# Scribble: Closing the Book on Ad Hoc Documentation Tools

Matthew Flatt
University of Utah and PLT
mflatt@cs.utah.edu

Eli Barzilay
Northeastern University and PLT
eli@ccs.neu.edu

Robert Bruce Findler
Northwestern University and PLT
robby@eecs.northwestern.edu

## Abstract

Scribble is a system for writing library documentation, user guides, and tutorials. It builds on PLT Scheme's technology for language extension, and at its heart is a new approach to connecting prose references with library bindings. Besides the base system, we have built Scribble libraries for JavaDoc-style API documentation, literate programming, and conference papers. We have used Scribble to produce thousands of pages of documentation for PLT Scheme; the new documentation is more complete, more accessible, and better organized, thanks in large part to Scribble's flexibility and the ease with which we cross-reference information across levels. This paper reports on the use of Scribble and on its design as both an extension and an extensible part of PLT Scheme.

***Categories and Subject Descriptors*** I.7.2 [*Document and Text Processing*]: Document Preparation—Languages and systems

***General Terms*** Design, Documentation, Languages

## 1. Documentation as Code

Most existing documentation tools fall into one of three categories: LaTeX-like tools that know nothing about source code; JavaDoc-like tools that extract documentation from annotations in source code; and WEB-like literate-programming tools where source code is organized around a prose presentation.

*Scribble* is a new documentation infrastructure for PLT Scheme that can support and integrate all three kinds of tools. Like the LaTeX category, Scribble is suitable for producing stand-alone documents. Like the other two categories, Scribble creates a connection between documentation and the program that it describes—but without restricting the form of the documentation like JavaDoc-style tools, and with a well-defined connection to the language's scoping that is lacking in WEB-like tools. Specifically, Scribble leverages lexical scoping as supplied by the underlying programming language, instead of ad hoc textual manipulation, to connect documentation and code. This connection supports abstractions across the prose and code layers, and it enables a precise and consistent association (e.g., via hyperlinks) of references in code fragments to specifications elsewhere in the documentation.

For example, `@scheme[circle]` in a document source generates the output text `circle`. If the source form appears within a lexical context that imports the `slideshow` library, then the rendered `circle` is hyperlinked to the documentation for the `slideshow` library—and not to the documentation of, say, the `htdp/image` library, which exports a `circle` binding for a different GUI library. Moreover, the hyperlink is correct even if `@scheme[circle]` resides in a function that is used to generate documentation, and even if the lexical context of the call does not otherwise mention `slideshow`. Such lexically scoped fragments of documentation are built on the same technology as Scheme's lexically scoped macros, and they provide the same benefits for documentation abstraction and composition as for ordinary programs.

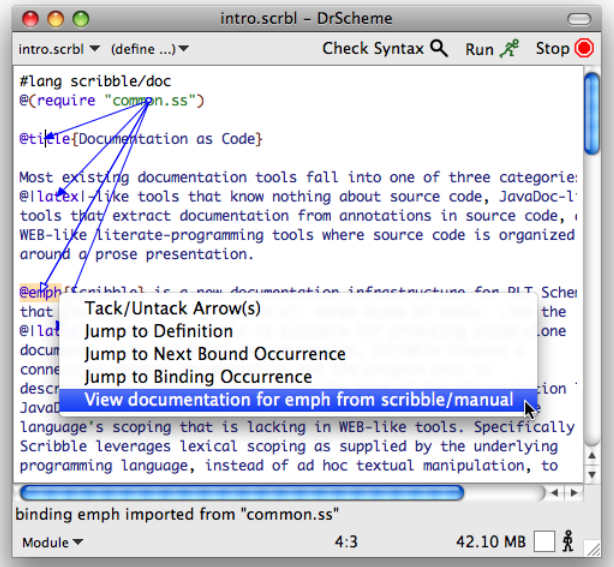



Figure 1: DrScheme with binding arrows and documentation links on Scribble code

To support documentation in the style of JavaDoc, a Scribble program can "include" a source library and extract its documentation. Bindings in the source are reflected naturally as cross-references in the documentation. Similarly, a source program can use module-level imports to introduce and compose literate-programming forms; in other words, the module system acts as the language that Ramsey (1994) has suggested to organize the composition of `noweb` extensions.

Scribble's capacity to span documentation-tool categories is a consequence of PLT Scheme's extensibility. Extensibility is an obstacle for JavaDoc-style tools, which parse a program's source text and would have to be extended in parallel to the language. Scribble, in contrast, plugs into the main language's extensibility machinery, so it both understands language extensions and is itself extensible. Similarly, Scheme macros accommodate a WEB-like



*ICFP'09,* August 31–September 2, 2009, Edinburgh, Scotland, UK.

organization of a library's implementation, and the same macros can simultaneously organize the associated documentation.

Indeed, Scribble documents are themselves Scheme programs, which means that PLT Scheme tools can work on Scribble sources. Figure 1 shows this paper's source opened in DrScheme. After clicking Check Syntax, then a right-click on a use of `emph` directly accesses the documentation of the `emph` function, even though the surface syntax of the document source does not look like Scheme. Such documentation links are based on the same lexical information and program-expansion process that the compiler uses, so the links point precisely to the right documentation.

We developed Scribble primarily for stand-alone documentation, but we have also developed a library for JavaDoc-style extraction of API documentation, and we have created a WEB-style tool for literate programming. In all forms, Scribble's connection between documentation and source plays a crucial role in cross-referencing, in writing examples within the documentation, and in searching the documentation from within the programming environment. These capabilities point the way toward even more sophisticated extensions, and they illustrate the advantages of treating documentation as code.

## 2. Scribbling Prose

The beginning of the PLT Scheme overview documentation demonstrates several common typesetting forms:

> **1 Welcome to PLT Scheme**
>
> Depending on how you look at it, **PLT Scheme** is
>
> - a *programming language* — a descendant of Scheme, which is a dialect of Lisp;
> - a *family* of programming languages — variants of Scheme, and more; or
> - a set of *tools* for using a family of programming languages.
>
> Where there is no room for confusion, we use simply "Scheme" to refer to any of these facets of PLT Scheme.

The Scribble syntax for generating this document fragment is reminiscent of LaTeX, using `@` (like texinfo) instead of `\`:

```
#lang scribble/doc
@(require scribble/manual)

@section{Welcome to PLT Scheme}

Depending on how you look at it, @bold{PLT Scheme}
is

@itemize[
  @item{a @emph{programming language} --- a
        descendant of Scheme, which is a dialect
        of Lisp;}

  @item{a @emph{family} of programming languages
        --- variants of Scheme, and more; or}

  @item{a set of @emph{tools} for using a family
        of programming languages.}
]

Where there is no room for confusion, we use
simply ``Scheme'' to refer to any of these facets
of PLT Scheme.
```

The initial `#lang scribble/doc` line declares that the module uses Scribble's documentation syntax, as opposed to using `#lang scheme` for S-expression syntax. At the same time, the `#lang` line also imports all of the usual PLT Scheme functions and syntax. The `@(require scribble/manual)` form imports additional functions and syntactic forms specific to typesetting a user manual. The remainder of the module represents the document content. The semantics of the document body is essentially that of Scheme, where most of the text is represented as Scheme strings.

Although we build Scribble on Scheme, a LaTeX-style syntax works better than nested S-expressions, because it more closely resembles the resulting textual layout. First, although all of the text belongs in a section, it is implicitly grouped by the section title, instead of explicitly grouped into something like a `section` function call. Second, the default parsing mode is "text" instead of "expression," so that commas, periods, quotes, paragraphs, and sections behave in the usual way for prose, while the `@` notation provides a uniform way to escape to a Scheme function call with text-mode arguments. Third, various automatic rules convert ASCII to more sophisticated typeset forms, such as the conversion of `---` to an em-dash and ``` ``...'' ``` to curly quotes.

Although LaTeX and Scribble use a similar syntax, the semantics are completely different. For example, `itemize` is a function that accepts document fragments created by the `item` function, instead of a text-parsing macro like LaTeX's `itemize` environment. The square brackets after `itemize` in the document source reflect that it accepts `item` values, whereas `item` and many other functions are followed by curly braces that indicate text arguments. The `@`-notation is simply another way of writing S-expressions, as we describe in detail in Section 4.

## 3. Scribbling Code

The PLT Scheme tutorial "Quick: An Introduction to PLT Scheme with Pictures" starts with a few paragraphs of prose and then shows the following example interaction:

```
> 5
5
> "art gallery"
"art gallery"
```

The `>` represents the Scheme prompt. The first `5` is a constant value in an expression, so it is colored green in the tutorial, while the second `5` is a result, so it is colored blue. In Scheme, the syntax for output does not always match the syntax for expressions, so the different colors are useful hints to readers—but only if they have the consistency of an automatic annotation.

The source code for the first example is simply

```
@interaction[5 "art gallery"]
```

where `interaction` is provided by a Scribble library to both evaluate examples and typeset the expressions and results with syntax coloring. Since example expressions are evaluated when the document is built, examples never contain bugs where evaluation does not match the predicted output.

The second example in the "Quick" tutorial shows a more interesting evaluation:

```
> (circle 10)
```

Here, again, the expression is colored in the normal way for PLT Scheme code. More importantly, the `circle` identifier is hyperlinked to the definition of the `circle` function in the `slideshow` library, so an interested reader can follow the link to learn more about `circle`. Meanwhile, the result is shown as a circle image, just as it would be shown when evaluating the expression in DrScheme.

The source code for the second example is equally simple:

```
@mr-interaction[(circle 10)]
```

The author of the tutorial had to implement the `mr-interaction` syntactic form, because `interaction` does not currently support picture results. The syntax coloring, hyperlinking, and evaluation of `(circle 10)`, however, is implemented by expanding to `interaction`. In particular, `circle` is correctly hyperlinked because the module containing the above source also includes

```
@(require (for-label slideshow))
```

which causes the `circle` binding to be imported from the `slideshow` module for the purposes of hyperlinking. Based on this import and a database mapping bindings to definition sites, Scribble can automatically insert the hyperlink.

A module that is imported only with `for-label` is *not* run when the documentation is built, because the time at which a document is built may not be a suitable time to actually run a module. As an extreme example, an author might want to document a module whose job is to erase all files on the disk. More practically, executing a GUI library might require a graphics terminal, while the documentation for the graphics library can be built using only a text terminal.

Pervasive and precise hyperlinking of identifiers greatly improves the quality of documentation, and it relieves a document author from much tedious cross-referencing work, much like automatic hyperlinking in wikis. The author need not specify where `circle` is documented, but instead merely import `for-label` a module that supplies `circle`, and the documentation system is responsible for correlating the use and the definition. Furthermore, since hyperlinks are used in examples everywhere, an author can expect readers to follow them, instead of explicitly writing "for more information on the `circle` procedure used above, see ..." These benefits are crucial when a system's documentation runs to thousands of pages. Indeed, PLT Scheme's documentation has 57,997 links between manuals, which is roughly 15 links per printed page (and which does not count the additional 105,344 intra-manual links).

Clicking the `circle` hyperlink leads to its documentation in a standard format:

---

`(circle` *diameter*`)` → `pict?`
  *diameter* : `real?`

Creates an unfilled ellipse.

---

In this definition, `real?` and `pict?` are contracts for the function argument and result. Naturally, they are in turn hyperlinked to their definitions, because suitable libraries are imported `for-label` in the documentation source.

The above documentation of `circle` is implemented using `defproc`:

```
@defproc[(circle [diameter real?]) pict?]{
 Creates an unfilled ellipse.
}
```

Alternatively, instead of writing the documentation for `circle` in a stand-alone document—where there is a possibility that the documented contract does not match the contract in the implementation—the documentation could be written with the implementation of `circle`. In that case, the documentation would look slightly different, since it would be part of the module's export declarations:

```
(provide/doc
 [circle ([diameter real?] . -> . pict?)
         @{Creates an unfilled ellipse.}])
```

With `provide/doc`, the single contract specification for `circle` is used in two ways: at run time to check arguments and results for `circle`, and when building the documentation to show the expected arguments and results of `circle`.

Although `defproc` and `provide/doc` are provided with Scribble, they are not built into the core typesetting engine. They are written in separate libraries, and Scribble users could have implemented these forms. We describe this approach to extending Scribble further in Section 8.

## 4. @s and []s and {}s, Oh My!

Users of a text-markup language experience first and foremost the language's concrete syntax. The same is true of any language, but in the case of text, authors with different backgrounds have arrived at a remarkably consistent view of the appropriate syntax: it should use blank lines to indicate paragraph breaks, double-quote characters should not be special, and so on. At the same time, a programmable mark-up language needs a natural escape to the programming layer and back.

From the perspective of a programming language, conventional notations for string literals are terrible for writing text. The quoting rules tend to be complex, and they usually omit an escape for arbitrarily nested expressions. "Here strings" and string interpolation can alleviate some of the quoting and escape problems, but they are insufficient for writing large amounts of text with frequent nested escapes to the programming language. More importantly, building text in terms of string escapes and operations like `string-append` distracts from the business of writing prose, which is about *text and markup* rather than *strings and function calls*.

Indeed, many documentation systems, like JavaDoc, avoid the limitations of string literals in the language by defining a completely new syntax that is embedded within comments. Of course, this approach sacrifices any connection between the text and the programming language.

For Scribble, our solution is the `@`-notation, which is a text-friendly alternative to traditional S-expression syntax. More precisely, the `@`-notation is another way to write down arbitrary S-expressions, but it is tuned for writing blocks of free-form text. The `@`-expression notation is a strict extension of PLT Scheme's S-expression syntax; the `@` character has no special meaning in Scheme strings, in comments, or in the middle of Scheme identifiers. Furthermore, since it builds on the existing S-expression parser, it inherits all of the existing source-location support (e.g., for error messages).

### 4.1 @-expressions as S-expressions

The grammar of an `@`-expression is roughly as follows (where `@`, `[`, `]`, `{`, and `}` are literal, and $\underline{x}_?$ means that $x$ is optional):

⟨*at-expr*⟩ ::= `@`$\underline{\langle op \rangle}_?$`[`$\underline{\langle S\text{-}expr \rangle^*}_?$`]`$\underline{\{\langle text \rangle\}}_?$
⟨*op*⟩ ::= ⟨*S-expr*⟩ that does not start with `[` or `{`
⟨*S-expr*⟩ ::= any PLT Scheme S-expression
⟨*text*⟩ ::= text with balanced `{`...`}` and with `@`-exprs

An @-expression maps to an S-expression as follows:

- An @⟨*op*⟩{...} sequence combines ⟨*op*⟩ with text-mode arguments. For example,

```
@emph{Yes!}
```

  is equivalent to the S-expression

```
(emph "Yes!")
```

  Also, since @ keeps its meaning inside text-mode arguments,

```
@section{Country @emph{and} Western}
```

  is equivalent to the S-expression

```
(section "Country " (emph "and") " Western")
```

- An @⟨*op*⟩[...] sequence combines ⟨*op*⟩ with S-expression arguments. For example,

```
@itemize[(item "a") (item "b")]
```

  is equivalent to the S-expression

```
(itemize (item "a") (item "b"))
```

- An @⟨*op*⟩[...]{...} sequence combines S-expression arguments and text-mode arguments. For example,

```
@title[#:style 'toc]{Contracts}
```

  is equivalent to the S-expression

```
(title #:style 'toc "Contracts")
```

  where `#:style` uses PLT Scheme notation for a keyword.

- An @⟨*op*⟩ sequence without an immediately following { or [ is equivalent to just ⟨*op*⟩ in Scheme mode. For example,

```
@username
```

  is equivalent to the S-expression

```
username
```

  so that

```
@emph{committed by @username}
```

  is equivalent to

```
(emph "committed by " username)
```

- An ⟨*op*⟩ can be omitted in any of the above forms. For example,

```
@{Country @emph{and} Western}
```

  is equivalent to the S-expression

```
("Country " (emph "and") " Western")
```

  which is useful in some quoted or macro contexts.

Another way to describe the @-expression syntax is simply @⟨*op*⟩[...]{...} where each of the three parts is optional. When ⟨*op*⟩ is included but both kinds of arguments are missing, then ⟨*op*⟩ can produce a value to use directly instead of a function to call. The ⟨*op*⟩ in an @-expression is not constrained to be an identifier; it can be any S-expression that does not start with { or [. For example, an argumentless `@(require scribble/manual)` is equivalent to the S-expression `(require scribble/manual)`.

The spectrum of @-expression forms enables a document author to use whichever variant is most convenient. For a given operation, however, one particular variant is typically used. In general, @⟨*op*⟩{...} or @⟨*op*⟩[...] is used to imply a typesetting operation, whereas @⟨*op*⟩ more directly implies an escape to Scheme. Hence, the form `@emph{Yes!}` is preferred to the equivalent `@(emph "Yes!")`, while `@(require scribble/manual)` is preferred to the equivalent `@require[scribble/manual]`.

A combination of S-expression and text-mode arguments is often useful to "customize" an operation that consumes text. The `@title[#:style 'toc]{Contracts}` example illustrates this combination, where the optional `'toc` style customizes the typeset result of the `title` function. In other cases, an operation that specifically leverages S-expression notation may also have a text component. For example,

```
@defproc[(circle [diameter real?]) pict?]{
 Creates an unfilled ellipse.
}
```

is equivalent to

```
(defproc (circle [diameter real?])
         pict?
         "Creates an unfilled ellipse.")
```

but as the description of the procedure becomes more involved, using text mode for the description becomes much more convenient.

An @ works both as an escape from text mode and as a form constructor in S-expression contexts. As a result, @-forms keep their meaning whether they are used in a Scheme expression or in a Scribble text part. This equivalence significantly reduces the need for explicit quoting and unquoting operations, and it helps avoid bugs due to incorrect quoting levels. For example, instead of `@itemize[(item "a") (item "b")]`, an itemization is normally written `@itemize[@item{a} @item{b}]`, since items for an itemization are better written in text mode than as conventional strings; in this case, `@item{a}` can be used directly without first switching back to text mode.

Overall, @-expressions are crucial to Scribble's flexibility in the same way that S-expressions are crucial to Scheme's flexibility—and, in the same way, the benefit is difficult to quantify. Furthermore, just as S-expressions can be used for more than writing Scheme programs, the @ notation can be used for purposes other than documentation, and the @-notation parser is available for use in PLT Scheme separate from the rest of the Scribble infrastructure. We use it as an alternative to HTML for building the `plt-scheme.org` web pages, more generally in a template system supported by the PLT Scheme web server, and also as a text preprocessor language similar in spirit to **m4** for generating plain-text files.

### 4.2 Documentation-Specific Decoding

The @ notation supports local text transformations and mark-up, but it does not directly address some other problems specific to organizing a document's source:

- Section content should be grouped implicitly via `section`, `subsection`, etc. declarations, instead of explicitly nesting section constructions.
- Paragraph breaks should be determined by empty lines in the source text, instead of explicitly constructing paragraph values.
- A handful of ASCII character sequences should be converted automatically to more sophisticated typesetting elements, such as converting ``` `` ``` and `''` to curly quotes or `---` to an em-dash.

These transformations are specific to typesetting, and they are not appropriate for other contexts where the `@` notation is useful. Therefore, the `@` parser in Scribble faithfully preserves the original text in Scheme strings, and a separate *decode* layer in Scribble provides additional transformations.

Functions like `bold` and `emph` apply `decode-content` to their arguments to perform ASCII transformations, and `item` calls `decode-flow` to transform ASCII sequences and form paragraphs between empty lines. In contrast, `tt` and `verbatim` do not call the decode layer, and they instead typeset text exactly as it is given.

For example, the source document

```
#lang scribble/doc
@(require scribble/manual)

@title{Tubers}

@section{Problem}

You say ``potato.''

I say ``potato.''

@section{Solution}

Call the whole thing off.
```

invokes the decode layer, producing a module that is roughly equivalent to the following (where a `part` is a generic section):

```
#lang scheme/base
(require scribble/struct)
(provide doc)

(define doc
 (make-part (list "Tubers")
  (list
   (make-part (list "Problem")
    (list
     (make-paragraph
      (list "You say \u201Cpotato.\u201D"))
     (make-paragraph
      (list "I say \u201Cpotato.\u201D"))))
   (make-part (list "Solution")
    (list
     (make-paragraph
      (list "Call the whole thing off."))))))))
```

## 5. Document Modules

Like all PLT Scheme programs, Scribble documents are organized into modules, each in its own file. A `#lang` line starts a module, and most PLT Scheme modules start with `#lang scheme` or `#lang scheme/base`. A Scribble document normally starts with `#lang scribble/doc` to use a prose-oriented notation with `@` syntax, but a Scribble document can be written in any notation and using any helper functions and syntax, as long as it exports a `doc` binding whose value is an instance of the Scribble `part` structure type. For example,

```
#lang scheme
(require scribble/decode)
(define doc (decode '("Hello, world!")))
(provide doc)
```

implements in Scheme notation a Scribble document that contains only the text "Hello, world!"

Larger documents are typically split across modules/files along section boundaries. Subsections are incorporated into a larger section using the `include-section` form, which expands to a `require` to import the sub-section module and an expression that produces the `doc` part exported by the module. Since document inclusion corresponds to module importing, all of the usual PLT Scheme tools for building and executing modules apply to Scribble documents.

When a large document source is split into multiple modules, most of the modules need the same basic typesetting functions as well as the same "standard" bindings for examples. In Scribble, both sets of bindings can be packaged together; since `for-label` declarations build on the module system's import mechanisms, they work with the module system's re-exporting mechanisms. For example, the documentation for a library that builds on the `scheme/base` library might use this `"common.ss"` library:

```
#lang scheme/base
(require scribble/manual
         (for-label lang/htdp-beginner))
(provide (all-from-out scribble/manual)
         (for-label
          (all-from-out lang/htdp-beginner)))
```

Then, each part of the document can be implemented as

```
#lang scribble/doc
@(require "common.ss")
....
```

instead of separately requiring `scribble/manual` and `(for-label lang/htdp-beginner)` in every file.

## 6. Modules and Bindings

As an embedded domain-specific language, Scribble follows a long tradition of using Lisp- and Scheme-style macros to implement little languages. In particular, Scribble relies heavily on the Scheme notion of *syntax objects* (Sperber 2007), which are fragments of code that have lexical-binding information attached. Besides using syntax objects in the usual way to implement macros, Scribble uses syntax objects to carry lexical information all the way through document rendering. For example, `@scheme[lambda]` expands to roughly `(typeset-id #'lambda)`, where `#'lambda` is similar to `'lambda` but produces a syntax object (with its lexical information intact) instead of a symbol.

At the same time, many details of Scribble's implementation rely on PLT Scheme extensions to Scheme macros. Continuing the above example, the `typeset-id` function applies PLT Scheme's `identifier-label-binding` function to the given syntax object to determine the source module of its binding. The `typeset-id` function can then construct a cross-reference key based on the identifier and the source module; the documentation for the binding pairs the same identifier and source module to define the target of the cross-reference.

A deeper dependence of Scribble on PLT Scheme relates to `#lang` parsing. The `#lang` notation organizes *reader extensions* of Scheme (i.e., changes to the way that raw text is converted to S-

expressions) to allow new forms of surface syntax. The identifier after **#lang** in the original source act as the "language" of a module.

To parse a **#lang** line, the identifier after **#lang** is used as the name of a library collection that contains a **"lang/reader.ss"** module. The collection's **"lang/reader.ss"** module must export a **read-syntax** function, which takes an input stream and produces a syntax object. The **"lang/reader.ss"** module for **scribble/doc** parses the given input stream in **@**-notation text mode, and then wraps the result in a **module** form. For example,

```
#lang scribble/doc
@(require scribble/manual)
It was a @bold{dark} and @italic{stormy} night.
```

in a file named **"hello.scrbl"** reads as

```
(module hello scribble/doclang
  doc ()
  "\n" (require scribble/manual) "\n"
  "It was a " (bold "dark") " and "
  (italic "stormy") "night." "\n")
```

where **doc** is inserted by the **scribble/doc** reader as the identifier to export from the module, and the **()** is a convenience explained below.

The **module** form is PLT Scheme's core module form, and it generalizes the standard **library** form (Sperber 2007) to give macros more control in transforming the body of a module. Within a **module**, the first identifier is the relative name of the module, and the second identifier indicates a module to supply initial bindings for the module body. In particular, the initial import of a module is responsible for supplying a **#%module-begin** macro that is implicitly applied to the entire content of the module.

In the case of **scribble/doclang**, the **#%module-begin** macro lifts out all import and definitions forms in the body, passes all remaining content to the **decode** function, and binds the result to an exported **doc** identifier. Thus, macro expansion converts the **hello** module to the following:

```
(module hello scheme/base
  (require scribble/doclang
           scribble/manual)
  (provide doc)
  (define doc
    (decode
     "\n"  "\n"
     "It was a " (bold "dark") " and "
     (italic "stormy") "night." "\n")))
```

A subtlety in the process of lifting out import and definition forms is that they might not appear directly, but instead appear in the process of macro expansion. For example, **include-section** expands to a **require** of the included document plus a reference to the document. The **#%module-begin** macro of **scribble/doclang** therefore relies on a PLT Scheme facility for forcing the expansion of sub-forms. Specifically, **#%module-begin** uses **local-expand** to expand each sub-form just far enough to determine whether it is an import form, definition form, or expression. If the sub-form is an import or definition, then **#%module-begin** suspends further work and lifts out the import or definition immediately; the import or definition can then supply bindings for further expansion of the module body. The need to suspend and continue lifting explains the **()** inserted in the body of a module by the **scribble/doc** reader; **#%module-begin** uses that position to track the sub-forms that have been expanded already to expressions.

Aside from (1) the ability to force the expansion of nested forms and (2) the ability of macros to expand into new imports, macro expansion of a module body is essentially the same as for libraries in the current Scheme standard (Sperber 2007). Where the standard allows choice in the separation of phases, we have chosen maximal separation in PLT Scheme, so that compilation and expansion as consistent as possible (Flatt 2002). That is, bindings and module instantiations needed during the compilation of a module are kept separate from the bindings and instantiations needed when executing a module for rendering.

Furthermore, to support the connection between documentation and library bindings, PLT Scheme introduces a new phase that is orthogonal to compile time or run time: the *label phase level*. As noted in Section 3, a **for-label** import introduces bindings for documentation without triggering the execution of the imported module. In PLT Scheme, the same identifier can have different bindings in different phases. For example, when documenting the Intermediate Scheme pedagogical language, a document author would like uses of **lambda** to link to the **lambda** specification for Intermediate Scheme, while procedures used to implement the document itself will more likely use the full PLT Scheme language, which is a different **lambda**. The two different uses of **lambda** are kept straight naturally and automatically by separate bindings in separate phases.

## 7. Core Scribble Datatypes

The **doc** binding that a Scribble module exports is a description of a document. Various tools, such as the **scribble** command-line program, can take this description of a document and render it to a specific format, such as LaTeX or HTML. In particular, Scribble defers detailed typesetting work to LaTeX or to HTML browsers, and Scribble's plug-in architecture accommodates new rendering back-ends.

Scribble's documentation abstraction reflects a least-common denominator among such document formats. For example, Scribble has a baked-in notion of itemization, since LaTeX, HTML, and other document formats provide specific support to typeset itemizations. For many other layout tasks, such as formatting Scheme code, Scribble documents fall back to a generic "table" abstraction. Similarly, Scribble itself resolves most forms of cross-references and document dependencies, since different formats provide different levels of automatic support; tables of contents and indexes are mostly built within Scribble, instead of the back-end.

A Scribble document is a program that generates an instance of a **part** structure type. A **part** can represent a section or a book, and it can have sub-parts that represent sub-sections or chapters. This paper, for example, is generated by a Scribble document whose resulting **part** represents the whole paper, and it contains sub-parts for individual sections. The **part** produced by a Scheme document for a reference manual is rendered as a book, where the immediate sub-parts are chapters.

Figure 2 summarizes the structure of a document under **part** in a UML-like diagram. When a field contains a list, the diagram shows a double arrow, and when a field contains a lists of lists, the diagram shows a triple arrow. The dashed arrows call attention to delayed fields, which are explained below.

Each **part** has a *flow* that is typeset before its sub-**part**s (if any), and that represents the main content of a section. A flow is a list of *blocks*, where each block is one of the following:

- a **paragraph**, which contains a list of elements that are typeset inline with automatic line breaks;
- a **table**, which contains a list of rows, where each row is a list of flows, one per cell in the table;

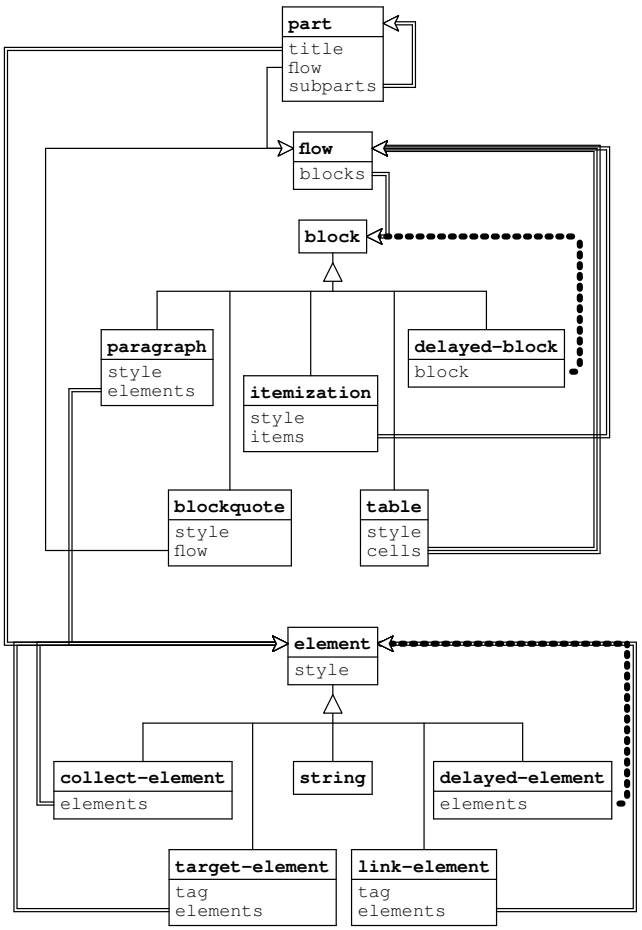



Figure 2: Scribble's core document representation

- an **itemization**, which contains a list of flows, one per item;
- a **blockquote**, which contains a single flow that is typically typeset with more indentation than its surrounding flow; or
- a **delayed-block**, which eventually expands to another block, using information gathered elsewhere in the document. Accordingly, the **block** field of a **delayed-block** is not just a **block**, but a function that computes a block when given that other information. For example, a **delayed-block** is used to implement a table of contents.

A Scribble document can construct other kinds of blocks that are implemented in terms of the above built-in kinds. For example, a **defproc** block that describes a procedure is implemented in terms of a **table**.

An *element* within a paragraph can be one of the following:

- a plain string;
- an instance of the **element** structure type, which wraps a list of elements with a typesetting style, such as **'bold**, whose detailed interpretation depends on the back-end format;
- a **target-element**, which associates a cross-reference tag with a list of elements, and where the typeset elements are the target for cross-references using the tag;
- a **link-element**, which associates a cross-reference tag to a list of elements, where the tag designates a cross-reference from the elements to elsewhere in the document (which is rendered in HTML as a hyperlink from the elements);
- a **delayed-element** eventually expands to a list of elements. Like a **delayed-block**, it typically generates the elements using information gathered from elsewhere in the document. A **delayed-element** often generates a **link-element** after a suitable target for cross-referencing is located.
- A **collect-element** is the complement of **delayed-element**: it includes an immediate list of elements, but also a procedure to record information that might be used elsewhere in the document. A **collect-element** often includes a **target-element**, in which case its procedure might register the target's cross-reference tag for discovery by **delayed-element** instances.
- A few other element types support more specialized tasks, such as communicating between phases and specifying tooltips.

A document as represented by a **part** instance is an immutable value. This value is transformed in several passes to eliminate **delayed-block** instances, **delayed-element** instances, and **collect-element** instances. The result is a simplified **part** instance and associated cross-reference information. Once the cross-reference information has been computed, it is saved for use when building other documents that have cross-references to this one. Finally, the **part** instance is consumed by a rendering back-end to produce the final document.

In the current implementation of Scribble, all documents are transformed in only two passes: a *collect* pass that collects information about the document (e.g., through **collect-element**s), and a *resolve* pass that turns delayed blocks and elements into normal elements. We could easily generalize to multiple passes, but so far, two passes have been sufficient within a single document. When multiple documents that refer to each other are built separately, these passes are iterated as explained in Section 9.

In some cases, the output of Scribble needs customization that is specific to a back-end. Users of Scribble provide the customization information by supplying a mapping from the contents of the **style** field in the various structures for the style's back-end rendering. For HTML output, a CSS fragment can extend or override the default Scribble style sheet. For LaTeX output, a **".tex"** file can extend or redefine the default Scribble LaTeX commands.

## 8. Scribble's Extensibility

Scribble's foundation on PLT Scheme empowers programmers to implement a number of features as libraries that ordinarily must be built into a documentation tool. More importantly, users can experiment with new and more interesting ways to write documentation without having to modify Scribble's implementation.

In this section, we describe several libraries that we have already built atop Scribble: for stand-alone API documentation, for automatically running examples when building documentation, for combining code with documentation in the style of JavaDoc, and for literate programing.

### 8.1 API Specification

Targets for code hyperlinks are defined by **defproc** (for functions), **defform** (for syntactic forms), **defstruct** (for structure types), **defclass** (for classes in the object system), and other such forms—one for each form of binding. When a library defines a new form of binding, an associated documentation library can define a new form for documenting the bindings.

As we demonstrated in Section 3, the **defproc** form documents a function given its name, information about its arguments,

and a contract expression for its result. Information for each argument includes a contract expression, the keyword (if any) for the argument, and the default value (if any). For example, a **louder** function that consumes and produces a string might be documented as follows:

```
@defproc[(louder [str string?]) string?]{
 Adds ``!'' to the end of @scheme[str].
}
```

The description of the function refers back to the formal argument **str** using **scheme**. In the typeset result, the reference to **str** is typeset in a slanted font both in the function prototype and description.

---

**(louder *str*) → string?**
**  *str* : string?**

Adds "!" to the end of ***str***.

As usual, lexical scope provides the connection between the formal-argument **str** and the reference. The **defproc** form expands to a combination of Scribble functions to construct a table representing the documentation and Scheme local-macro bindings to control the expansion and typesetting of the procedure description.

For the above **defproc**, the **for-label** binding of **louder** partly determines the library binding that is documented by this **defproc** form. A single binding, however, can be re-exported by many modules. On the reference side, the **scheme** and **scheme-block** forms follow re-export chains to discover the first exporting module for which a binding is documented; on the definition side, **defproc** needs a declaration of the module that is being documented. The module declaration is no extra burden on the document author, because the reader of the document needs some indication of which module is being documented.

The **defmodule** form both generates the user-readable explanation of the module being documented and declares that all definitions within the enclosing section (and sub-sections, unless overridden) correspond to exports from the declared module. Thus, if **louder** is exported by the **comics/string** library, it is documented as follows:

```
#lang scribble/doc
@(require scribble/manual
          (for-label scheme/base
                     comics/string))

@title{String Manipulations}

@defmodule[comics/string]

@defproc[(louder [str string?]) string?]{
 Adds ``!'' to the end of @scheme[str].
}
```

The **defproc** form is implemented by a **scribble/manual** layer of Scribble, which provides many functions and forms for typesetting PLT Scheme documentation. The **scribble/manual** layer is separate from the core Scribble engine, however, and other libraries can build up **defproc**-like abstractions on top of the core typesetting and cross-referencing capabilities described in Section 7.

### 8.2 Examples and Tests

In the documentation for a function or syntactic form, concrete examples help a reader understand how a function works, but only if the examples are reliable. Ensuring that examples are correct is a significant burden in a conventional approach to documentation, because the example expressions must be carefully checked against the implementation (often by manual cut and paste), and a small edit can easily introduce a bug.

The **examples** form of the **scribble/eval** library typesets an example along with its result using the style of a read-eval-print loop. For example,

```
@examples[(/ 1 2) (/ 1 2.0) (/ 1 +inf.0)]
```

produces the output

Examples:
```
> (/ 1 2)
1/2
> (/ 1 2.0)
0.5
> (/ 1 +inf.0)
0.0
```

Since building the documentation runs the examples every time, the typeset results are reliable. When an author makes a mistake, or when an implementation changes so that the documentation is out of sync, the example remains correct—though it may not reflect what the author intended. For example, if we misspell **+inf.0** in the example, then the output is still accurate, though unhelpful in describing the behavior of division:

Examples:
```
> (/ 1 +infinity.0)
reference to undefined identifier: +infinity.0
```

To guard against such mistakes, an example expression can be wrapped with **eval:check** to combine it with an expected result:

```
@examples[(eval:check (/ 1 +infinity.0) 0.0)]
```

Instead of typesetting an error message, this checked example will raise an exception when the document is built, because the expression does not produce the expected result **0.0**. In this way, documentation source can serve partly as a test suite.

Evaluation of example code mingles two phases that we have otherwise worked to keep separate: the time at which a library is executed, and the time at which its documentation is produced. For simple functional expressions, such as **(/ 1 2)**, the separation does not matter, and **examples** could simply duplicate its argument in both an expression position and a typeset position. More generally, however, examples involve temporary definitions and side-effects. To prevent examples from interfering with each other while building a large document, **examples** uses a sandboxed environment, for which PLT Scheme provides extensive support (Flatt et al. 1999; Flatt and PLT Scheme 2009, §13).

### 8.3 In-Code Documentation

For some libraries, the programmer may want to write documentation with the source instead of in a separate document. To support such documentation, we have created a Scheme/Scribble extension that is used to document some libraries in the PLT Scheme distribution.

Using this extension, the **comics/string** module could be implemented as follows:

```
#lang at-exp scheme/base
(require scheme/contract
         scribble/srcdoc)
(require/doc scheme/base
             scribble/manual)

(define (louder s)
  (string-append s "!"))

(provide/doc
 [louder
  ([str string?] . -> . string?)
  @{Adds ``!'' to the end of @scheme[str].}])
```

The **#lang at-exp scheme/base** line declares that the module uses **scheme/base** language extended with **@**-notation. The imported **scribble/srcdoc** library binds **require/doc** and **provide/doc**. The **require/doc** form imports bindings into a "documentation" phase, such as the **scheme** form that is used in the description of **louder**. The **provide/doc** form exports **louder**, annotates it with a contract for run-time checking, and records the contract and description for inclusion in documentation. The description is an expression in the documentation phase; it is dropped by normal compilation of the module, but combined with the **require/doc** imports and inferred **(require (for-label ...))** imports to generate the module's documentation.

The documentation part of this module is extracted using **include-extracted**, which is provided by the **scribble/extract** module in cooperation with **scribble/srcdoc**. The extracted documentation might provide the entire text of the document directly, or it might be incorporated into a larger document:

```
#lang scribble/doc
@(require scribble/manual
          scribble/extract
          (for-label comics/string))

@title{Strings}

@defmodule[comics/string]

The @schememodname[comics/string] library
provides functions for creating punchlines.

@include-extracted[comics/string]
```

An advantage of using **scribble/srcdoc** and **scribble/extract** is that the description of the function is with the implementation, and the function contract need not be duplicated in the source and documentation. Similarly, the fact that **string?** in the contract gets its binding from **scheme/base** is specified once in the code and inferred for the documentation. At the same time, a phase separation prevents document-generating expressions from polluting the library's run-time execution, and vice versa.

### 8.4 Literate Programming

The techniques used for in-source documentation extend to the creation of WEB-like literate programming tools. Figure 3 shows an example use of our literate-programming library; the left-hand side shows a screenshot of DrScheme editing the source code for a short, literate discussion of the Collatz conjecture, while the right-hand side shows the rendered output.

Literate programs written with our library look like ordinary Scribble documents, except that they start with **#lang scribble/lp** and use **chunk** to introduce a piece of the implementation. A use of **chunk** consists of a name followed by definitions and/or expressions:

```
@chunk[<name-of-chunk>
       ... definitions ...
       ... expressions ...]
```

The definitions and expressions in a chunk can refer to other chunks by their names.

Unlike a normal Scribble program, running a **scribble/lp** program ignores the prose exposition and instead evaluates the program in the chunks. In literate programming terminology, this process is called *tangling* the program. Thus, to a client module, a literate program behaves just like its illiterate variant. The compiled form of a literate program does not contain any of the documentation, nor does it depend on the runtime support for Scribble, just as if an extra-linguistic tangler had been used. Consequently, the literate implementation suffers no overhead due to the prose.

To recover the prose, the

```
@lp-include[filename]
```

form extracts a literate view of the program from ***filename***. In literate programming terminology, this process is called *weaving* the program. The right-hand side of Figure 3 shows the woven version of the code in the screenshot.

Both weaving and tangling with **scribble/lp** work at the level of syntactic extensions, and not in terms of manipulating source text. As a result, the language for writing prose is extensible, because Scribble libraries such as **scribble/manual** can be imported into the document. The language for implementing the program is also obviously extensible, because a chunk can include imports from other PLT Scheme libraries. Finally, even the bridge between the prose and the implementation is extensible, because the document author can create new syntactic forms that expand to a mixture of prose, implementation, and uses of **chunk**.

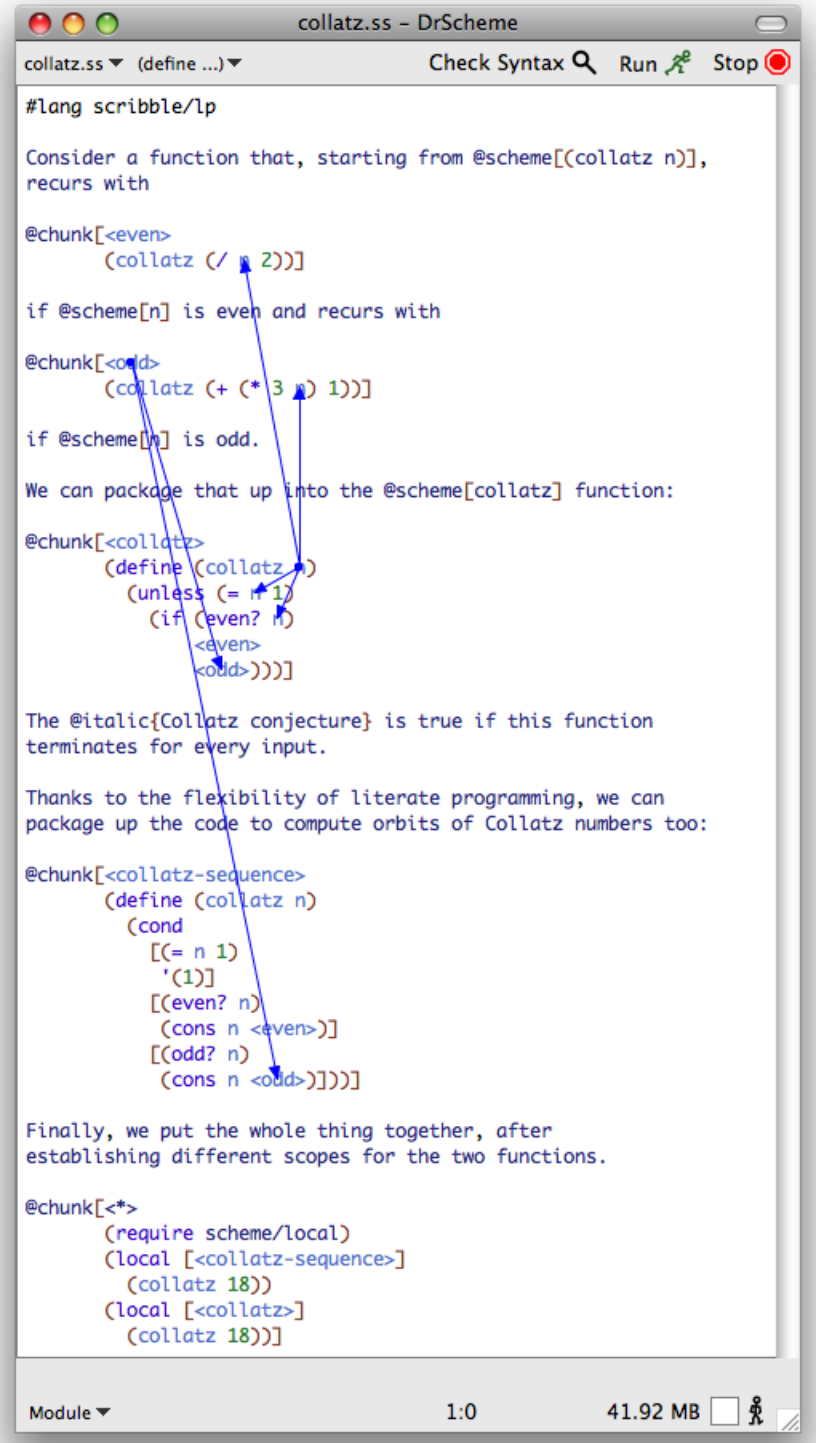
Tangling via syntactic extension also enables many tools for Scheme programs to automatically apply to literate Scheme programs. The arrows in Figure 3's screenshot demonstrate how DrScheme can draw arrows from chunk bindings to chunk references, and from the binding occurrence of an identifier to its bound occurrences, even across chunks. These latter arrows are particularly helpful with literate programs, where lexical scope is sometimes obscured by the way that textually disparate fragments of a program are eventually tangled into the same scope. DrScheme's interactive REPL, test-case coverage support, module browser, executable generation, and other tools also work on literate programs.

To gain some experience with non-trivial literate programming in Scribble, we have written a 34-page literate program that describes our implementation of the Chat Noir game, which is distributed with PLT Scheme. The source is included in the distribution as **"chat-noir-literate.ss"**, and the rendered output is in the help system and online at `http://docs.plt-scheme.org/games/chat-noir.html`.

## 9. Building and Installing Documentation

PLT Scheme documentation resides with the source code. The setup process that builds bytecode from Scheme source also renders HTML documentation from Scribble source. The HTML output is accompanied by cross-reference information that is used both for building more documentation when new libraries are installed and for online help in the programming environment.

Although many existing PLT Scheme tools help in building documents, the process of generating HTML is significantly different from compilation tasks. The main difference is that cyclic depen-




Consider a function that, starting from `(collatz n)`, recurs with

<*even*> ::=

```
(collatz (/ n 2))
```

if `n` is even and recurs with

<*odd*> ::=

```
(collatz (+ (* 3 n) 1))
```

if `n` is odd.

We can package that up into the `collatz` function:

<*collatz*> ::=

```
(define (collatz n)
  (unless (= n 1)
    (if (even? n)
        <even>
        <odd>)))
```

The *Collatz conjecture* is true if this function terminates for every input.

Thanks to the flexibility of literate programming, we can package up the code to compute orbits of Collatz numbers too:

<*collatz-sequence*> ::=

```
(define (collatz n)
  (cond
    [(= n 1)
     '(1)]
    [(even? n)
     (cons n <even>)]
    [(odd? n)
     (cons n <odd>)]))
```

Finally, we put the whole thing together, after establishing different scopes for the two functions.

<*> ::=

```
(require scheme/local)
(local [<collatz-sequence>]
  (collatz 18))
(local [<collatz>]
  (collatz 18))
```

Figure 3: Literate programming example

dencies are common in documentation, whereas library dependencies are strictly layered. For example, the core language reference contains many pointers into the overview and a few pointers to the GUI library and other extensions; all documents, meanwhile, refer back to the core reference. Resolving mutual dependencies directly would require loading all documents into memory at once, which is impractical for the scale of the PLT Scheme documentation. The setup processes therefore builds documents one at a time, reading and writing serialized cross-reference information until it arrives at a fixed point for all documents. A fixed point usually requires two iterations, so that all documents see the information collected from all other documents. A handful of documents require a third pass, because they contain section titles from other documents, where each section title is based on still other documents (e.g., by using an identifier whose typesetting depends on whether it is documented as a procedure or syntactic form).

Another challenge in building a unified set of documentation is that individual users might augment the main installation with user-specific libraries. The main installation includes a table of contents that is the default starting point for reading documentation, and this table is updated when a new package is placed into the main installation. When a user-specific library is installed, in contrast, its document is built so that hyperlink references go into the main installation's documentation, and a user-specific table of contents is created. When a user opens the documentation via DrScheme's Help menu, a user-specific table of contents is opened (if it exists).

Instead of explicitly installing a library, a user can implicitly install a package from the PLaneT repository (Matthews 2006) by using a library reference of the form `(planet ....)`. When a library is installed in this way, its documentation is installed as the library is compiled. PLaneT supports library versioning, and multiple versions of a package can be installed at a time. In that case, multiple versions of the documentation are installed; document references work with versions just as reliably as library references, since they use the same underlying module-import mechanisms to precisely identify the origin of a binding.

## 10. Experience

Scribble is part of the PLT Scheme distribution as of version 4.0, which was released in June 2008, and all PLT Scheme documentation is created with Scribble. Developing Scribble, porting old PLT Scheme documentation, and writing new documentation took about a year, but the `@` notation evolved earlier through years of experimentation.

The documentation at `http://docs.plt-scheme.org/` is built nightly by Scribble from a snapshot of the PLT Scheme source repository. The same documentation is available in PDF form at `http://pre.plt-scheme.org/docs/pdf/`. At the time of this writing, the 70 PDF files of current documentation total 3778 pages in a relatively sparse format, which we estimate would fit in around 1000 pages if compressed into a conference-style, two-column layout. This total includes documentation only for libraries that are bundled with PLT Scheme; additional libraries for download via PLaneT are also documented using Scribble.

PLT Scheme documentation was previously written in LaTeX and converted to HTML via **`tex2page`** (Sitaram 2007). Although **`tex2page`** was a dramatic improvement over our original use of **`latex2html`**, the build process relied on layers of fragile LaTeX macros, HTML hacks, and pre- and post-processing scripts, which made the documentation all but impossible to build except by its authors. Consequently, most library documentation used a plain-text format that was easier to write but inconsistent in style and difficult to index. The documentation index distinguished identifier names from general terms, but it did not attach a source module to each identifier name, so online help relied on textual search.

The Scribble-based documentation system is accessible to all PLT Scheme users, who write their own documentation using Scribble and often supply patches to the main document sources. Scribble produces output that is more consistent and easier to navigate than the old documentation, and the resulting documentation works better with online help. More importantly, the smooth path from API documentation to stand-alone documents has let us produce much more tutorial and overview documentation, helping users find their way through the volumes of available information.

## 11. Related Work

As noted in the introduction, most existing documentation tools fall into one of three categories: LaTeX-like tools, JavaDoc-like tools, and WEB-like tools.

The LaTeX category includes general word-processing tools like Microsoft Word, but LaTeX offers the crucial advantage of programmability, where macros enable automatic formatting of API details. Systems like Skribe (Gallesio and Serrano 2005) improve LaTeX by offering a sane programming language. Even in a programmable documentation language, however, a lack of connection to source code means that information is duplicated in documentation and source, and binding and evaluation rules inherent to the source language are not automatically reflected in documentation and in examples related to those bindings.

The JavaDoc category includes perldoc for Perl, RDoc for Ruby, Haddock (Marlow 2002) for Haskell, OCamlDoc (Leroy 2007), Doxygen (van Heesch 2007) for various languages (including Java, C++, C#, and Fortran), and many others. Such tools improve on the LaTeX category, in that they provide a closer connection to the programs that they document. In particular, they are specifically designed for library API documentation, where they shine in automatic extraction of API details from the source code. These tools are not suitable for other kinds of stand-alone documents, such as overview documents, tutorials, and papers (like this one), where prose and document structuring are more central than API details.

Literate programming tools such as WEB (Knuth 1984) and **`noweb`** (Ramsey 1994) are designed for documenting the implementation of a library as much as the API that a library exports. In a sense, these tools are an extreme version of the JavaDoc category, where the information communicated to a reader is drawn from both the prose and the executable source. In doing so, unfortunately, the tools typically revert to a textual slice-and-dice of the program and prose sources, instead of a programmable layer that spans the two halves.

Simonis and Weiss (2003) provide a more complete overview of existing systems and add **`Prog`***Doc*, which is similar to **`noweb`** in the way that it uses a pipeline of tools. Scribble builds on many ideas from these predecessors, but fits them into an extensible framework backed by an expressive programming language.

Skribe (categorized above in the LaTeX group) is by far the system most closely related to Scribble. Like Scribble, Skribe builds on Scheme to construct representations of documents using Scheme functions and macros, and it uses an extension of Scheme syntax to make it more suitable for working with literal text. (Skribe uses square brackets to quote strings, and within square brackets, a comma followed by an open parenthesis escapes back into Scheme.) Skribe's format-independent document structure and its use of passes to render a document influenced the design of Scribble. Skribe, however, lacks an integration with lexical binding and the module system that is the heart of Scribble. For example, a **`scheme`** form that typesets and links and identifier in a lexically sensitive way is not possible to implement in Skribe without building a PLT Scheme-style module and macro layer on top of Skribe.

Scribble builds on a long line of work in Lisp-style language extensibility, including traditional Lisp macros, lexically scoped

macros in Scheme (Dybvig et al. 1993), and readtable-based syntactic extension as in Common Lisp. Phase-sensitive binding through **`for-label`** is specific to PLT Scheme, as is the disciplined approach to reader extension embodied by **`#lang`**.

The SLaTeX (Sitaram 2007) system provides automatic formatting of Scheme code within a LaTeX document. To identify syntactic forms and constants, SLaTeX relies on **`defkeyword`** and **`defconstant`** declarations. In this mode, the author of a work in progress must constantly add another "standard" binding to SLaTeX's list; SLaTeX's built-in table of syntactic forms is small compared to the number of syntactic forms available in PLT Scheme. More generally, the problem is the usual one for "standards": there are many to choose from. Scribble solves this problem with **`for-label`** imports and by directly using the namespace-management functionality of PLT Scheme modules.

Many systems follow the Lisp tradition of *docstrings*, in which documentation is associated to run-time values and used for online help. Python supports docstrings, and its **`doctest`** module even extracts and executes examples as tests, analogous to Scribble's **`examples`** form. Scribble supports a docstring-like connection between run-time bindings and documentation, but using lexical-binding information instead of the value associated with a binding. For example, **`(help cons)`** in PLT Scheme's read-eval-print loop opens documentation for **`cons`** based on its binding as imported from **`scheme/base`**, and not based on the procedure obtained by evaluating **`cons`**.

Smalltalk programming environments (Kay 1993) have always encouraged programmers to use the source (with its comments) as documentation, and environments like Eclipse and Visual Studio now make code navigation similarly convenient for other languages. Such tools do not supplant the need for external documentation, however, such as guides and tutorials.

In terms of surface syntax, many documentation systems build on either S-expression notation (or its cousin XML) as a way to encode both document structure and program structure. Such representations are especially appropriate for an intermediate representation of documentation, as in DocBook (Walsh and Muellner 2008). S-expression encodings of documentation are especially common in Lisp projects, where data and code are mingled easily.

## 12. Conclusion

> A documentation language should be designed not by piling escape conventions on top of a comment syntax, but by removing the weaknesses and restrictions of the programming language that make a separate documentation language appear necessary. Scribble demonstrates that a small number of rules for forming documentation, with no restrictions on how they are composed, suffice to form a practical and efficient documentation language that is flexible enough to support the major documentation paradigms in use today.
>
> — Clinger's introduction to the R$^n$RS standards, adapted for Scribble

Our design for Scribble relies on a thread of language-extension work that starts in Lisp macros, runs through Scheme's introduction of lexically scoped syntax, and continues with PLT Scheme innovations on modules, phases, and an open syntax expander. Meanwhile, LaTeX and Skribe demonstrate the advantages of building a document system on top of a programming language, and tools like JavaDoc demonstrate the power of leveraging the information available in a program's source to automate and link documentation about the program.

Scribble combines all of these threads for the first time, producing a tool (or library, or language, depending on how you look at it) that spans and integrates document categories. We are aware of no programming system besides PLT Scheme that is distributed with tutorials, programmer guides, and detailed API documentation, all extensively and precisely cross-referenced. We also know of no other system that makes it so easy for third parties to add new documentation of all kinds with the same level of integration, to say nothing of being able to extend the documentation system itself.

## Trying Scribble

To install an HTML version of this paper where Scheme and Scribble identifiers are hyperlinked to their documentation, first install PLT Scheme 4.1.5 or later from `http://plt-scheme.org/`. Then, start DrScheme, enter the program

```
#lang scheme
(require (planet mflatt/scribble-paper))
```

and click Run. Running the program installs the paper and directs your default browser to the starting page. To view the document source, click Check Syntax and then right-click on **`mflatt/scribble-paper`** to open its source.

**Acknowledgements:** We would like to thank Matthias Felleisen and the anonymous reviewers for helpful feedback on this paper. This work is supported in part by the NSF.

## Bibliography

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), pp. 295–326, 1993.

Matthew Flatt. Compilable and Composable Macros, You Want it *When?* In *Proc. ACM Intl. Conf. Functional Programming*, pp. 72–83, 2002.

Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming Languages as Operating Systems (*or* Revenge of the Son of the Lisp Machine). In *Proc. ACM Intl. Conf. Functional Programming*, pp. 138–147, 1999.

Matthew Flatt, and PLT Scheme. Reference: PLT Scheme. PLT Scheme Inc., PLT-TR2009-reference-v4.2, 2009.

Erick Gallesio, and Manuel Serrano. Skribe: a Functional Authoring Language. *J. Functional Programming* 15(5), pp. 751–770, 2005.

Alan C. Kay. The early history of Smalltalk. *ACM SIGPLAN Notices* 28(3), 1993.

Donald E. Knuth. Literate Programming. *Computer Journal* 27(2), pp. 97–111, 1984.

Xavier Leroy. The Objective Caml System, release 3.10. 2007.

Simon Marlow. Haddock, a Haskell Documentation Tool. In *Proc. ACM Wksp. Haskell*, pp. 78–89, 2002.

Jacob Matthews. Component Deployment with PLaneT: You Want it *Where?* In *Proc. Wksp. Scheme and Functional Programming*, 2006.

Norman Ramsey. Literate Programming Simplified. *IEEE Software* 11(5), pp. 97–105, 1994.

Volker Simonis, and Roland Weiss. ProgDOC — A New Program Documentation System. In *Proc. Perspectives of System Informatics*, Lecture Notes in Computer Science volume 2890, pp. 438–449, 2003.

Dorai Sitaram. TeX2page. 2007. `http://www.ccs.neu.edu/home/dorai/tex2page/tex2page-doc.html`

Dorai Sitaram. How to Use SLaTeX. 2007. `http://www.ccs.neu.edu/home/dorai/slatex/`

Michael Sperber (Ed.). The Revised $^6$ Report on the Algorithmic Language Scheme. 2007.

Norman Walsh, and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly & Associates, Inc., 2008.

Dimitri van Heesch. Doxygen Source Code Documentation Generator Tool. 2007. `http://www.stack.nl/~dimitri/doxygen/`