JFP **15** (5): 751–770, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796805005575 Printed in the United Kingdom

Skribe: a functional authoring language

ERICK GALLESIO
Université de Nice, Sophia Antipolis,
930 route des Colles, BP 145, F-06903 Sophia Antipolis, Cedex, France
(*e-mail:* Erick.Gallesio@unice.fr)

MANUEL SERRANO
Inria Sophia Antipolis, 2004 route des Lucioles, BP 93F-06902 Sophia Antipolis Cedex, France
(*e-mail:* Manuel.Serrano@sophia.inria.fr)

Abstract

This paper presents SKRIBE, a functional programming language for authoring documents, especially technical documents such as web pages, technical reports, and API documentation. Executing Skribe programs can produce documents in various formats, such as PostScript, PDF, HTML, Texinfo, or Unix man pages. That is, the very same Skribe program can be used to produce documents in different formats. Skribe is a full featured programming language whose syntax makes it look like a markup language à la HTML.
For the sake of the example, here is the whole SKRIBE source code for the paragraph above:

```
(p [This paper presents ,(Skribe), a functional programming language
for authoring documents, especially technical documents such as web
pages, technical reports, and API documentation. Executing Skribe
programs can produce documents in various formats, such as PostScript,
PDF, HTML, Texinfo, or Unix man pages. That is, the very same Skribe
program can be used to produce documents in different formats. Skribe
is a full featured programming language whose syntax makes it look
like a markup language à la HTML.])
```

SKRIBE can be downloaded at: `http://www.inria.fr/mimosa/fp/Skribe`.

1 Introduction

SKRIBE is a functional programming language designed for authoring documents, such as web pages or technical reports. It is built on top of the Scheme programming language (Kelsey *et al.*, 1998). Its concrete syntax is simple and it looks familiar to anyone used to markup languages. Authoring a document with SKRIBE is as simple as with HTML or LaTeX. Because of the conciseness of its original syntax, it is even possible to use it without noticing that it is a programming language. In SKRIBE, the ratio *markup/text* is smaller than with the other markup systems we have tested.

Executing a SKRIBE program with a SKRIBE evaluator produces a target document. It can be HTML files for web browsers, LaTeX files for high-quality printed documents, *info* pages for on-line documentation, etc.

Elaborated documents are generally made of fixed texts and dynamic information that is automatically generated by arbitrary computation. This computation 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 wish to embed inside text some statistics about a document, such as its number of words, paragraphs, or sections. Skribe is highly suitable for these computations. A program is made of *static texts* (that is, *constants* in programming jargon) and various functions that dynamically compute (when the Skribe program runs) new texts. These functions are defined in the Scheme programming language. The Skribe syntax enables a sweet harmony between the static and dynamic components of a program.

The evaluation of a Skribe program involves several separate stages. During the first stage, the source expressions are read using the Skribe reader. These expressions are then evaluated using a conventional Scheme interpreter, which produces an internal representation of the source program. The second evaluation stage uses the internal representation and enables computations on the representation itself. That is, during the second stage, a Skribe program may compute properties about itself. The third stage produces the output document.

Authoring documents with a programming language is not a novel idea, of course, and many systems have used this approach, such as the TEX typesetting system (Knuth, 1986). PostScript (Adobe System Inc., 1985) can also be classified in this category, even though it is not generally directly used for authoring, because it represents a document as a program whose execution yields a set of printable pages.

On the other side, the SGML (Goldfarb, 1991) or XML (Harold & Means, 2001) technologies offer a model where all the computations on a document are expressed outside of the document itself. For instance, the DOM (World Wide Web Consortium, 1998) approach extols a strict dichotomy between documents and programs. This separation is presented as a virtue by its proponents, but it is our opinion that it is a heavier approach for simple documents, since it forces authors to use 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 (Lerdorf, 2000) 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 same time with both the underlying text markup system and the programming language. Furthermore, these tools cannot reify a document's structure, and they are generally limited to the production of web pages.

Our approach was initially inspired by the LAML system (Nørmark, 1999), which uses Scheme as a markup language. In LAML, as in Skribe, a document is a program and its evaluation yields its final form. Both languages permit the user to typeset documents using a *single* syntax.

Skribe user programs are independent of the target format. That is, using a single program, it is possible to produce an HTML version, a PostScript version, an ASCII version, etc. The Skribe API is general purpose in the sense that it is not limited to specific output formats. At the same time, independence with respect to the final document format does not limit the expressiveness of SKRIBE programs because specificities of particular formats are handled by dedicated back-ends. Back-ends are free to find convenient ways to implement SKRIBE features. For instance, intra document references are handled differently by the HTML and $\TeX$ back-ends. 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 not support some SKRIBE features. In that case, the back-end could possibly omit them (for instance, figures in ASCII formats, or dialog boxes in PostScript documents).

The rest of this document is organized as follows. In section 2 we present an overview of the SKRIBE system for authoring simple static documents. We show that a SKRIBE program looks like a document specified in a markup language. In section 3, we show that SKRIBE is actually a full-fledged functional programming language. This section presents the main characteristics of the language and some examples. Finally, in section 4, we compare SKRIBE with other functional programming languages used for authoring documents.

2 Skribe overview

This section presents an overview of the SKRIBE language. First, the syntax is described in section 2.1. Then, in section 2.2, the structure of a document is presented.

2.1 Sk-expressions

A SKRIBE document is a list of expressions (Sk-expression henceforth) that are extended S-expressions (McCarthy, 1960). An Sk-expression is:

- An *atom*, including a string or a number, ex: `"hello world!"`, 42.
- A *list* of Sk-expressions, ex: `(list "hello" (bold "world") "!")`
- A *text*, ex: `[hello ,(bold [world])!]`

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 similar to the standard *quasiquote* Scheme construction, which supports complex lists by automatically *quoting* the components of the list. It is to be used in conjunction of the *comma* operator that *unquotes* 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 SKRIBE bracket form collects all the characters between the brackets in a list of characters strings. Computation inside bracketsis triggered by the character sequence

"`,(`". For instance, the text:

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

is parsed by the SKRIBE reader as:

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

The SKRIBE syntax is unobtrusive, and easy to type with an editor aware of Lisp-like syntax, such as *Emacs*. We have designed the SKRIBE syntax so that it as *unobtrusive* as possible. We have found of premium importance to minimize the weight of meta information when authoring documents. A complex syntax would prevent use by non computer scientists. Documents expressed in SKRIBE are also generally shorter than their counterpart expressed in classical formatting languages. For instance, the size of the SKRIBE source files of this paper is about 52,500 characters long, and running it through SKRIBE produces 76,000 characters in LaTeX and 86,200 characters in HTML. Although it is somehow unfair to compare hand-written code against generated ones, these statistics hint at the compactness of SKRIBE documents.

2.2 Skribe, a markup language

In this section, we present how to build a document using SKRIBE, and we show how programming skill is not needed to write a document. In fact, non-programmer authors can view SKRIBE as a simple document formatting system, such as HTML, LaTeX , or nroff (Ossana, 1982).

SKRIBE provides an extensive set of pre-defined markups, which roughly correspond to 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. Interested readers can refer to the complete manual of SKRIBE which is available at `http://www.inria.fr/mimosa/fp/Skribe`.

2.2.1 Skribe Markups

SKRIBE markups resemble XML elements. In SKRIBE, the attributes are represented by keywords (identifiers whose first character is a colon). Keywords have been introduced by DSSL (ISO/IEC, 1996), the tree manipulation language associated to SGML. The following XML expression:

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

is represented in SKRIBE as:

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

2.2.2 Document structure

Among the Sk-expressions that compose a SKRIBE document, the `document` Sk-expression serves a special purpose, because it is used to represent the complete document. All the subdivisions of a document must appear inside the `document` Sk-expression. So, the general structure of a SKRIBE document looks like:

```
(document :title <sk-expr> :author <sk-expr>
  (abstract <sk-expr>)
  ...
  (include "introduction-section.skb")
  ...
  (section :title <sk-expr>
     ...
     (subsection :title <sk-expr>)
     ...
     (subsection :title <sk-expr>)
     ...)
   ...
   (section :title <sk-expr>)
   ...
   (include "conclusion-section.skb"))
```

As this example shows, all of the sectioning components of a document are embedded in their containing component (i.e. sections inside documents, subsections inside sections, and so on). Large documents can be split into several files by means of `include` forms. The strict nesting of document components is particularly useful for document introspection, as we will show in section 3.2.1.

2.2.3 Skribe standard library

SKRIBE provides with the usual facilities for text processing. Some of these are presented here. Skribe provides the usual text ornaments (bold, italic, underline) and list environments (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.*

The SKRIBE standard library also offers the usual tools for inter and intra document references, footnotes, tables, figures, etc. For instance,

```
(itemize
   (item [A reference to the subsection
          ,(ref :subsubsection "Skribe standard library").])
   (item [A reference to a URL:
          ,(tt (ref :url (skribe-url))).])
   (item [A reference to the
          ,(ref :figure "skribe-eval").])
   (item [A footnote
          ,(footnote [This is a ,(Skribe) footnote.]).]))
```

produces the following output text:

- *A reference to the subsection 2.2.3.*
- *A reference to a URL: `http://www.inria.fr/mimosa/fp/Skribe`.*
- *A reference to the 1.*
- *A footnote*[1].

Skribe provides also an original construction, the `program` markup, to *pretty-print* codes or algorithms. In contrast with other systems, such as LaTeX , there is no need in Skribe to use external pre-processors, such as SLaTeX (Sitaram) and Lisp2TeX (Queinnec, 1993) for pretty-printing programs inside texts. The `program` 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 Sk-expression:

```
(program :language C :file "src/C-code.c")
```

produces the following output:

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

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

3 Skribe, a functional authoring language

In addition to being a markup language (see section 2.2), Skribe is *also* a functional programming language. We have chosen to base Skribe on Scheme partly because the Scheme syntax is close to traditional markup languages. Like XML, Scheme syntax is based on the representation of trees. Skribe's modifications of the Scheme grammar are limited and simple, and merely improve Scheme to make it more suitable for representing literal text.

Like Scheme, Skribe relies on dynamic type checking. Unlike XML, it is not designed for supporting static verification of documents. The flexibility of dynamic types eases the production of dynamic texts. In particular, as illustrated in section 2.1, an Sk-expression can be a list whose elements are of different types. The first

[1] This is a Skribe footnote.

element of such a list could be a character string, and the next one a number. This generality enables compact representation of texts.

Skribe documents are programs, including the examples of section 2. Executing each program produces a target document, such as HTML pages or PostScript texts. In the rest of this article, to avoid the confusion between the source document and the output documents, we refer to the former as *source programs* or *programs*, and we refer to the latter as *output documents* or *documents*.

The rest of this section is organized as follows. First, in Section 3.1 we detail the role of functions in the Skribe programming language. Then, in section 3.2 we present the Skribe evaluation model. We conclude in section 3.3 with examples.

3.1 Skribe functions

Functions are ubiquitous in Skribe. They are the only means for defining the syntactic Skribe elements. For instance, functions are used to define convenience macros like the ones supported by most typesetting systems. 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 Skribe by the means of functions that produce Sk-expressions. For instance, a macro defining the typesetting of the word "Skribe" is used all along this paper. It is defined as follows:

```
(define (Skribe)
   (sc "Skribe"))
```

It can be used in a Sk-expression such as:

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

which produces the following output:

This text has been produced by Skribe.

It may also happen that a Skribe function is used in a more complex way. Instead of replacing a source expression with another, it may introduce a complex data structure, called an Sk-ast (see section 3.2), representing a whole document or a fragment of a document. In this case, the function plays the same role as a *markup* in languages such as XML or HTML. The Skribe functions implementing the standard library (see section 2.2.3) fall in this category.

In addition to naturally implemeting markups and macros, Skribe functions have another strength: they expose a single formalism for specifying the *static*, (i.e. declared as-is) and *dymamic*, (i.e. computed) parts of the documents. This combination is the main originality and the main strength of Skribe in which it is particularly easy to merge texts and computations for authoring documents. Moreover, because Skribe is a full-fledged programming language, arbitrarily complex computations can be easily programmed. This expressiveness is illustrated by the next example, which is inspired by the Skribe Web page. The example generates a download table that contains the name of the tar files containing the Skribe source code. The table also contains the size of each file and an estimate of its download time depending on the network speed.

		Estimated download times		
	size	*56kb*	*512kb*	*10mb*
skribe1.2b.tar.gz	*254k*	*464s*	*51s*	*3s*

This text is automatically produced by the user-defined markup `directory-download`. The function implementing this markup selects the files with the suffix "`tar.gz`" in the `dir` directory. It then sorts the files into decreasing alphabetic order. Using the size of these sorted files, it computes the estimated download times, which are used for filling a table with the local function `file-download`.

```
(define (directory-download dir base)
   (define (eta size speed)
      (format "~as" (/ size (* speed 10))))
   (define (file-download file)
      (let ((sz (file-size file)))
         (tr (td (tt (basename file)))
            (td (/ sz 1024) "k")
            (td (eta sz 56)) (td (eta sz 512)) (td (eta sz 10240)))))
   (table :cellspacing 1 :border 1 :frame 'border :rules 'all
      (tr (td :colspan 2) (th :colspan 3 (it "Estimated download times")))
      (tr (td) (th "size") (th "56kb") (th "512kb") (th "10mb"))
      (map file-download
           (sort (filter (lambda (x)
                                (and (prefix? x base) (suffix? x "tar.gz")))
                             (directory->list dir))
                 string>?))))
```

This example illustrates the strength of a full-fledged language for authoring documents, because it relies on computations that are familiar to general purpose languages (scanning files on disk, fetching files size, etc.) that are generally unavailable in domain-specific languages.

3.2 The Skribe evaluator

In this section, we detail how source programs are evaluated to produce output documents. In particular, we present the different execution times that take place during this process.

Most documents contain *self-references*. Self-references are references used in a document to point to subparts of itself. These references are used for indexes, bibliographic citations, table of contents, references to figures, chapters, and so on. On the one hand, these constructions can be built-in in an authoring system or, on the other hand, they can be implemented using a general mechanism that exposes the whole document as a data structure and that supports references to itself. The latter is SKRIBE's solution. In the rest of this text, we refer to any computation involving self-references as *document introspection* or *introspection.*

Self-references can be forward or backward. Hence, prior to introspection, the data structure representing the whole document must be built. To support this phasing, a SKRIBE program evaluates in two steps: one for building the data structure and one for resolving the self-references. Further, we have found it useful to add a third pass, which is devoted to the generation of the output document. Figure 1 illustrates the three stages for evaluating a SKRIBE program.

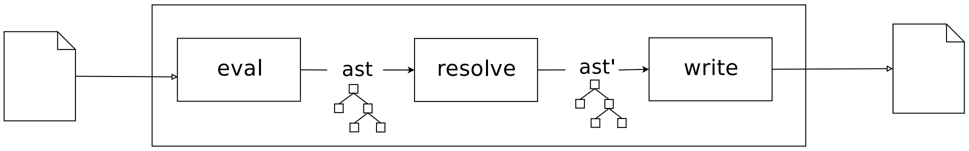

Fig. 1. Evaluation stages.

The first stage (named `eval` in Figure 1) parses the source program a builds an abstract syntax tree (henceforth AST). This stage roughly corresponds to a Scheme evaluator augmented with additional library functions. Because of its proximity with Scheme, this stage is not detailed here.
The second stage (RESOLVE) is presented Section 3.2.1. The third stage (WRITE) is presented section 3.2.2.

3.2.1 Resolve

The second stage (RESOLVE) of the SKRIBE evaluation process is in charge of resolving the *self-references*. These references can point to bibliographic entries, index entries, URL, sections, figures, etc. They are introduced in the source program by the `ref` function. Here is an excerpt of a SKRIBE program that contains two references, one to a section and one to a figure:

```
...
(section :title "Example of references"
   (p [Other examples can be found in Section
,(ref :section "Other examples") and in Figure
,(ref :figure "A big example").]))
...
```

The `ref` library function relies on the lower level function `unresolved`, which introduces nodes of type `unresolved` in the AST. Each of these nodes contains a function ρ, which accepts as argument the `unresolved` node itself. For the sake of the example, here is a simplified implementation of the `ref` function restricted to sections.

```
(define (ref-section sect)
   (let ((rho (lambda (node)
                 (let ((root (ast-root node)))
                    (section-number (find-section-from-title root sect))))))
      (unresolved rho)))
```

The argument `sect` is the title of the searched section. The functions `ast-root`, `section-number`, `find-section-from-title`, and `unresolved` are part of the SKRIBE library. As such they can be used in user programs (see example 3.3.1).
The purpose of the RESOLVE stage is to eliminate the `unresolved` nodes from the AST. To that end, it rewrites the AST by replacing each node v of type `unresolved` with the result of $\rho(v)$. Since $\rho(v)$ may introduce new `unresolved` nodes, RESOLVE requires a fixpoint iteration, and the user is ultimately responsible for ensuring that the fixpoint exists.

3.2.2 Write

After an AST is pruned of any unresolved nodes, it can be traversed to produce the output document, hence the Write stage. The Write stage traverses the AST, and at each node, it selects the appropriate *writer*. This selection depends on the type of the node and the format of the output document.

Skribe writers output the external representations of nodes. For instance, the writer in charge of outputting the HTML representation of `bold` nodes surrounds the HTML output of the node with <b> and </b>. A writer is a data structure containing three functions: `:before`, `:action`, and `:after`. These functions are applied before, during and after processing the node. Writers are introduced by the `define-writer` form. For instance, here is the definition of the Skribe writer handling `bold` nodes in HTML.

```
(define-writer 'bold 'html
   :before (lambda (n e) (display "<b>"))
   :action (lambda (n e) (emit (markup-body n) e))
   :after (lambda (n e) (display "</b>")))
```

The function `emit` actually implements the the Write stage. The function `markup-body` is part of the Skribe library, and it returns the content of a node. The functions `:before`, `:action`, and `:after` accept two arguments: `n`, which is the node to be written, and `e`, which is the writing *engine* or writing *context*.

This scheme of writer definitions is so frequent (print strings and recursively emit the node body) that the previous writer can also be defined with the more compact, equivalent definition:

```
(define-writer 'bold 'html :before "<b>" :after "</b>")
```

A set of writers constitutes a Skribe *back-end*. The standard Skribe distribution contains back-ends for popular formats, but users can also implement their own back-ends.

The second parameter of the three writer functions enables the writer to change *locally* the output representation of markups. This facility is used in the following example for implementing the markup `emph`. In this implementation, emphasized text is represented with an italic font, whereas nested emphasized text is represented with a bold font:

```
(define-writer 'emph 'html
   :before "<i>"
   :action (lambda (n e)
              (let ((ne (extend-engine e
                           (writer 'emph :before "<b>" :after "</b>"))))
                 (emit (markup-body n) ne)))
   :after "</i>")
```

The function `extend-engine` extends the given engine with a new writer. This new engine is used to recursively traverse the node that is emphasized. Note that this implementation does not conform to standard conventions, because it does not alternate italic and bold. This can be easily fixed by replacing the `:action` procedure of the `emph` writer with a recursive definition:

```
(define-writer 'emph 'html
   :before "<i>"
   :action (letrec ((rec (lambda (b)
                            (lambda (n e)
                               (let ((ne (extend-engine e
                                             (writer 'emph
                                             :before (if b "</i><b>" "</b><i>")
                                             :action (rec (not b))
                                             :after (if b "</b><i>" "</i><b>")))))
                                  (emit (markup-body n) ne))))))
               (rec #t))
   :after "</i>")
```

In this new writer definition, each nested invocation of `emph` extends the current context with a new definition for the `:before`, `:action`, and `:after` procedures.

3.3 Programming examples

Among the Skribe programs of Section 3.1, the first one illustrates how functions can be used to implement text macros. The second one shows that this single mechanism can also be used for introducing computed texts. The computations involved in these examples take place during the first stage of the evaluation of Skribe programs. In this section, we present example of computations that take place during the later stages, namely Resolve and Write stages.

3.3.1 User programmed self-references

As presented in section 3.2.1 user functions can be associated with `unresolved` nodes. In this example, we take benefit of this facility for implementing a custom table of contents. In the following example, we represent the table of contents as a tree. Applied to the present paper, it produces the tree of Figure 2.
This tree is built by the function `show-tree` which takes an Sk-ast as argument. It recursively walks the Sk-ast according to a depth-first traversal.

```
(define (show-container m s f)
  (list m
        "+-- "
        (f (container-title s))
        "\n"
        (map (lambda (x) (show-container (string-append m "|   ") x it))
             (containers-in s))))

(define (show-tree t)
  (map (lambda (x) (show-container "" x underline)) (containers-in t)))
```

The functions `show-tree` and `show-container` use Scheme functions and the following Skribe functions: `underline`, `it`, and `containers-in` (which returns the list of sub-containers of an Sk-ast).

The table of contents must be computed during the Resolve stage:

```
(define (my-document-toc)
   (unresolved (lambda (n) (show-tree (ast-root n)))))
```

```
+-- Introduction
+-- Skribe overview
|   +-- Sk-expressions
|   +-- Skribe, a markup language
|   |   +-- Skribe Markups
|   |   +-- Document Structure
|   |   +-- Skribe standard library
+-- Skribe, a functional authoring language
|   +-- Skribe functions
|   +-- The Skribe evaluator
|   |   +-- Resolve
|   |   +-- Write
|   +-- Programming examples
|   |   +-- User programmed self-references
|   |   +-- Blending output formats
+-- Related work
|   +-- SGML and XML
|   +-- LAML
|   +-- BRL
|   +-- Wash
+-- Conclusion and future work
```

Fig. 2. Custom table of contents.

The function `my-document-toc` introduces an `unresolved` node whose associated function calls the previously defined `show-tree` function.

3.3.2 Blending output formats

This example illustrates the computations that may take place during the Write stage. The Sk-ast is back-end independent and the specificities of output formats are only specified in the Write stage.

Let's assume the following itemize Sk-ast:

```
(define ast
   (itemize
      (item [foo])
      (item (bold [bar]))))
```

Used, as is, in the following document, it produces:

- *foo*
- ***bar***

The LaTeX source of this itemize is:

```
\begin{itemize}
 \item foo
 \item {\textbf{bar}}
\end{itemize}
```

The HTML source of this itemize is:

```
<ul class="itemize"><li>foo</li>
<li><strong>bar</strong></li>
</ul>
```

We show in the rest of this section how the previous LaTeX and HTML expressions are *automatically* generated from `ast`. For this, we introduce the new `proc` markup:

```
(define (proc body procedure)
   (make-markup 'proc procedure body))
```

A `proc` node contains a text (`body`) and a procedure (`procedure`). To invoke the `procedure` during the WRITE stage, we add the following writer definition:

```
(define-writer 'proc 'base
   :action (lambda (n e)
              (output ((markup-procedure n) (markup-body n) e) e)))
```

The `base` engine is the root of the engines. That is all other engines inherit from it. Let's assume a SKRIBE execution environment where two engines, `html-engine` and `latex-engine` have been loaded for extending the `base` engine with appropriate writers for the two formats. The procedure associated with the `proc` node is retrieved by the SKRIBE function `markup-procedure`. The functions `LATEX` and `HTML` which generate LaTeX and HTML sources of an Sk-ast are defined as:

```
(define (LATEX body)
   (proc body (lambda (n engine)
                 (with-output-to-string
                    (lambda () (output n latex-engine))))))

(define (HTML body)
   (proc body (lambda (n engine)
                 (with-output-to-string
                    (lambda () (output n html-engine))))))
```

Given the `LATEX` and `HTML` functions, the LaTeX and HTML expressions above are produced by expressions as simple as:

```
(LATEX ast)
(HTML ast)
```

This can be generalized. We can use the `HTML` and `LATEX` functions to generate the LaTeX source of the HTML expression:

```
$<$ul class="itemize"$>$$<$li$>$foo$<$/li$>$ %
$<$li$>$$<$strong$>$bar$<$/strong$>$$<$/li$>$ %
$<$/ul$>$
```

Which has been produced by:

```
(LATEX (HTML ast))
```

The LaTeX source of the LaTeX source of the HTML source of `ast` is:

```
\$$<$\$ul class="itemize"\$$>$\$\$$<$\$li\$$>$\$foo\$$<$\$/li\$$>$\$ \% %
\$$<$\$li\$$>$\$\$$<$\$strong\$$>$\$bar\$$<$\$/strong\$$>$\$\$$<$\$/li\$$>$\$ \%%
\$$<$\$/ul\$$>$\$
```

...

4 Related work

In this section, we compare SKRIBE and other markup languages based on functional programming languages. SKRIBE is meant for authoring polymorphic documents. That is, from one source document, SKRIBE can be used to produce several output files such as HTML files, PostScript, PDF documents, etc. Unlike many systems,

n=	fact
1	*1*
2	*2*
3	*6*
4	*24*
5	*120*
6	*720*
7	*5040*
8	*40320*
9	*362880*
10	*3628800*

Fig. 3. Factorial.

such as LaTeX or Texinfo, SKRIBE allows documents to rely on the specificities of a given output format. For instance, an HTML output may use a layout that suits Web browsing (e.g. navigation bar for fast browsing), a PostScript output may contain headers and footers displayed on each page. In consequence, documents can have different shapes according to the selected target format. Thus, complex web pages as well as high quality printed documents can be authorized in SKRIBE. To our knowledge SKRIBE is the only system supporting polymorphic documents. The others systems it compares to are devoted to the production of HTML documents only.

For each presented language we show how to program a simple HTML page containing the factorial table given Figure 3.

Assuming the standard definition of the factorial function, this table can be implemented using two additional 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 :frame 'box :rules 'all
          (tr (th "n=") (th "fact"))
          (map make-fact-row (upto 1 n))))
```

Thus, the figure 3 is generated with the expression:

```
(font :size -1 (make-fact-table 11))
```

4.1 SGML and XML

XML is a means to specify external representations for data structures (Harold & Means, 2001). It is a mere formalism for specifying grammars. XML can be used to represent texts, but this is not its only purpose. The most popular XML application 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 Skribe is that the former is not a programming language. In consequence, any processing (formating, rendering, extracting) over XML texts requires one or more external tools using different programming languages such as Java, Tcl, C, XSLT, or XQuery. Most functional programming languages have tools for handling XML texts. Haskell has HaXml (Wallace and Runciman, 1999), Caml has Px and Tony, and Scheme has SSax (Kiselyov, 2002).

In addition to parsers, Scheme has also SXML (Kiselyov, 2000) 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 (DSSSL) (ISO/IEC, 1996) defines several programming languages for handling SGML applications. The DSSSL suite plays approximatively the same role as XML XSLT, DOM and SSAX: it enables parsing and computing over SGML documents. The DSSSL languages are based on a simplified version of Scheme.

XEXPR (World Wide Web Consortium, 2000) is a scripting language that uses XML as its primary syntax. It has been defined to easily embed scripts inside XML documents, and it overcomes the usage of an external scripting language 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> <eq><n/>0</eq>
   1
   <multiply>
    <n/>
    <factorial> <substract><n/>1</substract> </factorial>
   </multiply>
 </if>
</define>
```

Obviously, writing by hand large scripts seems tedious in XEXPR. Furthermore, a careful reading of the report that defines this language seems to indicate that there is no way to manipulate the document itself inside an XEXPR expression. The language seems limited, therefore, to simple text generations inside an XML document.

XQuery (World Wide Web Consortium, 2003) is a functional programming language for handling XML documents. It enables a mixture of dynamic XML constructions and static XML fragments. XQuery documents mix traditional programming syntax with XML syntax. A purely static XQuery document is also a XML document. This approach minimizes the learning curve of XQuery. The factorial table (Figure 3) could be implemented in XQuery as:

```
define function fac ($n) {
   if ($n = 0)
      then 1
      else $n * fac($n - 1)
}

define function makefactrow ($n) {
   <tr>
     <td align="center">{$n}</td>
     <td align="right">{fac($n)}</td>
   </tr>
}

define function makefacttable ($n) {
  <table border="1">{
    <tr><th>n=</th><th>fact</th></tr>,
     for $x in 1 to $n
       return makefactrow($x)
  }</table>
}

<html>
  <head><title>Factorial</title></head>
  <body><font size="-1">{makefacttable(10)}</font></body>
</html>
```

The example shows the two different syntaxes used in XQuery documents. This approach is the opposite of the Skribe approach, which consists in a uniform syntax for the static and dynamic parts of the document.

4.2 LAML

LAML *Lisp Abstracted Markup Language* (Nørmark, 2002), 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. For the sake of the comparison, here is a possible implementation of the factorial table given figure 3 in LAML:

```
(define (fact n)
  (if (= 0 n)
      1
      (* n (fact (- n 1)))))

(define (make-fact-row n)
  (tr (td 'align "center" (b (as-string n)))
      (td 'align "right"  (i (as-string (fact n))))))

(define (make-fact-table n)
  (table 'border "1"
         (thead (tr (th (b "n=")) (th (b "fact"))))
         (tbody (map make-fact-row (upto 1 n)))))

(write-html '(pp)
 (html (head (title "Factorial in LAML"))
       (body (font 'size "-1" (make-fact-table 10)))))
```

As one may notice, LAML and Skribe are very close. They rely on Scheme's natural syntax, and they both consider a document as a program. However, there are two important differences between them. The first difference is about the syntax. In contrast with LAML that uses the regular Scheme syntax, Skribe uses an extended syntax. As presented Section 2.1, it introduces the `[...]` notation that, as we have shown, enables compact source texts. The second difference between Skribe and LAML is the lake a structure for representing the document in LAML. 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>"))
```

This direct mapping has three drawbacks:

- LAML sources cannot produce other formats than HTML.
- It is complex to implement efficiently a LAML interpreter. As reported in (Nørmark, 1999), the LAML evaluation process allocates many strings of characters. This allocation exercises intensively the memory manager (garbage collection and memory copies). These string manipulations are totally avoided by Skribe. One Skribe markup allocates one object that is a node of the Sk-ast. This node is used until the back-end has completed the file generation. It never happens that a node or the characters it contains are duplicated. In Skribe's model, building a large text from a smaller one is done by copying a pointer to the node, which is cheaper than building large strings from copies of smaller ones.
- Introspection over a LAML document is complex. In particular, introspection must 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.

4.3 BRL

The *Beautiful Report Language* BRL (Lewis, 2002) 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 mixes the text and the program that 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 strings. So,

```
]a string[
```

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

```
<html>
   <head><title>Pi</title></head>
   <body>The value of &pi; is [(* 4 (atan 1))].</body>
</html>
```

where we have the Scheme expression `(* 4 (atan 1))` surrounded by two strings. However, this syntax can be sometimes complex, as shown in the following excerpt from the reference manual.

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

The strength of the BRL syntax is more obvious on the BRL implementation of the factorial table given Figure 3:

```
[
(define (fact n)
   (if (= n 0)
       1
       (* n (fact (- n 1)))))

(define (make-fact-row n)
   (brl ]<tr><td align="center"><b>[n]</b></td>
         <td align="right"><it>[(fact n)]</it></td></tr>[))

(define (make-fact-table n)
   (brl ]<table border="1"><tr><th>n=</th><th>fact</th></tr>
         [(for-each make-fact-row (upto 1 n))]</table[))
]
<font size="-1">[(make-fact-table 10)]</font>
```

The idea of extending a standard Scheme reader for text processing has inspired the Skribe Sk-expression syntax (see section 2.1).

As XQuery, BRL mixes two syntaxes: an XML syntax for static parts and Scheme for dynamic parts. This approach is the opposite of the one chosen for Skribe. In our opinion, providing a single syntax and a single formalism for expressing static *and* dynamic parts of documents is more elegant. On the other hand, BRL's approach makes sense when existing HTML documents must be augmented with few computed parts.

4.4 Wash

Wash (Thiemann, 2000) 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. Wash and Skribe ensure the well-formedness of produced documents. In addition, thanks to the type system of the Haskell type checker, Wash can guarantee the validity of the generated HTML pages which Skribe does not. Using a Haskell interpreter it is possible with Wash to interactively create and manipulate web pages. Wash shares with Skribe the construction of a data structure representing the text to be rendered. Here is a possible implementation of the Factorial table given Figure 3 in Wash:

```
fact n = if n == 0
     then 1
     else n * fact (n - 1)

makeFactRow n =
  tr $ do td (A.align "center" ## b (text (show n)))
          td (A.align "right"  ## i (text (show (fact n)))

makeFactTable n =
  table $ do A.border "1"
             tr (th (text "n=") ## th (text "fact"))
             mapM makeFactRow [1..n]

factTable =
  font $ do A.size "-1"
            makeFactTable 10
```

Since this syntax is only accessible to programmers, Wash provides a pre-processor named wash2hs that enables a concise syntax. With this new syntax, the Factorial table could be written:

```
fact n =
  if n == 0
     then 1
     else n * fact (n - 1)

makeFactRow n =
  <tr>
    <td align="center"><b> <% text (show n) %> </b></td>
    <td align="right"><i> <% text (show (fact n)) %> </i></td>
  </tr>

makeFactTable n =
  <table border="1">
    <tr> <th>n=</th> <th>fact</th> </tr>
    <% mapM makeFactRow [1..n] %>
  </table>

factTable =
  <font size="-1"> <% makeFactTable 10 %> </font>
```

Wash uses the strong typing of Haskell to ensure that all produced documents are well-formed. SKRIBE follows a different path. by relying on dynamic type checking. Unlike Wash and XML, it is not designed for supporting static verification of documents. We have chosen to explore this path because it eases the production of dynamic texts. If the underlying language imposes a stronger typing system, the source program (that is, the user text) would be cluttered with cast operations that transform all values into strings.

5 Conclusion and future work

SKRIBE is a full-fledged programming language for authoring documents. It combines the expressiveness of a functional programming language and the declarative flavor of a markup language. SKRIBE relies on an original syntax that makes it look familiar to anyone who is used to markup languages, such as HTML.

SKRIBE has been deployed since 2002, and it is used daily for producing all kinds of documents, including large HTML and PostScript documents. For instance, this present paper, the whole web page `http://www.inria.fr/mimosa/fp/Bigloo`, and the documentation that it contains are implemented in Skribe.

Even if Skribe is already deployed and used, it still needs some improvements. In particular, the current version is not well suited for mathematical formulas. For bridging this gap, we plan to add support for mathematics formulas using a LaTeXlike syntax. The LaTeXback-end will consist in a straightforward mapping. The HTML back-end will be based on the MathML. Another area that deserves attention is the execution of Skribe programs in embedded applications. This point is currently studied. We are working on the design and implementation of a Skribe-aware web server. An early prototype seems promising.

References

Adobe System Inc. (1985) *PostScript Language Reference Manual.* Addison-Wesley.

Goldfarb, C. (1991) *The SGML Handbook*. Oxford University Press.

Harold, E. R. and Means, W. S. (2001) *XML in a Nutshell.* O'Reilly.

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

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

Kiselyov, O. (2000) *Implementing Metcast in Scheme*, Montréal, Canada.

Kiselyov, O. (2002) *A better XML parser through functional programming*. Portland, OR.

Knuth, D. (1986) *The TEXbook*. Addison-Wesley.

Lerdorf, R. (2000) *PHP Pocket Reference*. O'Reilly.

Lewis, B (2002) *BRL Reference Manualc.* Unpublished paper at URL http://brl.sourceforge.net/

McCarthy, J. (1960) Recursive functions of symbolic expressions and their computation by machine – I. *Comm. ACM*, **3**(1), 184–195.

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

Nørmark, K. (2002) Programmatic WWW authoring using Scheme and LAML. Honolulu, HI.

Ossana, J. (1982) *UNIX Programmer's manual: Supplementary Documents*, pp. 196–229. Holt, Rinehart and Winston.

Queinnec, C. (1993) *Literate programming from Scheme to TeX*. LIX RR 93.05, Laboratoire d'Informatique de l' Polytechnique, France.

Sitaram, D. SLaTeX. Unpublished paper at URL http://www.ccs.neu.edu/home/dorai/slatex/slatxdoc.html

Thiemann, P. (2000) *Modeling HTML in Haskell*, pp. 263–277. Of: Practical aspects of declarative languages, proceedings of the second international workshop, PADL '00. Lecture Notes in Computer Science No. 1753

Wallace, M. and Runciman, C. (1999) Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine), Paris, France.

World Wide Web Consortium (1998) Document Object Model (DOM) Level 1 Specification. W3C Recommendation.

World Wide Web Consortium (2000) XEXPR – A Scripting Language for XML. W3C Note.

World Wide Web Consortium (2003) XQuery – XML Query (XQuery) Requirements. W3C Working draft.