

Scheme MiniLab

Version 4.1.3

Matthew C. Jadud

April 8, 2009

The Scheme minilab gives you an opportunity to explore one of two things about the Scheme programming language.

- **Macros:** Using [syntax-rules](#), you will implement several simple macros. As has been mentioned before, the language CFWAE is remarkably similar to Scheme. That being the case, you can actually write macros to implement `if0`, `with`, and `fun`. This will allow you to then program in CFWAE without having to run your interpreter. A little double-bonus extra effort is required to implement FEARCOW in macros, but it can be done.
- **System Scripting:** If you're not interested in exploring the world of metaprogramming, you could instead explore some of PLT Scheme's features that make shell scripting pretty easy.

Either way, the hope is that you can carry out these explorations in roughly 45 minutes. Enjoy.

1 Exploring Macros

Macros can either be written using `syntax-rules` or `syntax-case`. The latter is more powerful, but the former is simpler for getting started. Hence, we'll use `syntax-rules`.

Remember the grammar for CFWAE?

```
<CFWAE2> ::= <num>
          || {<binop> <CFWAE2> <CFWAE2>}
          || <id>
          || {if0 <CFWAE2> <CFWAE2> <CFWAE2>}
          || {with {{<id> <CFWAE2>} ...} <CFWAE2>}
          || {fun {<id> ...} <CFWAE2>}
          || {<CFWAE2> <CFWAE2> ...}

<binop> ::= +,-,*,/
<num>   ::= number?
<id>    ::= symbol?
```

Lets say we want to define the syntax for `if0`. I'll do that as an example.

First, we're defining syntaxes, not functions, so we use `define-syntax`.

```
(define-syntax if0 ...)
```

Next, we need to define `if0` as one or more syntactic *patterns*. That sounds tricky, but you already know the pattern:

```
{if0 <test-expression> <>true-expression> <>false-expression>}
```

You know the pattern because you know the grammar and you've written the parser. Therefore, writing a macro for this in Scheme is going to be a piece of cake.

To start, we say that we're going to be writing some `syntax-rules`:

```
(define-syntax if0
  (syntax-rules ()
    [<pattern> <generated code>]
    .... <possibly more patterns> ...))
```

The empty "()" means that there aren't any keywords in our patterns. (In short, for these examples, there will always be an empty set of parens there.) The pattern is very straightforward:

```
(define-syntax if0
  (syntax-rules ()
    [{if0 test-exp true-exp false-exp}
     <generated code>]))
```

That might look way too easy. You should feel dirty right now... you've just implemented the parser for `if0`. Think about all that code you've written this semester..

Now, what should we output? We need to generate Scheme code that is *equivalent* to the `if0` in CFWAE. That says to me that we need to do the following:

- Generate an `if` statement.
- The `if` statement needs to compare the value of the `test-exp` to the number `0`.
- The rest of the generated code should just execute the code in our pattern.

Easy-peasy?

```
(define-syntax if0
  (syntax-rules ()
    [{if0 test-exp true-exp false-exp}
     (if (= test-exp 0)
         true-exp
         false-exp)]))
```

That's it. However, you have to remember something important: the `<generated code>` is **not** something that runs. Instead, you have to remember that this macro is saying "if the `<pattern>` is matched, then you should generate, at compile-time, the `<generated code>`." So, really, you can think of macros as something that *extend the compiler*. Or, if you prefer, they are Scheme functions that take Scheme syntax as input, and produce Scheme syntax as output. Along the way, you can re-arrange the syntax and produce all kinds of wacky new code.

1.1 More than one thing

To capture more than one piece of syntax in your pattern, you're going to need something more, however. For example, I have said before that in CFWAE (and, for that matter, in Scheme) that `with` is equivalent to `lambda`. In Scheme, `with` is called `let`.

As it turns out, we don't need `let` in our language directly: we can write it as a macro. I'll call my `let` something different (like `let-fu`) just so things are clear.

If I write:

```
(let-fu ((x 3) (y 5))
  (+ x y))
```

That is the same as saying

```
((lambda (x y) (+ x y)) 3 5)
```

We can write a macro called `let-fu` that will perform this transformation for us. Just as before, we start by defining a syntax transformer called `let-fu`:

```
(define-syntax let-fu
  (syntax-rules ()
    [<pattern> <code>]))
```

Now, we have to decide what the pattern will be.

```
(define-syntax let-fu
  (syntax-rules ()
    [(let-fu ((id exp) ...) body)
     <code>]))
```

The trick here is the `...`. The ellipsis means “zero or more.” So, we are looking for zero or more sub-expressions that have an identifier and an expression. In short, we’re looking for all the *binding pairs* in my `let-fu` expression.

Now, what do we generate? We need to generate a new `lambda`, and we want all the parameters of the function to be the `ids`, and the function to be applied to all of the `exps`.

```
(define-syntax let-fu
  (syntax-rules ()
    [(let-fu ((id exp) ...) body)
     ((lambda (id ...) body) exp ...)]))
```

The rule is that anywhere we used an ellipsis in the pattern, we have to use it with those variables in any code we generate. Because we captured a list of `ids`, that means we must use them in the same way in our generated code.

1.2 What is going on in there?

If you want to debug your macros, and see what the macro is generating, try throwing a `quote` around the output of your macro. This will cause it to generate a quoted list (or, if you prefer, a list of Scheme data) so you can see the generated code instead of having it evaluated.

```
(define-syntax let-fu
  (syntax-rules ()
    [(let-fu ((id exp) ...) body)
     '(lambda (id ...) body) exp ...]))
```

Now, when we write a `let-fu` expression like:

```
(let-fu ((x 3) (y 5)) (+ x y))
```

we get back:

```
((lambda (x y) (+ x y)) 3 5)
```

1.3 Exercises

Go ahead and implement macros for the following parts of CFWAE.

- **fun**: This should become a Scheme `lambda`.
- **with**: This should become a Scheme `let`.
- **blueprint**, **create**, and **invoke**: This may take a bit more effort; you will need to generate the appropriate object-oriented code using the PLT Scheme object system. Consider this double-extra-bonus if you do it.

Like the labs, you might start with single bindings, and then do multiple bindings. The difference in this case is very, very small, but it is still work starting with the simplest case.

And that's it. If you write these two macros, you should be able to Run them, and in the Interactions window, copy-paste some CFWAE programs and run them directly.

You will, at that point, have done the entire Extended Interpreter lab in roughly 9 lines of code and (hopefully) around 30 minutes.

1.4 That's it?!

Macros are non-trivial to implement and get right. They are the subject of decades of research, and incredibly powerful abstractions can be built using them. Currently, it is all the rage to talk about “domain-specific languages,” or DSLs. Macros are the original tool for defining DSLs, and have been around since the early days of Lisp.

readscheme.org has a number of papers on macros. (I can't figure out how to hyperlink to the site, so you'll just have to copy-paste the URL.)

<http://library.readscheme.org/page3.html>

For more practical information about writing Scheme macros, I recommend *The Scheme Programming Language, 3rd ed.* by R. Kent Dybvig.

<http://www.scheme.com/tspl3/>

2 System Scripting

You can do a lot of scripting of the Linux system in PLT Scheme. I prefer it infinitely to using Bash (bash), C Shell (tcsh), or any other *NIX shell for that matter. The reason is simple: Scheme is more powerful, better defined, and I'm less likely to screw up and delete half of my home directory. These all, for me, are good reasons to write my shell scripts in Scheme.

Choose one of the two following exercises. They'll each introduce you to something new in Scheme.

2.1 Traversing the filesystem

I'd like to know how big my home directory is. There are ways I could do this on the command line, but why use a single UNIX command when I could write a script, right? (Wrong, but just agree for the moment.)

PLT Scheme gives us a bunch of tools for interacting with the filesystem. You can write a function that walks through my home directory using the following tools:

- `directory-list`
- `file-exists?`
- `directory-exists?`
- `build-path`

I had 18 lines of code, including whitespace. The template for my code looked like this:

```
(define (calc path)
  (calc-size path (directory-list path)))

(define (calc-size current-path dir-ls)
  ....)
```

When I was done, I was able to use the function `calc` as follows:

```
> (calc "/Users/jadudm/Music")
11168925092
```

Like many problems, if you clearly understand the structure of the data, the solution should present itself. Directory hierarchies are trees, so the solution to this problem should look very much like the kind of code you've written a lot of this semester.

2.2 If you want...

If you knock that down, you could pretty it up a bit. I'd rather the result came back in a more human-readable form.

```
> (usage "/Users/jadudm/Music")
"11G 168M 925K"
```

2.3 Handling the command line

I'd say more, but I'm exhausted.

Take a look in the documentation for the `scheme/cmdline` library. There is quite a bit there, so I've provided a starter file:

```
(require scheme/cmdline)

(define say-hello
  (command-line
   #:program "hello"
   #:once-each
   [("-n" "--name") name
    "Say hello to someone."
    (printf "Hi there, ~a!~n" name)]))

say-hello
```

Save that to "hello.ss". Then, on the command-line, run:

```
mzscheme hello.ss
```

That will seem to do nothing. So, add a flag:

```
mzscheme hello.ss -h
```

That will tell you how to use your script.

If you'd like to compile your program, try typing:

```
mzc --exe hello hello.ss
```

Then, you should be able to say:

```
./hello -h
```

to run the program. You've now written and compiled Scheme program that takes command-

line arguments. Tie this together with the previous exercise, and you have a program that you can use to get the size of any directory.

2.4 What else?

If you look in the documentation for `system`, you'll find out how to spawn new processes, and probably a whole bunch more as well. Fun fun!

3 Wrapping Up

You may be asking yourself “Matt, is that it?”

For this minilab, the answer is “yes.” It is supposed to take you 30 minutes to do, if you only do one of these exercises. On the other hand, the answer is “no.” Scheme is a language, and therefore, can do most anything you might imagine doing. There are libraries and tools aplenty. I’m fond of it because the core language is small, but macros and other powerful tools let me express complex ideas simply. That, and a good compiler allows me to take those programs and then compile and distribute them on every major operating system.

If you want to see what kinds of libraries people have written for PLT Scheme, take a look at

<http://planet.plt-scheme.org/>

Your mileage with Scheme may vary. This minilab represents the last of our official explorations of the Scheme programming language. And, as minilabs go, this is just intended to let you see some of the other tools that the PLT Scheme implementation provides to you as a programmer.

3.1 About this document

This document was written in Scribble, a documentation system written in and for PLT Scheme. Think of it as a Schemey wrapper around LaTeX, and you’re on the right path.

This document was generally written late at night and/or very early in the morning. In-between, there were several diapers and some walking in circles around the house.