CSE 341, Spring 2008, Lecture 13 Summary

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.

For the next couple weeks, we will use the Scheme programming language in our lectures and homeworks rather than ML. Our focus will remain largely on key programming language constructs. We will "switch" to Scheme because several of these concepts shine better in Scheme. That said, Scheme and ML share many similarities — both are mostly functional languages, i.e., mutation exists but is discouraged — so it will be good to see functional programming in a somewhat different setting. This lecture is a bit more basic since we need to introduce Scheme before we start using it to study more advanced concepts. The most interesting topic in this lecture is Scheme's different rules for creating local environments (`let` vs. `letrec` vs. `let*`).

Before diving into Scheme specifics, here are some differences from ML we will be studying:

- Every binding is mutable
- The syntax is extremely minimal; everything is an atom or a parenthesized sequence
- The language is dynamically typed
- We can treat programs as data
- It has a (hygienic) macro system
- The DrScheme system/language we will use has a module system that takes a different approach to abstraction, using `define-struct`
- There is no pattern-matching
- (One feature we will not have time for is first-class continuations.)

We will use the DrScheme environment. There are a couple key notes for setting it up for this course:

- Set the language level to PLT→Pretty Big
- The current version is 372, but the labs have version 371 installed. Either version is fine (we won't rely on anything that has changed recently)
- If you use `attu`'s installation of DrScheme (which isn't necessary, DrScheme is installed on the Windows machines in the lab and it's very easy to install on your own machine), you will get a couple error messages when it starts. You can ignore these; they refer to an add-on tool we are not using.

A Scheme program is a sequence, most elements of which are definitions. A definition like

```
(define a 3)
```

extends the top-level environment so that `a` is bound to 3. Therefore a subsequent definition like

```
(define b (+ a 2))
```

would bind `b` to 5. In general, if we have `(define v e)` where `v` is a variable and `e` is an expression, we evaluate `e` to a value and change the environment so that `v` is bound to that value. Other than the syntax, this should seem very familiar. In Scheme, *everything* is prefix, such as the addition function used above.

An anonymous function that takes one argument is written `(lambda (x) e)` where the argument is the variable `x` and the body is the expression `e`. So this definition binds a cubing function to `cube1`:

```
(define cube1
  (lambda (x)
    (* x (* x x))))
```

In Scheme, different functions really take different numbers of arguments and it is a run-time error to call a function with the wrong number. A three argument function would look like `(lambda (x y z) e)`. However, many functions can take any number of arguments. The multiplication function, `*`, is one of them, so we could have written

```
(define cube2
  (lambda (x)
    (* x x x)))
```

We may learn later how to define our own variable-number-of-arguments functions. There is a very common form of syntactic sugar for defining functions where you do not need to use the `lambda` keyword:

```
(define (cube3 x)
  (* x x x))
```

This is more like ML's `fun` binding, but in ML `fun` is not just syntactic sugar since it is necessary for recursion. In Scheme, we can use `lambda` and a `define` for recursive functions, so this shorter and preferable form really is just sugar.

Here is some more code to demonstrate conditionals and currying:

```
(define (cube3 x)
  (* x x x))

(define (pow1 x y)
  (if (= y 0)
      1
      (* x (pow1 x (- y 1)))))

(define pow2
  (lambda (y)
    (lambda (x)
      (pow1 x y))))

(define cube4 (pow2 3))
```

The syntax for a conditional is `(if e1 e2 e3)` where `e1`, `e2`, and `e3` are expressions. If `e1` evaluates to `#f`, then `e2` is evaluated and the result returned, else `e3`. Everything that is not `#f` is "true", not just the constant `#t`. Notice how this is much more flexible type-wise than anything in ML.

While the code above used currying, there is no special syntax for it. If `f` is a curried 3-argument function, you would have to call it with `(((f e1) e2) e3)`.

You have probably noticed that Scheme uses lots of parentheses. Here are some relevant comments:

- For reasons your instructor has never fully understood, many people seem to dislike Scheme just because of this syntactic decision. It really does seem to be, "judging a book by its cover." Moreover, few of these people seem to mind XML, which is just as parenthetical, just with things like `<foo>...</foo>` instead of `(foo ...)` (and notice which is longer).

- *Parentheses matter!* You *cannot* just decide to take an expression like `(+ 2 3)` and add more parentheses. The expression `((+ 2 3))` means, "evaluate the addition to get 5 then try to apply 5 as a function to 0 arguments." This will understandably result in an error. Additional examples of wrong parenthesization are below.

- Scheme's parentheses lead to a *very* simple syntax for the language. Because everything is prefix and parenthesized, there is never any ambiguity about how things associate with each other. In other languages, you might wonder whether `if true then 1 else 3 + 4` evaluates to 1 or 5, but Scheme's `(if #t 1 (+ 3 4))` makes such questions obvious.

In general, the syntax of everything in Scheme is either an *atom* (some basic thing like a number, string, symbol, `#t` or `#f`, etc.) or a *sequence* of things `(t1 t2 ... tn)`. If `t1` is a *special form* like `define` or `lambda` or some other things we'll see shortly, then that affects what the remaining terms in the sequence mean and where the parentheses should be. Otherwise, the sequence is a function call – we evaluate `t1` to a value that should be a function closure, the other expressions to values, and we do the call.

Programmers new to Scheme sometimes struggle with remembering that *parentheses matter* and determining why programs fail when they are misparenthesized. As an example consider these six definitions. The first is a correct implementation of factorial and the others are wrong:

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))   ; 1
(define (fact n) (if (= n 0) (1) (* n (fact (- n 1))))) ; 2
(define (fact n) (if = n 0 1 (* n (fact (- n 1)))))     ; 3
(define fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))   ; 4
(define (fact n) (if (= n 0) 1 (* n fact (- n 1))))     ; 5
(define (fact n) (if (= n 0) 1 (* n ((fact) (- n 1))))) ; 6
```

Line	Error
2	calls `1` as a function taking no arguments
3	uses `if` with 5 subexpressions instead of 3
4	defines a variable `fact` by calling `n` and then something else
5	calls `*` with a function as one of the arguments
6	calls `fact` with 0 arguments

Scheme has built-in lists, much like ML, and Scheme programs probably use lists even more often in practice than ML programs. Scheme does not have pattern matching, so we just use built-in functions for building lists, extracting parts, and seeing if they are empty. The function names are a historical accident.

Primitive	Description	Example
`()`	The empty list	`()`
`cons`	Construct a list	`(cons 2 ())`
`car`	Get first element	`(car some-list)`
`cdr`	Get tail of a list	`(cdr some-list)`
`null?`	Return `#t` for `()` and `#f` otherwise	`(null? some-value)`

Here are three examples using these functions:

```
(define (sumlist lst)
  (if (null? lst)
      0
      (+ (car lst) (sumlist (cdr lst)))))
(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1) (append (cdr lst1) lst2))))
(define (map f lst)
  (if (null? lst)
      ()
      (cons (f (car lst)) (map f (cdr lst)))))
```

There is also a built-in function `list` for building a list from any number of elements, so you can write `(list 2 3 4)` instead of `(cons 2 (cons 3 (cons 4 ())))`.

Unlike in ML, we do not have a type-checker restricting things like the type of list elements. A list can hold any values. So `(list #t "hi" 14)` is no problem. If you build such things, you either have to "remember" what is where or use primitive testing-functions like `null?` to see what you have. For example, `number?` evaluates to `#t` when applied to a number and `#f` otherwise.

Here is an example where we assume we have a list that holds either numbers or other lists that hold either numbers or other lists, etc., and we sum up all the numbers in all the lists:

```
(define (sum1 lst)
  (if (null? lst)
      0
      (if (number? (car lst))
          (+ (car lst) (sum1 (cdr lst)))
          (+ (sum1 (car lst)) (sum1 (cdr lst))))))
```

While the code above is fine, when you have many nested conditionals, the `cond` special-form is easier and better style:

```
(define (sum2 lst)
  (cond [(null? lst) 0]
        [(number?(car lst)) (+ (car lst)(sum2 (cdr lst)))]
        [#t (+ sum2 (car lst)) (sum2 (cdr lst))]))
```

Using square brackets is just style – they are completely interchangeable with ordinary parentheses. A `cond` just has any number of parenthesized pairs of expressions, `[e1 e2]`. The first is a test; if it evaluates to `#f` we skip to the next branch. Otherwise we evaluate `e2` and that is the answer. As a matter of style, your last branch should have the test `#t`, so you do not "fall off the bottom" in which case the result is some sort of "void object" that you do not want to deal with.

As with `if`, the result of a test does not have to be `#t` or `#f`. Anything other than `#f` is interpreted as true for the purpose of the test. It is sometimes bad style to exploit this feature, but it can be useful.

Now let's consider (pun intended) how to create local bindings in Scheme. There are let-expressions, which have the form:

```
(let ([x1 e1]
      [x2 e2]
      ...
      [xn en])
  e)
```

where the `x`'s are variables, the `e`'s are expressions, and as always the choice of brackets versus parentheses is just style. As you might expect, the result of evaluating `e1` is bound to `x1` in the body `e` — and similarly for `x2`, ..., `xn` — and the result of evaluating `e` is the overall answer. However, unlike in ML, `x1` is *not* in the environment used to evaluate `e2`, and so on. That is, the environment for evaluating `e1`, `e2`, ..., `en` is the same environment from "before" the let-expression. Therefore, this silly function doubles its argument:

```
(define (double1 x)
  (let ([x (+ x 3)]
        [y (+ x 2)])
        (+ x y -5)))
```

For behavior more like ML's let-expression, we have the variant `let*`, which is syntactically the same but each variable is in scope for the subsequent bindings. For example, this silly function also doubles its argument:

```
(define (double2 x)
  (let* ([x (+ x 3)]
         [y (+ x 2)])
    (+ x y -8)))
```

Notice we do not *need* `let*` since we could always just use n nested `let` expressions with 1 binding each instead of a `let*` with n bindings, but `let*` is shorter and better style.

Neither `let` nor `let*` allows recursion since the `e1`, `e2`, ..., `en` cannot refer to the binding being defined or any later ones. To do so, we have a third variant `letrec`, which lets us write:

```
(define (double3 x)
  (letrec ([y (+ x 2)]
           [f (lambda (z) (+ z y w))]
           [w (+ x 7)])
    (f -9)))
```

One typically uses `letrec` to define one or more (mutually) recursive functions, such as this very slow method for taking a non-negative number mod 2:

```
(define (mod2 x)
  (letrec
      ([even?(lambda (x) (if (zero? x) #t (odd? (- x 1))))]
       [odd? (lambda (x) (if (zero? x) #f (even? (- x 1))))])
    (if (even? x) 0 1)))
```

Alternately, you can get the same behavior as `letrec` by using local defines. Your instructor doesn't like this style as much, but others do:

```
(define (mod2_b x)
  (define even? (lambda(x)(if (zero? x) #t (odd? (- x 1)))))
  (define odd?  (lambda(x)(if (zero? x) #f (even? (- x 1)))))
  (if (even? x) 0 1))
```

It is worth understanding how exactly the 3 forms of let differ because how variables are looked up in an environment is a fundamental feature of a programming language. Different languages have different approaches, and Scheme just happens to have multiple variations.

In fact, at top-level (outside of any `define`), Scheme's rules are a bit different than any of `let`, `let*`, or `letrec`. To a first approximation, all the `define`'s in a program are in "one big `letrec`" so (unlike ML) an earlier function can use a later one and there is no special syntax for (mutual) recursion. So this works to define a function that increments its argument:

```
(define f (lambda (x) (+ x a)))
(define a 1) ; order does not really matter
(define b (f 3)) ; b bound to 4
```

However, we do evaluate definitions in order and a variable must be in the environment when we look it up. So this does not work:

```
(define x (+ y 1)) ; y is not bound so you get an error
(define y 7)
```

In our first example, we did not look up `a` until it was in the top-level environment. The top-level environment is *changed* (mutated!) with each `define` and all closures have access to this changed environment.

While this is convenient for putting functions in whatever order you want, it is a problem if you redefine something at the top-level. In ML, that would not affect any earlier bindings and it would just shadow the redefined variable for later bindings. In ML, it *changes* what something is bound to for the whole program. Scheme programs typically just assume this doesn't happen and that's what you should assume too on your homework.

But just to understand, what redefinition means, consider something like this:

```
(define (g x) (+ x 13))
(define (f x) (+ (g x) 1))
(define g 37)
(f 12)
```

This program will fail since the body of `f` will look up `g` in the top-level environment and get 37, the redefined value, and this is not a function. Whenever you are programming and mutation can potentially make your code incorrect, a solution is *make a local copy before the mutation can occur*. In this case, a solution would be:

```
(define (g x) (+ x 13))
(define f
  (let ([g g])
    (lambda (x)
      (+ (g x) 1))))
(define g 37)
(f 12)
```

Now the closure bound to `f` looks up a non-top-level `g` that is bound to the original closure that adds 13 to its argument. Notice the `let` must be *outside* the `lambda` though; otherwise the local `g` won't get created until the function bound to `f` is *called* and that is too late.

However, our revised definition for `f` is still not enough to ensure the closure always adds 14 to its argument. After all, `+` is just a top-level function, so evil code like `(define + -)` could change the behavior of `f` to subtract 14 from its argument. We can protect against this as well:

```
(define f
  (let ([g g]
        [+ +])
    (lambda (x)
      (+ (g x) 1))))
```

Again, nobody actually does this in Scheme. The point is just that the possibility of mutation makes it very difficult to write correct code.