Building purely static texts, that is texts avoiding any kind of computation, is generally not sufficient for elaborated documents. Frequently one needs to automatically produce parts of the text. This ranges from very simple operations such as inserting in a document the date of its last update or the number of its last revision, to operations that work on the document itself. For instance, one may be willing to embed inside a text some statistics about the document, such as the number of words, paragraphs or sections it contains. Scribe is highly suitable for these computations. A program is made of *static texts* (that is, *constants* in the programming jargon) and various functions that dynamically compute (when the Scribe program runs) new texts. These functions are defined in the Scheme programming language. The Scribe syntax enables a sweet harmony between the static and dynamic components of a program.

Authoring documents with a programming language is of course not a novel idea, and a lot of systems have used this approach, such as the TeX [8] typesetting system. PostScript [1] can also be classified in this category. Even if it is not generally directly used for authoring, it represents a document as a program whose execution yields a set of printed pages.

On the other side, solutions based on the SGML [2] or XML [3] formats propose a model where all the computations on a document are expressed outside of the document itself. For instance, the DOM [20] approach extols a strict dichotomy between documents and programs. This dichotomy is presented as a virtue by its proponents, but it is our opinion that it makes simple documents harder to code than with a general linguistic tool because it requires the usage of several different languages with different semantics and different syntax.

With the development of dynamic content web sites, a great number of intermediate solutions based on programming languages have been proposed. These solutions generally consist in giving a way to embed calls to a programming language inside a document. PHP [9] is probably the most representative of this kind. A document is a mix of text and code expressed with different syntaxes. This implies that the author/programmer must deal at the very same time with the underlying text markup system as well as the programming language. Furthermore, these tools do not permit to reify a document structure and are generally limited to the production of web pages only.

The approach we propose is inspired by the LAML system

[12] which uses Scheme as a markup language. In LAML as in SCRIBE, a document is a program and its evaluation yields its final form. Both languages permit the user to typeset documents using an *unique* syntax. However, LAML is limited to the production of HTML, whereas, as said before, the evaluation of a SCRIBE program can produce several output formats.

In Section 2 we present an overview of the SCRIBE system for authoring simple static documents. We show that a SCRIBE program looks like a document specified in a markup language. A more complex usage of the language is shown in Section 3, where some simple text generations are done, as well as some text inclusions built by introspecting the document itself. Section 4 shows various customizations that can take place during the execution of a SCRIBE program. Finally, we compare in Section 5 SCRIBE with various tools or programming languages used for authoring documents.

2. SCRIBE OVERVIEW

This section presents an overview of the SCRIBE programming language and its implementation. First, the syntax is presented in Section 2.1. Then, in Section 2.2, the structure of a program is presented. Finally, Section 2.3 contains some few words about the current state of the SCRIBE implementation.

2.1 Sc-expressions

We have designed the SCRIBE syntax so that it as *unobtrusive* as possible. We have found of premium importance to minimize the weight of meta information when authoring documentations. A complex syntax would prevent it to be used by non computer scientists. A SCRIBE program is a list of expressions (Sc-expression henceforth) that are extended S-expressions [11]. An Sc-expression is:

- An *atom*, such as a string or a number.
- A *list* of Sc-expressions.
- A *text*.

Atomic expressions and *lists* are regular Scheme expressions. A *text* is a sequence of characters enclosed inside square brackets. This is the sole extension to the standard Scheme reader. The bracket syntax is very similar to the standard *quasiquote* Scheme construction. In Scheme, the *quasiquote* syntax allows to enter complex lists by automatically *quoting* the components of the list. It is to be used in conjunction of the *comma* operator that allows to *unquote* the expressions. For instance, the Scheme form:

```
`(compute pi = ,(* 4 (atan 1)))
```

is equivalent to the expression:

```
(list 'compute 'pi '= (* 4 (atan 1)))
```

which evaluates to:

```
(compute pi = 3.1415926535898)
```

The SCRIBE bracket form collects all the characters between the brackets in a list of characters strings. Computations inside brackets are handled by the characters sequence "`,(`". For instance, the text:

```
[text goodies: ,(bold "bold") and ,(it "italic").]
```

is parsed by the SCRIBE reader as:

```
(list "text goodies: " (bold "bold")
      "and" (it "italic") ".")
```

The SCRIBE syntax is unobtrusive, and easy to typeset with an editor aware of Lisp-like syntax, such as *Emacs*. Documents expressed in SCRIBE are also generally shorter to type-in than their counterpart expressed in classical formatting languages. For instance, the size of the SCRIBE source files of this paper is about 42,200 characters long, whereas it is 53,000 characters in LaTeX and 72,000 in HTML. Even if it is somehow unfair to compare hand-written code against generated ones, these figures give the intuition of the compactness of SCRIBE programs. The idea of extending a standard Scheme reader for text processing comes from the BRL system [10].

2.2 Scribe as a markup language

In this section, we present how to build a document using SCRIBE. As said before, programming skill is not needed to produce a document. In fact, non programmer writers can see SCRIBE as a simple document formatting system such as HTML or nroff [14].

SCRIBE provides an extensive set of pre-defined markups. These roughly correspond to the HTML markups. The goal of this section is to give an idea of the *look and feel* of this system. It will avoid the tedious presentation of an extensive enumeration of all the markups available. For a complete manual of SCRIBE, interested readers can have a look at `http://www-sop.inria.fr/mimosa/fp/Scribe`.

2.2.1 Scribe Markups

A SCRIBE markup is close to an XML element. The attributes that can appear inside an XML element are represented by Scheme keywords. They are identifiers whose first (or last character) is a colon. Scheme keywords have been introduced by DSSSL [4], the tree manipulation language associated to SGML. So, the following XML expression:

```
<elmt1 att1="v1" att2="v2">
   Some text <elmt2>for the example</elmt2>
</elmt1>
```

is represented in SCRIBE as:

```
(elmt1 :att1 v1 :att2 v2
       [Some text ,(elmt2 [for the example])])
```

2.2.2 Document Structure

As said before, a SCRIBE program consists in a list of Sc-expressions. Among these, the `document` one serves a special purpose. It is used to represent the complete document. All the subdivisions of a document must appear as arguments of the `document` call. So, the general structure of a SCRIBE document looks like:

```
<sc-expr>
...
(document :title <sc-expr> :author <sc-expr>
  (abstract <sc-expr>)
  (section :title <sc-expr>
     ...
     (subsection :title <sc-expr>)
     ...
     (subsection :title <sc-expr>)
     ...)
   ...
   (section :title <sc-expr>))
```

As we can see, all the sectioning components of a document are embedded in their containing component (i.e. subsections are embedded in sections, sections are inside chapters, and so on). This strict nesting of document components is particularly useful when one wants to do introspection on the structure of the document, as we will see in Section 3.2.

2.2.3 Scribe standard library

SCRIBE is provided with the usual functions for text processing. Some of these are presented here.

The *Lists* offered in SCRIBE are classical: itemization, enumeration and description. For instance, the following expression:

```
(itemize (item [A first item.])
         (item [A ,(bold "second") one.])
         (item (description
                 (item :key (bold "foo")
                       [is a usual Lisp identifier.])
                 (item :key (bold "bar")
                       [is another one.])))
         (item (enumerate (item "One.")
                          (item "Two."))))
```

produces the following output text:

- *A first item.*
- *A* ***second*** *one.*
- **foo** *is a usual Lisp identifier.*
  bar *is another one.*
- 1. *One.*
  2. *Two.*

Of course, all the usual text ornaments are available in SCRIBE, that is one can easily produce text in **bold**, *italic*, underline or combine ***them***.

The SCRIBE standard library also offers the usual tools for inter and intra document references, footnotes, tables, figures, ... It provides also an original construction, the **prgm** markup, to *pretty-print* codes or algorithms. In contrast with previous systems such as LaTeX there is no need, in SCRIBE to use external pre-processors such as SLaTex [17] and Lisp2TeX [15] for pretty-print programs inside texts. The **prgm** form takes as an option the language in which the code is expressed and its evaluation yields a form that is the pretty-printed version of this code. For instance, the following call

```
(prgm :language c (from-file "ex/C-code.c"))
```

produces the following output

```
int main(int argc, char **argv) {
  /* A variant of a classical C program */
  printf("Hello, Scribe\n");
  return 0;
}
```

if the C program source is located in file `ex/C-code.c`.

2.3 Front-ends and Back-ends

The current version of SCRIBE which is available at `http://www-sop.inria.fr/mimosa/fp/Scribe` contains two front-ends which are used to translate existing document sources into SCRIBE documents:

- `scribeinfo` compiles Texinfo into SCRIBE. An example of such a compilation can be browsed at `http://www.inria.fr/mimosa/fp/Bigloo/doc/r5rs.html`. It is an on-line version of the Scheme definition, automatically produced from a Texinfo source.
- `scribebibtex` translates Bibtex bibliography databases into SCRIBE sources. This tool is, for instance, used to produce the bibliographic references of this paper.

SCRIBE can produce various kinds of document formats. Currently five back-ends are supported:

- **HTML**: It is extensively used on the SCRIBE web page.
- **PS** or **PDF** (via LaTeX): That is, for instance, used to produce the PostScript version of this paper.
- **Man**: which is the format of Unix "man pages".
- **Text**: which is a plain text format.
- **Info**: which is the format of the Emacs documentation.

SCRIBE user programs are independent of the target formats. That is, using one unique program, it is possible to produce an HTML version, and a PostScript version, and an ASCII version, etc. The SCRIBE API is general purpose.

It is not impacted by specific output formats. Independence with respect to the final document format does not limit the expressiveness of SCRIBE programs because specificities of particular formats are handled by dedicated back-ends. Back-ends are free to find convenient ways to implement SCRIBE features. For instance, intra document references are handled differently by the HTML back-end and the TEX back-end. In HTML, they appear as hyper-links whose text is the title of the section. In TEX they appear as section numbers. An output target may even not support some SCRIBE features. In that case, the back-end could possibly omit them (for instance, figures in ASCII formats, or dialog boxes in PostScript documents).

When customization of the produced documents is required, the SCRIBE `hook` form must deployed. It enables to insert characters in the final document. Coupled with conditional evaluation, the `hook` form can be used to implement fine grain tuning aware of the idiosyncrasies of the target format (see Section 3.3).

3. DYNAMIC TEXTS

We show in this section various situations where *dynamic texts*, that is texts not written *as is* in the SCRIBE sources, can be used when authoring documents. We have isolated two kinds of computations. The ones that produce some parts of the document being processed (Section 3.1). The ones that involve *introspection* on the source text (Section 3.2). These computations correspond to two different evaluation stages of the SCRIBE evaluator. The first ones are *front-end* computations that take place at the very beginning of the execution of a program. The second ones are *back-end* computations that take place at the very end of the execution while an internal representation of the whole SCRIBE program has been loaded in memory.

3.1 Computing Sc-expressions

Many typesetting systems such as LATEX enable users to define convenience *macros*. In its simplest form, a *macro* is just a name that is *expanded into*, or *replaced with*, a text that is part of the produced document. Macros are implemented in SCRIBE by the means of functions that produce Sc-expressions. For instance, a macro defining the typesetting of the word "SCRIBE" is used all along this paper. It is defined as follow:

```
(define (Scribe.tex)
   (sc "Scribe"))
```

That can be used in a Sc-expression such as:

```
[This text has been produced by ,(Scribe.tex).]
```

That produces the following output:

"*This text has been produced by* SCRIBE."

The function `Scribe.tex` is overly simple because it merely *inserts* in the SCRIBE program one new string each time it is called. Sometimes we need to compute more complex parts of a document and some texts are better to be computed. Either because they contain pattern repetitions or because they are the result of the evaluation of an algorithm, such as the table of Figure 1.

n=	fact
3	*6*
4	*24*
5	*120*
6	*720*
7	*5040*
8	*40320*
9	*362880*
10	*3628800*
11	*39916800*

Figure 1: Factorial

This table can be *statically* declared in a program using a Sc-expression such as:

```
(table :border 1
       (tr (th "n=") (th "fact"))
       (tr (td :align 'center (bold 3))
           (td :align 'right (it 6)))
       (tr (td :align 'center (bold 4))
           (td :align 'right (it 24)))
       (tr (td :align 'center (bold 5))
           (td :align 'right (it 120)))
       (tr (td :align 'center (bold 6))
           (td :align 'right (it 720)))
       (tr (td :align 'center (bold 7))
           (td :align 'right (it 5040)))
       (tr (td :align 'center (bold 8))
           (td :align 'right (it 40320)))
       (tr (td :align 'center (bold 9))
           (td :align 'right (it 362880)))
       (tr (td :align 'center (bold 10))
           (td :align 'right (it 3628800)))
       (tr (td :align 'center (bold 11))
           (td :align 'right (it 39916800))))
```

Obviously the table construction can be automated. The factorial values can be computed and the table rows can be generated. Unlike many other markup languages, SCRIBE enables this computation to take place inside the document itself. Let us assume the standard definitions for the `upto` and `fact` functions:

```
(define (upto min max)
   (if (= min max)
       (list max)
       (cons min (upto (+ min 1) max))))

(define (fact n)
   (if (< n 2)
       n
       (* n (fact (- n 1)))))
```

The generation of the factorial table requires two additional SCRIBE functions. The first one builds table rows:

```
(define (make-fact-row n)
   (tr (td :align 'center (bold n))
       (td :align 'right (it (fact n)))))
```

The second one is in charge of creating the table:

```
(define (make-fact-table n)
   (apply table :border 1
          (tr (th "n=") (th "fact"))
          (map make-fact-row (upto 3 n))))
```

3.2 Computing Sc-ast

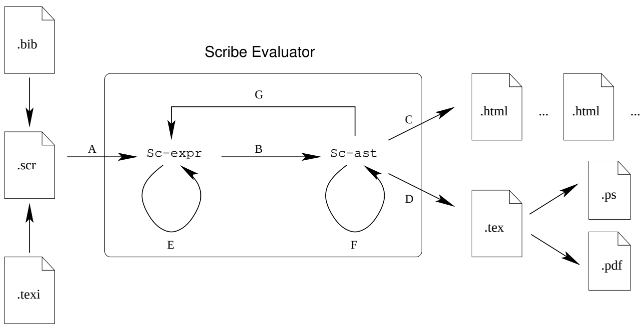

Figure 2: The Scribe process

The evaluation of a SCRIBE program involves three steps (see Figure 2):

- First, the source file is read and represented as a list of Sc-expressions (edge "A").
- Second, the Sc-expressions are evaluated using the standard SCRIBE library. This produces an abstract syntax tree named Sc-ast (edge "B").
- Third, the Sc-ast is translated into the target format *i.e.,* HTML, LaTeX , ... (edges "C" and "D").

The computations previously presented in Section 3.1 take place on the edge "E". This section focuses now on the computations that are involved on edges "F" and "G".

Frequently some parts of a document may refer to the document itself. For instance, *introspection* is needed to compute a table of contents. SCRIBE is provided with introspection facilities that can be used in user programs. For instance, it enables the computation of such a sentence:

"*This document contains 9 sections.*"

The actual number of sections is the result of a user computation. The whole sentence is computed by the following Sc-expression:

```
[This document contains
,(hook :after
       (lambda ()
          (display (length
                     (document-sections*
                      (current-document))))))
sections.]
```

It uses the SCRIBE library function `hook` which enables computations to take place while the Sc-ast is built, that is on the edge "F" of Figure 2. The `:after` argument is a function which is executed once the Sc-ast is translated into the target format. It *prints* a string that is inserted in the target. Obviously, the dynamic text of the previous example cannot be computed earlier in the SCRIBE evaluation process since the number of sections cannot be computed until all the sections are built! The function of the standard library `current-document` returns a structure that describes the document being processed. The function `document--sections*` returns the list of sections contained in a document. Not that, since the `hook` function enables arbitrary characters insertion, it can be used to introduce low level back-end commands such as TeX commands or HTML commands in the target. For instance, the SCRIBE command `LaTeX` which produces the following "LaTeX" is implemented as:

```
(define-markup (LaTeX)
   (if (scribe-format? 'tex)
       (hook :after (lambda () (display "\\LaTeX")))
       "LaTeX"))
```

Sometimes, instead of printing characters into the target, it is needed that the evaluation of a `hook` node produces a fresh Sc-expression. That is, an expression that has to be evaluated by the SCRIBE engine (the edge "G" of Figure 2)[1]. This is illustrated by the following example. The user function `document-tree` computes the hierarchical structure of a document. Applied to the current document it produces:

```
+--ABSTRACT
+--1 Introduction
+--2 Scribe overview
|  +--2.1 Sc-expressions
|  +--2.2 Scribe as a markup language
|  |  +--2.2.1 Scribe Markups
|  |  +--2.2.2 Document Structure
|  |  +--2.2.3 Scribe standard library
|  +--2.3 Front-ends and Back-ends
+--3 Dynamic texts
|  +--3.1 Computing Sc-expressions
|  +--3.2 Computing Sc-ast
|  +--3.3 Conditional execution
+--4 Customization
+--5 Related work
|  +--5.1 SGML and XML
|  +--5.2 Scheme vs. other functional languages
|  +--5.3 LAML
|  +--5.4 BRL
|  +--5.5 Wash
+--6 Conclusion
+--7 References
+--APPENDIX
```

Figure 3: Tree

The tree branches are displayed using a typewriter font and a layout that preserves spaces and line breaks. The tree

[1]Introducing a fresh Sc-expression in the tree may introduce incoherences for cross-references. When iterations are needed, it belongs to the programmer to implement it.

nodes are displayed underlined and in italic. The computation involved in document-tree produces a regular Sc-expression that is evaluated by the SCRIBE engine. This ensures back-end independence because it prevents the hook call to specify how underline and italic have to be rendered for each specific target format. The function document-tree is defined as:

```
(define (document-tree)
   (hook :process #t
         :after (lambda ()
                  (prgm
                   (make-tree (current-document))))))
```

The argument :process #t means that the result of the application of the :after function has to be evaluated back by the SCRIBE engine. This function constructs a new Sc-expression which is made of a prgm call. The definition of make-tree is:

```
(define (make-tree doc)
   (let loop ((s (scribe-get-children doc))
              (m "")
              (f underline))
       (if (null? s)
           '()
           (append (make-row m (car s) f)
                   (loop (scribe-get-children (car s))
                         (string-append m "|  ")
                         it)
                   (loop (cdr s) m f)))))
```

The function make-row is:

```
(define (make-row m s f)
   (list (string-append m "+--")
         (f (scribe-get-title s))
         "\n"))
```

The library function scribe-get-children returns the elements contained in a section or a subsection. The library function scribe-get-title returns the title of a section or a subsection.

In addition to illustrating SCRIBE introspection, this example also shows how suitable functional programming languages are to compute over texts: the whole implementation of Figure 3 is a simple recursive traversal of the tree representing the document (function make-tree).

3.3 Conditional execution

Conditional execution is required when the text to be produced depends on some properties of the target format. The scribe-format? predicate checks which target format is to be produced. It is used several times in the paper. For instance, in Section 3.1 we have presented the definition of the Scribe.tex macro. The actual macro used in the sources of this paper is slightly more complex. Instead of rendering the word "Scribe", when targeting HTML, it introduces a reference to the SCRIBE home page. Moreover, because of our poor English style, we have also decided to introduce an URL link only *once* per section. So, the actual function used in the paper source is defined as:

```
(define Scribe
   (let ((sec #f))
      (lambda ()
         (if (scribe-format? 'html)
             (hook :after
                   (lambda ()
                      (let ((s (current-section)))
                         (if (eq? s sec)
                             (Scribe.tex)
                             (begin
                                (set! sec s)
                                (ref :url (scribe-url)
                                     "Scribe")))))
                   :process #t)
             (Scribe.tex)))))
```

4. CUSTOMIZATION

A *real* and *practical* programming language is useful when considering *customizations* (in SCRIBE they usually take place in *style* files). SCRIBE customizations enable users to change the way documents are rendered. They are ubiquitous in the standard SCRIBE API. For instance, one may setup the way a bold text is rendered, configure the header and the footer of the document, or even define margins. One may also specify the structure of the produced documents. In this section we illustrate how one may benefit from the expressiveness of SCRIBE in order to achieve complex customizations. In particular, we will show how computers program can be rendered.

Depending of the specified language, SCRIBE uses different colors and fonts when rendering computer programs. The standard implementation supports several languages such as SCRIBE, Scheme, C, or XML. Computer programs are specified by the prgm markup (see Section 2.2.3) which accepts one optional argument which is a function implementing the rendering of the program. This function is called a *pretty-printer*. One may define its own pretty-printers.

For the sake of the example, let us implement a pretty-printer for rendering *makefiles* which uses some colors for make targets, variables, and comments. In addition, for back-ends supporting hyper links (such as HTML) a reference to its definition is added to the text when a variable is used. For other back-ends, variable references are underlined.

```
SCRIBE= scribe
SFLAGS= -J style

MASTER= main.scr
INPUT= abstract.scr intro.scr what.scr why.scr this.scr
EXAMPLE= ex0 ex1 ex2 ex3 ex4 makefile
STYLE= style/local.scr

# main entry
all: scribe.tex

scribe.tex: $(MASTER) $(INPUT) $(STYLE) $(EXAMPLE)
$(SCRIBE) $(SFLAGS) $(MASTER) -o scribe.tex
```

A pretty-printer function is a SCRIBE function accepting one parameter. This formal parameter is bound to a string representing the text to be pretty-printed. A pretty-printer returns a Sc-expression representing the pretty-printed program that must be included in the target document. The definition of the `makefile` pretty-printer is:

```
(define (makefile obj)
  (parse-makefile (open-input-string obj)))
```

In order to implement the pretty-printer we are using Bigloo regular parser [16]. This mechanism enables a lexical analysis of character strings.

```
(define (parse-makefile port::input-port)
   (read/rp
    (regular-grammar ()
       ((: #\# (+ all))
        ;; makefile comment
        (let ((cmt (the-string)))
           (cons (it cmt) (ignore))))
       ((bol (: (+ (out " \t\n:")) #\:))
        ;; target
        (let ((prompt (the-string)))
           (cons (bold prompt) (ignore))))
       ((bol (: (+ alpha) #\=))
        ;; variable definitions
        (let* ((len (- (the-length) 1))
               (var (the-substring 0 len)))
           (cons `(list ,(mark var)
                        (color :fg "#bb0000" (bold ,var))
                        ,"=") (ignore))))
       ((+ (out " \t\n:=$"))
        ;; plain strings
        (let ((str (the-string)))
           (cons str (ignore))))
       ((: #\$ #\( (+ (out " )\n")) #\))
        ;; variable references
        (let ((str (the-string))
              (var (the-substring 2 (- (the-length) 1))))
           (cons (ref :mark var (underline str))
                 (ignore))))
       ((+ (in " \t\n:"))
        ;; separators
        (let ((nl (the-string)))
           (cons nl (ignore))))
       (else
        ;; default
        (let ((c (the-failure)))
           (if (eof-object? c)
               '()
               (error "prgm(makefile)"
                      "Unexpected character"
                      c)))))
    port))
```

5. RELATED WORK

In this section we compare SCRIBE and other markup languages. We also compare it with other efforts for handling texts in functional programming languages.

5.1 SGML and XML

As stated in [3] *"XML, the Extensible Markup Language, is W3C-endorsed standard for document markup. It defines a generic syntax used to mark up data with simple, human-readable tags. It provides a standard format for computer documents"*. In other words, XML is a mean to specify external representations for data structures. It is a mere formalism for specifying grammars. It can be used to represent texts but this is not its main purpose. The most popular XML application used for representing texts (henceforth XML texts) is XHTML (a reformulation of HTML 4.0). XML can be thought as a simplification of SGML. They both share the same goals and syntax.

The fundamental difference between XML and SCRIBE is that the first one is definitely not a programming language. In consequence, any processing (formating, rendering, extracting) over XML texts requires one or several external tools using different programming languages which appear to be, most of the time, Java, Tcl, and C. A vast effort has been made to provide most of the functional programming languages with tools for handling XML texts. It exists XML parsers for mostly all functional programming languages. Haskell has HaXml [19], Caml has Px and Tony, and Scheme has SSax [7].

In addition to parsers, Scheme has also SXML [6] which is either an abstract syntax tree of an XML document or a concrete representation using S-expressions. SXML is suitable for Scheme-based XML authoring. It is a term implementation of the XML document.

The document style semantics and specification language (aka DSSSL [4]) defines several programming languages for handling SGML applications. The DSSSL suite plays approximatively the same role as XML XSLT, DOM and SSAX do: it enables parsing and computing over SGML documents. The DSSSL languages are based on a simplified version of Scheme.

XEXPR [21] is a scripting language that uses XML as its primary syntax. It has been defined to easily embed scripts inside XML documents and overcomes the usage of an external scripting language in order to process a document. The language defines itself to be *very close to a typical Lisp or combinator-based language where the primary means of programming is through functional composition.* XEXPR allows the definition of functions using the `<define>` element. Hereafter is a definition of the factorial function expressed in XEXPR:

```
<define name="factorial" args="n">
 <if>
   <lt><n/>2</lt>
   <n/>
   <multiply>
    <n/>
    <factorial>
      <substract><n/>1</substract>
    </factorial>
   </multiply>
 </if>
</define>
```

which must be compared with the Scheme version given in Section 3.1. Obviously, writing by hand large scripts seems hardly achievable in XEXPR. Furthermore, a careful reading of the report defining this language seems to indicate that there is no way to manipulate the document itself inside an XEXPR expression. The language seems then limited to simple text generations inside an XML document, as the ones presented Section 3.1

Besides deploying one unique formalism and syntax for authoring documents we have found that SCRIBE enables more compact sources than XML (see Section 2.1). The SCRIBE syntax is less verbose than the XML one mainly because the closing parenthesis of a Sc-expression is exactly one character long when it is usually much more in XML.

5.2 Scheme vs. other functional languages

We have chosen to base SCRIBE on Scheme mainly because its syntax is genuinely close to traditional markup languages. Such as XML, the Scheme syntax is based on the representation of trees. The modifications to apply to the Scheme grammar are very limited and simple. This makes this language suitable for text representation. The other functional languages such as, Caml and Haskell, rely on LALR syntaxes that do not fit the markup look-and-feel.

In addition, we think that the Scheme type system is an advantage for SCRIBE programs. It is convenient to dispose of fully polymorphic data types. As presented in Section 2.1, an Sc-expression can be a list whose elements are of different types. For instance, the first element of such a list could be a character string and the next one a number. This enables compact representation of texts. If the underlying language imposes a stronger typing system, the source program, that is the user text, will be polluted with cast operations that transform all the values into strings.

We have considered using a *call-by-name* semantics for SCRIBE function application in order to implement the nesting of Sc-expressions. As presented in Section 3.2 the SCRIBE library proposes introspection functions. For instance, the `document-sections*` returns the list of sections contained in a document structured such as:

```
(document ...
   (chapter ...)
   (chapter
      ...
      (section ...)
      ...)
   ...)
```

The container nodes (representing documents, chapters, sections, ...) of the Sc-ast are provided with pointers to the children they contain and *vice versa*. Since laziness enables to postpone the computation of expressions until they are required, it can be used to delay the evaluation of inner elements of a document until the whole document is declared. We have obtained the same effect by adding a second traversal of the Sc-ast (see 3.2).

5.3 LAML

LAML (*Lisp as a Markup Language*) [13] is an attempt to use Scheme as a markup language. It mirrors the HTML markups in Scheme. That is, for each HTML markup there is a corresponding Scheme function in LAML. The HTML document:

```
<html>
  <head><title>An example</title></head>
  <body>
    <br>
    This is an <em>HTML</em>example.
    <br>
  </body>
</html>
```

is mirrored in LAML as:

```
(html
  (head (title "An example"))
  (body (br)
        "This is an" (em "HTML") "example."
        (br)))
```

So, LAML and SCRIBE are very close. They rely on the natural Scheme syntax and they both consider a document as a program. However, there is two important differences between them:

- The syntax: SCRIBE uses an extended Scheme syntax. As presented Section 2.1, it introduces the `[...]` notation that, as we have shown, enables compact source texts.
- The Sc-ast: The evaluation of a LAML function call directly produces an HTML expression. For instance, the definition of the LAML `em` function of the previous example is:

  ```
  (define (em str)
     (string-append "<EM>" str "</EM>"))
  ```

  Contrarily to SCRIBE, LAML does not build a tree representing the text to be generated. This direct mapping has three drawbacks:

  1. LAML sources cannot produce other formats than HTML.
  2. It is complex to implement efficiently a LAML interpreter. As reported in [12], the LAML evaluation process allocates a lot of strings of characters. This exercises intensively the memory manager (garbage collection and memory copies). These string manipulations are totally avoided by SCRIBE. One SCRIBE markup allocates one object that is a node of the Sc-ast. This node is used until the back-end has completed the file generation. It never happens that a node nor the characters is contains are duplicated.
  3. Introspection over a LAML document is complex. In particular, it has to take place before the string representing an HTML expression is built. That is, it has to take place before LAML functions are called. In other words, LAML is of no help for computing on documents. LAML users have to implement their own data representation before using LAML functions.

5.4 BRL

The *Beautiful Report Language* BRL [10] defines itself as a database-oriented language to embed in HTML and other markups. In some extent BRL approach is very similar to the PHP one: it proposes to mix the text and the program which form the document in the same source file. For BRL, a document is a sequence of either strings or Scheme expressions. BRL displays strings *as is* and *evaluates* Scheme expressions. To alleviate document typesetting using this conventions, BRL has introduced a new syntax for character strings: there is no need to put a quote for a string starting a file or terminating a file. Furthermore, "]" and "[" can be used to respectively open and close a string. So,

```
]a string[
```

is a valid string in BRL. The interest of this notation seems more evident in a construction such as

```
The value of pi is [(* 4 (atan 1))].
```

where we have a Scheme expression enclosed between two strings ("`The value of pi is`" and "`.`"). However, this syntax can be sometimes complex as it is shown in the following excerpt from the reference manual.

```
[(define rowcount (sql-repeat ...)
  (brl ]<li><strong>
<a href="p2.brl?[
(brl-url-args brl-blank? color)
]">(brl-html-escape color)]</a></strong>
[)))]
```

As we can see, BRL is just a sort of *preprocessor* and as such it cannot be used to do introspective work on a document.

5.5 Wash

Wash [18] is a family of embedded domain specific languages for programming Web applications. Each language is embedded in the functional language Haskell, which means that it is implemented as a combinator library. The basic idea of Wash is to build a data structure that can be rendered to HTML text. Because of the type system of the Haskell type checker, Wash guarantees the well-formedness of the generated HTML pages. Using a Haskell interpreter it is possible with Wash to interactively create and manipulate web pages.

If Wash shares with Scribe the construction of a data structure representing the text to be rendered, no effort is made to provide it with a concise syntax. A "hello, word" page which is in HTML:

```
<html>
  <head>
   <title>Hello, World</title>
  </head>
  <body>
    This is the traditional &#34;Hello, World!&#34; page.
    <hr>
  </body>
</html>
```

and that can be implemented in Scribe as:

```
(define *title* "Hello, World!")
(document :title "Hello, World" [
This is the traditional ,(begin *title*) page.
,(hrule)])
```

would be written in Wash as:

```
doc_head :: HEAD
doc_head =
  make_head
   `add` (make_title `add` "Hello, World")

doc_body :: BODY
doc_body =
  make_body
    `add` (make_heading 1 `add` title)
    `add` ("This is the traditional \""
           ++ title ++
           "\" page.")
    `add` make_hr
     where title = "Hello, World!"

doc :: HTML
doc = make_html `add` doc_head `add` doc_body

putStr (show_html doc)
```

It is obvious that Wash is designed *for* programmers. Unlike Scribe it cannot be as easily used in replacement of traditional markup languages.

6. CONCLUSION

Scribe is a functional programming language for authoring various kind of electronic documents. It can be used to produce target formats such as HTML and PostScript. It relies on an original syntax that makes it looking familiar to anyone used to markup languages such as HTML.

We have shown that the evaluation of a Scribe program involves two separate stages. During the first one the source expressions are read using the Scribe reader. These expressions are then evaluated using a classical Scheme interpreter. This stage produces an internal representation of the source program. The second evaluation stage uses that representation and, as a consequence, enables computations on the representation itself. That is, during the second stage a Scribe program may compute properties about itself.

Scribe is used on daily basis to produce large documents. For instance, the whole web page `http://www.inria.fr-/mimosa/fp/Bigloo` and the documentations it contains are implemented in Scribe. Obviously, the current paper *is* a Scribe program. An HTML version can be browsed at `http://www.inria.fr/mimosa/fp/Scribe/doc/scribe.html`.

7. REFERENCES

[1] Adobe System Inc. – **PostScript Language Reference Manual** – *Addison-Wesley, Readings, Massachusetts*, 1985.

[2] Goldfarb, C. – **The SGML Handbook** – *Oxford University Press*, 1991.

[3] Harold, E.R. and Means W.S. – **XML in a nutshell** – *O'Reilly*, Jan, 2001.

[4] ISO/IEC – **Information technology, Processing Languages, Document Style Semantics and Specification Languages (DSSSL)** – 10179:1996(E), *ISO*, 1996.

[5] Kelsey, R. and Clinger, W. and Rees, J. – **The Revised(5) Report on the Algorithmic Language Scheme** – *Higher-Order and Symbolic Computation*, 11(1), Sep, 1998.

[6] Kiselyov, O. – **Implementing Metcast in Scheme** – *Scheme workshop*, Montréal, Canada, Sep, 2000.

[7] Kiselyov, O. – **A better XML parser through functional programming.** – *Practical Aspects of Declarative Languages*, Portland, Oregon, USA, Jan, 2002.

[8] Knuth, D. – **The TEXbook,** – *Addison-Wesley, Readings, Massachusetts*, 1986.

[9] Lerdorf, R. – **PHP Pocket Reference** – *O'Reilly & Associates*, Jan, 2000.

[10] Lewis, B – **BRL Reference Manual** – `http://brl.sourceforge.net/`2002.

[11] McCarthy, J. – **Recursive functions of symbolic expressions and their computation by machine – I** – *Communications of the ACM*, 3(1), 1960, pp. 184–195.

[12] Nørmark, K. – **Programming World Wide Web Pages in Scheme** – *Sigplan Notices*, 34(12), 1999.

[13] Nørmark, K. – **Programmatic WWW authoring using Scheme and LAML** – *The Eleventh International World Wide Web Conference*, Honolulu, Hawaii, USA, May, 2002.

[14] Ossana, J. – **UNIX Programmer's manual: Supplementary Documents** – 1982.

[15] Queinnec, C. – **Literate programming from Scheme to TeX** – LIX RR 93.05, *Laboratoire d'Informatique de l' Polytechnique*, 91128 Palaiseau Cedex, France, Nov, 1993.

[16] Serrano, M. – **Bigloo user's manual** – 0169, *INRIA-Rocquencourt*, France, Dec, 1994.

[17] Sitaram, D. – **SLaTeX** – `http://www.ccs.neu.edu/home-/dorai/slatex/slatxdoc.html`.

[18] Thiemann, P. – **Modeling HTML in Haskell** – *Practical Aspects of Declarative Languages*, 2000, pp. 263–277.

[19] Wallace, M. and Runciman, C. – **Haskell and XML: Generic Combinators or Type-Based Translation?** – *Int'l Conf. on Functional Programming*, Paris, France, 1999.

[20] World Wide Web Consortium – **Document Object Model (DOM) Level 1 Specification** – *W3C Recommendation*, Oct, 1998.

[21] World Wide Web Consortium – **XEXPR - A Scripting Language for XML** – *W3C Note*, Nov, 2000.

APPENDIX

For the sake of the example, we present in this Annex, the whole SCRIBE source code for the abstract of this paper:

```
(paragraph [
This paper presents ,(Scribe), a functional programming
language for authoring documents. Even if it is a general
purpose tool, it best suits the writing of technical
documents such as web pages or technical reports, API
documentations, etc.  Executing ,(Scribe) programs can
produce documents of various formats such as PostScript,
PDF, HTML, Texinfo or Unix man pages. That is, the very
same program can be used to produce documents in different
formats. ,(Scribe) is a full featured programming language
but it ,(emph "looks") like a markup language ,(emph "à la")
HTML.
]))
```

This paper has been generated by Scribe (`http://www-sop.inria.fr-/mimosa/fp/Scribe`) (via LaTeX and the `ACMproc` class.)