Hygienic Literate Programming: Lessons from ChezWEB

Aaron W. Hsu
awhsu@indiana.edu
Indiana University

Abstract. Literate programming systems are a class of domain specific languages designed to encourage writing programs specifically to be read as essays or books by humans instead of by machines. Systems like CWEB, WEB, and ChezWEB allow the user to associate arbitrary code bodies with a concise but natural language description. That description may then be referred to in other sections of the program source, and the code body associated with the description is substituted for the reference in the program source. Using Scheme macros, we describe how this substitution is performed unhygienically. We then describe and implement an innovation unique to ChezWEB that extends the notion of code reordering to provide hygiene and referential transparency guarantees for section references and code substitutions in the system. Our implementation strategy serves as a design pattern for implementing other language forms for which the programmer desires similar hygienic guarantees.

1. Introduction. Literate programming [5] is a disciplined approach to program design and documentation. The literate programmer composes programs with the intention that they will be read by humans in ways not dissimilar to the ways one may read a book. Such a programmer interleaves code and prose within a single document to be executed by the computer or read by another programmer. Literate programming shifts the focus from creating programs that run on computers towards programs written to carefully exposit code for the sake of a human audience. Integral to this exposition, literate programming tools allow a programmer to rearrange pieces of code in ways that a computer would reject, but that, presumably, make the code clearer or more easily explained to a human.

Literate programming systems usually take the form of simple domain specific languages that admit two interpretations, one as code that a computer can compile or interpret, and another as a source for typesetting an essay-like document for human reading. These languages map nicely to the notion of preprocessor macros common throughout general purpose programming languages. Like most meta-languages for programming in general, however, all literate programming systems of which the author is aware do textual level copying when reordering code. This is equivalent to unhygienic macro expansion and suffers the same limitations and problems.

The Scheme language, with its powerful macro facility, presents an excellent opportunity to address the problems of unhygienic code reordering in literate programming. We have developed and used a literate programming system called ChezWEB which introduces a notion of hygiene for literate programs. In its second major version, we have learned a number of things about implementing such hygiene conditions using the Scheme macro language, which are presented here. We discuss the desirability and usability of a hygienic literate programming system. We also describe in detail the implementation of hygiene for ChezWEB. This implementation has gone through a number of revisions and improvements, and we believe this process is especially elucidating to programmers who may wish to implement similar guarantees for other user-defined extensions to Scheme.

We begin by delving into background information on literate programming and Scheme macro programming. We then discuss the motivations for ChezWEB; we discuss its notion of hygiene in detail. An exposition of the ChezWEB runtime system follows, which implements the hygiene semantics of the system. We identify limitations of our current approach and lay out future work. We also discuss related work on literate programming throughout the Scheme community and other communities, and briefly comment on their relation to ChezWEB.

This paper is itself a ChezWEB program and the reader is encouraged to obtain a copy of the ChezWEB system[1] and investigate all of the examples, implementations, and illustrations in the paper. Organizational aids such as an index, a list of code sections, and a table of major

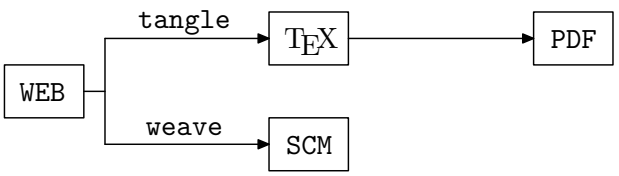
[1] http://www.cs.indiana.edu/~awhsu/

sections are included at the back of the paper. Cross references are littered throughout the code and body of the paper, and point to other sections in the paper that contain useful or important information. A copy of the web for this paper is available online at the author's home page.

2. Background and Terminology. This and the following few sections describe the basic terminology that will be used throughout this paper, as well as any background information that will be helpful to understanding the work in later sections. The reader already familiar with concepts related to hygiene, `syntax-case`, literate programming and `WEB` systems, and with Chez Scheme's `module` and `with-implicit` forms can safely skip to the Chez`WEB` design.

3. Literate Programming. Donald Knuth first conceived of literate programming, creating the `WEB` literate programming system for Pascal [5]. The basic literate programming tenets emphasize the close association between documentation, code, and the style of documentation. Literate programming is the construction of a program much as the construction of an essay or paper, where code and prose work to exemplify some particular topic or behavior. The code is designed to be read as a rendered paper (maybe as a web page). Literate programming systems usually connect a documentation language, often TEX [4] or XML, together with some programming language. Some systems are mostly agnostic towards one or both of these languages [8].

Literate programming systems enable a programmer to deconstruct a program into chunks that can be reordered. The code itself is "tangled" into a form suitable for program execution, or "woven" into a document that may be rendered for human consumption. Some literate programming systems (sometimes called semi-literate) do not have explicit code reordering primitives in their meta-language [8].



Most literate programming systems are implemented as preprocessors, a separate layer above the code or documentation language, and, in some sense, implement a specialized macro system on top of the programming language. In most of these systems, the literate programming meta-language is unhygienic and code reordering is done at the textual/string level like the C preprocessor.

Literate programming systems usually allow one to give a name to a string of code, and then to use that name in other strings of code, where the system will then find references to this named chunk in the pieces of code and replace them with the literal string associated with the name. This is usually done simply by replacing the reference to the name with the string in a preprocessor step before the compiler runs. This makes the copying and pasting of code from place to place entirely unhygienic and not at all referentially transparent.

4. WEB-style Literate Programming. In the style of literate programming espoused and encouraged by the `WEB` family of systems, of which Chez`WEB` is a member, the programmer makes minimal annotations to the code, through the use of control codes delimited by the `@` character. A web is a text file containing a series of sections, where each section consists of a text part and a code part, possibly with other elements, depending on the system. In this paper, we refer to the code parts sometimes as chunks.

When a web is woven together, a TEX file is produced that typesets the documentation and the code. In these systems, pretty printing plays an important role in the presentation of the code, and they contain pretty printers that work hard to present the code in an esthetically pleasing manner. Additionally, cross-referencing of various sections, where the code in one section may be referenced or extended in another section are managed automatically. Indexing of identifiers and a listing of all the section names are also part of the metadata that the system tracks. This is in contrast to other systems, like noweb [8], which provide the code reordering functionality of a full literate programming system, but which do not provide any default support for language specific metadata or cross-referencing. Systems like noweb usually have some way of incorporating this sort of indexing and pretty printing should the user desire this functionality.

In a `WEB` program, there are named sections, which associate code parts with a specific, usually short but descriptive natural language name. This name may then be referenced in other sections with the appropriate annotation. As an example, in this section we define a section that can be used in place of a message string. The source code for this section definition looks like this:

```
@<Load message@>=
"Hello, you've loaded the system."
```

This is then typeset as follows when the document is woven together.

⟨ The load message 4 ⟩ ≡
```
"Hello, you've loaded the system."
```
This code is used in section 5.

5. Notice that the system provides cross references and indicates the section number where the code part is defined. In ChezWEB we have not yet implemented a pretty printer, so the code in woven ChezWEB documents is printed in a typewriter font.

We can use the above message by referencing it in the body of another section. Named sections as in the above are not evaluated by default, but are saved for use in later sections. Top-level program constructs are indicated by using the `@p` control code. These are output directly to the top-level of the file when the web is tangled together. If we then `load` this file, these expressions will be evaluated. If we reference a named section somewhere in a top-level code fragment, then that section will also be evaluated and expanded. In this section, for demonstration purposes, we will put a print message at the top level that prints the message above. The source code for this looks like this:

```
@p
(printf "~a~n" @<The load message@>)
```

In the woven output of the program, this will look as follows:

```
(printf "~a~n" ⟨ The load message 4⟩)
```

6. Semi-literate Programming. There are other types of systems that are sometimes referred to as literate programming systems. In actuality, these are considered semi-literate systems. In these systems, which include Literate Haskell [7] and SchemeWEB [9], there is no special support for the arbitrary reordering of fragments of code. Instead, the system only supports the intermingling of top level code and top-level documentation. This documentation usually takes the form of either marked up comments, or, as in SchemeWEB, it may be plain text, where the system determines what elements of the text file were meant as program source. These systems sometimes include a pretty printer for the code, but this is not essential to the concept of semi-literate systems. The important distinction between semi-literate and literate programming systems is the ability to arbitrarily reorder chunks of code as a part of the literate system. In this paper, we deal specifically with fully literate programming systems, though we discuss the relation of our work to semi-literate systems when appropriate.

7. Scheme macros. Scheme macros are transformers from source code to source code. They provide two important guarantees over most other macro systems for general programming languages. These are known as the hygiene condition and referential transparency. Specifically, in other macro languages, we may inadvertently capture bindings at the call site of the macro that we do not wish to capture, or we may introduce bindings that pollute the call site namespace in bad ways.

8. The hygiene condition assures the programmer that any binding that is introduced in the body of a macro expansion will be visible only within the scope of that macro. In the following example, the variable `hide-me` is guaranteed to only be visible in the body of the macro.

⟨ Hygiene example, define `hidden` 8 ⟩ ≡
```
(define-syntax hidden
  (syntax-rules ()
    [(_ id x)
     (begin
       (define hide-me x)
       (define (id) hide-me))]))
```
This section exports hidden.

This code is used in section 9.

9. In the above example, we can use this macro as many times as we want, and each id will be associated with its own `hide-me` variable. Moreover, the `hide-me` variable will not be visible in the surrounding context. We can test that with the following code:

⟨ Hygiene example, test `hidden` 9 ⟩ ≡
```
(let ()
  ⟨ Hygiene example, define hidden 8⟩
  (hidden a 3)
  (let ([x (a)])
    (hidden b 4)
    (list x a b)))
;=> '(3 3 4)
```

10. We can think of the hygiene condition as ensuring that bindings don't flow out of the context of the macro body in which they are introduced. Referential transparency is a safety guarantee about the free variables in the body of a macro. Specifically, any free reference in the body of a macro will be scoped at the definition site, rather than at the call site. This means that the expansion of a macro given the same identifiers as input is invariant to the context of the macro call. As an example, suppose we define a macro that uses `if`.

⟨ Ref. trans. example, define `reftrans` 10 ⟩ ≡
```
(define-syntax reftrans
  (syntax-rules ()
    [(_ x) (if x 1 0)]))
```
This section exports reftrans.

This code is used in section 11.

11. Assuming that we define `reftrans` in a context where `if` is bound to the standard Scheme `if`, then we want to be assured that calling `(reftrans x)` will return `1` if the `x` value is a true value, and `0` otherwise. Referential transparency allows us to make this claim, even when the context could give us trouble, as in the following code, where we redefine `if` in the calling context.

⟨ Ref. trans. example, test `reftrans` 11 ⟩ ≡
```
(let ()
  ⟨ Ref. trans. example, define reftrans 10 ⟩
  (let ([if 5]) (reftrans if)))
;=> 1
```

12. The `if` that we bind will be considered a different identifier from the one used internally to the macro, and so even passing `if` as the `x` value for the macro will not cause a conflict.

13. The SYNTAX-CASE Macro System. The R6RS [12] standard extends the purely hygienic `syntax-rules` system that was defined in the R5RS standard. The system is called `syntax-case` for the pattern matching construct of the same name provided by the system. Importantly, hygiene is still enforced automatically by the `syntax-case` system, and hygiene is the default operation. [14]

The `syntax-case` system introduces a few extensions to the concept of Scheme macros that greatly increase its expressive power. A macro is now a procedure bound to an identifier using the `define-syntax` form. This procedure takes a single argument and returns a single value. These procedures may perform arbitrary computation and may use the whole Scheme language. The `syntax-case` system defines a new data type called syntax objects that encode the structure of an expression together with the lexical information necessary to fully determine the scoping of each identifier in the expression. Macro procedures, called macro transformers, are passed syntax objects as their single argument, and must return syntax objects.

The following code defines a `reftrans` macro equivalent to the example above using `syntax-case`.

⟨ Define `reftrans` with `syntax-case` 13 ⟩ ≡
```
(define-syntax reftrans
  (lambda (x)
    (syntax-case x ()
      [(_ x) #'(if x 1 0)])))
```
This section exports reftrans.

14. The `#'` form in the above is a lexical shorthand where `#'x` is the same as saying `(syntax x)` and evaluates to a syntax object that represents the identifier `x` scoped as if it were the body of a macro expansion that had been defined there. That is, the `x` will refer to the nearest lexical binding in scope. In the above code, for example, the `if` identifier will still refer to the standard Scheme `if` and `x` will refer to the pattern variable in the `syntax-case` pattern.

In the `syntax-case` system, despite the name, the `syntax-case` form is entirely optional. It is simply a means of destructoring and working with the syntax objects. The important element here is that we have syntax objects as first-class citizens in the language.

Moreover, `syntax-case` defines a way of extracting the datum value of the syntax object (such as one might pass to `eval`) as well as allowing a programmer to change the lexical information of the syntax object so that identifiers will be scoped at different locations of the program. In this paper, the concept of changing the lexical scope will be referred to as recoloring or rewrapping, where one thinks of lexical regions or contexts as each having a unique color.

In `syntax-case` one uses `syntax->datum` to extract the datum representation of a syntax object without any lexical information. For example, `(syntax->datum #'x)` will return the symbol `'x`. Given this symbol, we may recolor the colorless symbol by using `datum->syntax`. Here `(datum->syntax #'k 'x)` will color `x` with the same color as the `#'k` syntax object. The returned syntax object will be an identifier that will capture the binding for `x` at the lexical scope of the `#'k` form.

In the following example, we define a macro that returns the value of `x` as defined at its call site. Recall from above that in purely hygienic systems, we are not able to do this. If we have a free reference to `x` in the body of our macro, then it must refer not to the `x` bound at the macro call site, but instead to the binding at the definition site. Thus, `syntax-case` allows us to selectively bend hygiene to insert and extract bindings from given contexts.

⟨ Demonstrate `syntax-case` capture 14 ⟩ ≡

```
(let ()
  (define-syntax sc-x
    (lambda (x)
      (syntax-case x ()
        [(k) (datum->syntax #'k 'x)])))
  (let ([x 3]) (sc-x)))
;=> 3
```

This section exports sc-x.

15. In the following sections, we use an extension to standard Scheme that mirrors the procedure syntax of `define`. That is, a `define-syntax` form of the shape `(define-syntax (m x) ...)` is equivalent to the form `(define-syntax m (lambda (x) ...))`. We also use a form `with-implicit` to simplify the recoloring of identifiers. We give a simple definition for `with-implicit` below. The `with-syntax` form used in the following code is a binding form for pattern variables, where each left hand side clause in the bindings is a pattern such as in the left hand sides of `syntax-case` patterns, and the right hand side is an arbitrary expression that must evaluate to a syntax object.

⟨ Define `with-implicit` 15 ⟩ ≡

```
(define-syntax (with-implicit x)
  (syntax-case x ()
    [(_ (k id ...) b1 b2 ...)
     #'(with-syntax
           ([id (datum->syntax #'k 'id)]
            ...)
         b1 b2 ...)]))
```

This section exports with-implicit.

16. Chez Scheme Module Form. We also rely in this paper on the Chez Scheme `module` form. This is a simple syntactic form that encapsulates a body of code and selectively makes certain bindings in that body visible to the surrounding context. The `module` form has the following syntax that we use in this paper:

```
(module (e ...) body+ ...)
```

Where the `e ...` pattern maps to zero or more identifiers and `body+ ...` maps to one or more expressions, with definition expressions coming before normal value expressions. Thus, `body+ ...` is like the body of a `let`.

⟨ Example using `module` 16 ⟩ ≡

```
(let ()
  (module (x) (define x 3))
  x)
;=> 3
```

17. The Design of ChezWEB. There are two major versions of ChezWEB. The first generation was designed as a set of libraries and scripts that bootstrapped those libraries together. This meant that all ChezWEB programs were written as S-expressions with all of the documentation embedded in Scheme strings. In this case, writing TEX code was tremendously inconvenient. Moreover, the main benefit of doing this was to enable one to type a literate program directly into the REPL. This was useful for some applications, but it was so rarely used that it did not really represent a significant feature. The use of S-expressions as the main element lifted the documentation to a second-class citizen.

In the second generation of the program, which generated this document itself, is instead implemented with the same workflow and style as `CWEB` described in the background section. Namely, there are two programs `chezweave` and `cheztangle` that process webs. These two programs do the same thing that the equivalently named programs do in `CWEB` and have similar command signatures. Thus, ChezWEB programs must be preprocessed into either a Scheme file or a TEX file. We still need some runtime support in order to enforce our hygiene condition, but the ChezWEB runtime is embedded directly in the output files, rather than being a separate library, as was required by the first generation of the software. Because we implement the most significant element of the ChezWEB system as a runtime element, we are also able to provide a library exposing this runtime functionality to users. This gives us the benefit of the hygienic code chunking that we get from the normal ChezWEB directly in the REPL. This mitigates the disadvantage of having a separate preprocessor step in our experience.

18. The novel component of ChezWEB is the way in which is handles named sections of code, such as those that have been used in this paper thus far. The reader will note annotations below the code parts that indicate what bindings are exported from a chunk. This is part of the hygienic guarantee that ChezWEB provides. The ChezWEB system provides two guarantees, called, analogously to Scheme macro guarantees, the hygiene and referential transparency conditions. We can state these guarantees informally as follows:

Hygiene. No definition in the body of a code section shall be visible to the surrounding context of a section reference unless that binding is explicitly exported by the code section at the point of definition.

Referential Transparency. Free variables in the body of a code section will always refer to the nearest lexical binding at the definition site of the code

section, unless the identifier is explicitly noted as a capture at the definition site, in which case the binding to a captured variable will refer to the nearest lexical binding at the reference site of the code section.

These mirror the guarantees provides by the `syntax-case` system. Put more concisely, no binding enters or leaves the code section at the reference site without explicit direction from the programmer. Our system notes these explicit exports and captures as annotations at the bottom of each named code section in the woven output. It should be noted that in the normal case of tangling a web file using the `cheztangle` program, the referential transparency guarantee is even more restricted such that all free variables not explicitly captured will always refer to the top-level binding for that identifier in the program. This is due to the way that we lay out tangled code. However, even when the runtime system enforcing these is used in non-global contexts, such as when it may be used internally in the REPL, then the general referential transparency guarantee will hold.

19. The above is all well and good, but one may reasonable question the value of introducing a notion of hygiene into literate programming, which, at some level, is about reordering programs. Indeed, a literate programming system that did not allow for the export and capture of variables would not be particularly useful. Literate programs are in their nature unhygienic to some extent, so why do we even bother with a set of hygienic guarantees?

While it is absolutely true that we need to be able to thread data back and forth out of a chunk, it is not true that we always wish to do this. Similar to the problems that macro writers face in Scheme, one may encounter these same problems in a literate program. The following example demonstrates a common and desirable methodology that we have found useful and productive in writing our literate programs. In this example, suppose that we wished to define a function to compute factorial over a number of elements in a list. It would be nice to separate out the process of computing factorial from that of actually performing that operation on every element of the list.

⟨ Define `map-fact` 19 ⟩ ≡

```
(define (factorial n)
  (if (zero? n)
      1
      (* n (factorial (-1+ n)))))
(define (map-fact lst)
  (map factorial lst))
```

This section exports map-fact.

This code is used in section 21.

20. In this example, would not want the `factorial` function bleeding out into the outer context of the reference site, perhaps because we are doing this for many implementations of an algorithm, and each version uses slightly different internal function implementations, but the same names. In this case, we have to know ahead of time when we write the above code whether or not we are going to conflict with anything at the call site, and choose names that don't conflict. Moreover, we may be in a context where we want to return a value, and we are not in a definition context. It would still be nice to be able to use internal definitions.

Both of these situations could be accommodate by using `module` and `let` appropriately, but then our code is littered with `module` and `let` everywhere. It is much cleaner in the presentation of code and in terms of reliability to have our system handle this automatically. After all, the point of literate programming is to make it easier to make your programs look good and easy to read. Thus, the user should not be burdened with watching their backs when reordering chunks of code, just as they should not be burdened with manually dealing with name conflicts in macros. Wrapping `module` and `let` around your unhygienic code sections is the philosophical equivalent to `defmacro` and `gensym` style macro programming.

21. The following example demonstrates these conditions. The ⟨Define `map-fact`⟩ chunk will export `map-fact`, but we can be assured that its internal definition of `factorial` will not conflict with our binding of the same name in this code section.

⟨ Illustrate guarantees 21 ⟩ ≡

```
(define factorial 'nothing)
⟨ Define map-fact 19 ⟩
(map-fact (iota 6))
;=> '(1 1 2 6 24 120)
```

22. Another reasonable objection is to the use of a WEB system at all. Indeed, if one doesn't buy into typeset programs, then any form of literate programming will seem pointless. However, people have seen the value of these types of documentation systems, as evidenced by systems like noweb, SchemeWEB and Literate Haskell, as well as numerous others. Why then did we choose the admittedly more complicated model of WEB instead of a semi-literate approach? In response, we can only say that we want better weaving support for code reordering, indexing, and the like. We would have ended up rolling our own indexing, cross-referencing, and so forth for any semi-literate system, as typeset programs are less useful if you cannot navigate through the program quickly, so this makes the features of WEB systems a good, direct fit for our needs.

We do not, however, wish to neglect the semi-literate systems. Since we make available the code-reordering primitive as a library, users are welcome to use them in whatever code they want, literate or not. The ChezWEB runtime system fits very nicely with semi-literate programs if all that is desired is to enable clean and reliable code-reordering inside of one of these systems; the tangle and weave programs for ChezWEB can be safely ignored.

23. Implementation. We will now walk through a progression of implementations for the hygiene guarantees of ChezWEB. These somewhat mirror the actual implementations that we went through in developing ChezWEB from the first generation through its current version. This progression illustrates many of the important elements of the syntax-case macro system that may not be immediately apparent when reading about the macro system. It also serves as a design pattern and set of warnings for programmers attempting to do the same sort of thing in other languages. Specifically, there are certain pitfalls into which a macro writer may fall, and these should be studiously avoided. Fortunately, the syntax-case system makes these pitfalls more apparent than might otherwise be possible.

Our target is a macro defsec that binds a name to a body of code. That name can then be referenced in other definition contexts that will expand to the body of the code.

```
(defsec (name caps ...) => (exps ...)
  body+ ...)
```

In this pattern, we want to explicitly capture the caps ... identifiers and explicitly export the exps ... identifiers. A simple example illustrates our design:

⟨ Simple defsec Example 23 ⟩ ≡

```
(defsec (defx y) => (x) (define x y))
(let ([y 3]) defx (list y x))
;=> '(3 3)
```

This section captures defsec.

This code is used in sections 25, 35, and 40.

24. To begin with, let us first encode the semantics of the traditional literate programming model of code reordering. This will serve as a model for us to extend and improve upon. To make things simpler, we will also restrict the type of code sections we deal with to only those which may export bindings, rather than to those which may expand to values.

The normal model of code reordering binds the name of a section, which we will encode as an identifier, to a code body, such that the reference to that identifier will expand into the code body and colored the same as the color of the surrounding context of the call, that is, unhygienically. We will ignore captures and exports here. This is trivial in syntax-case.

⟨ Define unhygienic defsec 24 ⟩ ≡

```
(define-syntax (defsec x)
  (syntax-case x (=>)
    [(_ (n c ...) => (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (datum->syntax x
           '(begin b1 b2 ...)))]))
```

This section exports defsec.

This code is used in sections 25 and 27.

25. We can test our simple example against this implementation using == which we define in the "Testing" appendix, and it does indeed work.

```
⟨ Define testing primitive == 52 ⟩
(== '(3 3) "Simple unhygienic defsec"
  ⟨ Define unhygienic defsec 24 ⟩
  ⟨ Simple defsec Example 23 ⟩)
```

26. It should be obvious at this point why this macro will also exhibit the same capture problems of existing literate programming systems. If we were to redefine define, then the above case would simply fail to work entirely.

⟨ Break unhygienic defsec 26 ⟩ ≡

```
(defsec (defx y) => (x) (define x y))
(let ([define 3] [y 3]) defx (list y x))
```

This section captures defsec.

This code is used in sections 27, 35, and 40.

27. Here, because we are recoloring the entire body, we see the exact same problems as we would expect from any unhygienic macro approach. The capture problem in literate programming is just a special case of normal macro variable captures problems.

```
(== '(3 3) "Break unhygienic defsec"
  ⟨ Define unhygienic defsec 24 ⟩
  ⟨ Break unhygienic defsec 26 ⟩)
```

28. We can then go to the opposite approach, by making the entire system completely hygienic. However, in doing so, this means that we cannot have any implicit captures or exports when we call the macro. We will have to explicitly pass all of the exports and imports through the macro. This changes the signature of `defsec` slightly.

```
(define-syntax (defsec/hyg x)
  (syntax-case x (=>)
    [(_ (n c ...) => (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [(_ c ... e ...)
            #'(begin b1 b2 ...)]))]))
```

29. This will then provide us a means of doing the exports and captures because we explicitly indicate to the system, as the programmer, what variables go where. This changed behavior also means that it is possible to provide identifiers of different names to our macro, instead of just `x` and `y`. In fact, this behaves exactly like the `define-syntax-rule` form and its kin that are found in various Scheme implementations.

```
(== '(3 3 3) "Fully hygienic"
  (defsec/hyg (defx y) => (x) (define x y))
  (let ([define 3] [y 3])
    (defx y x) (list define x y)))
```

30. This actually works pretty well if we assume that we have a completely hygienic system. That is, assuming that we never insert identifiers anywhere that they do not occur explicitly in the source, then the above macro works remarkably well. Unfortunately, this macro starts to fall apart as soon as our assumptions start to change. Indeed, with much of the code that programmers write, we cannot assume such identifier "purity." Take for instance a case where one might wish to define a new record type using the R6RS `define-record-type`. If we use the long form, then things are all fine:

```
(== #t "Full hygiene, Long record form"
  (defsec/hyg (def-x) => (make-x x?)
    (define-record-type (x make-x x?)))
  (def-x make-x x?)
  (x? (make-x)))
```

31. Many programmers, however, find the long form of `define-record-type` to be inconvenient and unnecessarily verbose. Rather, they prefer to use the short form, where we can collapse the three bindings into just one identifier reference. In doing so, `define-record-type` then generates the appropriate identifiers based off the one that it was given, and inserts them into the calling context. In this case, our fully hygienic system will fail. Because we only know about the identifiers visible in the body, the implicitly defined `make-x` and `x?` identifiers, which do not occur in the body, will never be substituted for the outer context identifiers when the pattern substitution occurs.

```
(== #t "Full hygiene breaks"
  (defsec/hyg (def-x) => (make-x x?)
    (define-record-type x))
  (def-x make-x x?)
  (x? (make-x)))
```

32. It may be clearer to see why this happens if we show the expanded output. Suppose we expand the following code:

```
(let ()
  (defsec/hyg (def-x) => (make-x x?)
    (define-record-type x))
  (def-x make-x x?)
  (x? (make-x)))
```

We will get output something like this:

```
(letrec* ([rtd.4 ---]
          [rcd.5 ---]
          [make-x.6 (constructor rcd.5)]
          [x?.7 (predicate rtd.4)])
  (x? (make-x)))
```

In the above example, because the identifiers `make-x` and `x?` that are defined by the `define-record-type` form take on the colors of the `x` in the body of the macro, which is only visible inside the body of the `defsec/hyg` form (hygiene condition), then they will not refer to any binding outside of that context, even though they have the same symbolic representation; the outer context is of a different color. In the long form, this wasn't a problem because we explicitly passed in the variables with the correct coloring that we wanted already on them, and

`define-record-type` just used those identifiers. Without that explicit threading, however, the fully hygienic approach falls apart.

33. Above and beyond the literal problems that we see above, there is also an issue of usability when dealing with the fully hygienic system. It requires that we indicate in two places the set of identifiers that we are capturing and exporting, and it is possible that these identifiers may not actually have the same textual representation. Moreover, any changes to the definition site of the code that alters the list of captures or exports requires changing all of these signatures everywhere else. This is very inconvenient when dealing with these chunks. In the first generation of ChezWEB the programmer was in charge of partially threading through the identifiers in a manner much like the fully hygienic approach. This caused a number of problems on more than one occasion with identifier mismatches and changes to captures lists, and so forth. It is much more convenient to be able to specify captures and exports at the definition site and not to worry about explicitly mentioning them at the call site.

Let's improve on the fully hygienic version by eliminating the need for the explicit identifier passing. The most common idea and technique when dealing with this is to make use of the `with-implicit` paradigm to recolor the set of identifiers you capture or export based on the outer context. Doing this, we should be able to remove the need to explicitly thread the identifiers through.

⟨ Define `with-implicit` based `defsec` 33 ⟩ ≡

```
(define-syntax (defsec x)
  (syntax-case x ()
    [(_ (n c ...) => (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [n (identifier? #'n)
            (with-implicit (n c ... e ...)
              #'(begin b1 b2 ...))]))]))
```

This section exports defsec.

This code is used in sections 35 and 37.

34. The above allows us to reformulate our tests before into the standard signature that we used for `defsec`.

⟨ Test `defsec` with long record 34 ⟩ ≡

```
(defsec (def-x) => (make-x x?)
  (define-record-type (x make-x x?)))
def-x (x? (make-x))
```

This section captures defsec.

This code is used in sections 35 and 40.

35. This successfully addresses most of the problems that we had with the previous examples. Now all of the identifiers are colored correctly, and we don't have to explicitly list all of the captures and exports each time that we use the code section.

```
(== '((3 3) (3 3) #t) "Using with-implicit"
  ⟨ Define with-implicit based defsec 33 ⟩
  (list
    ⟨ Simple defsec Example 23 ⟩
    ⟨ Break unhygienic defsec 26 ⟩
    ⟨ Test defsec with long record 34 ⟩))
```

36. Unfortunately, this is not enough to address the issue with the short form `define-record-type`. We can recast our fully hygienic example to one that uses `defsec` instead of `defsec/hyg`.

⟨ Test `defsec` with short record 36 ⟩ ≡

```
(defsec (def-x) => (make-x x?)
    (define-record-type x))
def-x (x? (make-x))
```

This section captures defsec.

This code is used in sections 37 and 40.

37. This will still fail to give us the expected result that we want.

```
(== #t "With-implicit fails on short form"
  ⟨ Define with-implicit based defsec 33 ⟩
  ⟨ Test defsec with short record 36 ⟩)
```

38. If we examine the expansion, we see the the exact same issue as happens with the fully hygienic version happens when using `with-implicit`. That is, if we think of `with-implicit` as recoloring all of the identifiers in its first clause that appear in `syntax` forms of its body, then it becomes obvious that even though we are recoloring `make-x` and `x?` using `with-implicit`, in the short form of the expansion, there is nothing there to recolor, so nothing gets recolored the way we want.

In the first generation of ChezWEB, our approach was to introduce two forms `@<` and `@<<` that were wrapped around the names of chunks at call sites. Using `with-implicit` as above, we would always recolor `@<<` to have the outside context, and `@<` would always have the inside context. These wrappers would then be in charge of passing the right coloring information to the chunk, which could rewrap its identifiers as appropriate. Unfortunately, while this fixes the problem to some extent for chunk exports, it doesn't fix it for forms that ChezWEB may not know about, such as `define-record-type`. Moreover, it unnecessarily restricts where identifiers can be visible; either they are visible in the outer context or the inner context, but not both.

In the second generation of ChezWEB, we take a different approach. Instead of introducing two extra forms, we change the body shape of our expansion. Specifically, we make use of modules to better encapsulate the body code. Until now, we were wrapping the body code with a `begin` form. This means that we cannot arbitrarily mix expressions and definitions inside of the body of our chunks without knowing where and how the chunks will be used at the call site. Instead of doing this, we now wrap our code in a `module` form that exports what we want.

```
(module (e ...) body+ ...)
```

The `module` form inherits the environment for the `body+ ...` from the enclosing context, like `let` does. In the above template, we are not recoloring any identifiers, so everything is colored with the inner context. This means that all the code in the body is capable of getting access to all of the exported code that we intend to also be visible in the outer context. This is a step up from our approach using `with-implicit` which only let us make exported identifiers visible in either of the contexts, but not both.

To make the exported identifiers visible to the outer context, we want to alias them together. Fortunately, Chez Scheme has a form `alias` which allows us to do just that. We can apply the same technique to bring in bindings from the outer calling context to the inner context. This leads us to the following expansion template:

```
(module (oe ...)
  (alias ic oc) ...
  (module (ie ...) body+ ...)
  (alias oe ie) ...)
```

In this template, the `i` and `o` prefixes stand for inner and outer contexts.

We can simulate the behavior of the `alias` function in R6RS by using identifier syntaxes. This does not fully match the semantics that `alias` gives us, but it works if the Scheme system in question does not have an equivalent `alias` syntax.

```
(define-syntax (alias/r6rs x)
  (syntax-case x ()
    [(_ t s)
     #'(define-syntax t
         (identifier-syntax
           [t s]
           [(set! t e) (set! s e)]))]))
```

39. We are now in a good position to implement a working `defsec` that does not exhibit the weaknesses described above.

```
(define-syntax (defsec x)
  (syntax-case x (=>)
    [(_ (n c ...) => (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [n (identifier? #'n)
            (with-syntax
                ([(ic (... ...))
                  #'(c ...)]
                 [(ie (... ...))
                  #'(e ...)]
                 [(oc (... ...))
                  (datum->syntax x
                    '(c ...))]
                 [(oe (... ...))
                  (datum->syntax x
                    '(e ...))])
              #'(module (oe (... ...))
                  (alias ic oc) (... ...)
                  (module (ie (... ...))
                    ((... ...) b1)
                    ((... ...) b2) ...)
                  (alias oe ie)
                  (... ...)))]))]))
```

40. In the above macro, we do one extra thing besides the new aliasing to make the macro more reliable. We wrap the body in appropriate ellipses to make sure that we can also define macros that themselves have ellipses inside of these sections. The above macro now successfully works in all of the previously troublesome conditions.

```
(== '((3 3) (3 3) #t #t)
    "Using modules and alias"
  (list
    ⟨ Simple defsec Example 23 ⟩
    ⟨ Break unhygienic defsec 26 ⟩
    ⟨ Test defsec with long record 34 ⟩
    ⟨ Test defsec with short record 36 ⟩))
```

41. Limitations. As it stands, the system defined above has two important limitations. Firstly, you cannot alpha rename identifiers from the input to the output. In the first generation of ChezWEB and in the above fully hygienic example `defsec/hyg`, because we are passing the captures directly through at the call site, we can associate different names with the captures variables in the body of the referenced section.

⟨ Example of alpha renaming 41 ⟩ ≡
```
(defsec/hyg (def-x) => (x) (define x 3))
(def-x y) (list y)
;=> '(3)
```

42. While it may be convenient at times to have different names, we have found the trouble of maintaining these threads throughout the program to be more trouble than they are worth, and the convenience of not explicitly managing them is useful more often than it is a limitation.

43. The system is also limited by where you can actually put a chunk. In this sense, our approach to chunk reordering is less flexible than a traditional literate program, because a traditional approach to code reordering allows arbitrary text to be substituted anywhere. In our approach, only places that lead to the reference being evaluated have significant meaning. Otherwise, the chunk references degenerate into identifiers. This means that a reference to a code section can appear only in definition or value contexts. We believe the benefits of hygienic code reordering outweigh this limitation. It is also possible to have two constructs for defining sections: one that is hygienic, and one that does purely textual substitution. This would give you the best of both approaches.

44. Discussion. Since the runtime system is written entirely in Scheme macros, the code reordering primitives can be used in their own right, without needing to use the entire system. This allows for simple experimentation and prototyping without incurring some of the overhead of the `WEB` workflow, when desirable. The fact that the Scheme macro system can so readily accommodate the requirements of implementing a system like Chez`WEB` demonstrates the appeal that systems like `syntax-case` have when dealing with domain specific languages. We cannot begin to imagine how we might implement a system like this for C or C++, for example, even should it be possible to do so without implementing a separate expander on top of these languages.

Chez`WEB` differs from a few semi-literate programming systems in that it implements the concept of named chunks, enabling code to be re-ordered in a fashion more suited to human consumption, as opposed to the ordering dictated by the compiler. Scheme itself is an extremely flexible language, though, and much of the complexity of traditional systems can be removed, since Scheme itself allows such a flexibility of expression beyond a language like Pascal, for which `WEB` was the original target. Nonetheless, Chez`WEB` makes it more convenient to move code around than would be normally found in standard Scheme. It condenses and simplifies things by hiding away some features of the normal macro system, and doing some extra work for the user. We believe that wrapping this abstraction into something like Chez`WEB` makes this paradigm more accessible, more productive, and more useful than it would be should one simply try to achieve a similar result with comments and without something like `defsec`.

Enforcing hygiene in code sections provides a level of safety that Schemers usually get from macros, and allows a more precise reasoning about the behavior of the chunks that would not normally be possible. That is, an user can now gain the benefits of Scheme's hygiene and referential transparency inside of the literate programming system as well as the macro system proper. This brings macros and documentation systems closer together, rather than making them two distinct things; a documentation system becomes just another set of macros, with the same rules as any other Scheme program.

45. As a meta note, we do not believe that it should have been necessary to iterate over two generations to reach the macro that we eventually defined. We did pursue the literature on macro programming, but we were unable to find enough guidance to help us avoid the pitfalls that we demonstrated in this paper. Fortunately, these pitfalls are quick to manifest in `syntax-case` so we were able to make progress quickly, despite these missteps. We hope that this paper and others like it may serve to guide macro programmers who may be searching for similar intuitive grasps of macro design, something that we believe is somewhat lacking in the programming literature available today, especially concerning non-trivial macros.

46. We would also like to note that the implementation of these macros was greatly simplified by the `module` form in Chez Scheme. The ease and appropriateness with which it allows one to compose and create syntaxes surrounding namespace management should be considered, and we would encourage other implementations to provide similar functionality.

47. The syntax that we have chosen for our system closely mirrors that of the `CWEB` system, but there is no reason not to support other syntaxes on top of our basic runtime. The Scribble system [10] is particularly attractive in this regard, and we hope to pursue this further.

48. Related Work. Literate programming is not new in the Scheme community. The SchemeWEB [9] system is a semi-literate programming system that allows one to combine LaTeX and Scheme code together. It is semi-literate because it does not have any features for reorganizing code. Indeed, SchemeWEB is remarkably simple in that it has no forms at all. Instead, top-level Scheme forms are identified using a heuristic and everything else is treated as documentation. This makes it rather easy to begin using it. However this does limit some of the ability to produce higher quality outputs without more work on the documentation side.

The Racket Scribble system [10] provides a reader syntax for writing documentation that nicely combines with literate programming systems. Indeed, Scribble/LP is an implementation of a traditional literate programming system. It implements the non-hygienic semantics for code relocation, and is Racket [3] specific, since it relies on the reader features available in Racket. There is also an Ikarus port of the Scribble system.

`CWEB` [6] is the traditional literate programming system that inspired Chez`WEB`. It is targeted especially to C/C++, and contains many special forms for extending the abilities of the C language. It is implemented as a preprocessor for both tangling and weaving.

SLaTeX [11] and STeX [13] are two distinct systems that provide means of typesetting Scheme code. They are actually more modes for Scheme typesetting, since SLaTeX has no native means of evaluating a SLaTeX file as a Scheme program. STeX has interesting modes that let the typesetter use Scheme to generate output to be typeset, but similarly focuses on typesetting, rather than on code evaluation.

49. Future Plans. We are now in the second generation of Chez`WEB` and have a number of plans for where we want to take Chez`WEB`. At the top of that list is providing a pretty printer, but we also wish to enable more syntaxes, such as Scribble to be used with the system. We would also like to expand the sorts of indexing that we can do and deal with libraries properly. The first generation architecture allowed us to work directly with libraries and format them specially, but our current system is designed in such a way that libraries must remain as Scheme constructs, rather than Chez`WEB` having any special knowledge of them. We hope that this can change. The interaction of chunks with libraries poses an interesting situation, since our approach to code section substitution requires that the code chunks be visible in the namespace of the Scheme program.

50. Conclusion. We have discussed the implementation and motivation for Chez`WEB`, a hygienic literate programming system. We discussed the design limits and philosophy that have lead to the current generation of the system. In particular, we examined the runtime system for Chez`WEB` in detail through a series of refinements on a basic macro. We have demonstrated the interaction and synergy that can be achieved with syntax based modular abstractions. Some effort has been made to elucidate the issues of code documentation and how meta-languages may be developed in Scheme that compose simply with other macros, as opposed to the use of monolithic preprocessors.

We hope that the detailed discussion of the implementation of named chunks demonstrates the practical construction of a fairly involved, but deceptively simple looking macro. It is hoped that while the construction will convince the reader of the care necessary to accomplish such a thing, the reader will yet be convinced of the power and accessibility of such macros given the excellent macro system provided by R6RS Scheme.

51. Acknowledgments. I readily acknowledge and thank the reviewers and commenters that examined numerous drafts of this paper, both anonymously and not so much. Their feedback greatly improved the paper, but I take full responsibility for its current state, good or bad.

References

[1] Dybvig, R. Kent, Robert Hieb, and Carl Bruggerman. 1992. "Syntactic Abstraction in Scheme." *Lisp and Symbolic Computation* 5, no. 4 (December): 83-110.

[2] Dybvig, R. Kent. 2011. *Chez Scheme*. `http://www.scheme.com`

[3] Flatt, Matthew and PLT. "Reference: Racket." *PLT Technical Report 1* (2010): PLT Inc. `http://www.racket-lang.org`.

[4] Knuth, Donald E. 1986. *TeX: The Program*. Vol. 2 of Computers **T**ypesetting. Addison-Wesley Professional (January).

[5] Knuth, Donald E. 1992. "Literate Programming." In *CSLI Lecture Notes* (27), xvi+368.

[6] Knuth, Donald E. and Silvio Levy. 1993. *The CWEB System of Structured Documentation*. Massachusetts: Addison-Wesley. `http://www-cs-faculty.stanford.edu/~uno/cweb.html`.

[7] Literate Haskell. `http://www.haskell.org/haskellwiki/Literate_programming`.

[8] Ramsey, Norman. 1994. "Literate Programming Simplified." *IEEE Software* 11, no. 5 (September): 97-105.

[9] SchemeWEB. 2001. `http://www.ctan.org/tex-archive/web/schemeweb2`

[10] Scribble. `http://docs.racket-lang.org/scribble/`.

[11] Sitaram, Dorai. 2009. *How to Use SLaTeX*. `http://evalwhen.com/slatex/slatxdoc.html`.

[12] Sperber, Michael, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, eds. 2010. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge: University Press (April). http://www.r6rs.org

[13] Dybvig, R. Kent. *STeX*. Internal System used at Indiana University.

[14] Waddell, Oscar and R. Kent Dybvig. 1999. "Extending the Scope of Syntactic Abstraction." *Symposium on Principles of Programming Languages* (January): 203-213. http://www.cs.indiana.edu/ dyb/pubs/popl99.pdf.

52. Testing. We use the following `==` macro as a means of verifying our results. It takes in an expression that should evaluate to the expected value, and a string that is a description of the test, followed by a code body.

```
(== expected desc-str body+ ...)
```

It should print out one of three messages:

```
desc-str: Passed.
desc-str: Failed with bad value ----.
desc-str: Failed with error...
```

If the test fails it should either print the error message or it should print the bad value that it encountered.

⟨ Define testing primitive `==` 52 ⟩ ≡

```
(define (c->s c)
  (with-output-to-string
    (lambda () (display-condition c))))
(define errstr "Failed with error~n~8,8t~a")
(define badstr "Failed with bad value ~s.")
(define-syntax (== x)
  (syntax-case x ()
    [(_ exp str b1 b2 ...)
     #'(let ([val exp])
         (printf "~a: ~a~n" str
           (guard
             (c [else
                  (format errstr (c->s c))])
             (let ([act (let () b1 b2 ...)])
               (if (equal? act val)
                   "Passed."
                   (format badstr
                           act)))))))))
(indirect-export == c->s errstr badstr)
```

This section exports ==.

This code is used in section 25.

53. The above code relies on `format` and `printf` from Chez Scheme, which is just a Common Lisp compatible implementation of those procedures.

54. Index.

⟨ Break unhygienic `defsec` 26 ⟩ Used in sections 27, 35, and 40.
⟨ Define testing primitive `==` 52 ⟩ Used in section 25.
⟨ Define unhygienic `defsec` 24 ⟩ Used in sections 25 and 27.
⟨ Define `map-fact` 19 ⟩ Used in section 21.
⟨ Define `reftrans` with `syntax-case` 13 ⟩
⟨ Define `with-implicit` 15 ⟩
⟨ Define `with-implicit` based `defsec` 33 ⟩ Used in sections 35 and 37.
⟨ Demonstrate `syntax-case` capture 14 ⟩
⟨ Example of alpha renaming 41 ⟩
⟨ Example using `module` 16 ⟩
⟨ Hygiene example, define `hidden` 8 ⟩ Used in section 9.
⟨ Hygiene example, test `hidden` 9 ⟩
⟨ Illustrate guarantees 21 ⟩
⟨ Ref. trans. example, define `reftrans` 10 ⟩ Used in section 11.
⟨ Ref. trans. example, test `reftrans` 11 ⟩
⟨ Simple `defsec` Example 23 ⟩ Used in sections 25, 35, and 40.
⟨ Test `defsec` with long record 34 ⟩ Used in sections 35 and 40.
⟨ Test `defsec` with short record 36 ⟩ Used in sections 37 and 40.
⟨ The load message 4 ⟩ Used in section 5.

Table of Contents