The Scribble Reader

An Alternative to S-expressions for Textual Content

Eli Barzilay
Northeastern University
eli@barzilay.org

Abstract

For decades, S-expressions have been one of the fundamental advantages of languages in the Lisp family — a major factor in shaping these languages as an ideal platform for symbolic computations, from macros and meta-programming to symbolic data exchange and much more. As convenient as this minimalist syntax has proven to be, it is unfitting for dealing with textual content. In this paper we describe the reader used by Scribble — the PLT Scheme documentation system. The reader implements a syntax that is easy to use, uniform, and it meshes well with the Scheme philosophy. The syntax makes "here-strings" and string interpolation easy, yet it is more powerful than a combination of the two.

1. Introduction

In Scheme, textual content comes in the form of plain old double-quoted strings, with simple backslash escapes. Dealing with rich textual content is possible, but highly inconvenient to the point of making such kind of programing nearly impractical. Writing a complete textual document in plain Scheme syntax is possibly more difficult than implementing a symbolic interpreter in Fortran. In contrast, practically all modern languages come with a variety of tools that promote textual content: multiple kinds of string quotations, "here-strings" (also called "here-documents"), and string interpolation syntax. In this regard, Scheme is severely lagging behind the times.

The convenience that comes with these tools is more than a mere technicality; it is important for enabling textual computing in much the same way that S-expressions, quotes, and quasiquotes in Scheme enable symbolic computing. Neither of these facilities is required, yet without them, textual manipulation and symbolic programming becomes a hair-pulling experience. Being limited to Scheme strings means that all text must be modified by escaping double quotes and backslashes (a major hassle for text that has Scheme code in it), and to mix text and code, we need to split the text into separate strings, then recombine them with `string-append` — making this quite similar in nature to an implementation of meta-programming when all you have is `make-symbol` and `make-pair`.

In PLT Scheme, we have designed a new concrete syntax to address the problem of textual content. This is implemented as a reader macro that is used as part of the Scribble documentation system[5]. The new syntax is similar in spirit to S-expressions, and indeed it is both elegant and useful in a similar way. At the conceptual level, the syntax builds on a similar uniformity to that of S-expressions — the result is even more convenient and general than popular textual facilities, specifically, both here-strings and string interpolation are achieved in a nearly trivial way. The new syntax has proven itself in a massive migration of thousands of documentation pages from a messy LaTeXbased system to Scribble, extending it with much more text, and the result is higher quality renderings, with much improved utility to end users.

An additional similarity with S-expressions is in the syntax's versatility and independent utility: the Scribble syntax is useful in many textual contexts, going well beyond "a documentation language"; parallel to the utility of S-expressions for many tasks that revolve around symbolic computations and more. It is essentially an alternative S-expression "skin" that can ease textual applications at the concrete syntax level while keeping the usual Scheme flexibility. As a matter of fact, this approach can be used to extend *any* language, it is the conceptual approach and the use of S-expressions that makes it particularly fitting for Scheme.

In short, the Scribble syntax, not only brings PLT Scheme up to speed with respect to textual programming: it provides it with the same edge that Scheme always had with respect to symbolic programming.

The Scribble syntax was not made up in a vacuum; various Scheme implementations have come up with a few solutions to varying degrees of completeness. Indeed, PLT Scheme itself has SCSH-style "here-strings"[10], two different preprocessor tools, a PLaneT library for string interpolation, and all of these come in addition to a decade-long sequence of various experiments with syntaxes in trying to find a good solution.

Many Scheme implementations provide a readtable-like facility for extending their concrete syntax, and can therefore implement the Scribble extensions conveniently and achieve the same benefits. Moreover, if adopted by more implementations, these benefits can carry to the Scheme community as a whole.

2. Scribble Syntax by Example

In this section we present the Scribble syntax through a not-so-short sequence of examples. As usual with readtable-based parsers, the syntax is hooked onto a "reader macro" character, which makes it an extension of the existing S-expression reader — and allowing a natural mix of Scheme code in both S-expression and Scribble syntax, as well as making the implementation relatively simple by localizing the required extension rather than requiring a completely new parser.

Scheme and Functional Programming August 22nd 2009, Cambridge, MA

2.1 Basic @-Expressions

Scribble forms are often called @-expressions or @-forms because they begin with a @ character. Embedded in Scheme code, a simple @-expression can look as follows:

```
(define (greetings) @list{Hello, world!})
```

Following the @, the reader proceeds to read `list` as a Scheme identifier, and any text that appears within the following curly braces gets parsed as a sequence of Scheme strings, and finally the identifier and the strings are wrapped in a list. We can use the fact that this is an extension of the Scheme reader, and inspect how it parses an @-form using `quote`[1]:

```
> '(define (greetings) @list{Hello, world!})
(define (greetings) (list "Hello, world!"))
```

The @ character was chosen by counting the frequency of the first character used for symbols in the complete PLT code-base, and choosing the least frequent one. Another (unexpected) advantage of @ is that it is illegal to start an identifier with it according to the R6RS[11] standard. The Scheme reader is extended by introducing @ as a "non-breaking" reader macro character, so less code may be broken when switching to it (i.e., a @ inside an identifier has no special meaning). In the rare case where @ *is* needed at the beginning of a symbol, PLT Scheme's escape conventions for symbols (also used by several other implementations) can be used — for example, `\@foo` and `|@foo|` are plain Scheme identifiers. This is a direct benefit of using a readtable extension: these escapes mean that the Scribble reader will never see these symbols.

Reading @-forms as plain S-expressions is crucial to the utility of the Scribble syntax. It allows us to leave the meaning of these expressions to the usual Scheme semantics — using any kind of bindings: existing or imported, procedures or macros.

```
> @display{Hello, world!}
Hello, world!
> (define (greet str) (printf "Hello, ~a!\n" str))
> @greet{world}
Hello, world!
```

The textual content of an @-expression is nearly free-form text. It can have newlines, double and single quotes, backslashes, etc. The S-expression that it parses to will have a number of strings: usually one string per line and one for each newline. The indentation of the whole form is ignored, but indentation *inside* the form is preserved and is parsed as a separate string of spaces.

```
> '@string-append{Backslashes escape quotes:
                  "x\"y"}
(string-append "Backslashes escape quotes:" "\n" "\"x\\\"y\"")
> '@string-append{1. First
                     - sub
                  2. Second}
(string-append "1. First" "\n" "   " "- sub" "\n" "2. Second")
```

This makes @-forms extremely convenient for common tasks where text contains code snippets. In almost all case, the author only needs to deal with forms (function applications, macros, or quoted lists) that have multiple strings as a "textual body".

```
> (define (show . strings)
    (for-each display strings))
> @show{(define (show . strings)
          (for-each display strings))}
(define (show . strings)
  (for-each display strings))
```

The reason for separating the text into separate strings for newlines and indentation is that in some cases it is useful to process the text based on them. We also use syntax properties (a PLT Scheme feature) that contain additional information, making it possible to know the precise original syntax. In some rather rare cases where this is needed, this information can be used by macros[2].

Several subtle yet important decisions are made here. Ignoring the indentation of an @-forms makes it possible to preserve the indentation structure of Scribble-extended Scheme code, a priceless feature for Schemers. In fact, the "overly verbatim" nature of Scheme strings and popular here-string syntaxes greatly reduces their popularity, and indeed, they are mostly used only in toplevel positions, and still, Schemers often prefer `string-append` to preserve the textual layout of their code. The importance of preserving this textual layout leads to another feature of the Scribble reader: a newline that begins the textual body or one that ends it are ignored *if* the body contains text. If the body contains no text, then it is assumed that the newlines are all intentional.

```
> @list{
    blah blah blah
  }
("blah blah blah")
> @list{
  }
("\n")
> @list{

    blah

  }
("\n" "blah" "\n")
```

This decision is a typical case that demonstrates an important principle of the Scribble syntax design: it should be convenient and natural to use for *humans authors* facing tasks that involve writing text in code. This stands in contrast to plain Scheme strings, where uniformity and terseness is strongly favored over convenience. (It is also interesting to compare this with textual quotations in modern languages.) Obviously, there is an important tradeoff here: uniformity and terseness are important for any feature of a programming language, even more so for its concrete syntax. As a result, designing the Scribble syntax has been a tedious experience of finding the golden line between uniformity and convenience[3].

The issue of indentation demonstrate this tension in two additional decisions. First, the indentation of a whole @-form is ignored, but what if after the opening brace there are some spaces and then text? In this case, the assumption is that the spaces are intentional, and they are not ignored. Second, when @-forms are used as markup language, a textual body might be *outdented* relative to the first line, to avoid disturbing the flow of text. A simple solution to both of these issues is to simply ignore any text that follows the opening brace when determining the indentation of the whole @-expression.

```
> @list{  One
          Two}
("  One" "\n" "  " "Two")
> @list{This sentence is split
  across two lines.}
("This sentence is split" "\n" "across two lines.")
```

In a similar way to its treatment of indentation, Scribble ignores spaces at the end of lines — except for spaces right before the closing brace, if there is text on that line.

```
> @list{  One
          Two  }
("  One" "\n" "  " "Two  ")
```

[1] To try these examples in a DrScheme or MzScheme REPL, enable the Scribble reader with `(require scribble/reader)` and `(use-at-readtable)`.

[2] A macro is required since the source and its properties do not exist at runtime.

[3] There are still a number of corner cases where it is not clear whether the Scribble syntax has followed the right choice.

2.2 Escapes and "Here Strings"

Clearly, the textual content is not completely free form: it is terminated by a closing brace. However, balanced braces are still allowed. This greatly reduces the need for escaping — most uses of a textual container that require braces will have balanced braces, so we favor not requiring escapes for this case over the alternative of always requiring them, or the asymmetric alternative of requiring them *only* for closing braces.

```
> @list{int add1(int i) {
          return i+1;
        }}
("int add1(int i) {" "\n" "  " "return i+1;" "\n" "}")
```

Typical cases that require an unbalanced single brace are programs that construct text programmatically, and for such uses we can still use conventional Scheme strings. In other words, we choose yet again the option that has the greater advantage for most texts, over the more uniform but less convenient alternatives.

Still, we wish for the syntax to be complete and to accommodate an unbalanced brace in *some* way when it is needed. A good rule-of-thumb to see how such a syntax scale is to observe how it handle reflective texts, for example, writing texts *about* the system itself[4].

A common sequence of events at this point is (a) decide to use an escape character, (b) go with the familiar backslash, (c) require backslashes to be escaped too, (d) end up with yet another level of backslash-escapes. An alternative that we considered is backslashes that escape only braces, and otherwise are part of the text: a sequence of n backslashes followed by a brace would stand for a text that consists of $n-1$ backslashes and the brace, for example "\{" quotes "{", and "\\\{" quotes "\\{". This non-uniform rule comes at a cost: it is confusing in its non-uniform behavior, and it is impossible to have a backslash appear before an *unescaped* brace (as the last character of a textual body). Regardless of the choice, an escape character is a poor choice for reflective texts (failing our rule-of-thumb), and was therefore rejected.

Instead of an escape mechanism, we have turned to a more convenient approach that fits cases that call for extra "freedom" in the text part of an `@`-expression — a set of alternative delimiters can be used, making braces lose their special role. Using a different shape of parentheses, as done in some languages that have both single- and double-quoted strings, only shifts the problem elsewhere, forcing the author to be aware of the current delimiters. Instead, we have settled on alternative delimiters that are longer than single braces: we still use braces (so delimiters still look similar, making them easy to read and to remember), but the open/close delimiters we use `|{` and `}|`.

```
> @list|{ }-{ }|
(" }-{ ")
```

The vertical bars are not an arbitrary choice: they have a similar role in delimiting symbols in the PLT Scheme reader, so playing a similar role here makes this choice more memorable.

Going back to our rule-of-thumb, it becomes apparent that just a single alternative is insufficient, for example when this alternative is itself documented. The delimiter syntax is therefore further generalized to arbitrary user-specified delimiters in a way that is analogous to here-strings: the opening delimiter can have any sequence of punctuation characters (excluding curly braces) between the vertical bar and the brace, making the expected closing delimiter to hold the same sequence (in reversed order, and with reversed parentheses)[5]. Using such delimiters makes it possible to have the text arbitrarily free-form.

```
> @show|--<<{Use @foo|{...}| to type free braces}>>--|
Use @foo|{...}| to type free braces
```

For extreme cases, the Scribble reader is generalized on the macro character to use. For example, the textual domain might use `@` excessively (e.g., documenting Scribble), or you might want to intentionally make a syntax that looks like an existing language (e.g., create a language that is close in its look to LaTeX)[6]. Yet even with our extensive experience of (re)writing the PLT Scheme documentation in Scribble, there was no need for this feature. It is therefore only available through procedures in the reader API that can construct such custom readers.

Finally, the Scribble syntax has yet another way to locally escape text using the usual Scheme string syntax. This is done by following a `@` with a Scheme string. This can be handy in some difficult cases, where it is used for very short strings, or when authors prefer it over using the alternative delimiter syntax.

```
> @show{An open brace: #\@"{".}
An open brace: #\{.
```

As we shall see next, this is actually a limited (and slightly modified) example of nested Scribble forms.

2.3 Nested `@`-Forms

So far, the Scribble syntax that we have covered serves as only a convenient alternative to quoting and to "here-strings". Specifically, we did not address the problem of combining text and code, commonly addressed via "string interpolation". Schemers quickly recognize string interpolation as similar in nature to quasi-quoting. In fact, such a facility is sometimes named "quasi-strings", and implemented as a string-like extension using similar characters.

Our syntax implements a conceptually different solution — a general approach that is powerful enough to support string interpolation as a trivial byproduct. At its core, the basic property of the Scribble syntax is that `@`-forms have the *same meaning* whether they appear in Scheme code or nested in other `@`-forms. The direct implication of this principle is simple: if the textual body of an `@`-expression has a nested expression, then the nested one is read recursively, and the resulting expression contains the nested form among the strings of its textual body, which means that the textual body of an `@`-form is no longer just a sequence of strings.

```
> '@foo{abc @bar{ijk}
        xyz}
(foo "abc " (bar "ijk") "\n" "xyz")
```

This approach is the primary key to the convenience and flexibility of the Scribble syntax. The first thing to note is a convenience point: `@`-expressions preserve their meaning when they move into and out of other `@`-expressions. But this is important at a much more fundamental level: it means that `@`-expressions follow the same rules as S-expressions — an occurrence of `@foo{...}` denotes an application of the procedure bound to `foo` (or a `foo` macro form), regardless of where it appears in the code (barring quoted contexts). This is what makes the Scribble syntax be an *alternative* to S-expressions, one that is well suited for textual content — not just a mere combination of string quotation and unquotation.

```
> (define mytext @list{A @vector{B} C})
> mytext
("A " #("B") " C")
```

If this is compared to a conventional string interpolation (using a fictitious syntax),

[4] For example, this paper and the Scribble documentation pages are both written using the Scribble reader.

[5] Alphanumeric characters are forbidden here to avoid mistakes; curly braces are forbidden to avoid ambiguity; and the reason for reversing parentheses is to make it more convenient to edit the text and making it fit well inside the `|{...}|` delimiters

[6] To experiment with this, the source of this paper uses a Scribble reader customized to use a backslash, making the source mostly compatible with LaTeX.

```
(define mytext (list "A $(vector "B") C"))
```

several differences become apparent:

- In such mechanisms, the interpolated expressions are expected to evaluate to a string, or are coerced to a string value, to keep the value a string. In contrast, nested `@`-forms can be any expression and have any kind value.
- This is an indication of a more important difference: in the string interpolation example, there is no single interpolated *expression* — only an interpolated *value*. To put this in rough Scheme terms, to get an interpolated expression, we would need to somehow "expand the string" into a sequence of expressions that are spliced into the expression that contains the string — and make sure that such strings always appear in expressions that expect one or more string values. (This kind of expansion requires a global transformation, or a kind of a macro expander that can expand string literals and deals with transformers that return a to-be-spliced value.)
- Finally, note that the vector expression needs to be escaped, while the list expression must not be escaped. We therefore need to be aware of the lexical surrounding of an expression to know if it should be escaped or not — which can be difficult with bigger pieces of code, and it therefore encourages restricting interpolated expressions to relatively small bits. This is similar in nature to quasi-quoting, where we have a "textual context" for text, and we can escape out of it back to Scheme code. This is in sharp contrast to Scribble *expressions* — which are an extension of the language rather than just a new kind of context.

Going back to our reader, nested `@`-forms make perfect sense: they can be conveniently used with any kind of code, and any kind of context. We only need to make sure that every `@`-expression that we use has a proper binding, and that these bindings expect a variable number of "textual body" arguments, mostly strings. For example, a simple adjustment to the previous definition of `show` makes the following example work as expected:

```
> (define (show . text)
    (let loop ([x text])
      (if (list? x) (for-each loop x) (display x))))
> (define (greet str) (list "Hello, " str "!"))
> @show{Once again: @greet{world}}
Once again: Hello, world!
```

As the `@` character turns into a special character in a textual body, it is also subjected to the alternative delimiter syntax as the open and end braces. Specifically, when `|{...}|` are used for a textual body, then nested expressions should similarly begin with a `|@`, and the same goes for the variants with extra punctuations in the delimiter.

```
> @list|{123 @vector{456} 789}|
("123 @vector{456} 789")
> @list|{123 |@vector{456} 789}|
("123 " #("456") " 789")
> @show|{@greet{world} --> |@greet{world}}|
@greet{world} --> Hello, world!
> @show|--{@greet|{world}| --> |--@greet|{world}|}--|
@greet|{world}| --> Hello, world!
```

Note that the nested form determines its own variant of delimiters: the `greet` expressions above do not have to use the alternative delimiters.

At this point the Scribble syntax combines convenient facility for quoting free-form text, with an extension that addresses the same kind of problems as string interpolation devices. But it is not a complete replacement for string interpolation — yet.

2.4 Expression Escapes, String Interpolation

With Scheme S-expressions, parenthesized expressions denote procedure applications (we ignore macros for now), with the head of the expression denoting the procedure to apply. Of course, identifiers can appear in places other than the head of a parenthesized expression, making it a simple reference to a value. The Scribble syntax builds on S-expressions by reading an `@`-form as an S-expression, where the identifier after the `@` stands in the "head position" of the expression, and the following curly braces denote the (textual body) arguments. It is therefore intuitive to make the Scribble syntax correspond to S-expressions: if `@`$\langle id \rangle$ is not followed by a curly-braced textual body, then the result of reading the form is just $\langle id \rangle$.

```
> '@foo{x @y z}
(foo "x " y " z")
> (define name "earth")
> @show{Using "@name": @greet{@name}}
Using "earth": Hello, earth!
```

Furthermore, there is no reason to restrict the head part of an `@`-expression to identifiers only — it can just as well have *any* S-expression. With a textual body, this is equivalent to an application expression that has an application expression in the function position.

```
> '@(foo){bar}
((foo) "bar")
> @show{I repeat: @(compose greet string-upcase){@name}}
I repeat: Hello, EARTH!
```

Without a textual body, we get a generic escape for arbitrary Scheme expressions.

```
> @show{I repeat: @(greet (string-upcase name))}
I repeat: Hello, EARTH!
> '@show{1+2 = @(+ 1 2)}
(show "1+2 = " (+ 1 2))
> @show{1+2 = @(+ 1 2)}
1+2 = 3
```

An interesting result of making the syntax uniform via uses of the Scheme reader is that the head part of an `@`-expression can itself be an `@`-expression. With an input code of `@@foo{bar}{baz}`, the Scribble reader gets invoked by the first `@`, it will then read the head part for the `@`-form which is `@foo{bar}` (resulting in `(foo "bar")`), and finally it reads the textual body and constructs the resulting expression, `((foo "bar") "baz")`. In some cases, this can be useful for procedures that expect more than a single textual body: write the appropriate curried definition where each step consumes a "textual body" as a rest argument, and to use it, specify the right number of `@`s at the beginning of the `@`-form.

```
> (define ((features . pros) . cons)
    @list{Pros: @|pros|.
          Cons: @|cons|.})
> @show{@@features{good}{bad, ugly}}
Pros: good.
Cons: bad, ugly.
```

There is a small exception to such Scheme escapes: when the escaped expression is a literal Scheme string, the string is combined with the surrounding text rather than become a separate expression. As discussed above, this allows using the same syntax for escaping arbitrary text using familiar Scheme quotes.

```
> @list{1 @2 @"3" 4}
("1 " 2 " 3 4")
```

This is another case where Scribble favors utility over uniformity. In general, `@`-forms are not used when precise control over the number of arguments is needed — instead, it is used with a sequence of "textual body arguments", with functions (or macros) where `(f "x" "yz")`, `(f "xy" "z")`, `(f "x" "y" "z")`, and `(f "xyz")` are all equivalent. Given this, the escaped string exception is both

harmless and redundant; however, it is useful in some cases where a single string argument is expected, and in cases where it is important that a chunk of text be kept as a single string (e.g., when each string is later post-processed).

Using the Scribble reader's expression escapes, we can now get string interpolation as a relatively boring special case of using `@string-append{...}` and `@`-expression escapes that evaluate to plain strings. Combining this further with configurable delimiters, we get the functionality of here-strings too.

```
> (define qs string-append) ; qs is for quasi-string
> @qs{... name=@name ...}
"... name=earth ..."
> @qs|{...} @name=|@name {...}|
"...} @name=earth {..."
```

Since the Scheme reader is used to read escaped expressions, we need some way to separate identifiers from adjacent text that will otherwise be read as part of the identifier, or from braces that should not be part of the `@`-form. One solution that a few Scheme implementations of a textual syntax choose is to avoid the problem: require using an escaped `begin` expression when delimiting identifiers. In Scribble, we have avoided such solutions for a few reasons: (1) it is inconvenient, especially when it breaks the textual flow with an otherwise redundant identifier; (2) a `(begin id)` expression is not always equivalent to `id` (e.g., in a quoted list); and (3) given the uniform Scribble syntax, it makes more sense to have a way to avoid a following brace from becoming part of an `@`-expression, rather than leave it to the less convenient escapes.

The Scribble solution for this is to use `|...|` delimiters. The vertical bar character was chosen partly because it is also used by the alternative delimiter syntax (so the motivation is to keep a small set of "special" characters), and partly because in PLT Scheme, `|id|` is read as the `id` identifier.

```
> @show{Hello, @name.}
reference to undefined identifier: name.
> @show{Hello, @|name|.}
Hello, earth.
> @show{Hello, @|name|{}.}
Hello, earth{}.
```

Note, however, that the two uses are different: if the vertical bars were left for the Scheme reader, the above examples would not have worked (the second would still read the period as part of the identifier, and the third would still use the braces as the form's textual body). This is not a problem in practice: `@`-forms are for human-authored texts, so unconventional identifiers are not used; even if there is a case where such an identifier is needed, it is possible to use backslash escapes as usual in the PLT reader.

2.5 Further Extensions

2.5.1 Scheme-Mode Arguments

While we have a convenient way for specifying textual `@`-forms and Scheme S-expressions in both code and text contexts, there are cases where it is convenient to have a form with both S-expression subforms and a textual body. Such a mixture of the two kinds of subforms is particularly useful in a documentation system, where many typesetting procedures consume text (as a 'rest' argument) as well as keyword arguments for various customization options, but also for functions that expect a few non-text argument before the textual body. Yet, using the Scribble syntax that we have seen so far make this inconvenient and error prone.

```
> (define (shout #:level n . text)
    (list text (make-list n "!")))
> @show{@shout{@|#:level|@|3|Greetings}}
Greetings!!!
```

The way that this is addressed in Scribble is by introducing another part to `@`-forms where additional "Scheme-mode" arguments can appear. This part is specified using square bracket delimiters.

```
> @show{@shout[#:level 3]{Greetings}}
Greetings!!!
```

The textual body part of an `@`-form is still optional — making it is possible to specify only the Scheme-mode arguments with no textual body at all. Such `@`-expressions are read as a simple parenthesized list with the head followed by the arguments, making it an alternative form for plain S-expressions.

```
> @list{@list[1 2 3] @(list 1 2 3)}
((1 2 3) " " (1 2 3))
```

This apparent redundancy is a by-product of making the Scribble syntax uniform. While both of the `@`-expression and the escaped S-expression read as exactly the same code, we use a convention where the first form is for expressions that produce text, and the second for other expressions. This makes more sense in places where the default reader mode for a whole file is the Scribble reader's "text mode", which is used, for example, in the PLT documentation sources. In these files, a function like `itemize` that expects items as arguments is written using the first form, while `require` and `define` expressions are written using the second.

```
#lang scribble/manual
@(require some-module)
...text...
@itemize[@item{...text...}
         @item{...text...}]
...text...
```

It is interesting to note that initially, we intended to use Scheme-mode part of `@`-forms only for keyword arguments, and used `@itemize{...}`, which means that the `itemize` function needs to ignore all-whitespace string arguments (and throw an error for other string arguments). Only later we 'discovered' that the square-bracket form makes more sense.

2.5.2 Headless Expressions

The full syntax of an `@`-form is therefore an `@`, a Scheme-syntax form head, a sequence of Scheme-syntax arguments in square brackets, and a sequence of text-mode arguments delimited by square braces. Only *one* of these is required — if only the head is included, we get an expression escape; and if the head is not included, then the Scribble reader will read it as a parenthesized expression with only the Scheme-mode and/or textual-mode arguments. This is mostly useful inside a Scheme quote:

```
> '@{Hello,
     world!}
("Hello," "\n" "world!")
> '@{Hello @{world}!}
("Hello " ("world") "!")
> `@{Hello @|,name|!}
("Hello " "earth" "!")
```

2.5.3 Dealing with Scheme Punctuations

There is a minor problem that is related to Scheme quasiquotes. Say that we want to have a quasiquoted list, and use unquoted `@`-forms in this list. The `@` character in `,@foo` is going to be read as the short notation for `unquote-splicing` — a problem that appears in a few places in the PLT documentation sources[7]. An ugly hack around this problem is to add a space after the comma.

```
> `(html ,@string-titlecase{hello world!})
unquote-splicing: expected argument of type ⟨proper list⟩; given #⟨procedure:string-titlecase⟩
> `(html , @string-titlecase{hello world!})
(html "Hello World!")
```

[7] This is similar to one reason that identifiers are not allowed to begin with a `@` in standard Scheme, as it makes `,@foo` ambiguous.

A related problem occurs when we try to unquote @-forms: if we follow the rules, we need `‘@{...@, @f{}...}` — but Schemers want their unquotes short and convenient (e.g., reader shorthands).

Scribble solves both problems by “pulling out” any of the shorthand punctuations (`’ ‘ , ,@ #’ #‘ #, #,@`) outside of an @-form's head when they are found there, and wraps them around the whole expression. This solves both problems conveniently:

```
> ‘(html @,string-titlecase{hello world!})
(html "Hello World!")
> ‘@html{... @,string-titlecase{hello world!} ...}
(html "... " "Hello World!" " ...")
```

2.5.4 Extending Scheme Escapes

Since we have a specific syntax for specifying Scheme expression escapes (`@|...|`), it is natural to generalize it: instead of a single Scheme expression, it can hold more expressions, or none at all. When more than one expressions are used, they are all spliced into the containing @-form.

```
> @list{foo @|(+ 1 2) (* 3 4)| bar}
("foo " 3 12 " bar")
```

Note that this means that it possible to use this for keyword arguments, e.g., `@para{@|#:style 'small|blah blah}` — but this is inconvenient: it is heavy on punctuation (therefore hard to remember and easy to mistakes), and it is unnatural in the sense that the Scheme-mode part is nested inside the textual part. The square-bracket syntax is vastly easier to remember and simpler to use.

A Scheme escape with no expressions serves a special purpose as a “separator token” that can be used to split some text into two strings. This is rarely used since, as mentioned above, Scribble code tend to not rely on a particular separation of the textual-body arguments. A slightly more useful use of zero-expression escapes is forcing the reader to include an initial or a final newline, more spaces at the beginning or at the end of a line.

```
> @list{
    foo @|| bar
    }
("foo " " bar")
> @list{@||
    @|| foo @||
    @||}
("\n" " foo " "\n")
```

2.5.5 Comments

Finally, the Scribble reader has syntax for `#||#`-like balanced comments and `;`-like line comments. Note that we cannot use Scheme comments in escape expressions; it seems that an escape expression can be used with a balanced Scheme comment (making it an empty escape expression): `@|#|...|#|`. Ignoring the hieroglyphic look of this construct, the problem is that the commented portion is a comment in Scheme syntax. For example, if the commented part contains `|#`, then the comment will terminate prematurely. A proper way to achieve balanced comments without an extension to the Scribble syntax is therefore even more hieroglyphic: `@|#;@{...}|`. Line comments are impossible, since we will still need to insert something on the following line.

A balanced comment in Scribble is written as `@;{...}`. The body is still parsed in text-mode, so it can hold balanced braces, and it can use the alternative delimiter syntax.

```
> @list{foo@;{commented
             text}bar}
("foobar")
> @list{foo@;{...{}...}bar}
("foobar")
> @list{foo@;|{...}...}|bar}
("foobar")
```

A line comment starts with a `@;` (and followed by a character other than `;`), and it ends at the first non-whitespace character in the following line. This kind of comment can be useful in the same way as LaTeXcomments.

```
> @list{Some text, @; a comment
        more text.}
("Some text, more text.")
```

There is no syntax for an “expression comment”, because defining an “expression” in a textual context is more difficult, and also because we had no need for this so far.

3. Text-Mode Source Code

While the Scribble extension is extremely useful, there are cases where we want to change our “perspective”. Instead of thinking about source code as a Scheme file that contains some textual data, we wish to view it as a text file that contains some embedded Scheme code.

There are a number of applications where this shift in view is desirable. The most obvious case in PLT Scheme, and in fact one of the major motivations for the Scribble syntax, is the Scribble documentation system itself. Files are written in one of the Scribble languages (e.g., `#lang scribble/manual`), where all text in the source (except for the `#lang` line itself) is read as strings by default, and rendered as text in the target manual. The exception to this is @-forms (including `@` escape expressions) which are read as usual in the Scribble syntax. For example, the source of the Scribble manual starts with:

```
#lang scribble/manual
@(require scribble/bnf "utils.ss")
@title{@bold{Scribble}: PLT Documentation Tool}
@author["Matthew Flatt" "Eli Barzilay"]
Scribble is a collection of tools for creating prose
...
```

Unfortunately, implementing a reader for such languages cannot be done with a readtable as it is no longer an extension of the Scheme reader. Doing so will require overriding *all* characters (including Unicode characters). But there is no need for a separate reader implementation: at the conceptual level, the syntax of these files is as if the source is all contained in the textual body part of an @-form that surrounds the whole contents. This description provides a concrete hint for implementing such a reader — the only thing that we need is to invoke the part of the Scribble reader that parses a textual body. Indeed, the Scribble reader's API makes its textual body parser available as an “inside” kind of reader function, and this function can be used as a module reader (via the `#lang` mechanism) to parse the source. When invoked this way, the toplevel textual parser is only different in that it is expecting an end-of-file to end the text, rather than a `}` closing delimiter.

The result of this inside reader has a different type from the normal Scheme reader (`read` and `read-syntax` in PLT Scheme) — instead of returning a single syntax value, it returns a list of such values. Most items in this list are strings and the rest are @-forms. The list then becomes the body expressions in the module that is constructed by the `#lang`-specified language. The textual languages then use a special macro that can easily change the semantics of the module's toplevel expressions — `#%module-begin`; this macro is essentially wrapped around the module's body, and therefore it can rewrite these toplevel expressions.

3.1 Specific Use Cases

The “inside reader” feature is a powerful tool for creating textual languages. In the PLT codebase, it is used in a few places in addition to the documentation language. Each of these languages

```
#lang scribble/text
@(require scheme/list scheme/string)
@(define map/nl (compose add-newlines map))
@(define (itemize #:bullet [b "*"] . items)
   (map/nl (lambda (item) @list{@b @item})
           items))
@(define (pseudo-loop statements)
   @list{begin
           @; mix braces and begin/end: the joy of pseudo
           while (true@;{use NIL for dramatic effect}) {
             @(map/nl (lambda (s)
                        (let* ([s (string-append* s)]
                               [s (string-downcase s)]
                               [s (regexp-replace*
                                   #px"\\s+" s "_")])
                          @list{@s();}))
                      statements)
           }
         end})
@(define (both . items)
   @itemize[@list{In text:
                  @(apply itemize #:bullet "-" items)}
            @list{And repeating in a pseudo-code:
                  @pseudo-loop[items]}
            ])
@(define summary
   @list{If that's not enough,
         I don't know what is.})
Todo:
@both[@list{Hack some}
      @list{Hack more}
      @list{Sleep some}
      @list{Hack some
            more}]
@summary
```

```
Todo:
* In text:
  - Hack some
  - Hack more
  - Sleep some
  - Hack some
    more
* And repeating in a pseudo-code:
  begin
    while (true) {
      hack_some();
      hack_more();
      sleep_some();
      hack_some_more();
    }
  end
If that's not enough,
I don't know what is.
```

Figure 1. Preprocessor example

has a specific twist on the concept of text files with embedded Scheme code.

- In the documentation languages, the (mostly textual) expressions are all collected into an implicit global definition that is made available for later rendering,

  ```
  (define doc (list ...textual-contents...))
  (provide doc)
  ```

  except for definition expressions and `require` declarations that are kept at the top-level.

  As usual, the contents of the `doc` definition expression is made of strings and `@`-forms, which evaluate to a hierarchy of structs that the documentation system uses to represent the text, and the Scribble renderers can translate the resulting value to HTML, LaTeX, or text. The documentation languages come with bindings for typesetting markup, facilities to do LaTeX-like processing (grouping parts, splitting to paragraphs, and shorthands like ``` `` ``` and `--`). Additional libraries provide manual-specific bindings (e.g., forms for documenting procedures and for producing examples with automatic sandbox evaluation to show the results of these examples); bindings for articles, and more.
- Another case where the textual file reader is used is the a preprocessor language, `#lang scribble/text`. In this language toplevel expressions are displayed one at a time on the standard output, using a procedure that is similar to the `show` definition in the above examples. In fact, this article is written using this language, where the source can be programmable in Scheme, an obvious improvement over the underlying LaTeX. (In addition, the reader uses `\` as the character for `@`-forms, making it a hybrid language, where both LaTeXand Scheme can be used to define new commands.)

  The actual procedure that displays the text has the added capability of handling indentation and more[8], making it possible to tackle difficult tasks where whitespace matters. For example, this language is used as a preprocessor for the PLT foreign interface, where it needs to gracefully handle oddities such as the requirement that C preprocessor (CPP) directives start at the beginning of a line.

  (Note that our preprocessor cannot *replace* the C preprocessor. This will require implementing C-specific functionality such us finding include files, and knowing which CPP symbols are defined by the local C compiler. In other words, CPP is not just a preprocessor tool — it also serves as a an extension language of the C compiler, and indeed it is implemented as part of the compiler.)

  The Scribble preprocessor language can also be used from Scheme files (extended with the `@`-form syntax) — we use this approach in the source code of the `plt-scheme.org` web pages.
- In addition to these, the PLT web server implements template-based servlets using this facility: a servlet "includes" a template file using `include/reader`, which injects the textual content into a lexical scope in the servlet — for example, one that binds identifiers that are used in the template. The resulting language is similar in nature to existing template systems like the Cheetah template engine for Python[12] — where template files do not even have a `#lang` line.

The Scribble system is still quite young, and we expect to have additional uses for textual languages in the future, in addition to using it in extended-Scheme-syntax files. For example, most uses of the slideshow[3] language still use strings in the source code; switching to `@`-expressions can make writing slides considerably easier, and a textual language might further improve it. Also, we consider implementing some form of a wiki, where `@`-forms (and Scheme) are used as for convenient markup.

To get a rough feeling of how working with Scribble in a textual language looks like, see Figure 1 (real code tend to have much more textual content, of course).

4. Syntax Design, DSLs, and Why Macros are So Great

Scribble is essentially the last version in a decade of various experiments with different approaches to combining textual content and Scheme code. Two older systems are still part of the PLT distribution: `mzpp` is a template based preprocessor, which allows interleaving of text and Scheme code in a conventional way; `mztext` is a preprocessor that is similar in nature to TeX— when `@` is found in the input stream, a function name is read and applied on the stream, allowing it to parse arbitrary amount of text in arbitrary ways (of-

[8] This requires a PLT-specific feature: syntax values contain the position of the expression in the source.

ten a `{...}` piece of text), and return a modified stream that can contain new tokens.

Our experience of the development and implementation of the Scribble syntax demonstrates that extending a language at the concrete syntax level is *hard*. The end result seems sensible now, but the road that leads up to it is paved with subtle decisions. A few examples:

- How do we deal with quoting delimiters? As discussed above, there are several options with different trade-offs.
- Should we stick to the Syntax of scheme identifiers even though it sometimes require extra quoting? Maybe change it to break on periods, colons, etc?
- What is the best way to solve the possible problem with `,@`?
- Even the seemingly minor issue of how whitespaces are handled can be important. For example, if a space is either allowed or forbidden before the braced textual body (either parsing `@foo {x}` as a single `@`-expression or as `foo` followed by `" x"`), we may run into confusing mistakes; which choice is more consistent and/or natural? (The Scribble chooses the latter.) Perhaps such occurrences should just lead to a read error?
- Another question is how should the concrete syntax translate to S-expressions. An earlier implementation used a `dispatch` form, with subforms for the head, the Scheme sub-expressions, and the textual body. The idea was that it would be convenient to have a central point of control for assigning meaning to Scribble forms, by importing or defining `dispatch` — but this did not work out well. Having a central point was not useful (the only `dispatch` definition that was used expanded to an application form), and is better left for the underlying language (in this case, to Scheme macros and/or PLT's `#%app` form); meanwhile, the `dispatch` symbol would show up in quoted forms, making a common Scheme idiom (quoting and quasiquoting) less useful.

  This particular issue has lead to the "principle of least surprise", and a decision that the Scribble reader should not inject any new identifiers into the input that were not originally there. In a sense, this is the same confusion Scheme newbies run into with the `quote` character shorthand, e.g., `(define 'x 4)`.

There is an important lesson about (Scheme) macros and language design here. If the only tool you have to extend your language is a concrete syntax extension, then the difficulty involved in that considerably raises the bar for implementing such extensions, and at the same time extension code is more fragile. At the S-expression level, Scheme gets a clear win by separating the concrete level parser from syntactic extensions, which means that there is only one place to worry about the concrete syntax. Similarly, adding hygiene continues this trend of going higher than the concrete text of the code, as lexical scope is also separated into an independent lower level.

Considering concrete syntax in this light, the utility of `@`-expressions seems even greater. We get to keep the advantage of a separate layer to do the concrete parsing, while making more "text-oriented" information available at the higher level (i.e., in user code). With this, the Scribble reader helps in providing an additional bridge to the textual level of the source code. But this should not come as a surprise, as strings have always been this kind of flexible data containers. (Perhaps too flexible, as seen in languages like Perl and TCL where strings are used for arbitrary semi-structured data, similarly to S-expression abuse in Scheme and Lisp.)

This additional utility of `@`-expressions can be demonstrated by a simple use of here-strings, to specify code in a DSL. For example, a Scheme interface to generate equation images specifies shell commands in one string, and the LaTeXequation in another string.

```
(define commands
 @string-append{
   pdflatex x.tex
   convert -density 96x96 x.pdf -trim +repage x.png})
(define (latex . body) ...)
...
@latex{\sum_{i=0}^{\infty}\lambda_i}
```

An obvious extension to this use is to add interpolation with escape expressions. Taking it further, we can gradually choose textual constructs to abstract over in the foreign syntax using Scheme functions to generate thos syntax, while keeping the text-friendly property of our source code. For example, the preprocessed C code of our foreign interface has a `cdefine` function

```
@cdefine[ffi-lib 1 2]{
  ...C code...
}
```

that generates the boilerplate C function header, adds a CPP `#define` before the function for error messages and an `#undef` after it, and finally registers the function's name and arity to be used later in the C initialization code to create the proper binding.

5. PLT Specifics

Generally speaking, the Scribble reader does not require any features specific to PLT Scheme. In fact, we *hope* that other implementors will consider doing so, which will provide their implementations with the same benefits, as well as benefit the whole Scheme community.

There are, however, a number of specific PLT features that are worth mentioning in this context.

- The syntax objects that are used in PLT Scheme contain source location information. Making the Scribble reader use and record locations is an important feature: parsing errors are properly reported and easy to find, and even in the case of syntax errors and runtime errors, we know where the error is (e.g., with highlights in DrScheme) without resorting to dumping the S-expression that were read for manual inspection. In other words, it makes the Scribble syntax be a real part of the language, rather than a loose add-on.
- The Scribble reader records information about the original form in the parsed syntax values using syntax properties. For example, we can distinguish syntax that was written as an S-expression from one that was written as an `@`-expression, and we can tell which of its subforms were specified as part of a textual body. This feature can be useful in some rare cases where we want the choice of concrete form to affect the meaning of the code.
- The implementation of keyword arguments in PLT Scheme[4] allows keywords to come before other arguments, which is convenient for use in textual forms, where the customization options are better kept next to the function name. This is not specific to the Scribble syntax — it fits well with any similar markup language (e.g., XML attributes).
- In PLT, a `#lang` line at the top of a file is used to specify the language for the file. This specification works by choosing a reader that will wrap the code in a `module` form, which is how both the syntax and the semantics of the language are determined. As mentioned above, there are several languages that use the "inside reader" — for example, Scribble documentation files start with `#lang scribble/manual` and preprocessor files start with `#lang scribble/text`. In addition, there is a 'special' `at-exp` language that is used as a prefix for other languages, for example: `#lang at-exp scheme`. The `at-exp` language will delegate to the `scheme` reader, but will mix-in

the Scribble `@`-expression reader as an extension. This makes it easy to enrich your language with `@`-forms. In both case, the main benefit of `#lang` is localizing the reader to a single file, making it possible to use different concrete syntaxes for different source files, without a damaging global effect.

- Some of the special syntax identifiers that are used in PLT Scheme – like `#%app` and `#%module-begin` – can be used to create customized languages like the preprocessor language, where the result of evaluating a toplevel expression is printed using the preprocessor printer, or the documentation language where they are collected into a definition.

Again, missing these features does not prevent implementing the Scribble reader and getting its benefits. These are features that are generally useful, and enhance the utility of the reader in various ways — not having them means that the respective benefit is lost, but nothing else.

6. Alternative Approaches and Related Work

There are numerous approaches to representing textual content in code — and more than a few have been used by Schemers.

- Many Schemers still use plain Scheme syntax. Some attempts at making things a little better include quasiquoting, and the simple idea of using no spaces around double quotes, which can be seen as a very limited form of interpolation.

```
(string-append "Today is "(date)".")
```

  There are also uses of multi-line Scheme strings, yet it seems that these are disliked enough to be rather rare.

- SCSH[10] was the first implementation to popularize here strings, with a syntax that was adopted by several implementations, including PLT Scheme

```
#<<END
...
END
```

- The Skribe[6] documentation system has greatly influenced the design of Scribble, though not at the concrete syntax level. Skribe has a simple string interpolation facility, where square brackets delimit a string, and a `,(` starts a Scheme expression escape, requiring all such escapes to be parenthesized expressions. Other implementations have a similar functionality, for example, Gauche[8] and JScheme[1] have a built-in facility, and other implementations (PLT included) have add-on libraries. Implementations are mostly simple readtable-based extensions, which usually expand to a `string-append` expression. However, there is no consensus for either the syntax or the details (e.g., whether any value can be used in an expression escape or just strings).
- BRL ("Beautiful Report Language"[9]) and later Kawa[2], use the textual template approach: a source file is parsed as text by default, with Scheme code in square brackets. An interesting aspect of the BRL syntax is that in Scheme code, reversed square brackets delimit strings, which makes it possible to view brackets as marks around Scheme code which is possible not continuous. For example:

```
Are we there yet?
[(if (< (time) eta) ]no[ ]yes![)]
```

  This approach is based on several server-side page generation systems like PHP, JSP, and ASP.

- Finally, we have mentioned a few precursors to the current Scribble syntax, two of which (`mzpp` and `mztext`) are still include in the PLT preprocessor collection. Several other experiments were never made public, including an early template implementation where escapes could be nested in a way that is similar to Kawa, except that expression escapes can nest, forming a tower of interpreters, and in addition each level can run a *different* language. This was an example of a powerful system that was not useful at all for practical purposes.

7. Conclusion

In PLT Scheme, the Scribble reader has proven itself as a valuable tool. It is more powerful than similar facilities in modern languages, and at the same time it is a relatively simple and uniform syntax, which is critical to its acceptance. The reader plays a major role in the success of our documentation system: in our content migration from LaTeXto Scheme, and in adding significant amounts of text — we now have thousands of pages, and the quality of the documentation is much higher than it ever was before. It was also successful in creating a better preprocessor tool, and a template-based generation in our web server. The design process has been long and difficult; getting the advantages of string syntaxes in modern languages, doing so in a framework that fits well with the Scheme midset, and improving on it, are all factors that make this a real challenge.

Any Scheme implementation can gain the same benefits, and in fact, the Ikarus implementation[7] has been recently extended with the Scribble syntax. We hope that additional implementations will follow, leading to an even greater value of the syntax. Scribble is easy to try out in PLT, and it is even possible to use PLT to "manually expand" code with Scribble syntax to plain S-expressions (by simply using `read` on source files), so users of other implementations can try out the syntax using MzScheme as a preprocessor before implementing it.

References

[1] Ken Anderson, Tim Hickey, and Peter Norvig. JScheme. `http://www.norvig.com/jscheme.html`.

[2] Per Bothner. The Kawa language framework. `http://www.gnu.org/software/kawa/`.

[3] Robert Bruce Findler and Matthew Flatt. Slideshow: Functional presentations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 224–235, 2004.

[4] Matthew Flatt and Eli Barzilay. Keyword and optional arguments in PLT Scheme. In *Proceedings of the Tenth Workshop on Scheme and Functional Programming*, 2009.

[5] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. In *ACM SIGPLAN International Conference on Functional Programming*, 2009.

[6] Erick Gallesio and Manuel Serrano. Skribe: a functional authoring language. *Journal of Functional Programming*, 15(5):751–770, 2005.

[7] Abdulaziz Ghuloum. Ikarus Scheme. `http://ikarus-scheme.org/`.

[8] Shiro Kawai. Gauche. `http://practical-scheme.net/gauche/`.

[9] Bruce R. Lewis. BRL: the beautiful report language. `http://brl.sourceforge.net/`.

[10] Olin Shivers. A Scheme shell. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1994.

[11] Michael Sperber (Ed.). The revised^6 report on the algorithmic language Scheme, 2007.

[12] Cheetah, the python powered template engine. `http://www.cheetahtemplate.org/`.