CSE 341, Spring 2011, Lecture 15 Summary

Last update: 5/1/11 to use Racket's mutable cons cells in `delay/force`.

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

This lecture discusses *macros*, which let programmers extend the *syntax* of a programming language. A macro definition describes how a macro use is *rewritten* into other syntax that is (already) in the programming language. There are many inappropriate uses of macros and many languages with bad macro systems. Fortunately, Scheme has a extraordinarily good and powerful approach to macros and there are some good idioms using macros. So we will study Scheme's macro system and through it learn some of the pitfalls of macros in general. More specifically, we will see how Scheme's macro system is *hygienic* (a technical term described below), which is usually what you want and helps avoid many (but not all) of macros' problems.

The first thing we can consider with macros is how the rewriting is defined. For example, consider a macro that, "replaces every use of `car` with `hd`." In macros systems that does *not* mean some variable `cart` would be rewritten as `hdt`. So the implementation of macros has to at least understand how a programming language's text is broken into *tokens* (i.e., words). We can then ask if macros do or do not understand parenthesization. For example, in C/C++, if you have a macro

```
#define ADD(x,y) x+y
```

then `ADD(1,2/3)*4` gets rewritten as `1 + 2 / 3 * 4`, which is *not* the same thing as `(1 + 2/3)*4`. So in such languages, macro writers generally include lots of explicit parentheses in their macro definitions, e.g.,

```
#define ADD(x,y) ((x)+(y))
```

In Scheme, macro expansion preserves the code structure so this issue is not a problem. Finally, we can ask if macro expansion happens for binding occurrences. If not, then local variables can shadow macros, which is probably what you want. For example, suppose we have:

```
(let* ([hd 0] [car 1]) hd) ; evaluates to 0
```

If we replace the `car` above with `hd`, then the expression now evaluates to 1. In Scheme, macro expansion does not apply to variable definitions, i.e., the `car` above is different and shadows any macro for `car` that happens to be in scope.

Let's now walk through the syntax we will use to define macros in Scheme. (There have been many variations just in Scheme over the years; this is a modern approach we will use in this course.) Here is a macro that lets users write `(my-if e1 then e2 else e3)` for any expressions `e1`, `e2`, and `e3` and have it mean exactly `(if e1 e2 e3)`:

```
(define-syntax my-if
  (syntax-rules (then else)
    [(my-if e1 then e2 else e3)
     (if e1 e2 e3)]))
```

- `define-syntax` is the special form for defining a macro.
- `my-if` is the name of our macro. It adds to the environment so that expressions of the form `(my-if ...)` will be rewritten according to the syntax rules in the rest of the macro definition.
- `syntax-rules` is a keyword
- The next parenthesized list (in this case `(then else)`) is a list of "keywords" for this macro, i.e., any use of `then` or `else` is just syntax whereas anything not in this list (or `my-if` itself) represents an arbitrary expression.

- The rest is a list of pairs: how `my-if` might be used and how it should be rewritten if it used that way.
- In this example, our list has only one option, `my-if` must be used in an expression of the form `(my-if e1 then e2 else e3)` and that becomes `(if e1 e2 e3)`. Otherwise an error result. Note the rewriting occurs *before* any evaluation of the expressions `e1`, `e2`, or `e3`, unlike with functions. This is what we want for a conditional expression like `my-if`.

Whereas `my-if` creates a more verbose way of writing something we already could do, these macros provide a more concise way to use "delay" and "force" as described in the previous lecture:

```
(define-syntax my-delay
  (syntax-rules ()
    [(my-delay e)
     (mcons #f (lambda () e))]))
(define-syntax my-force
  (syntax-rules ()
    [(my-force e)
     (let ([x e])
       (if (mcar x)
           (mcdr x)
           (begin (set-mcar! x #t)
                  (set-mcdr! x ((mcdr x)))
                  (mcdr x))))]))
```

We can write `(my-delay some-computation)` to get back something and `some-computation` will not be executed until that something is "passed" to `my-force`. The convenience here is that the user of `my-delay` does not write the thunk, that is inserted by the rewriting in the definition of the `my-delay` macro.

In the definition of `my-force` it is very good style that we use a local variable (`x`) to hold the result of evaluating `e`. Otherwise, we would evaluate `e` multiple times. In code like:

```
(let ([t (my-delay some-complicated-expression)])
  (my-force t))
```

this does not matter since `t` is already bound to a value, but in code like:

```
(my-force (my-delay some-complicated-expression))
```

not using `x` would lead to creating multiple different thunks. Now code like `(my-force (my-delay ...))` is not very common, but that is no reason for `my-force` not to work as expected even for this case.

As a simpler but less useful example that lets us investigate the issue of when macro arguments are evaluated and in what environment, let's consider a macro that doubles its argument. This is poor style because if you want to double an argument you should just write a function: `(define (double x) (* 2 x))` or `(define (double x) (+ x x))` which are both equivalent. That said, the "macro versions" are *not* equivalent:

```
(define-syntax double1
  (syntax-rules ()
        [(double1 e)
         (* 2 e)]))
(define-syntax double2
  (syntax-rules ()
    [(double2 e)
     (+ e e)]))
```

The reason is `double2` will evaluate its argument twice. So `(double1 (begin (print "hi") 17))` prints `"hi"` once but `(double2 (begin (print "hi") 17))` prints `"hi"` twice. The function versions print `"hi"` once, simply because, as always, function arguments are evaluated to values before the function is called.

To fix `double2` without "changing the algorithm" to multiplication instead of addition, we use a local variable like we did in `my-force`:

```
(define-syntax double3
  (syntax-rules ()
    [(double3 e)
     (let ([x e])
       (+ x x))]))
```

Using local variables in macro definitions to control if/when expressions get evaluated is exactly what you should do, but in less powerful macro languages (again, C/C++ is an easy target for derision here), local variables in macros are typically avoided. The reason has to do with scope and something that is called *hygiene*. For sake of example, consider this silly variant of `double3`:

```
(define-syntax double4
  (syntax-rules ()
    [(double4 e)
     (let* ([zero 0]
            [x e])
       (+ x x zero))]))
```

In Scheme, this macro always works as expected, but that may/should surprise you. After all, suppose I have this use of it:

```
(let ([zero 17])
   (double4 zero))
```

If you do the syntactic rewriting as expected, you will end up with

```
(let ([zero 17])
  (let* ([zero 0]
         [x zero])
       (+ x x zero)))
```

But this expression evaluates to 0, not to 34. The problem is a *free variable* at the macro-use (the `zero` in `(double4 zero)`) ended up in the scope of a local variable in the macro definition. That is why in C/C++, local variables in macro definitions tend to have funny names like `__x_hopefully_no_conflict` in the hope that this sort of thing will not occur. In Scheme, the rule for macro expansion is more sophisticated to avoid this problem. Basically, every time a macro is used, all of its local variables are *rewritten* to be fresh new variable names that do not conflict with anything else in the program. This is "one half" of what by definition make Scheme macros hygienic.

The other half has to do with free variables in the *macro definition* and making sure they do not wrongly end up in the scope of some local variable where the macro is used. For example, consider this strange code that uses `double3`:

```
(let ([+ *])
   (double3 17))
```

The naive rewriting would produce:

```
(let ([+ *])
  (let ([x 17])
     (+ 17 17)))
```

Yet this produces 17^2, not 34. Again, the naive rewriting is *not* what Scheme does. Free variables in a macro definition always refer to what was in the environment where the macro was defined, not where the macro was used. This makes it much easier to write macros that always work as expected. Again macros in C/C++ work like the naive rewriting.

There are situations where you do not want hygiene. For example, suppose you wanted a macro for for-loops where the macro user specified a variable that would hold the loop-index and the macro definer made sure that variable held the correct value on each loop iteration. Scheme's macro system has a way to do this, which involves explicitly violating hygiene, but we won't go into the features needed.

Finally, let's consider two macro definitions that use multiple cases for how to do the rewriting. First, here is a macro that lets you write up to two let-bindings using `let*` semantics but with fewer parentheses:

```
(define-syntax let2
  (syntax-rules ()
    [(let2 () body)
     body]
    [(let2 (var val) body)
     (let ([var val]) body)]
    [(let2 (var1 val1 var2 val2) body)
     (let ([var1 val1])
       (let ([var2 val2])
         body))]))
```

As examples, `(let2 () 4)` evaluates to 4, `(let2 (x 5) (+ x 4)` evaluates to 9, and `(let2 (x 5 y 6) (+ x y))` evaluates to 11.

In fact, given support for recursive macros, we could redefine Scheme's `let*` entirely in terms of `let`. We need some way to talk about "the rest of a list of syntax" and Scheme's `...` gives us this:

```
(define-syntax my-let*
  (syntax-rules ()
    [(my-let* () body)
     body]
    [(my-let* ([var0 val0]
               [var-rest val-rest] ...)
              body)
     (let ([var0 val0])
       (my-let* ([var-rest val-rest] ...)
                body))]))
```

Since macros are recursive, there is nothing to prevent you from generating an infinite loop or an infinite amount of syntax during macro expansion, i.e., before the code runs. The example above does not do this because it recurs on a shorter list of bindings.